Design of Service Robots

Experiences Using Software Engineering

© PHOTODISC, DIGITAL STOCK, & JOHN FOXX

According to the International Federation of Robotics (IFR), "a service robot is a robot which operates semi or fully autonomously to perform services useful to the well being of human and equipment, excluding manufacturing operations" [1]. These devices are typically complex systems requiring the input of knowledge from numerous disciplines. The authors have been using different software engineering techniques for the last 15 years, integrating new paradigms in the service robot development process as they emerged. This has made it possible to achieve rapid development of applications and subsequent maintenance. During the early years (1993–1998), our efforts were directed at the development of software for various kinds of teleoperated robots to perform maintenance tasks in nuclear power plants [2]; during a second phase (1999–2006), we built applications for ship-hull cleaning robots [3]. All this time, we have been applying all the possibilities of software engineering, from the use of paradigms for structured and object-based programming in early developments to the adoption of the current model-driven approach [model-driven engineering (MDE)] [4].

Digital Object Identifier 10.1109/MRA.2008.931635

In the course of the research conducted by the Division of Electronics Engineering and Systems (DSIE) Research Group, we can see a parallelism between each new need and the software development paradigm that has been applied to meet that need. The first applications were intended for teleoperated systems, each specializing in a highly specific maintenance task within a fully structured operational environment. The challenge lays in how to reuse as much as possible the code of one application in another application. To that end, we developed an architecture founded on the use of object-based programming and the development of generic control modules. With the Ada95 language, it was possible to put these ideas directly into practice, and therefore, it was used as the implementing language. However, when at a later stage it was decided to develop robots to clean ship hulls, the architecture was no longer useful because the new field entailed not fully structured environments, only partly defined tasks, semiautomated systems, already developed industrial systems, and so forth. Moreover, when developing the new robotic applications for these new systems, it proved impossible to define a single common architecture for them all. As the software components

BY ANDRÉS IBORRA, DIEGO ALONSO CÁCERES, FRANCISCO J. ORTIZ,
JUAN PASTOR FRANCO, PEDRO SÁNCHEZ PALMA, AND BÁRBARA ÁLVAREZ

1070-9932/09/$25.00©2009 IEEE

(in this case, generic Ada packages) were designed to be used in a software architecture that imposed strong dependences among them in the absence of such an architecture, we lacked the framework that would allow the components to be reused.

The challenge was then to find an approach whereby the code could be reused in applications with different architectures. The solution arrived at was to adopt the component-based development (CBD) paradigm [5]. CBD is conceived for the purpose of speeding up the software development process. It states that such development would be achieved by linking independent parts, the components, in the same way as in mechanics and electronics.

Following in this line, the DSIE Research Group developed an abstract framework called ACRoSeT (reference architecture for teleoperated service robots) [6], in which it is possible to define software components for robotic applications independent of the architecture and of the ultimate implementing technology. Despite the benefits it brought, the use of ACRoSeT posed new challenges arising out of the conceptual leap from the predominant object-based technology to component-based design concepts. Each of the components defined in ACRoSeT had to be encoded manually in the chosen programming language. There was therefore a need for an approach that came with a set of tools to facilitate automatic or semiautomatic generation of applications. These needs can be satisfied by adopting a model-driven development approach.

Table 1 summarizes the ideas presented in the foregoing paragraphs and some of the chief characteristics of the robotic systems that have been developed. As shown in the table, three major software engineering paradigms were adopted successively: reference architecture, component-orientation, and model-driven development. The following sections deal with all these issues in detail, describing our own experience since 1993 in the use of the fundamental concepts of software engineering.

Specialized Teleoperation Systems: Reference Architecture

The first applications for which software engineering was used systematically were for the maintenance work in nuclear power plants. These applications [2] may be classified into two broad groups: those intended to furnish new control software for the Westinghouse ROSA III robot and those intended for new robotic systems being developed by us. The methodology used to develop all these applications was based on the domain-engineering process to develop a common reference architecture for all these systems. This methodology consists of three stages, namely, domain analysis, domain design, and domain implementation.

The principal functional features that were identified as a result of domain analysis are the following: 1) the operating environments are thoroughly understood and structured; 2) the robot's movements are directed by the operator at all times (no autonomy); 3) a user-graphic interface is needed to display the robot's status and how it is interacting with its surroundings in real-time and three-dimensional (3-D) display; and 4) the system should be fault-tolerant. The nonfunctional features include the possibility of adapting applications to new tools, operating environments and user interfaces, and portability with respect to the operating system and communications links.

The challenge lays in how to reuse as much as possible the code of one application in another application.

With all these requirements in mind, we defined a reference architecture in which we can distinguish the graphical display subsystem (3-D display of the robot and its environment), kinematics subsystem (movement simulation and collision detection), user interface, communications subsystem, and robot controller (assures the feasibility of the operator's commands and controls their execution).

The architecture was implemented using the Ada95 programming language (high-level tasks running on the teleoperation platform), C language (low-level control tasks running on onboard robot processors), Motif and X-Windows libraries (user interface), and the ROBCAD commercial tool (3-D graphic display of the robot and its environment, cinematic calculations, and collision detection). The programming paradigms used were the ones available at the time: object orientation, abstract data types, and generic units.

One of the major advantages of this approach is the ease with which subsystems from one application can be used in another, simply by appropriate instantiation of the generic parameters. To give a more precise idea of how the different models are reused or configured, Table 2 shows some scenarios entailing modification and extension of applications. When something new is to be added to an existing application (e.g., an environment, a model, a robot tool), all that is needed is to define the new elements and incorporate them in the applications. Moreover, such additions are carried out systematically following a set of rules defined in the architecture. If it is wished to develop an application for a new device (as in the last of the scenarios), generic modules have to be defined and specific algorithms programmed, but all the code for the interactions between subsystems can be reused. Despite the aforementioned advantages, the architecture cannot be adapted to any requirements other than those described in this section.

These drawbacks became apparent with the development of robots for ship-hull maintenance, where in addition it proved impossible to define a generic reference architecture for that domain. A different approach had to be found necessarily because the challenge was different. The problem now was to come up with a means of reusing code with different architectures. To do that, we adopted the paradigm of CBD.

Robot Control Architectures and Frameworks: ACRoSeT

The development of various service robots to perform maintenance tasks in the shipping sector (hull cleaning) was addressed in the context of the European project environmental friendly and cost-effective technology for coating removal (EFTCoR), under the European Union's fifth framework programme (EU 5th FP) (GROWTH, G3RD-CT-00794). The various robots that were considered in this project would

Table 1. Robotic systems developed and software engineering paradigms used.

Robot/ Project/ Year	Application Domain	Software Engineering Strategy/ Purpose	Examples of Robotic Devices Controlled	Features	Scheme of the Proposal
ROSA III EUREKA 1993-1998	Nuclear power plants	**Software engineering strategy:** Reference architecture **Purposes:** • Define a common architecture for all systems for a given domain (in our case pure teleoperation, structured environments, and specialized tools). • Reuse of code. **Related approaches:** • Hierarchical-deliberative architectures • Reactive architectures • Hybrid architectures • See [11] for a complete discussion about software architectures for robotics systems.	Vehicle for retrieving foreign objects from the nozzles of primary circuits of PWR nuclear plants. Graphic subsystem of teleoperation applications (implemented with ROBCAD).	↑ All systems share the same structure. ↑ Generic components readily reused in different applications. ↓ The architecture cannot be used in other domains (autonomous systems or nonstructured environments). ↓ Components cannot be reused in other architectures. ↓ It is very difficult to design a single architecture flexible enough to deal with the heterogeneity of requirements of robotic applications.	Reference architecture (which preserves the original notation) where we can distinguish the different subsystems or components.
GOYA ERDF 1999-2001	Shipyard repair industry				
EFTCoR VFP 2002-2006		**Software engineering strategy:** Architectural frameworks Component-based development **Purposes:** • To provide an architectural umbrella for defining components and architectures. • To define abstract components and integration strategies. • Reuse of components regardless of architecture. **Related approaches:** • Robotic development frameworks: OROCOS[7], ORCA[8]	XYZ table mounted on tower during functional tests. Jointed tower for cleaning large surfaces.	↑ Includes the reference architecture approach, but within a broader framework. ↑ Possibility of defining and reusing specific architectures. ↑ Components are independent of both system architecture and implementation platform. ↓ Components cannot readily be implemented with object-oriented technology. ↓ Manual encoding is highly laborious and error-prone.	
EFTCoR v2 2007-Today		**Software engineering strategy:** Model-driven engineering **Purposes:** • Architectures as models. • Models conform to metamodels. and transformations between them. **Related approaches:** • Not used extensively in robotics nor in reactive systems (as far as we know), but several proposals exist [10].	Lazaro climbing vehicle for cleaning shaped surfaces of bow and stern.	↑ Includes the CBD approach, but within a broader framework. ↑ Automatic translation of abstract components into concrete ones. ↑ The architecture is designed at a high level of abstraction in a platform-independent way and models are translated semiautomatically into code. ↓ Lack of maturity. ↓ Lack of tools.	

have to perform cleaning operations on ships' hulls at shipyards with heterogeneous facilities for vessels of different types. In view of the difficulty of designing a single robot that would meet all the requirements, we opted to design a family of robots. All these robots consist of a primary positioning system, which can be a vertical tower with up to five degrees of freedom, a climbing vehicle, or an elevator table. The cleaning head can be a turbine or several blasting nozzles with a confining hood. Some of the systems also have a secondary element (XYZ table), which augments the number of degrees of freedom of the primary element and improves operating times, especially when cleaning small areas or spots. The different applications and the devices that perform them are summarized in Table 3. Table 1 shows various images of the EFTCoR systems.

Architectural Framework for Control Units (ACRoSeT)

When it came to tackling the problem of designing the software architecture for the family of robots, we encountered the following difficulties: 1) the reference architecture used for the robots in the nuclear environment could not be used; 2) the functional requirements varied very much from one system to another; 3) programming languages and platforms had to be different for each device; and 4) it was necessary to use commercial components (motor controllers, programmable logic controllers, frequency variators, etc.).

In view of these problems, it was clearly necessary to define an architectural framework that would 1) impose no particular architecture, but would permit the definition of different architectures to fit the particular restrictions of each application; 2) facilitate the reuse of code; and 3) allow for highly diversified final implementation of components, both software and hardware, and including commercial off-the-shelf (COTS) components.

The solution arrived at was to adopt the CBD paradigm. It states that application development should be achieved by linking independent parts, the components. There is a great confusion about the meaning of software component. Such meaning depends on the underlying component model that describes the semantics of components, which can be roughly classified into two categories: 1) component models where components are classes or objects (e.g., JavaBeans, .NET, Corba Component Model) and 2) component models where components are architectural units (e.g., UML 2.0, Koala, KobrA). The first category is supported by the current technologies, and the components are directly executable in their respective programming languages; the second compels to implement the components manually or develop tools that generate the code that implements the component (see [5] for a complete and very understandable discussion). Because composition is the central issue in CBD, the selection of an approach will depend on the compositionality of the resulting components, and in this sense, the second

Table 2. First stage (1993–1998): Change scenarios for nuclear power plants maintenance applications.

Change Scenario	Changes to be Made
New steam generator maintenance operation (i.e., make a new application for welding plugs).	Graphical modeling of tool (ROBCAD). Instantiation of generic tool controller with features specific to the new tool. Tool controller included in application. No need to modify any additional software subsystem.
Change of operating environment (i.e., updating of a new water box model).	Graphical modeling of the new environment (ROBCAD).
Change of robot model (i.e., updating of a new water robot model with an added axis).	Graphical modeling of the new robot (ROBCAD). Change corresponding characteristics in robot controller (generic Ada95 modules).
Define a new robot [i.e., A new robot to retrieve objects in the vessel bottom (TRON)].	Graphical modeling of the new environment (ROBCAD). Graphical modeling of the new robot and tools (ROBCAD). Instantiation of generic robot and tool controller modules with the features of the new devices.

Table 3. Second stage (1999–2006): Maintenance operations and devices developed in the context of the EFTCoR project.

	Hull Area Considered		
Cleaning Operation	Vertical Surfaces	Fines	Bottoms
Full blasting *Large surfaces*	*Primary system:* Vertical towers *Head:* Turbines	*Primary system:* Vertical towers *Head:* Nozzle *Primary system:* Climbing vehicle *Head:* Nozzle	*Primary system.:* Elevator table *Head:* Turbine *Primary system:* Climbing vehicle *Head:* Nozzle
Spotting *Small multiple surfaces scattered over the underwater body.*	*Primary system:* Vertical towers *Secondary system:* XYZ table *Head:* Nozzle *Primary system:* Climbing vehicle *Head:* Nozzle	*Primary system:* Vertical towers *Secondary system:* XYZ table *Head:* Nozzle *Primary system:* Climbing vehicle *Head:* Nozzle	*Primary system:* Elevator table *Secondary system:* XYZ table *Head:* Nozzle *Primary system:* Climbing vehicle *Head:* Nozzle

approach is clearly better. Components such as architectural units allow 1) specifying very precisely, using the concept of port, both the services provided and the services required by a given component and 2) defining a composition theory based on the notion of a connector. None of these features can be directly achieved using objects because classes only publish the provided services, and the unique way of interaction among objects is method invocation. Anyway, components can be implemented using objects and design patterns as long as the resulting code implements the semantics associated to the components and their interactions.

The CBD paradigm has been adopted by several existing frameworks for robot development (e.g., Orocos [7], ORCA [8], etc.), of which some use objects and others use architectural units as components. Frameworks offer high rates of reusability and ease of use, but little flexibility with regard to the implementation platform: most of them are linked to C/C++ and Linux, although some achieve more independence, thanks to the use of some middleware. For industrial purposes, the EFTCoR project required the use of commercial devices and programming languages for programmable logic controllers (PLCs) not catered for by the available frameworks. Moreover, the frameworks usually have an implicit architecture and offer the principal control loop for the application. Since one of the objectives pursued is the ability to define different architectures, the use of a commercial framework posed additional problems.

With these ideas in mind, ACRoSeT was designed as a component-based architectural framework to guide the design of control software for teleoperated service robots. ACRoSeT provides a framework of abstract components, the architectural units mentioned earlier, which can be implemented in various different ways (by integrating different software or hardware solutions, or even COTS components).

We needed a way to define interfaces and behavior at a higher level of abstraction so that they could be used in systems with different platforms. This is what prompted the idea of abstract components, which would be independent of the implementation platform but could be translated into an executable software or hardware component. In opting for these abstract components, we were trusting that the tools associated with the unified modeling language (UML) for generating code would evolve favorably, and this would make it possible to generate the code automatically from ACRoSeT diagrams.

Instantiation of ACRoSeT in the EFTCoR Project

By way of example we shall present two architectures that have been defined with ACRoSeT, one for the XYZ table controller and the other for the climbing vehicle. Note in Figures 1 and 2 how both the components and their interactions and connections become explicit, following the CBD approach mentioned earlier. The fundamental components we can see in the figures represent physical device controllers (which control one or more axes, a tool, etc.). When a controller refers to a single actuator it is called a simple controller (SC); when it coordinates several SCs it is called a mechanism controller (MC). The aggregate of SCs and MCs following a given architecture is the robot controller (called RC in ACRoSeT terminology). Sometimes, these controllers only encapsulate access to a real physical device, for instance the MC component of the XYZ table (Figure 1), which corresponds to the access interface of a SIEMENS commercial controller for a 315-2DP programmable logic controller. In the case of the Lazaro climbing vehicle (Figure 2), the MC is composed of software modules written in Ada95 and is executed on an embedded PC.

The control software for the XYZ towers (Table 1) has been implemented on a development infrastructure supplied by SIEMENS, using a PLC 315-2DP and the STEP7 development environment. In this case, ACRoSeT components have

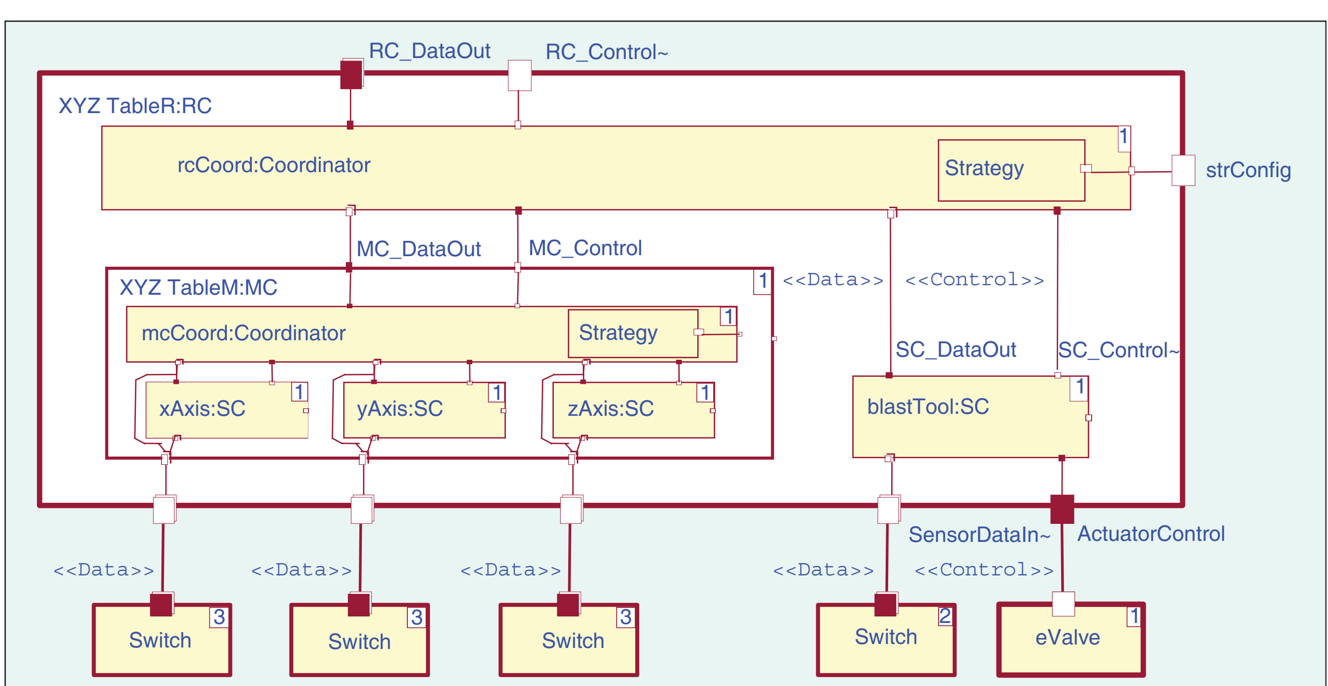

Figure 1. *Second stage (1999–2006): ACRoSeT component-based architecture for the XYZ table control unit.*

been translated into KOP programming language function blocks. On the other hand, the Lazaro climbing vehicle (Table 1) has been implemented on an embedded PC using the Ada95 language. In this case, the implementation of the components was much simpler, thanks to 1) the level of abstraction this language provides and 2) the use of the Active Object [9] architectural design pattern, which has guided the translation of the ACRoSeT components to object-oriented technology.

Using ACRoSeT has enabled us to significantly reduce the time devoted to system analysis and architectural design by making it possible to 1) design controllers and their interactions at a level of abstraction independent of the platform; 2) reuse the abstract components defined by ACRoSeT in systems that use different implementation platforms; 3) reuse the same particular components in systems with different architectures that share the same implementation platform; and 4) facilitate the extension of systems with additional functionality by considerably simplifying the addition and substitution of components.

Nevertheless, although the ACRoSeT's capacity to describe the architecture of different robotic systems has brought an improvement in the designs executed by the DSIE, manual translation of abstract ACRoSeT components into particular platform-specific components is a difficult and error-prone task. To give an idea of the size and complexity of this process, a simple design with only two components offering two services (e.g., move and stop) is translated to 16 collaborating objects using the Active Object pattern. In the case of the mentioned Lazaro vehicle, the implementation of the ACRoSeT architecture (Figure 2) in Ada would result in about 50 classes. Therefore, ACRoSeT will only display its full potential if it is possible to automatically translate abstract components into executable code. In our opinion, which is shared by other members of the scientific community [10], the solution may be to adopt the MDE approach.

The architecture was implemented using the Ada95 programming language.

A Metamodel for Components: V^{3}Studio

In light of the situation described earlier, we adopted the MDE approach to deal with the limitations detected in ACRoSeT. This approach is consistent with the model-driven architecture (MDA) of the Object Management Group (OMG) and is a highly promising alternative to the traditional software development process.

MDE proposes the use of models as the principal artifact for software development, a model being a simplified depiction of reality that shows only the aspects that are of interest. In this

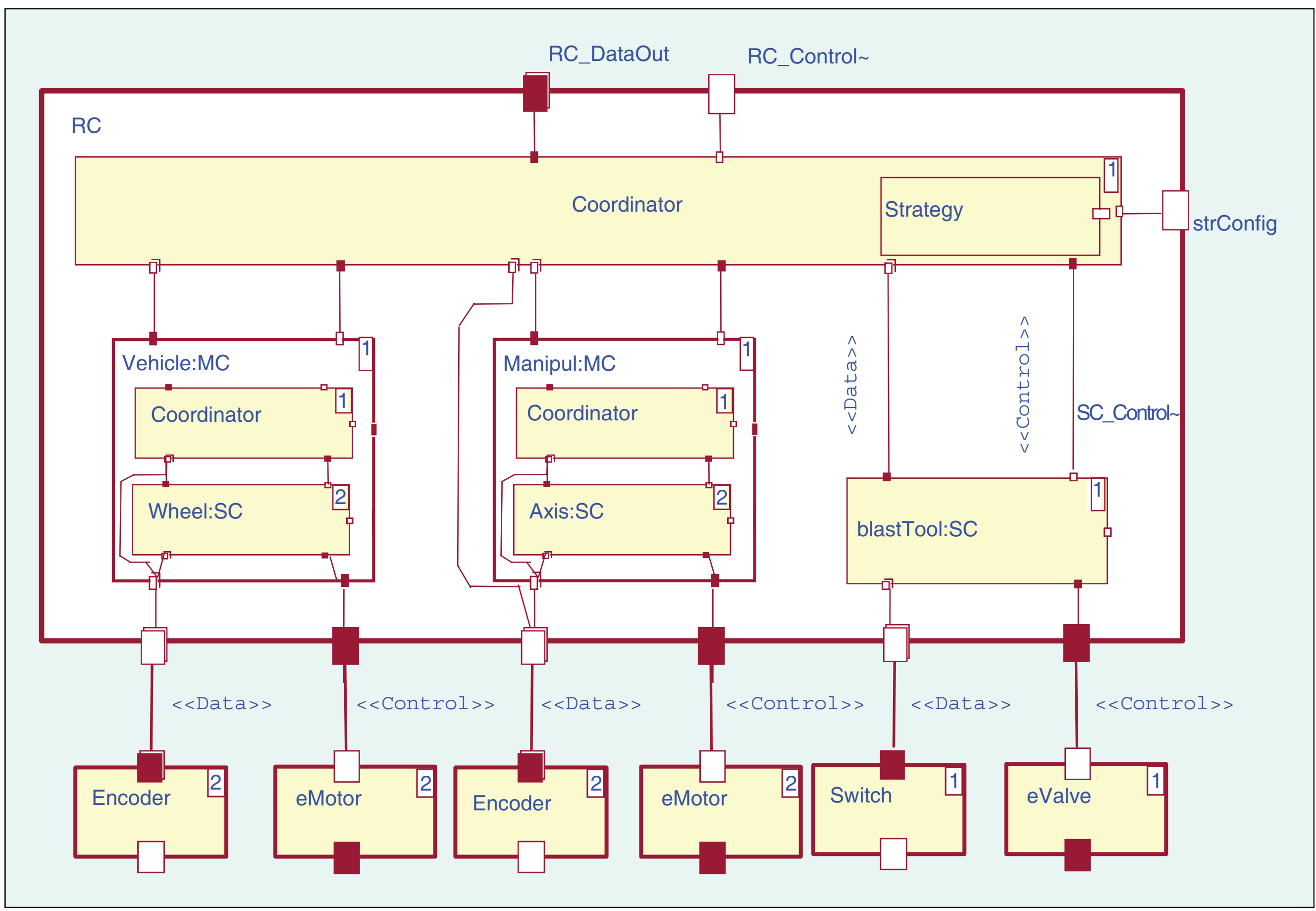

Figure 2. *Second stage (1999–2006): ACRoSeT component-based architecture for the Lazaro II climbing robot.*

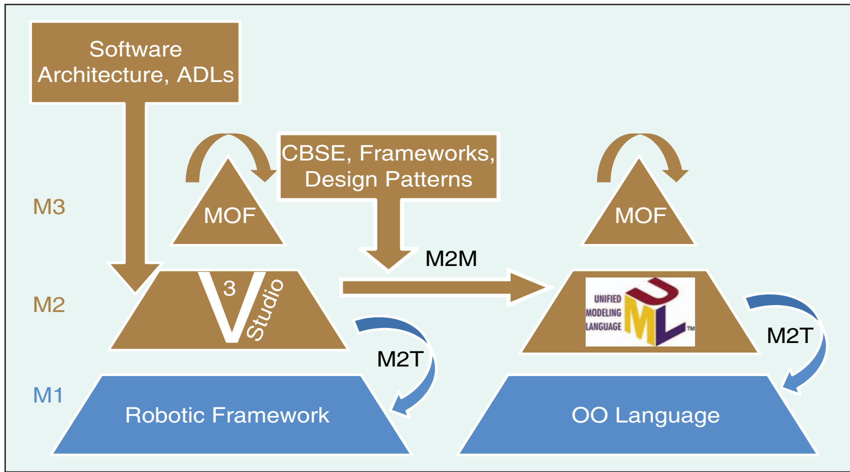

Figure 3. *Third stage (2007–2008): Overview of the V^{3}Studio development process.*

approach, a model is defined in accordance with a metamodel, which defines the abstract syntax of a modeling language and establishes the concepts and relationships between them, including the rules that determine when a model is properly formed. As a given system can be described by different models at different levels of abstraction, model transformations are one of the key issues of this approach.

The use of the MDE approach and its supporting tools and technologies will allow us to

1) define a metamodel for ACRoSeT that will enable us to formally define the concepts that are important for the designer and their relationships. Tools linked to MDE supply the necessary supporting infrastructure to create and manipulate models of robotic systems on the basis of these elements
2) semiautomate the entire development process, from specification of the architecture to the final generation of code, thanks to the various model-to-model transformation (M2M) and model-to-text (M2T) tools. The highly abstract models can later be specialized by executing various different model transformations, until finally there is a model close enough to the platform to generate the associated code.

After weighing the advantages and disadvantages of abandoning UML as the principal notation vehicle, it was eventually decided to 1) define a metamodel for components, called V^{3}Studio, adapted to the modeling needs imposed by ACRoSeT rather than using UML directly and 2) design a development process that would make it possible to semiautomate the generation of code associated with the ACRoSeT model. With V^{3}Studio, it is possible to describe an application's architecture on the basis of its components and also the behavior and algorithms that are implemented by these components. V^{3}Studio defines the minimum set of elements necessary to describe the architecture of applications and dispenses with all those which experience shows to be unnecessary. The V^{3}Studio metamodel is divided into three interrelated views that describe each of the aspects of the application listed earlier: one view for the system's architecture and two views for the behavior of its components (state machines for describing component behavior and activity diagrams for describing the algorithms executed in states and transitions). This fact also makes V^{3}Studio to be easier to use for designers who lack a thorough knowledge of UML, as it defines a total of 51 concepts when compared with more than 200 in UML.

Figure 3 provides a schematic of the process of developing robotic applications using V^{3}Studio. Each level of the pyramids represents a model that conforms to the metamodel located on the level mentioned earlier; the cycle is closed in the Meta Object Facility (MOF defined by OMG) metamodel, which conforms to itself. As we can see in the left-hand pyramid, V^{3}Studio is the metamodel in which the building elements of ACRoSeT are defined. Once the V^{3}Studio model is generated, it can be directly translated into code (M2T transformation in the left-hand pyramid), or it can be translated into an object-oriented model expressed with UML (M2M transformation). Both transformations can be defined and executed inside the Eclipse development environment by using the following plug-ins: 1) the Atlas Transformation Language (ATL) for M2M transformations and 2) Java Emitter Templates (JET) for M2T transformations.

The object-oriented model is then automatically transformed into code (M2T transformation in the right-hand pyramid). The explanation for this last route, which requires an intermediate transformation step expressed in UML, is that it is simpler and more convenient to arrive at the final code by way of such an intermediate transformation than directly. It is simpler to perform two transformations that gradually lower the level of abstraction than to perform a single more complex one. We should stress that, in this development scheme, UML is not used as a design language; the M2M transformation is automatic and can be concealed from the end user.

There are other situations in which the direct M2T transformation (left-hand pyramid) is suitable. These are situations where the application to be developed allows the use of any of the existing frameworks (OROCOS, ORCA, etc.), which are component-oriented in any case. No intermediate transformation is advisable in these cases.

Figure 4 shows a schematic of the complete process of developing an application with V^{3}Studio for the XYZ table. As the figure shows, the development steps are as follows:

Step 1) *Design of the application architecture using the architectural view*: 1) Define the components' interfaces and the services contained in these interfaces; 2) define the components by assigning interfaces to the component ports, in the form of offered and requested interfaces; and 3) proceeds to link up the components' compatible ports. The components may in turn contain other components.

Step 2) *Design of the component state machines using the behavior view*: For each component, the designer creates a state machine describing the component's

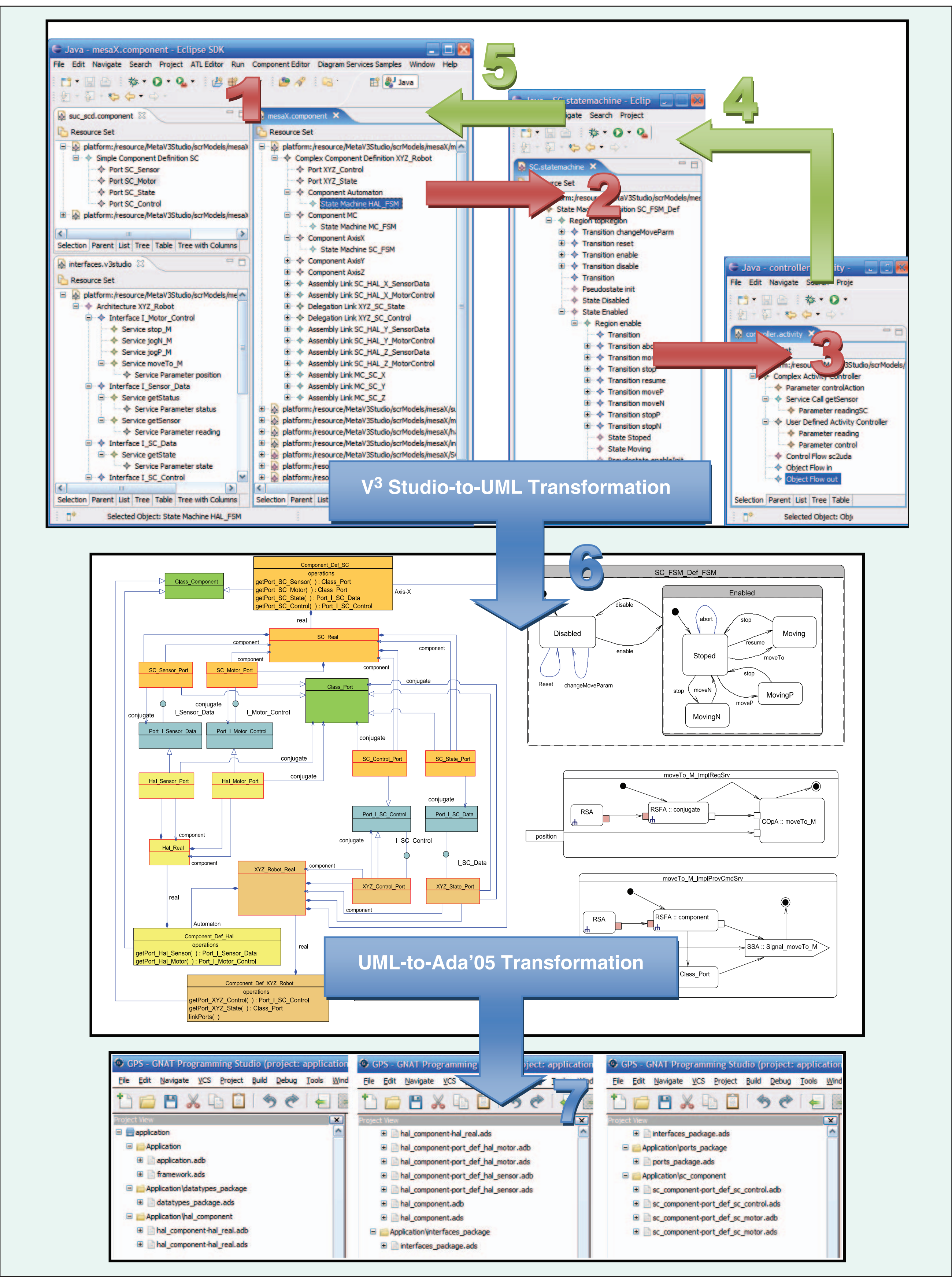

Figure 4. Third stage (2007–2008): Example of the V^3*Studio development process for the XYZ table. Numbers refer to the steps of the proposed MDE development process.*

CBD is conceived for the purpose of speeding up the software development process.

internal behavior and its reaction to the messages it receives from other components.

Step 3) *Design of algorithms using the algorithmic view*: This view, based on the UML activity diagrams, describes the algorithms that will be implemented by the component depending on its current status.

Step 4) *Association of activities and state machines*: Once each of the application's three views has been separately defined, it only remains to run the process backward to complete the model. Each state and transition of the state machines must be associated with the corresponding activity that describes the algorithm that is implemented whenever the component is in that state or triggers that transition.

Step 5) *Association of state machines and components*: Each component must be associated with whichever of the state machines designed in Step 4 defining its behavior.

Step 6a) *Transformation of the V^3Studio model to a UML model*: The translation is based into concepts handled by object-oriented languages. A set of classes and additional operations is generated, derived from the different design patterns that have been used to perform this translation (mainly Active Object, but also Composite, Command, Proxy, and others).

Step 6b) *Transformation of the V^3Studio model to a robotic framework*: Alternatively, it would be possible to design transformations that generate code for any robotic framework on the basis of the V^3Studio model, since both use similar concepts (for example, component and connector). This step has not yet been implemented.

Step 7) *Transformation of a UML model to a programming language*: From the UML model generated in Step 6, we can directly generate code for any object-oriented programming language, since that model contains only concepts used by languages of this type.

One of the requirements imposed on the V^3Studio design was that it facilitates and allows the reuse of models designed in as many scenarios as possible. Components can therefore be reused as models in different architectures; state machines can be reused in different components; and finally, activities can be reused in different state machines. Moreover, the way in which a component requests services from other components has been parameterized. In this connection, the V^3Studio caters for two types of services (synchronous and asynchronous) and two communication patterns for every case (polling or subscription). In V^3Studio, it is also possible to specify concurrency policies for the components, indicating whether the component will have its own implementation thread or, on the contrary, will it be implemented on its container's thread.

This process has been applied to generate code previously obtained manually. To give the reader an idea of the time savings, the Ada code corresponding to the ACRoSeT design of Figure 2 took about a week to one full-time programmer. With the V^3Studio tools, it is simply a process of compilation that only takes several seconds. Other issue is that the development of V^3Studio has taken several months.

Finally, it should be stressed that designing the software of any system is a complex task that involves more steps rather than just the design phase we have described. There are other very important phases such as verification and validation of both models and code, which have not been described because they are outside the scope of this article. MDE is a relatively new approach that still has not reached a mature status, and, as such, there are still many areas that are subject to intense research, such as 1) model verification and validation, i.e., how it can be assured that model transformations produce the correct result and 2) model testing, i.e., how it can be assured that generated models still conform to application requirements, to mention a few. As far as we know, the MDE approach has not been applied to the field of robotics, except in small case studies like the one shown here. Nevertheless, there are important initiatives that are promoting its use [10].

Conclusions

This article relates our experiences over the last 15 years in the development of robotic applications within the field of service robotics, using the techniques proposed by software engineering. The process began with domain engineering and reference architectures, moved on to component-oriented development, and currently centered on model-driven design. Table 1 summarizes the ideas and characteristics of the developed robotic systems as well as the software engineering paradigms that have been used and described here.

One of the key problems in software development for robotic systems is that the possibilities of reusing software in new applications are frequently limited. This means that we are forced over and over to solve the same problems starting practically from zero every time. The possible causes of this include the following: 1) robotics specialists normally concentrate more on developing algorithms and the way to solve concrete problems than on organizing the software; 2) lack of good standards for the development of robotic software and implementations of these standards; 3) the case studies conducted to demonstrate the viability of software engineering techniques traditionally deal with information management systems; and 4) the robotics community see software engineering not as a solution but as another problem that adds complexity to already-complex problems.

This research has helped to demonstrate the viability of using software engineering techniques in real industrial applications, albeit using academic tools that cannot readily be accepted by industry. There is a need for the development of commercial tools incorporating these ideas, particularly CBD and MDE, designed for the development of robotic applications. Our experience tells us that software engineering has made a decisive contribution to improving the quality of our applications and to reducing the effort entailed in development. Our hopes for the future are basically pinned on the MDE approach and the development of tools

to provide automated support for code generation. One of the challenges we are facing in the short term is to turn V^{3}Studio into a robust tool that can be offered to other research groups and is of more than strictly academic use. In the long term, we plan to integrate V^{3}Studio with a Software Product Line approach to define a common architecture for a given family of products. It will be then possible to derive a concrete architecture for one product by selecting components (in the form of V^{3}Studio models) from a component repository. After this step, the developer can use the process described in "A Metamodel for Components: V^{3}Studio" section to generate the application code.

Acknowledgments

This work has been supported by EU and Spanish Government research programmes: 5th FP (GROWTH G3RD-CT-00794), CICYT-FEDER Program (MEDWSA, TIN2006-15175-C05-02). Additional funds have been supplied by the Government of Murcia (Fundación Séneca) and the Spanish Ministry of Industry (PROFIT programs).

Keywords

Service robots, software engineering, software architectures, frameworks, model-driven engineering.

References

[1] J. Karlsson, "UN world robotics statistics 1999," *Ind. Robot.*, vol. 27, no. 1, pp. 14–18, 2000.

[2] A. Iborra, J. A. Pastor, B. Álvarez, C. Fernández, and J. M. Fernández, "Robots in radioactive environments," *IEEE Robot. Automat. Mag.*, vol. 10, no. 4, pp. 12–22, 2003.

[3] C. Fernández, A. Iborra, B. Álvarez, J. Pastor, P. Sánchez, J. M. Fernández, and N. Ortega, "Ship shape in Europe: Co-operative robots in the ship repair industry," *IEEE Robot. Automat. Mag.*, vol. 12, no. 3, pp. 65–77, 2005.

[4] T. Stahl and M. Vöelter, *Model-Driven Software Development: Technology, Engineering, Management*, 1st ed. New York: Wiley, 2006.

[5] K. Lau and Z. Wang, "Software component models," *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 709–724, 2007.

[6] B. Álvarez, P. Sánchez, J. A. Pastor, and F. Ortiz, "An architectural framework for modelling teleoperated service robots," *ROBOTICA Int. J. Inf. Educ. Res. Robot. Artif. Intell.*, vol. 24, no. 4, pp. 411–418, 2006.

[7] H. Bruyninckx, "Open robot control software: The OROCOS project," in *Proc. IEEE Int. Conf. Robotics Automation*, 2001, vol. 3, pp. 2523–2528.

[8] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Oreback, "Towards component-based robotics," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, 2005, vol. 1, pp. 163–168.

[9] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, vol. 2. New York, Wiley, 2000.

[10] H. Bruyninckx, "Robotics software: The future should be open," *IEEE Robot. Automat. Mag.*, vol. 15, no. 1, pp. 9–11, 2008.

[11] E. Coste-Manière and R. Simmons, "Architecture, the backbone of robotic systems," in *Proc. 2000 IEEE Int. Conf. Robotics and Automation*, pp. 67–72.

Andrés Iborra received a Ph.D. degree in 1993 and an M.S. degree in 1989 in industrial engineering from the Technical University of Madrid. He has worked as a research engineer in robotics for nuclear power plants during 1993–1998. In 1998, he joined the DSIE. He is full professor and the head of the Electronics Technology Department at the Technical University of Cartagena, where he has been since 1998. His current research interests include mechatronic systems design and analysis, computer vision, robotics, and engineering education.

Diego Alonso Cáceres received an M.S. degree in electrical engineering from the Universidad Politécnica de Valencia in 2001 and a Ph.D. degree from the Universidad Politécnica de Cartagena in 2008, where he is currently a lecturer. He joined the DSIE in 2004. His research interests focus on the application of the model-driven engineering approach and to the development of component-based reactive systems with real-time constraints.

Francisco J. Ortiz received his Ph.D. degree in industrial engineering from the Technical University of Cartagena, Spain, in 2005. Since 1999, he has participated in different projects focused in computer-assisted surgery, service robotics, and software engineering in the DSIE. He is currently an associate professor at the Electronics Technology Department of the Technical University of Cartagena. His research interests include mechatronic systems design, software architectures for robotics, and model-driven engineering.

Juan Pastor Franco received his Ph.D. degree in telecommunication engineering from the Technical University of Cartagena, Spain, in 2002. He is currently an associate professor at the Technical University of Cartagena in the field of computer science. He has worked as a research engineer in robotics for nuclear power plants from 1995–2000. In 2000, he joined the DSIE. His research interests include mechatronic systems design and analysis, robotics, design patterns, and model-driven development.

Pedro Sánchez Palma received his Ph.D. degree in computer science from the Technical University of Valencia, Spain, in 2000. Since 1996, he has participated in different projects focused on software engineering and conceptual modeling. In 2000, he joined the DSIE. He is currently an associate professor at the Technical University of Cartagena in the field of computer science. His current research interests include model-driven engineering, real-time systems, and conceptual modeling.

Bárbara Álvarez received his Ph.D. degree in telecommunication engineering from the Technical University of Madrid, Spain, in 1997. Since 1995, she has participated in different projects focused in robotics applications for the industry. In 1998, she joined the DSIE. She is currently an associate professor at the Technical University of Cartagena in the field of computer science. Her research interests include real-time systems and software architectures for teleoperation and computer vision systems.

Address for Correspondence: Andrés Iborra, Universidad Politécnica de Cartagena, División de Sistemas e Ingeniería Electrónica (DSIE), Campus Muralla del Mar, s/n. Cartagena E-30202, Spain. E-mail: andres.iborra@upct.es.