Journal of Software Engineering for Robotics 1(1), January 2010, 3-17
ISSN: 2035-3928

V^3CMM: a 3-View Component Meta-Model for Model-Driven Robotic Software Development

Diego ALONSO[1] Cristina VICENTE-CHICOTE[1] Francisco ORTIZ[1] Juan PASTOR[1] Bárbara ÁLVAREZ[1]
[1] División de Sistemas e Ingeniería Electrónica (DSIE), Universidad Politécnica de Cartagena, Cartagena 30202 (SPAIN)

Abstract—There are many voices in the robotics community demanding a qualitative improvement in the robotics software development process and tools, in order to increase product flexibility, adaptability, and overall quality, while reducing its cost and time-to-market. This article describes a first step towards a model-driven approach to robotics software development, based on the definition of highly reusable and platform-independent component-based design models. The proposed approach revolves around the V^3CMM modeling language and the definition of different model transformations for deriving both special purpose models (e.g., models suited for analysis or simulation purposes) and lower-level design models, in which platform-specific and application-dependent details can be progressively included. The article describes the tool chain implemented to support the different stages of the proposed model-driven process, including (1) the definition of component-based architectural models, defined using the V^3CMM platform-independent modeling language, (2) the automatic transformation of the V^3CMM component-based models into equivalent object-oriented designs, described in terms of the UML standard, and (3) the transformation of the UML models into the Ada 2005 object-oriented programming language. In order to show the feasibility and the benefits of the proposal, a simple (yet complete) case study regarding the design of a Cartesian robot is presented.

Index Terms—Robotics Software, Model-Driven Engineering, Component-Based Software Development, Model Reuse.

1 Introduction and Motivation

Robots are software intensive systems, that is, systems in which *"software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole"* [1]. Nowadays, the number and complexity of robotic systems is rapidly growing, while pressures are mounting to increase their flexibility, adaptability, and overall quality, and to reduce their cost and time-to-market. In order to meet this growing demand, new robotics software development methods and tools are needed [2]. Being aware of this, the robotics community keeps continuous track of the advances achieved in Software Engineering, in order to exploit the benefits that the new trends in this field might bring to robotic software development. In this vein, it is worth reminding that, in the last decade, robotics software has been strongly influenced by both the Object-Oriented (**OO**) and the Component-Based (**CB**) software development paradigms.

Regular paper – *Manuscript received June 1st 2009; revised October 6th and December 23rd, 2009: online since January 20th 2010.*

- *The authors would like to thank the reviewers for their helpful suggestions and comments. This research has been funded by the Spanish CICYT Project EXPLORE (ref. TIN2009-08572), the Fundación Séneca Regional Project COMPAS-R (ref. 11994/PI/09), and the Spanish Research Network on Model-Driven Software Development (ref. TIN2008-00889-E).*

Quite recently, the Model-Driven Engineering (**MDE**) paradigm [3], [4] is also starting to catch the attention of the robotics community [5], [2], mainly due to the very promising results it has already achieved in other application domains (e.g., automotive, avionics, or consumer electronics, among many others) in terms of improved levels of reuse, higher software quality, and shorter product time-to-market [6].

MDE enables designers to focus on domain concepts, relegating implementation details to a secondary level. In MDE, models are the primary artifacts leading the whole software development process [8]. Models [9] are simplified representations of reality in which the superfluous details are abstracted away, thus easing the understanding and communication of the underlying reality. Models are defined in terms of formal meta-models, which bring together the concepts relevant to a particular application domain, and the syntactic relationships existing among these concepts (i.e., each meta-model defines the abstract syntax of a modeling language). Model transformations [10] are also key artifacts in MDE, since they define how models shall be interpreted and translated into other artifacts. Both model-to-model (**M2M**) and model-to-text (**M2T**) transformations provide software developers with formal model compilers, which allow them to automatically translate models from one modeling language to another (modeling or programming) language.

It is worth noting that although MDE provides a formal foundation for improving the whole software development process [8], it does not prescribe how to select or design the most appropriate modeling languages and transformation paths. Thus, in order to exploit all the benefits that MDE may bring to robotics, there is an urgent need to define a MDE robotics-specific software development process, and to build a set of reliable MDE tools (including robotics-specific modeling languages and transformations) that fully support it. In this vein, it is worth highlighting some initiatives, like the one described in [7], aimed at discovering and organizing a stable set of concepts related to robotics software design.

MDE owes a great part of its success to the Model-Driven Architecture (**MDA**) [11] initiative, launched by the Object Management Group (**OMG**) in 2001. As shown in Fig. 1, models in MDA can be classified into the following levels: Computation Independent Models (**CIM**), Platform-Independent Models (**PIM**), and Platform-Specific Models (**PSM**). CIMs (also known as domain models) specify the high-level system requirements, without mentioning or taking into account any design or implementation decision. The primary users of CIM languages are domain experts (rather than software engineers or application developers) and, as a consequence, CIMs are commonly specified using Domain-Specific Languages (**DSL**) [12]. PIMs are defined at a lower level of abstraction than CIMs, providing a high-level system design solution in a platform-independent way. PSMs are the lowest level models and, thus, the closest to the final system implementation. The concept of “platform” is quite vaguely defined in MDA, making it difficult to draw a clear line between PIMs and PSMs. For instance, the OMG standard Common Object Request Broker Architecture (**CORBA**), is considered both as a low-level PIM and as a high-level PSM in different OMG specifications.

Nowadays, most robotics software is either hand-coded (generating the glue-logic needed to bring together the functionality provided by different robotics-specific libraries), or developed on top of one of the existing robotic frameworks [2]. Although frameworks are excellent examples of code reuse and provide a higher level of abstraction than most programming languages, they depend on an specific platform (or middleware) [13]. This dependency makes design reuse across different frameworks almost impossible. Besides, current frameworks can not be considered to be model-driven, since they have no meta-model foundation supporting them. The work described in this article proposes a qualitative improvement in the way robotic software is developed, overcoming many of the current limitations of CB frameworks, both in level of abstraction and in tool support by designing a complete MDE solution.

Although, from the point of view of robotic software developers, having a language defined at the CIM level would be the most desirable scenario, in this article the authors describe a PIM language, developed as a first step towards a complete MDA process. In this vein, the authors propose the 3-View Component Meta-Model (V^{3}CMM), as an expressive yet simple platform-independent modeling language for component-based application design. As further discussed in Section 3, the main design drivers behind V^{3}CMM are simplicity, economy of concepts, and component reuse. V^{3}CMM is aimed at allowing developers (1) to model high-level reusable components, including both their structural and behavioral facets (modeling *for* reuse), (2) to build complex platform-independent designs up from the previous components (modeling *by* reuse), and (3) to automatically translate these high-level designs into lower level models (e.g., into different CB or OO designs, bounded or not to a specific development framework, or into analysis models), or into different ad-hoc implementations, isolating functionality from platform details. V^{3}CMM is based on the previous experiences [14] of the *Division of Electronics Engineering and Systems* (**DSIE**) Research Group, and is supported by a tool chain developed using the free and open-source Eclipse platform. This tool chain supports the MDE process previously outlined, allowing designers to define platform-independent CB models (both *for* and *by* reuse), and to generate lower level OO models and code from them (in UML and in Ada 2005, respectively).

This article is structured in seven sections. The following section briefly reviews the state-of-the-art in robotics software development, and explains how the proposed MDE approach can help overcoming some of its current limitations. Section 3 describes the main characteristics and design drivers

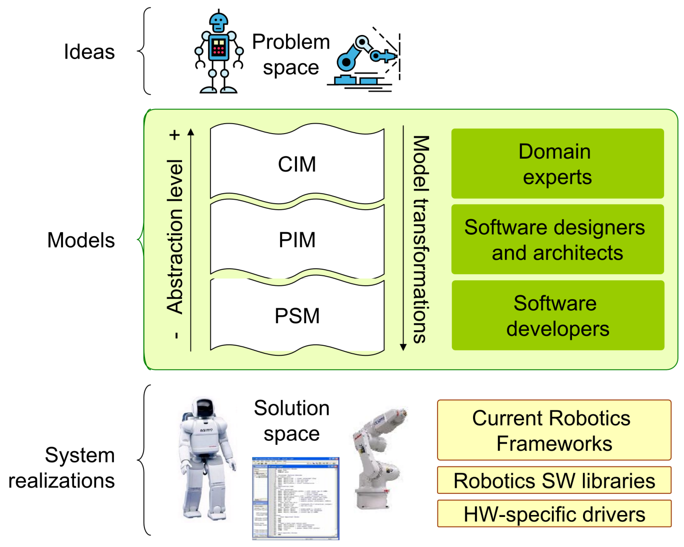


Figure 1. MDA in the context of robotic software development.

behind V^{3}CMM, while Section 4 presents the two model transformations implemented to support one of the possible paths to generate code from V^{3}CMM models. The approach is illustrated in Section 5, where the proposed MDE development approach (V^{3}CMM and the model transformations) is applied for generating the skeleton of the control software of a Cartesian robot from the structural and behavioral description of its architecture. In order to show the use of the tool chain, this section is written from the point of view of the application developer. Section 6 discusses the most controversial aspects of the proposal, such as why not using the Unified Modeling Language (**UML**) [15] or the System Modeling Language (**SysML**) [16] or why V^{3}CMM provides just three views, among others. Section 7 summarizes the lessons learnt from our experience with V^{3}CMM and draws the conclusions and the future research lines.

2 Limitations of Current Approaches

There is a well established tradition of applying Component Based Software Development (**CBSD**) [17] principles in the robotics community, which has resulted in the appearance of several toolkits and frameworks for developing robotic applications. Particularly, frameworks are excellent examples of the application of good software engineering practices [18]. They are semi-complete applications that provide designers (1) with an architecture tailored to the specific requirements of the domain, (2) with more advanced reuse and extension mechanisms than those provided by libraries, (3) with a set of predefined components providing the typical functionality of the domain, and (4) with the run-time support for executing the resulting applications. The extension mechanisms, which are based on inheritance or composition, define the rules for both designing new components and for integrating them in the framework architecture. Some of these frameworks rely on a middleware technology for achieving a certain degree of platform independence, for minimizing component coupling, and for easing component distribution. An actualized state-of-the-art with references to the most important robotic frameworks and toolkits, like OROCOS [19], ORCA2 [20], Player/Stage [21], SmartSoft [2], etc. can be found in the RoSta project (Robot Standards and Reference architecture) [22].

The main drawback of frameworks is that, despite being CB in their conception, designers develop, integrate and connect components using OO technology. That is, the problem is that OO is used as both a design and implementation language. There is no problem in using OO as the implementation language as, in fact, most of the existing robotic frameworks are implemented on top of them (e.g., C++ or Java). The problem comes from the fact that CB designs require more (or rather different) abstractions and tool support than OO technology can offer. For instance, the lack of explicit "required" interfaces makes it impossible to compilers assure that the components are correctly composed (linked). Also, component interaction protocols are not explicitly defined when using an OO language. Thus, we think that OO languages must not be used for expressing CB concepts, although OO technology can be perfectly used for implementing them. That is, components have to be designed as architectural units, not as object [23].

Many frameworks enable designers to model their CB application structure, but most of them impose the overall internal behavior of their components (e.g. standardized interaction and configuration ports, a common behavior that must be followed by all components, etc.), and therefore they lack of formal mechanisms to specify the internal component behavior. In this way, robotic specific components are polluted with platform-specific details, making it almost impossible to reuse the aforementioned components among frameworks [13]. Some frameworks rely on middleware technologies to achieve a certain degree of platform independence and to ease component deployment (both for adding or removing components at run-time and for distributing them). But this flexibility comes with the price of a more complicated configuration process. Besides, the middleware publish/subscribe mechanism does not assure that the components are correctly composed or that all required interfaces are provided by any component either, since middleware were designed for easing the communication among objects (and not components). And everything is furthermore aggravated by the fact that robotics comprises heterogeneous hardware (e.g. sensors and actuators) that need specific software drivers, which contributes to increase the platform-specific part added by frameworks [13]. The main conclusion that can be drawn is that one framework does not fit all robotics problems, and that it is mandatory to clearly separate and isolate the functionality that is specific to the robotics domain from the details of the execution platform and from the application specific requirements. This separation shall provide users with a greater control over what the system is really doing (as frameworks suffer from the "inversion of control" problem), as well as over important non-functional properties (e.g. real-time issues), which are normally imposed by the chosen framework.

In this article the authors propose the use of MDE as a way of achieving the aforementioned separation of concerns. Up to date, there are not many initiatives for applying MDE principles to robotic software development. One of such initiatives is the work related to the Sony Aibo robot presented in [24]. In this work, the authors propose a modeling language for expressing the behavior of the robot using a a kind of sequential script composed of blocks of Aibo actions, which are linked to joins and sensors of the Aibo robot model. This model is then transformed into the Universal Real-time Behavior Interface (URBI) language in order to finally obtain the code for the robot. The scope of this work is limited to the Aibo robot or, at most, to any other robot compliant with URBI. Another initiative, described in [25], revolves around the use of the Java Application Building Center (jABC) for developing robot control applications. This approach is illustrated on a Lego Mindstorm robot. Although jABC provides a number of early error detection mechanisms, such as animation, analysis, simulation, and formal verification, it only generates Java code and, thus, its application is very limited.

Finally, one of the most interesting initiatives is the one described in [2]. Although their approach is very similar to the one described in this article, there are two main differences that should be noted: (1) they only target the SmartSoft framework, while V^{3}CMM does not impose any platform nor application specific requirements, and (2) they rely on the use of UML for modeling SmartSoft components, while V^{3}CMM is directly defined using the OMG's Meta-Object Facility (**MOF**) [26]. In addition, it must be highlighted that V^{3}CMM does not impose any restriction on the component design, such as the kind of ports that must appear, common interfaces and operations, or a common behavior for all components, to mention a few. We think that these kind of details are application specific and, thus, they must be added by the designer. We realize that this can result in a very tedious and error-prone work, but which can be solved by developing a model transformations that super-imposes the required common characteristics. The flexibility of the MDE approach for doing this kind of things is immense.

The current state of the application of MDE to robotic software development contrasts with what happens in other similar domains, where big efforts are being carried out in this line. For instance, the ArtistDesign Network of Excellence on Embedded Systems Design [27] and the OpenEmbeDD [28] project address highly relevant topics regarding real-time and embedded systems, while the automotive industry has standardized AUTOSAR [29] for easing the development of software for vehicles.

3 The 3-View Component Meta-Model

This section briefly describes the main characteristics and design drivers of the 3-View Component Meta-Model (V^{3}CMM). V^{3}CMM is an extended and improved version of a previous work [30], aiming to provide designers with an expressive yet simple CB modeling language for general-purpose (i.e., domain-independent) and platform-independent architectural design. In this line, it is worth noting that V^{3}CMM is currently being used by the DSIE group in other domains, like Wireless Sensors and Actuators Networks (WSAN) [31] or Home Automation Systems [32]. V^{3}CMM provides the kind of components we need and an absolute control over their semantics. This control is needed in order to ease further model transformation steps and to guarantee the properties of the final applications. V^{3}CMM components have very simple semantics, aimed to ease the derivation of analyzable models and code. Simplicity, economy of concepts and component reuse are the main design drivers behind V^{3}CMM.

Simplicity and economy of concepts. This first driver is achieved by using a reduced and focused set of concepts. V^{3}CMM includes the minimum set of elements for modeling highly reusable CB applications, and dispenses with all those which our experience has shown to be unnecessary [14]. V^{3}CMM does not "reinvent the wheel", but instead adopts and adapts some of the concepts included in UML, which also contributes to ease its use by the software community. With these objectives in mind, V^{3}CMM comprises three complementary views (see Fig. 2), namely: (1) a *structural* view for describing the static structure of both simple and complex components, (2) a *coordination* view for describing the event-driven behavior of each component, and (3) an *algorithmic* view for describing the algorithm executed by each component depending on its current state. In addition to the modeling concepts included in these three views, V^{3}CMM also contains other elements (namely, interfaces and data types), which are simultaneously related to the three views and, thus, have been separately defined.

The coordination view is strongly based on UML state-machines, while the algorithmic view is based on a simplified version of UML activity diagrams. Unlike UML, which considers that behavior can be expressed using either state-machines or activity diagrams (or interaction diagrams), V^{3}CMM considers that both views are compulsory, as they model different aspects of component behavior. As such, state-machines in V^{3}CMM model the event-driven and concurrent aspects of component behavior, while activity diagrams model only the sequential flow of execution. This design decision greatly simplifies the design of the algorithmic view. The structural view, on the other hand, is the one that differs the most from UML, and thus we will put the focus on the characteristics of this view.

Component reuse is achieved in two aspects: by reusing both the structural and behavioral facets (modeling *for* reuse), and by building complex platform-independent designs up from the previous components (modeling *by* reuse).

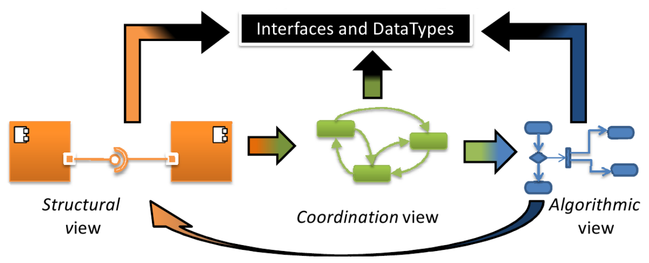


Figure 2. Schematic representation of the V^{3}CMM views showing the kind of concepts appearing in each view and the loosely coupled relationships existing among them.

Modeling for reuse is achieved by loosely coupling the three V^{3}CMM views by means of unidirectional plain associations (shown in Fig. 2), which allows designers (1) to separately model and store the three views of their components, and (2) to change any of the component views by simply updating the association among them (provided the new selected view is also compatible), thus reusing them as many times as needed. This kind of loosely coupled relationship is defined between the structural and the coordination views, the coordination and the algorithmic views, the algorithmic and the structural views, and between the additional concepts included in V^{3}CMM (e.g., interfaces and data types) and each of the three views.

Modeling by reuse is achieved by separately modeling definitions and instances (in a similar way as OO differentiates classes and objects), both for the structural (components) and the coordination (state-machines) views. Definitions are fully reusable artifacts that model all the relevant information, while instances are light-weight artifacts that only contain a reference to their definition. This reuse mechanism is much more efficient than the *copy-paste* reuse provided by most repository-based CB approaches, where reusing a component implies creating a full replica of it each time it has to be added to a design.

The tool chain supporting V^{3}CMM has been integrated in Eclipse by using the following MDE-related plug-ins: **EMF** (*Eclipse Modeling Framework* [33], which adds MOF support to Eclipse), **EMF OCL** (*Object Constraint Language*, **OCL** [34], which provides a formal language for defining constraints and queries on models), **ATL** (*Atlas Transformation Language* [35], which adds a declarative M2M transformation language), and **JET** (*Java Emitter Templates* [33], which adds a template-based M2T transformation language).

3.1 V^{3}CMM Structural View

An excerpt of V^{3}CMM, focused on its structural view, can be found in Fig. 3. As can be seen in the figure, metamodels resemble UML class diagram. The OMG decided to reuse the graphical notation used for depicting UML class diagrams, which can be quite confusing at first. The concepts highlighted in black (i.e., `StateMachineDefinition` and `StateMachine`) belong to the coordination view, although they have been included in the figure to illustrate the loosely coupled relationship existing between both views (e.g., the plain unidirectional association labeled `behavior` between `SimpleComponentDefinition` and `StateMachineDefinition`).

As said before, V^{3}CMM differentiates definitions from instances for reusing purposes. This fact is shown in Fig. 3 with the `ComponentDefinition` and `Component` concepts, respectively. Component definitions contain `Ports`, which exhibit the `Interfaces` (which are globally defined) `provided` and `required` by the component. Ports define the component communication points, while interfaces define the concrete messages they can exchange. On the other hand, components (instances) are defined according to a given component definition (see the association relationship entitled `type`). Components have additional attributes that widen the number of potential scenarios in which component definitions can be reused. These attributes are later processed by model transformations, and thus do not affect the structural description.

V^{3}CMM provides two types of component definitions: `SimpleComponentDefinitions` and `ComplexComponentDefinitions`. While simple components are atomic architectural units with their own behavior, complex components are composite architectural units that encapsulate component instances (whether simple or complex, up to any level), enabling component composition. At the highest level of abstraction, complex components can also be envisaged as application models, as they contain component instances, which represent run-time entities. For the sake of simplicity, it was decided that complex components would not have behavior on their own, but rather it is derived from the combined behavior of the components (instances) they contain. This decision eliminates the risk of creating inconsistent and contradictory models, where the complex component behavior is not consistent with the combined behavior of its inner components, and thus the behavior of the complex component do not represent its real behavior. When complex components have some behavior on their own, the solution lies in including an additional internal (simple) component that models this extended behavior (e.g., an internal coordinator or synchronizer component). Nevertheless, we realize that having a model of the complete behavior of a complex component can be desirable for analysis purposes, but it should be extracted by means of a model transformation.

3.2 V^{3}CMM Behavioral Views

As said before, V^{3}CMM has adopted and adapted UML state-machines and activity diagrams for its coordination and algorithmic views, respectively. The coordination view incorporates many of its modeling concepts from the UML 2.0 state-machines (e.g. state, transition, region, etc.), keeping their meaning. However, like in the structural view, V^{3}CMM distinguishes between `StateMachineDefinitions`, which describe the behavior of `ComponentDefinitions`, and `StateMachines` (instances), which describe the behavior of `Components` (instances). State machine definitions contain the `States` and `Transitions` that model the event-driven behavior of component definitions.

Unlike UML 2.0 activity diagrams, the V^{3}CMM algorithmic view only enables designers to model sequential execution, as concurrent behavior is already

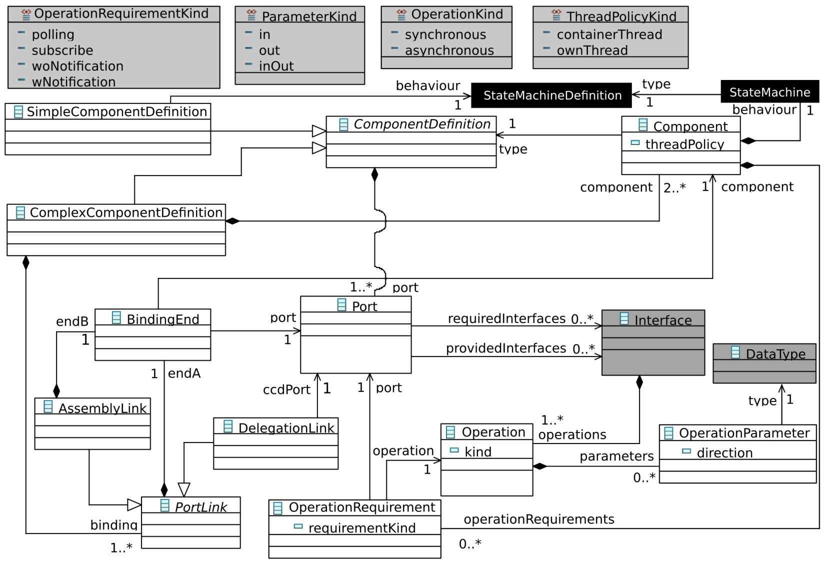


Figure 3. Excerpt of V^{3}CMM, focused on its structural view. This view is related to the coordination view by means of the `StateMachineDefinition` and `StateMachine` concepts (highlighted in black), which are linked to `ComponentDefinition` and `Component`.

modeled in the coordination view by means of orthogonal regions. The V^{3}CMM algorithmic view enables designers to model both simple activities (atomic units) and complex activities (including conditionals and loops), together with the data-flow and object-flow links that enable to connect them. V^{3}CMM defines four types of simple activities, namely: `OperationCall` (to request operations provided by other components), `ConstantActivity` (to produce a single predefined value), `LibraryCall` (to request the execution of an algorithm already implemented in an external library), and `UserDefinedActivity`.

The first three types of simple activities describe specific behaviors for which a full implementation can be later generated. Conversely, when a `UserDefinedActivity` is included in an algorithm, only an empty skeleton of code will be generated, so designers will need to later specify its final implementation manually. V^{3}CMM provides designers with this mechanism just in case the activity they need to describe is neither provided by other component, nor available in a predefined library, nor can be defined as a constant activity. However, the use of this type of activities is strongly discouraged by the authors, since coding manually part of the behavior may cause model erosion (i.e. design models might not reliably represent the real implementations once they have been manually altered) and, as a consequence, the system might become less scalable and more difficult to maintain. Besides, there is no way to ensure that the manually coded behavior does not violate the overall design patterns described in the V^{3}CMM models.

Before concluding this section, it is worth noting that the `LibraryCall` activities embody a wrapping mechanism aimed to encapsulate heterogeneous Commercial Off-The Shelf (COTS) library functions in order to build homogeneous (and thus inter-connectable) implementation units.

4 Model Transformations: from V^{3}CMM Components to Code

As said before, one of the main objectives of V^{3}CMM is to provide a CB modeling language that is solely focused on those parts of a component that are independent of the underlying platform, and thus highly reusable. The objectives of the model transformations involved in a development process that revolves around V^{3}CMM are twofold. On the one hand, they have to provide a formal mapping between the component concepts included in V^{3}CMM and the primitives of the chosen platform (programming language or framework). On the other hand, they have to add the application specific details, thus completing the model and preparing it for the code generation step. There are some cases, for instance when targeting a robotic framework, where only one transformation step is needed, since the rest of the steps are covered by the framework tool chain.

The automation of this translation by means of formal model transformations frees developers from the tedious and error-prone task of manually writing and debugging the final application code every time, and enables them to reuse their designs when targeting different platforms. This article is focused on one of the possible transformation path for V^{3}CMM models, where they are translated into an *ad-hoc* UML 2.0 OO model (by executing an ATL M2M transformation) and this model, in turn, into OO Ada code (by executing a JET M2T transformation).

4.1 From CB to OO Designs

This section describes the M2M transformation from V^{3}CMM components to an OO platform-independent implementation expressed in UML, as UML provides the required concepts for modeling OO software. This intermediate transformation reduces the abstraction level and eases the generation of several M2T transformations for generating code for different programming languages. The generated UML model can even be the input of any of the available Computer Aided Software Engineering (CASE) tools that can generate code from UML models.

The M2M transformation defines the correspondence between the three V^{3}CMM views and their OO implementations, as well as the infrastructure of the execution and run-time support for the components. It also generates the infrastructure for linking the code generated for each view with the generated run-time and with the code that should be added by the user. It is worth noting that, in fact, the part of the transformation that defines the infrastructure represents a framework. While the transformation of the views is rather stable and reusable, the generation of the infrastructure is strongly determined by the platform features and the application requirements. In this vein, the M2M transformation completes the semantics of V^{3}CMM depending on the application characteristics.

The OO design resulting from the proposed M2M transformation maintains both the encapsulation of the component and its run-time independence. On the one hand, component encapsulation is achieved by using the *Façade* [36] design pattern and the visibility property provided by UML. On the other hand, run-time independence is achieved by using the well-known and complex *Active Object* [37] architectural pattern, which aims to decouple method invocation and execution, enhance concurrency, and simplify the synchronized access to objects running in their own threads.

An excerpt of the ATL code of the M2M transformation is shown in Fig. 4. The transformation starts by creating a base UML package, where all the generated artifacts will be stored. This package contains (1) the abstract base classes Port and Component, which provide the common functionality needed to support these basic concepts, (2) a set of packages that contain the data types, the interfaces and their operations, as they appear in the V^{3}CMM models, (3) one auxiliary package that contains the infrastructure needed to associate operation calls to state-machine transitions, and (4) one package for each component, which contains the classes that implement its structure and behavior.

Regarding component behavior, while the structural models are transformed into UML class diagrams, the behavioral models are transformed into UML state-machines and activity diagrams, respectively, making this part of the transformation straightforward. This is particularly interesting in the case of state-machine models, since it allows designers to delay the selection of the particular design pattern they want to implement in a further transformation from the UML state-machine models into code. Finally, it is worth noting that the M2M transformation also generates additional elements and auxiliary operations (together with their implementations) that do not appear in the V^{3}CMM models, such as constructors for all classes, methods for sending, receiving and forwarding operation requests, etc.

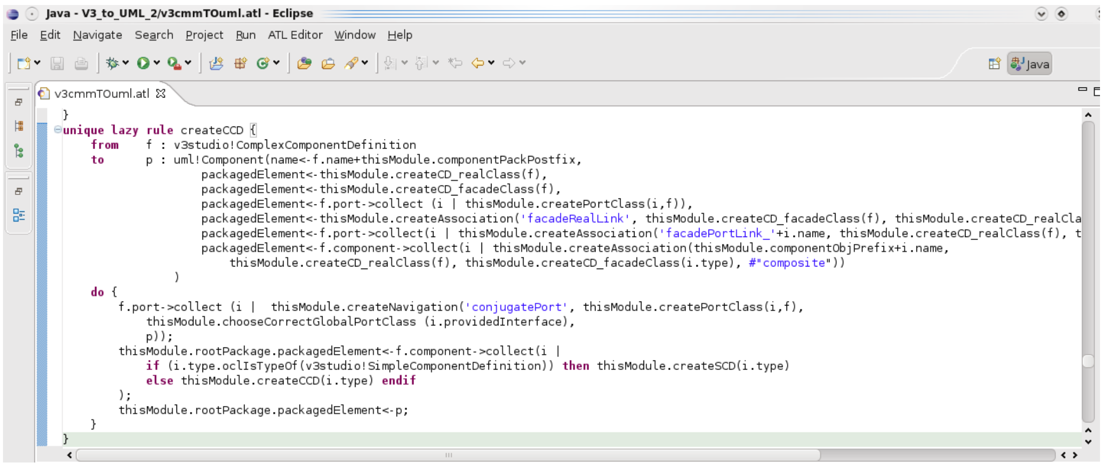
4.2 Generation of Ada 2005 Code

Code for any OO programming language can be easily generated from the UML model previously obtained, since this is the main objective of the M2M transformation. Although any programming language could have been used, we decided to target the Ada language because of the robust mechanisms it provides and for its support for concurrent programming. The UML to Ada M2T transformation generates the skeleton of the application, where designers must just add the Ada code for the `UserDefined` activities. An excerpt of the JET code for the M2T transformation is shown in Fig. 4. The transformation generates a clear, structured, and easy to follow code, which isolates the parts where designers should add their code from the generated code they must not modify, by making extensive use of the package primitive provided by Ada.

5 Case Study

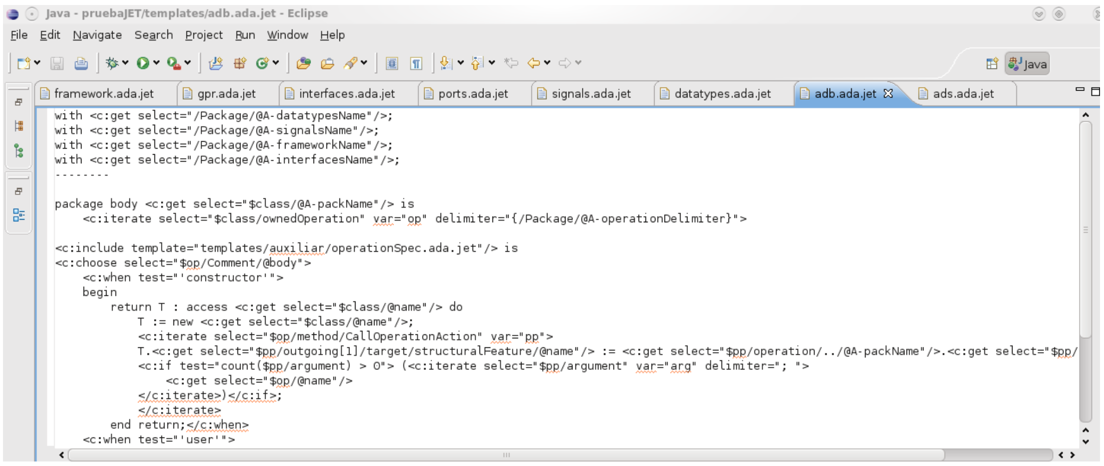
This section illustrates the application of the proposed MDE approach from the point of view of the tool chain user. It comprises a simple yet complete case study regarding a Cartesian robot developed in the context of EU 5th FP project *EFTCoR* (Growth, G3RD-CT-00794) for automating ship hull maintenance operations [14] (see Fig. 5). The EFTCoR addressed the development of a family of robots for grit-blasting, whose mission is to retrieve and confine the paint, oxide and adherences from ship hulls and recycle the blasting material. This case study was selected because it was previously developed using a traditional software development process, and thus the architecture and the algorithms were available. Also, it is simple enough so the reader can focus on the benefits of the approach instead of on the details of the case study.

The Cartesian robot holds a cleaning tool that consists of an enclosed nozzle projecting grit and recovering the residues to be recycled. This tool cleans an area previously identified by a computer vision system. Each axis is moved by a servo-motor and is limited by mechanical switches. The robot is controlled by a *Programmable Logic Controller* (**PLC**) connected to a PC running the high-level control software, which is in charge of identifying the spots that must be blasted, planning the path



```
}
unique lazy rule createCCD {
    from    f : v3studio!ComplexComponentDefinition
    to      p : uml!Component(name<-f.name+thisModule.componentPackPostfix,
                    packagedElement<-thisModule.createCD_realClass(f),
                    packagedElement<-thisModule.createCD_facadeClass(f),
                    packagedElement<-f.port->collect (i | thisModule.createPortClass(i,f)),
                    packagedElement<-thisModule.createAssociation('facadeRealLink', thisModule.createCD_facadeClass(f), thisModule.createCD_realCla
                    packagedElement<-f.port->collect(i | thisModule.createAssociation('facadePortLink_'+i.name, thisModule.createCD_realClass(f), t
                    packagedElement<-f.component->collect(i | thisModule.createAssociation(thisModule.componentObjPrefix+i.name,
                        thisModule.createCD_realClass(f), thisModule.createCD_facadeClass(i.type), #"composite"))
                )
    do {
        f.port->collect (i |  thisModule.createNavigation('conjugatePort', thisModule.createPortClass(i,f),
            thisModule.chooseCorrectGlobalPortClass (i.providedInterface),
            p));
        thisModule.rootPackage.packagedElement<-f.component->collect(i |
            if (i.type.oclIsTypeOf(v3studio!SimpleComponentDefinition)) then thisModule.createSCD(i.type)
            else thisModule.createCCD(i.type) endif
        );
        thisModule.rootPackage.packagedElement<-p;
    }
}
```

(a) This ATL rule creates part of the UML structure corresponding to a V^{3}CMM ComplexComponentDefinition. This rule invokes other ATL rules, which are in charge, in turn, of creating the UML classes corresponding to the ComponentDefinition (e.g., createCD_realClass and createCD_facadeClass) and its ports (e.g., createPortClass).



```
with <c:get select="/Package/@A-datatypesName"/>;
with <c:get select="/Package/@A-signalsName"/>;
with <c:get select="/Package/@A-frameworkName"/>;
with <c:get select="/Package/@A-interfacesName"/>;
--------

package body <c:get select="$class/@A-packName"/> is
    <c:iterate select="$class/ownedOperation" var="op" delimiter="{/Package/@A-operationDelimiter}">

<c:include template="templates/auxiliar/operationSpec.ada.jet"/> is
<c:choose select="$op/Comment/@body">
    <c:when test="'constructor'">
    begin
        return T : access <c:get select="$class/@name"/> do
            T := new <c:get select="$class/@name"/>;
            <c:iterate select="$op/method/CallOperationAction" var="pp">
            T.<c:get select="$pp/outgoing[1]/target/structuralFeature/@name"/> := <c:get select="$pp/operation/../@A-packName"/>.<c:get select="$pp/
            <c:if test="count($pp/argument) > 0"> (<c:iterate select="$pp/argument" var="arg" delimiter="; ">
                <c:get select="$op/@name"/>
            </c:iterate>)</c:if>;
            </c:iterate>
        end return;</c:when>
    <c:when test="'user'">
```

(b) This excerpt of the JET M2T transfomation generates an Ada package body for each component. Specifically, it creates the constructor subprogram for each component.

Figure 4. Screenshots showing an excerpt of the two model transformations implemented as part of the proposed Eclipse-based tool chain: (a) M2M transformation from V^{3}CMM to UML, and (b) M2T transformation from UML to Ada.

to blast them, and sequentially moving the cleaning tool to the identified spot positions.

The case study comprises three main steps, namely (1) modeling the CB architecture of the application using the three views included in V^{3}CMM , (2) executing the ATL M2M transformation in order to generate a UML OO design from the previous CB model, and (3) executing the JET M2T transformation in order to generate an OO implementation (code) from the previous UML model.

5.1 Modeling the Architecture of the Cartesian Robot with V^{3}CMM

Part of the CB architecture is depicted in Fig. 6. As V^{3}CMM does not impose any concrete architecture, we used **ACRoSeT** (*Reference Control Architecture for Service Robots*) [38], which was developed in a previous work. The components appearing in the figure are defined according to the component types defined by ACRoSeT. Although not the objective of this article, it is worth defining their meaning: **HAL** (*Hardware Abstraction Layer*) components model the interface with the control hardware, **SC** (*Simple*

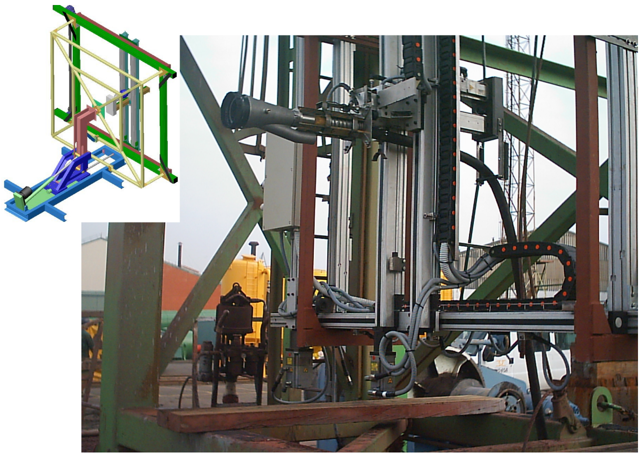


Figure 5. Cartesian robot developed in the context of the EU 5^{th} FP project EFTCoR for ship hull cleaning.

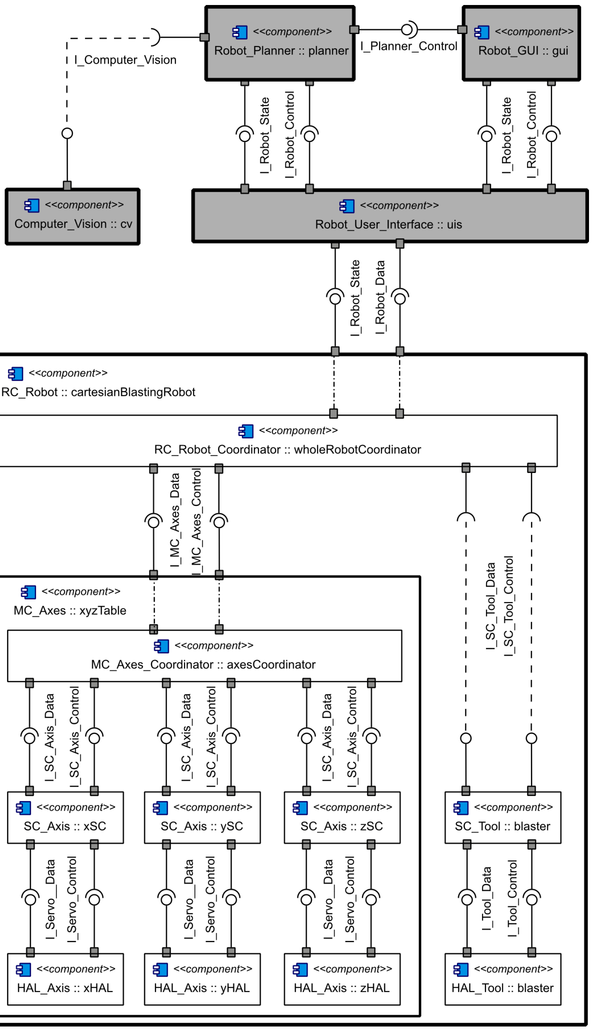
Controller) components model a controller of one actuator, **MC** (*Mechanism Controller*) components model a coordinator of several SCs (and also MCs), and **RC** (*Robot Controller*) component model the coordinator of a whole robot. Both HALs and SCs are simple components, while MCs and RCs are complex components. As complex components do not have behavior on their own in V^{3}CMM, both MCs and RCs require the definition of an additional *Coordinator* component, which is itself a simple component, for coordinating their inner components.

The architecture of the Cartesian robot also comprises the following components: a graphic user interface (*Robot_Gui*), a path planer and sequencer (*Robot_Planner*), a computer vision component (*Computer_Vision*), and a user interface (*Robot_User_Interface*) that controls the commands that are sent to the RC component. These components are all complex components that are depicted as black-box components (on the contrary of the RC component) for the sake of simplicity. The definition of the architecture comprises the following steps.

Step 1) Define the common data types and interfaces (together with their operations) of the whole application, in one or more files. For instance, the definition of the interfaces and data types for the SC_Axis component of the Cartesian robot is shown in Fig. 7.

Step 2) Create the simple component definitions using the data types and interfaces previously defined, each in its own file. SC_Axis, HAL_Axis, MC_Axes_Coordinator, SC_Tool, HAL_Tool and RC_Robot_Coordinator are all simple components.

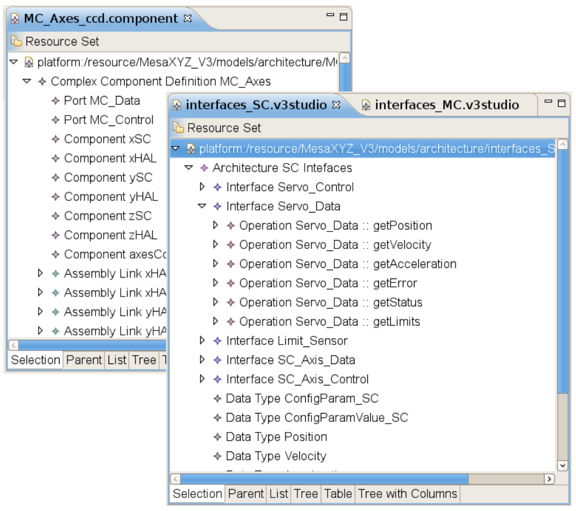
Step 3) Create the complex component definitions using the data types, interfaces, and component definitions previously defined. Complex components are defined in their own files, and they contain instances of component definitions (simple or complex) and links between the *compatible* ports of these component instances. The meaning of "compatible" depends on the components being connected. In case both components are contained in the same complex component, their ports can be connected by means of an `AssemblyLink` (see Fig. 3) as long as the interfaces required by one port are provided by the other and *vice-versa*. When connecting a complex component to a component it contains, their ports can be connected by means of a `DelegationLink` as long as the interfaces required by one port are required by the other and *vice-versa*.



<<component>> Robot_Planner :: planner
I_Planner_Control
<<component>> Robot_GUI :: gui
I_Computer_Vision
I_Robot_State
I_Robot_Control
<<component>> Computer_Vision :: cv
<<component>> Robot_User_Interface :: uis
I_Robot_Data
<<component>> RC_Robot :: cartesianBlastingRobot
<<component>> RC_Robot_Coordinator :: wholeRobotCoordinator
I_MC_Axes_Data
I_MC_Axes_Control
I_SC_Tool_Data
I_SC_Tool_Control
<<component>> MC_Axes :: xyzTable
<<component>> MC_Axes_Coordinator :: axesCoordinator
I_SC_Axis_Data
I_SC_Axis_Control
<<component>> SC_Axis :: xSC
<<component>> SC_Axis :: ySC
<<component>> SC_Axis :: zSC
<<component>> SC_Tool :: blaster
I_Servo_Data
I_Servo_Control
I_Tool_Data
I_Tool_Control
<<component>> HAL_Axis :: xHAL
<<component>> HAL_Axis :: yHAL
<<component>> HAL_Axis :: zHAL
<<component>> HAL_Tool :: blaster

Figure 6. ACRoSeT based software architecture for the Cartesian robot shown in Fig. 5. Grey components are complex components shown as black-boxes.

In the case study, components MC_Axes and RC_Robot (as well as the black-box components) are complex components. As a sample of the reuse mechanism embedded in V^{3}CMM, it is worth noting that the complex component MC_Axes (shown in Fig. 7) contains three instances of the SC_Axis simple component definition, three instances of the HAL_Axis, and one instance of the MC_Axes_Coordinator.



```
MC_Axes_ccd.component
Resource Set
▽ platform:/resource/MesaXYZ_V3/models/architecture/M
  ▽ Complex Component Definition MC_Axes
      Port MC_Data
      Port MC_Control
      Component xSC
      Component xHAL
      Component ySC
      Component yHAL
      Component zSC
      Component zHAL
      Component axesC
    ▷ Assembly Link xHA
    ▷ Assembly Link xHA
    ▷ Assembly Link yHA
    ▷ Assembly Link yHA
Selection | Parent | List | Tree | T
```

```
interfaces_SC.v3studio | interfaces_MC.v3studio
Resource Set
▽ platform:/resource/MesaXYZ_V3/models/architecture/interfaces_S
  ▽ Architecture SC Intefaces
    ▷ Interface Servo_Control
    ▽ Interface Servo_Data
      ▷ Operation Servo_Data :: getPosition
      ▷ Operation Servo_Data :: getVelocity
      ▷ Operation Servo_Data :: getAcceleration
      ▷ Operation Servo_Data :: getError
      ▷ Operation Servo_Data :: getStatus
      ▷ Operation Servo_Data :: getLimits
    ▷ Interface Limit_Sensor
    ▷ Interface SC_Axis_Data
    ▷ Interface SC_Axis_Control
      Data Type ConfigParam_SC
      Data Type ConfigParamValue_SC
      Data Type Position
      Data Type Velocity
Selection | Parent | List | Tree | Table | Tree with Columns
```

Figure 7. Eclipse screenshots showing the definition of some models with V^{3}CMM. The foreground image shows the definition of the data types, the interfaces and the operations of the SC_Axis simple component, while the background image shows the definition of the MC_Axes complex component.

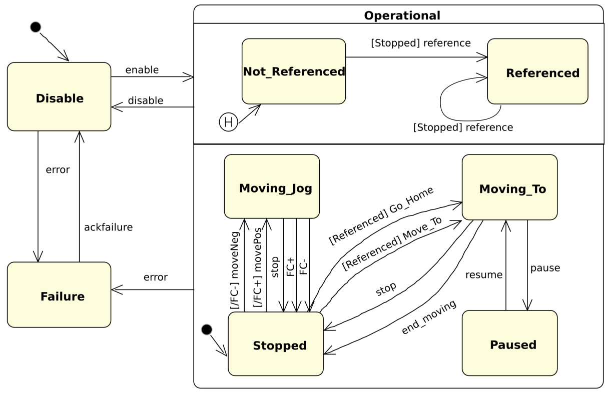


Figure 8. State-machine model for the SC_Axis simple component.

Step 4) Design the state-machines using the behavioral view, each in its own file. For each simple component, the designer creates a state-machine describing the component internal behavior and its reaction to the messages it receives from other components. Fig. 8 depicts the state-machine model for the SC_Axis simple component. State-machines are driven by the operation requests issued by other components. These operation requests trigger the transitions of the state-machine, provided the guard condition (when present) is satisfied.

Step 5) Design the algorithms using the algorithmic view. This view, based on the UML activity diagrams, describes the algorithms that will be executed by the component depending on its current state. Fig. 9 depicts the activity

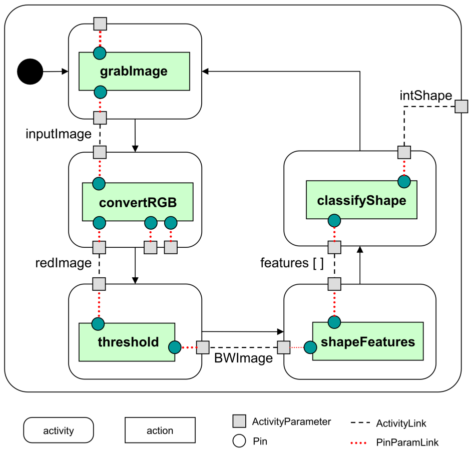


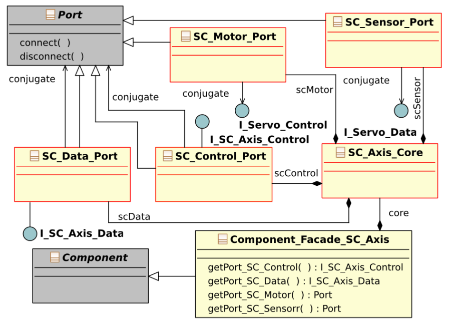
Figure 9. Activity diagram associated to one of the states of the Computer_Vision component, aimed at identifying visual targets for cleaning operations.

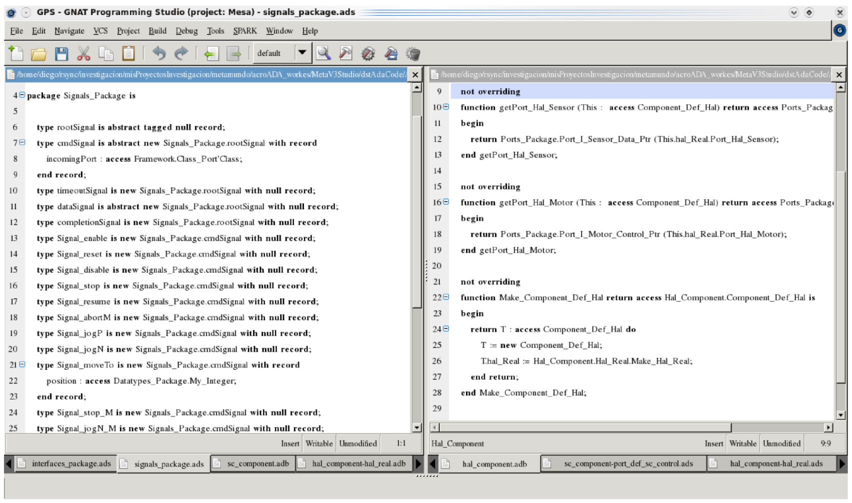
diagram associated to one of the states of the Computer_Vision component.

Step 6) Link activities to state-machines. Each state and transition of the state-machine must be associated with the corresponding activity that describes the algorithm that must be executed whenever the component is in a given state, enters or exits the state, or a transition is triggered. This step, as well as the next one, illustrates the loosely coupling relationship existing between the V^{3}CMM views, since each is defined in a different file.

Step 7) Link state-machines to components. Each component definition must be associated with one of the state-machines definitions designed in Step 4. The only constraint is that the state-machine must conform with the interfaces provided and required by the component. In the same way as the component definitions of the HAL_Axis and SC_Axis simple components have been reused three times (as described in step 3), the state-machine definitions associated to those simple components are also instantiated three times, since the event-driven behavior of the three axes is identical.

Note that defining the V^{3}CMM models in separate files is possible thanks to the loosely coupling mechanism provided by the meta-model, and that this enables designers to load and reuse these models as many times as needed, either in the same or in different designs. The description of the CB software architecture of the Cartesian robot comprises the following separate files: (1) one file containing the ten interfaces appearing in the design, together with their operations and the data types they use, (2) eight files, each one containing a component definition, (3) eight files, each one containing the state-machine definition describing the behavior associated to the previous component definitions, and (4) twenty files, each one containing a group of related activities (less than one hundred in total).




GPS - GNAT Programming Studio (project: Mesa) - signals_package.ads
File Edit Navigate VCS Project Build Debug Tools SPARK Window Help
default
/home/diego/rsync/investigacion/misProyectosInvestigacion/metamundo/acroADA_workes/MetaV3Studio/dstAdaCode/
Insert Writable Unmodified 1:1
interfaces_package.ads signals_package.ads sc_component.adb hal_component-hal_real.adb
Hal_Component
Insert Writable Unmodified 9:9
hal_component.adb sc_component-port_def_sc_control.ads hal_component-hal_real.ads

```
package Signals_Package is

   type rootSignal is abstract tagged null record;
   type cmdSignal is abstract new Signals_Package.rootSignal with record
      incomingPort : access Framework.Class_Port'Class;
   end record;
   type timeoutSignal is new Signals_Package.rootSignal with null record;
   type dataSignal is abstract new Signals_Package.rootSignal with null record;
   type completionSignal is new Signals_Package.rootSignal with null record;
   type Signal_enable is new Signals_Package.cmdSignal with null record;
   type Signal_reset is new Signals_Package.cmdSignal with null record;
   type Signal_disable is new Signals_Package.cmdSignal with null record;
   type Signal_stop is new Signals_Package.cmdSignal with null record;
   type Signal_resume is new Signals_Package.cmdSignal with null record;
   type Signal_abortM is new Signals_Package.cmdSignal with null record;
   type Signal_jogP is new Signals_Package.cmdSignal with null record;
   type Signal_jogN is new Signals_Package.cmdSignal with null record;
   type Signal_moveTo is new Signals_Package.cmdSignal with record
      position : access Datatypes_Package.My_Integer;
   end record;
   type Signal_stop_M is new Signals_Package.cmdSignal with null record;
   type Signal_jogN_M is new Signals_Package.cmdSignal with null record;
```

```
   not overriding
   function getPort_Hal_Sensor (This :  access Component_Def_Hal) return access Ports_Packag
   begin
      return Ports_Package.Port_I_Sensor_Data_Ptr (This.hal_Real.Port_Hal_Sensor);
   end getPort_Hal_Sensor;

   not overriding
   function getPort_Hal_Motor (This :  access Component_Def_Hal) return access Ports_Package
   begin
      return Ports_Package.Port_I_Motor_Control_Ptr (This.hal_Real.Port_Hal_Motor);
   end getPort_Hal_Motor;

   not overriding
   function Make_Component_Def_Hal return access Hal_Component.Component_Def_Hal is
   begin
      return T : access Component_Def_Hal do
         T := new Component_Def_Hal;
         T.hal_Real := Hal_Component.Hal_Real.Make_Hal_Real;
      end return;
   end Make_Component_Def_Hal;

```

Figure 10. Class diagram generated by the ATL M2M transformation for the SC_Axis simple component (left). Screenshot including an excerpt of the Ada code generated by the JET M2T transformation (right).

5.2 Execution of the Model Transformations

After the user has modeled the architecture of the Cartesian robots, he must configure and execute the model transformations. Firstly he must execute the M2M transformation. Fig. 10 depicts an excerpt of the generated UML model for the SC_Axis simple component after the execution of the ATL M2M transformation. The Component_Façade_SC_Axis class provides the public interface of the SC_Axis component, which comprises a constructor (not shown in the figure) and four methods for obtaining a reference to each of its ports. These references will be later used for connecting the ports to other components, as the architecture of the application models. The component functionality is implemented by the SC_Axis_Core class. Each port is translated into a class, which provides and requires interfaces as stated in the V^3CMM model. Only the façade has 'public' visibility, while the rest of the classes implementing the component are kept safe from external access by giving them 'package' visibility. After the UML model has been generated, the user must execute the M2T transformation. Fig. 10 depicts a screenshot of the Ada code generated after the execution of the JET M2T transformation.

5.3 Result Analysis and Comparison

The two-stage transformation, from the Cartesian robot architectural design into Ada code, resulted into fifty seven packages, which contain a total of sixty five classes. The whole application contains 4800 Lines Of Code (LOC), from which more than 3500 (nearly a seventy five percent) have been automatically generated. Although the number of generated packages and classes may seem too high, it is worth noting that this is the result of applying the selected design patterns, aimed to produce a well structured and readable code. It took the authors one month to obtain the final application code, namely: one week for formally defining and validating the initial V^3CMM architectural model, and three weeks for manually completing the automatically generated Ada skeleton. It is worth noting that the development time and effort would have sensibly decreased if the authors would have counted on a set of Ada libraries, which functionality could have been readily reused by including `LibraryCall` (instead of `UserDefined`) activities in the V^3CMM design. Besides, the need for manually completing the generated application may lead to the problems already described in Section 3.2.

As formerly stated, the authors have a previous implementation of the system available, which is currently being used is several shipyards. This implementation, developed in C++, contains around 4500 LOC, grouped into seventeen classes. The time needed to manually implement this application was of eight months. It is worth noting that, in spite of the efforts for defining the CB software architecture of the Cartesian robot using ACRoSeT, the lack of tools supporting both the formal definition and validation of the models and their translation into code, ended up with a quite error-prone manual implementation of the design. As a consequence, a lot of time was expended manually testing and patching the successive versions.

6 Discussion

The MDE approach provides the theoretical support for creating, manipulating and transforming models (sets of concepts), but it provides no guidelines for selecting these concepts or for organizing a complete MDE-based development process. In order to leverage all the potential of MDE in robotics software development, a significant research effort needs to be done in order to discover, organize, and precisely define the set of concepts needed to model robotics software requirements at the highest level of abstraction [7] (see Fig. 1). Additionally, it is also needed to implement automated model transformations enabling designers to refine their high-level models into lower level representations (until generating the final application code or a part of it).

Up to date, the definition of such core concepts has proved elusive, and, even provided with such a CIM language, one or more intermediate PIM levels will be required in order to progressively reduce the semantic gap existing between the high-level concepts and the platform details. For these reasons the authors decided to develop a PIM language as a first step towards a complete MDA process. The level of abstraction provided by components and CBSD was selected since (1) CBSD is a very mature approach with a broad tradition in the community, (2) CBSD enables describing (and checking) the software architecture of the applications, (3) CBSD is aimed to reuse by its very nature, and (4) because CBSD provides many advantages over OO, particularly a higher abstraction level as stated in Section 2.

Once components and CBSD were selected for the PIM language, the next step is to precisely define, by means of a meta-model, *what* components are, *how* they can be composed, and *how* they interact. At this point we considered two options: use or profile a general purpose modeling language like UML/SysML, or develop a new one. This last option was not available until recently, and thus UML profiles were the only solution for generating graphical tools and for manipulating models. But with the standardization of the MOF and the support provided by Eclipse, designers no longer have to rely on UML for generating such tools.

The adoption of UML or SysML (for instance, UML «component» or SysML «block» concepts) seems the most immediate and correct option. However, the generality of these languages brings, in our opinion, an important problem: they provide many and very generally defined concepts, which semantics must be completed by users at some points (the so-called "semantic variation points" in UML). Besides, such general concepts must be sometimes combined in order to make more concrete definitions. This in turn makes developing model transformations a more difficult task, since in this case it is necessary to check that all elements are correctly combined to express a given concept. As an alternative, the UML profiling mechanism can be used to specialize the semantics of some of its concepts. Profiles provide a straightforward mechanism for adapting an existing meta-model with constructs that are specific to a particular domain, platform, or method. However, profiling does not reduce the number of concepts, and making a profile of such a big meta-model as UML is neither easy nor efficient. In addition, models built from this profile would be rambling (plenty of tags and stereotypes) and difficult to inspect and debug.

It is worth highlighting that we do not encourage to give up using UML. It is an excellent and very complete modeling language when dealing with the whole development life-cycle of OO applications, but it has several limitations when applied to other domains that have been already pointed out by other authors [39], [8], [30], [40].

Nevertheless, since V^{3}CMM adopts and adapts many UML concepts, it is possible to design model transformations in order to convert V^{3}CMM components into UML components, but the opposite way is not always possible, as UML provides many ways of modeling a component that are not always compliant with V^{3}CMM components. This discussion about the differences between UML and V^{3}CMM resembles the differences between English and the mathematical notation: $3 + 4 = 7$ and "three plus four equals seven" have the same meaning, although not both notations are equally useful to express it. At the end, it is a matter of economy of symbols and formality of the underlying language. Besides, since V^{3}CMM is based on UML, it is easy to use, and because it comprises few concepts, it is easy to understand.

Finally, Section 2 described the three views that comprise V^{3}CMM. Defining just three views may seem insufficient for modeling CB applications, specially when languages like UML or SysML provide many more diagrams that target the whole development life-cycle (e.g., requirement specification, deployment, etc.). V^{3}CMM comprises what we think is the minimum set of concepts needed to model the platform independent aspects of CB applications. Higher level concepts (like requirements for robotic applications) must be modeled by a "stable" CIM language as proposed in [7], and afterward transformed into V^{3}CMM (or any other modeling language). Platform and application specific details (like deployment) or implementation concepts (like classes and objects) must be added in subsequent transformation steps in order to keep the original V^{3}CMM models as much platform independent as possible.

We considered other alternative artifacts for behavior modeling, like Petri Nets [41] or Communicating Sequential Processes (CSP) [42], but they were discarded because they are not as widely known and easy to use as state-machines. Furthermore, interaction diagrams were also discarded because they do not model the behavior of a single component but rather the exchange of messages among components. From our point of view, interaction diagrams can be derived from V^{3}CMM behavioral models through an appropriate model transformation, and thus they should not be part of V^{3}CMM, since having redundant views makes maintaining model consistency extremely difficult. A very important view, specially when designing concurrent applications, is the tasks view. But this view strongly depends on the platform features (operating system, scheduling policies, and inter-process communication mechanisms) and on the timing requirements of the application (hard and soft deadlines, event arrival rates, etc.), and thus these characteristics must be added by the transformations according to the characteristics of the execution platform and the application requirements.

7 Conclusions, Lessons Learnt, and Future Research Lines

This article proposes a new reuse-centric, platform independent and model-driven approach to CB robotic system design and implementation. The proposed approach clearly separates the reusable platform independent aspects from the platform dependent details and from the application specific requirements. In this vein, the work described in this article comprises (1) a meta-model that precisely defines and formalizes the basic architectural units of design (V^3CMM), (2) a M2M transformation from V^3CMM to OO concepts expressed in UML, and (3) a M2T transformation from the mentioned UML model to the Ada 2005 programming language. These facilities constitute a small, but yet operative, development framework for generating well structured Ada applications from CB designs. A simple yet complete case study in robotics has been presented to illustrate both the modeling capabilities and the reuse benefits of V^3CMM, as well as the final results obtained by the two-stage model transformation.

The small and simple components considered in the case study provide less reuse benefits than large and complex examples. However, even for these components, the MDE approach proposed in the article can be considered beneficial, since: (1) designers are provided with an expressive yet simple to use modeling language that enables them to formally define their components and CB designs at a high level of abstraction, (2) they can effortlessly obtain different implementations from their designs (either for a single or for different platforms), provided the corresponding model transformations are available, and (3) the formal foundations on which both models and model transformations are defined allow them to formally validate all the artifacts involved in the process, achieving more dependable and less error prone solutions in a shorter time. As an example of these benefits, consider that the manual implementation of state-machines, even when they are relatively simple, is an intensive, tedious, and error-prone work.

The adoption of a new paradigm always involves facing many challenging issues regarding its application. MDE technologies eases the way in which applications can be designed, implemented and tested, but it does not provide any criteria for developing a complete MDE solution. From our experience, the two most challenging issues we faced were the selection of the concepts for V^3CMM (and its definition directly from MOF) and the definition of the run-time support according to the application specific details by means of different model transformations. Moreover, while V^3CMM can be used in multiple types of applications, not only in the robotics domain, transformations have a more limited scope and are very sensitive to changes in the meta-model. In this last case, it is very likely that all the transformations have to be redefined, and, unfortunately, during research such changes are almost inevitable. This fact enforces our idea that meta-models should be as simple and stable as possible.

Developing model transformations requires a deep knowledge of both the source and the target languages (including both their syntax and precise semantics), and the selection of the most appropriate transformation patterns. As a result, model transformation design and implementation are quite time consuming tasks, which require highly skilled people. It is also worth noting that the learning curve associated to the use of model transformation tools increases development time. Nevertheless, model transformations provide the required flexibility to target any platform and programming language. As an example, the V^3CMM-to-UML ATL transformation took about four man–month, while the UML-to-Ada JET transformation required just one man–month.

Another challenge is the integration of the existing robotics frameworks and libraries in the proposed MDE development process. In this line, three complementary approaches can be considered. The first one relies on a direct model transformation from V^3CMM to the chosen framework. In this case the difficulties lie in finding the correct correspondences between the behavioral V^3CMM views and the way in which behavior is defined in the chosen framework, since the structural part uses to be similar in all CB frameworks. The second approach relies on the application of the recent OMG's Architecture-Driven Modernization (ADM) [43] initiative, aimed at generating models directly from source code. In this way, ADM is a very promising way to integrate the existing legacy code into a MDE process. Finally, the third approach relies on the integration of the libraries that provide the functionality of the domain without using the framework run-time support. In order to reuse these libraries, the less dependencies they have, the easier their integration will be [13]. The existence of standards, at least for some of the data usually managed by robotics applications, will be also very valuable [44].

Regarding future research lines, we plan to turn V^3CMM and the developed tools into a more robust, general-purpose and open source environment. In the short-term we will develop a graphical editor for V^3CMM models, and provide two M2T transformations: one that targets C/C++ and another tailored to systems with hard real-time requirements. In the mid-term we will (1) develop a mapping from V^3CMM to one robotic framework, being the candidates those closer to the CBSD paradigm, (2) develop a transformation from V^3CMM models to one analysis tool, such as the Architecture Analysis & Design Language (AADL) [45], the Kernel LAnguage for PErformance and Reliability (KLAPER) [46], or Palladio [40], and (3) extend the transformations to consider component distribution using middleware technologies. In the long-term we plan to create several component and model transformation catalogs and integrate V^3CMM in a Software Product Line [47] approach in order to define common architectures for different families of products. It will then be possible to derive a concrete architecture for one product by selecting components (in the form of V^3CMM models) and transformations from their repositories.

References

[1] The ANSI-IEEE 1471-2000 Standard, "IEEE Recommended Practice for Architectural Description of Software-Intensive Systems", The Institute of Electrical and Electronics Engineers (IEEE), Sep. 2000. 1

[2] C. Schlegel, T. Hassler, A. Lotz, and A. Steck, "Robotic software systems: from code-driven to model-driven designs", in *Proc. International Conference on Advanced Robotics ICAR 2009*. IEEE, 2009, pp. 1–8. 1, 1, 2

[3] C. Atkinson and T. Kühne, "Model-driven development: a metamodelling foundation", *IEEE Softw.*, vol. 20, no. 5, pp. 14–18, 2003. 1

[4] B. Selic, "The pragmatics of model-driven development", *IEEE Trans. Software Eng.*, vol. 20, no. 5, pp. 19–25, 2003. 1

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

[6] OMG, "MDA success stories", Jun. 2008. [Online]. Available: http://www.omg.org/mda/products_success.htm 1

[7] D. Brugali and P. Salvaneschi, "Stable aspects in robot software development", *International Journal of Advanced Robotic Systems*, vol. 3, no. 1, pp. 17–22, Mar. 2006. 1, 6

[8] J. Bézivin, "On the unification power of models", *Journal of Systems and Software*, vol. 4, no. 2, pp. 171–188, May 2005. 1, 6

[9] E. Seidewitz, "What models mean", *IEEE Softw.*, vol. 20, no. 5, pp. 26–32, 2003. 1

[10] T. Mens and P. van Gorp, "A taxonomy of model transformation", *Electronic Notes in Theoretical Computer Science*, vol. 152, pp. 125–142, 2006. 1

[11] OMG, *Model Driven Architecture Guide, version v1.0.1, omg/2003-06-01*, Jun. 2003. [Online]. Available: http://www.omg.org/docs/omg/03-06-01.pdf 1

[12] A. van Deursen, P. Klint, and J. Visser, "Domain-specific languages: an annotated bibliography", *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000. 1

[13] A. Makarenko, A. Brooks, and T. Kaupp, "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. 1, 2, 7

[14] A. Iborra, D. Alonso, F. Ortiz, J. Franco, P. Sánchez, and B. Álvarez, "Design of service robots", *IEEE Robot. Automat. Mag., Special Issue on Software Engineering for Robotics*, vol. 16, no. 1, pp. 24–33, Mar. 2009. 1, 3, 5

[15] OMG, *Unified Modeling Language (UML) Superstructure specification v2.1.1, formal/2007-02-05*, Feb. 2007. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?formal/07-02-03.pdf 1

[16] OMG, *OMG Systems Modeling Language (SysML) specification v1.1, formal/2008-11-01*, Nov. 2008. [Online]. Available: http://www.omg.org/spec/SysML/1.1/ 1

[17] C. Szyperski, *Component software: beyond object-oriented programming*, 2nd ed. A-W, 2002. 2

[18] M. Fayad, D. Schmidt, and R. Johnson, *Building Application Frameworks: Object-Oriented Foundations of Framework Design*. John Wiley & Sons, 1999. 2

[19] H. Bruyninckx, "Open Robot Control Software: the OROCOS project", in *Proc. of the IEEE International Conference on Robotics and Automation*, vol. 3. IEEE, 2001, pp. 2523–2528. 2

[20] A. Makarenko, A. Brooks, and T. Kaupp, "Orca: Components for robotics", in *Proc. of the International Conference on Intelligent Robots and Systems*. IEEE, 2006. 2

[21] T. Collett, B. MacDonald, and B. Gerkey, "Player 2.0: Toward a practical robot programming framework", in *Proc. of the Australasian Conference on Robotics and Automation*, 2005. [Online]. Available: http://www.araa.asn.au/acra/acra2005/papers/collet.pdf 2

[22] Robot Standards and Reference Architectures (RoSTa), Coordination Action funded under EU's FP6. [Online]. Available: http://wiki.robot-standards.org/index.php/Current_Middleware_Approaches_and_Paradigms 2

[23] K. Lau and Z. Wang, "Software component models", *IEEE Trans. Software Eng.*, vol. 33, no. 10, pp. 709–724, Oct. 2007. 2

[24] X. Blanc, J. Delatour, and T. Ziadi, "Benefits of the MDE approach for the development of embedded and robotic systems. Application to Aibo", in *Proc of the* 3^{rd} *National Conference onControl Architectures of Robots*, 2007. 2

[25] S. Jorges, C. Kubczak, F. Pageau, and T. Margaria, "Model driven design of reliable robot control programs using the jABC", in *Proc. Fourth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems EASe '07*. IEEE, 2007, pp. 137–148. 2

[26] OMG, *Meta-Object Facility (MOF) specification v2.0, formal/2006-01-01*, Oct. 2006. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?formal/06-01-01.pdf 2

[27] "ArtistDesign - European Network of Excellence on Embedded Systems Design", 2008-2011. [Online]. Available: http://www.artist-embedded.org/ 2

[28] "OpenEmbeDD project, Model Driven Engineering open-source platform for Real-Time & Embedded systems", 2008-2011. [Online]. Available: http://openembedd.org/home_html 2

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

[30] D. Alonso, C. Vicente-Chicote, and O. Barais, "V^3Studio: A Component-Based Architecture Modeling Language", in *15th Annual IEEE International Conference and Workshop on Engineering of Computer Based Systems*. IEEE, Mar. 2008, pp. 346–355. 3, 6

[31] C. Vicente-Chicote, F. Losilla, A. Álvarez, B.and Iborra, and P. Sánchez, "Applying MDE to the Development of Flexible and Reusable Wireless Sensor Networks", *International Journal of Cooperative Information Systems (IJCIS)*, vol. 16, no. 3/4, pp. 393–412, Sep.-Dec. 2007. 3

[32] M. Jiménez, F. Rosique, P. Sánchez, B. Álvarez, and A. Iborra, "Habitation: a domain-specific language for home automation", *IEEE Softw., Special Issue on Domain Specific Languages*, Jul. 2009. 3

[33] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework*, 2nd ed., A.-W. Prof., Ed., 2008. 3

[34] OMG, *Object Constraint Language (OCL) specification v2.0, formal/06-05-01*, May 2006. [Online]. Available: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf 3

[35] F. Jouault, F. Allilairea, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool", *Science of Computer Programming*, vol. 72, no. 1-2, pp. 31–39, 2008. 3

[36] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. A-W Prof., Jan. 1995. 4.1

[37] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, volume 2: patterns for concurrent and networked objects*. Wiley, 2000. 4.1

[38] B. Álvarez, P. Sánchez, J. Pastor, and F. Ortiz, "An architectural framework for modeling teleoperated service robots", *Robotica*, vol. 24, no. 4, pp. 411–418, Dec. 2005. 5.1

[39] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins, "Modeling software architectures in the unified modeling language", *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 1, pp. 2–57, Aug. 2002. 6

[40] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction", *Journal of Systems and Software*, vol. 82, no. 1, pp. 3–22, 2009. 6, 7

[41] T. Murata, "Petri nets: Properties, analysis and applications", *Proc. IEEE*, vol. 77, no. 4, pp. 541–580, Apr. 1989. 6

[42] C. Hoare, *Communicating Sequential Processes*. Prentice Hall, 1985. 6

[43] OMG, "Architecture-Driven Modernization." [Online]. Available: http://adm.omg.org 7

[44] K. Nilsson, T. Olsson, and H. Bruyninck, "Basic Robotics Standards (BRoS) - Motivations and examples", in *Proc. of the IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07)*. IEEE, 2007. 7

[45] P. Feiler, D. Gluch, and J. Hudak, "The Architecture Analysis & Design Language (AADL): an introduction", SEI (CMU),

Tech. Rep. CMU/SEI-2006-TN-011, 2006. [Online]. Available: http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn011.pdf 7

[46] V. Grassi, R. Mirandola, and A. Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: a model-driven approach", *Journal of Systems and Software*, vol. 80, no. 4, pp. 528–558, 2007. 7

[47] L. Northrop, "SEI's Software Product Line Tenets", *IEEE Softw.*, vol. 19, no. 4, pp. 32–40, 2002.

7

Diego Alonso received his M. Sc. degree in Industrial Engineering from the Technical University of Valencia (Spain), in 2001, and a Ph. D. degree (European Mention) from the Technical University of Cartagena (Spain), in 2008. Currently, he is a lecturer in the Department of Information and Communication Technologies at the Technical University of Cartagena. He has published about 20 refereed journal and conference papers. His research interests focus on the application of the model-driven engineering approach to the development of component-based reactive systems with real-time constraints. He is a member of IFAC.

Cristina Vicente-Chicote received her M. Sc. in Computer Science from the University of Murcia (Spain) in 1997, and the Ph. D. degree in Telecommunication Engineering (European Mention) from the Technical University of Cartagena (Spain), in 2005. Currently, she is an associate professor in the Department of Information and Communication Technologies at the Technical University of Cartagena. She has published about 30 refereed journal and conference papers. Her research interest covers Model-Driven Engineering and Component-Based Software Development applied to Robotics, Mechatronics, and Computer Vision Systems.

Francisco J. Ortiz received his B. Sc and M. Sc. degrees in industrial engineering from the University of Murcia (Spain) in 1993 and 1997, respectively, and the Ph. D. degree in Industrial Engineering from the Technical University of Cartagena (Spain) in 2005. He is currently an associate professor at the Technical University of Cartagena, in the Department of Electronics Technology. Since 1999, he has participated in different projects focused in Computer Assisted Surgery, Service Robotics and Software Engineering. His current research interests include Mechatronic Systems, Software Architectures for Robotics and Model-Driven Engineering.

Juan A. Pastor received his M. Sc. degree in Telecommunication Engineering from the Technical University of Madrid (Spain), in 1997, and the Ph. D. degree in Telecommunication Engineering from the Technical University of Cartagena (Spain), in 2002. Currently, he is an associate professor in the Department of Information and Communication Technologies at the Technical University of Cartagena. 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.

Bárbara Álvarez received her M. Sc. degree in Telecommunication Engineering from the Technical University of Madrid (Spain), in 1993, and the Ph. D. degree in Telecommunication Engineering from the Technical University of Madrid, in 1997. In 1998, she was a faculty member at the University of Murcia (Spain), and in 1999 at the Technical University of Cartagena (Spain). Currently, she is an associate professor in the Department of Information and Communication Technologies at the Technical University of Cartagena. She has published about 50 refereed journal and conference papers. Her research interest covers Robotics and Software Engineering. She is a member of Ada-Europe, IFAC and IEEE.