V^{3}Studio: A Component-Based Architecture Modeling Language

Diego Alonso, Cristina Vicente-Chicote
Dpto. de Tecnologías de la Información y las Comunicaciones (TIC)
Universidad Politécnica de Cartagena, Ed. Antigones, 30202 Cartagena (Spain)
{diego.alonso, cristina.vicente}@upct.es

Olivier Barais
Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA)
Campus de Beaulieu, F-35042 Rennes Cedex (France)
barais@irisa.fr

Abstract

Component-Based Software Development (CBSD) has proven to obtain highly reusable, extensible and evolvable designs. This paper presents a Model-Driven Engineering approach to CBSD which revolves around the definition of the V^{3}Studio component-based meta-model and a set of graphical modeling tools implemented to support it. V^{3}Studio has been designed to model the structure and behavior of a wide variety of reactive systems by means of three complementary views, namely: one for describing the components of the architecture (structural view), and two for describing their behavior (coordination and algorithmic views). Dividing the V^{3}Studio meta-model into these three loosely coupled views considerably simplifies the overall design process, allowing designers to reuse previously defined models. In order to show the feasibility and the benefits of the proposal, a simple but complete case study regarding the design of a vision guided robotic system will be presented.

1. Introduction

Physical systems capable of perceiving and responding to external stimuli coming from its environment are considered reactive systems. This definition encompasses all physical devices that exhibit some organized behaviour, as well as interconnections of these devices into large networks of dynamic and interacting components.

Our research group is interested in developing reactive system following a Component-Based Software Development (CBSD) approach. Specifically, we have gained quite a lot of experience developing robotic systems [22], machine-vision systems [21] and, more recently, Wireless Sensors and Actuators Network (WSAN) applications [9]. We have followed a bottom-up CBSD approach to build our systems trying to incorporate some of the related Off-The-Self (OTS) components currently available (commonly as part of domain-specific libraries) in the marketplace.

As stated in [19], "*software architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution*". This definition points out the close relationship existing between software architecture and component specification [5].

Component-Based Software Development (CBSD) has proven to obtain highly reusable, extensible and evolvable designs [18]. According to this approach, systems can be built by selecting and assembling appropriate *Commercial Off-The-Shelf* (COTS) components. However, integrating CBSD (bottom-up) and software architecture design (top-down) is a non-trivial task which commonly requires adapting the architecture to make COTS components fit.

The *Model-Driven Engineering* (MDE) [17] approach offers an effective solution for bridging the gap between architecture design and CBSD. MDE owes part of its success to the number of standards the *Object Management Group* (OMG) has developed as part of its *Model-Driven Architecture* (MDA) [12] proposal. The MDE approach revolves around the definition of models and model transformations. Models represent part of the functionality, structure and/or behavior of a system [6], and they are defined in terms of formal meta-models. A meta-model includes the set of concepts needed to describe a domain at a certain level of abstraction, together with the relationships existing between them. Model transformations, commonly described as meta-model mappings, enable the automatic transformation and evolution of models into (1) other

978-0-7695-3141-0/08 $25.00 © 2008 IEEE
DOI 10.1109/ECBS.2008.9

models, defined at any level of abstraction, or (2) any given textual format (e.g. code) [11].

In the context of MDE and CBSD, this paper presents the V^{3}Studio meta-model and a set of graphical modeling tools implemented to support reactive system component-based architecture design. In spite of having been conceived to fit our needs regarding reactive system design and implementation, V^{3}Studio is a general purpose and platform-independent meta-model, which allows designers to model both the structure and the behavior of a wide range of applications. The reduced set of concepts included in the meta-model makes it simple to use, and its division in three loosely coupled views allows designers to easily reuse previously defined models.

Before presenting the meta-model in detail, the following section briefly reviews the two current trends in MDE regarding meta-model design (i.e. profiling general purpose meta-models such as UML or SysML and defining new domain-specific ones), and justifies why V^{3}Studio has been defined following a combined approach. Then, the rest of the paper is organized as follows. Firstly, section 3 introduces the V^{3}Studio meta-model and the three loosely coupled views it has been divided into, while section 4 details the three graphical modeling tools implemented to support these views. A case study, regarding the design of a vision guided robotic system, is presented in section 5. Finally, section 6 reviews some related works, and section 7 presents the conclusions and outlines some future research lines.

2. V^{3}Studio: neither a DSL nor a UML profile

As stated in [1], nowadays there are two main trends in MDE. The first one promotes the use of standard modeling languages like UML 2.x [16] or SysML [15], while the second one advocates the benefits of *Domain-Specific Languages* (DSLs) [20]. UML 2.x provides a rich set of modeling notations and offers some restriction and extension mechanisms (stereotypes, tags and constraints) which allow developers to adapt it to their particular domains. These customized versions of UML are commonly known as *profiles*. On the other hand, DSLs commonly provide a reduced and well-focused set of concepts and tailored notations for describing specific domains.

As previously stated, this research is focused on the definition of a meta-model which should exhibit the following characteristics: (1) it should be a general purpose modeling language, which enable the description of component-based software architectures for a wide range of reactive systems; (2) it should be expressive but simple, dealing with as few concepts as possible; (3) it should be independent of the target platform and of the specific reactive application domain (robotics, computer vision, etc.); and (4) it should be easy to integrate with other meta-models as part of an overall MDE software development process. In order to obtain such a meta-model, the following options were considered:

Option 1: select UML 2.x [16] as the target meta-model. UML fulfills all the previous requirements but the one regarding simplicity. As its name states, UML supports general purpose system modeling, covering the whole software development life cycle, from analysis to deployment. As a consequence, UML includes hundreds of elements, although just a few of them are directly involved in the software design step and related to component structure and behavior specification. Thus, it is not easy (or even possible) to take only the required elements and diagrams and leave the rest of UML behind (for instance, all the elements related to Use Case modeling). Besides, the semantics associated to some UML elements should be completed, as they remain undefined in the UML specification (this lack of a unique definition is called "semantic variation point").

Option 2: define a UML *profile*, which will only take those elements needed for specifying component-based reactive system designs, and which will precisely define the variation points associated to them (if any). However, obtaining a reduced set of concepts and relationships by profiling a huge meta-model like UML 2.x is neither easy nor efficient. Actually, as stated in [3] by Bézivin "*it is much more difficult to work by restriction than by extension*". Besides, although the resulting meta-model would be simpler and more precise than UML, models built from this profile would be rambling (plenty of tags and stereotypes) and difficult to inspect and debug. In fact, the UML 2.x Superstructure [16] standard states the following about profiles:

> *"The profiles mechanism is not a first-class extension mechanism (i.e., it does not allow for modifying existing meta-models). Rather, the intention of profiles is to give a straightforward mechanism for adapting an existing meta-model with constructs that are specific to a particular domain, platform, or method [...] First-class extensibility is handled through MOF, where there are no restrictions on what you are allowed to do with a meta-model: you can add and remove meta-classes and relationships as you find necessary [...]"*

Option 3: define a DSL which only includes the required modeling elements, precisely defined with the desired meaning. However, the reactive systems domain is quite wide (actually it includes many very different application domains), and we are interested in modeling general-purpose component-based reactive systems rather than applications belonging to a very specific domain. In other words, we do not want concepts such as robot, camera or wireless-link to appear in our meta-model. As a consequence, considering V^{3}Studio a DSL and a general-purpose meta-model at a time could seem a bit artificial and confusing, unless component-based reactive systems are considered a “wide although specific” domain.

The benefits and the drawbacks of these three options were carefully analyzed and, finally, we decided to define the V^{3}Studio meta-model using a combined approach: we have selected those UML 2.x concepts needed to describe the structure and behavior of reactive system components but, instead of using a profiling technique, V^{3}Studio has been designed as a stand-alone meta-modeled, directly from MOF [13] (like most DSLs). This allowed us to define the precise meaning of the selected UML elements, and to slightly modify some of them to improve, for instance, model reuse thanks to the definition of three loosely coupled component views. In spite of the minor changes that will be described in the following section, the resulting V^{3}Studio meta-model remains very close to the original subset of UML it was extracted from and, in this sense, it can be considered UML-compliant. Actually, a model transformation from V^{3}Studio to UML 2.x (and vice versa) is straight forward.

3. Description of the V^{3}Studio meta-model

As stated in the previous section, the resulting V^{3}Studio meta-model was built to help us design component-based reactive systems. V^{3}Studio is aimed at modeling both component structure and behavior, as components provide more reusable, extensible, and evolvable designs [18].

Even at the risk of being repetitive, we want to make it clear again that UML would have been also a valid option for modeling the kind of applications we are interested in, but we decided not to use it because it contains many concepts we do not need, and that would have only polluted the resulting models. In fact, as it will be further explained in this section, V^{3}Studio takes and adapts many of the concepts already included in UML (in fact, V^{3}Studio is largely based on UML). V^{3}Studio contains only the essential concepts and relationships required to describe the kind of systems we are interested in. However, we think that V^{3}Studio is general enough to be used in other domains to design component-based applications.

In order to model component behavior, two alternatives were considered, namely: state-machines and activity diagrams, both of them available in UML 2.x. State-machines describe how components react to different stimulus coming, either from outside (e.g. from other component which require some service), or from the component itself (e.g. when an error condition is detected). Conversely, activity diagrams describe a data-flow oriented behavior, enumerating the sequence of activities performed by the component when it is in a certain state (e.g. idle, working, onError, etc.). Thus, state-machines and activity diagrams do not offer equivalent but complementary behavioral views.

Although UML 2.x allows designers to choose only one of these conventions to model component behavior, in most cases a combined approach is required to provide a complete behavioral description. This is why V^{3}Studio provides the facilities to model both, state-machines and activity diagrams, although using only a subset of the concepts included in the equivalent UML 2.x views. Besides, V^{3}Studio makes these two views compulsory, that is, a component is not complete (and thus, not valid) until (1) a state-machine defining its behavior is specified, and (2) for each state included in this state-machine, an activity diagram defining the behavior of the state is associated to it.

In order to enable model reuse, the three views the V^{3}Studio meta-model has been divided into (structural, coordination, and algorithmic), are not just highly cohesive, but also very loosely coupled. To achieve these two characteristics, inter-view relationships have been modeled using plain association links instead of composition ones, as it will be further explained in the following subsections. Such a meta-model partition exhibits several advantages, namely: (1) it eases the overall design process since designers only deal with a reduced set of concepts at a time; (2) the loosely coupling of the three views promotes model reuse, and (3) the inter-model consistency is assured, since all models are build from the same meta-model.

3.1. Component structure description

As stated before, the V^{3}Studio meta-model describes the static architecture of a system in terms of its components and the connections existing between them. V^{3}Studio components are modeled according to the UML 2.x [16] definition of component: “*a component is a self contained unit that encapsulates certain behavior. It specifies the services provided to its clients and those required from other components, defined in terms of its provided and required interfaces. Components are substitutable units that can be replaced, at design time or run-time, by a component that offers equivalent functionality (based on compatibility of its interfaces)*”. This definition is captured in Figure 1, which shows the structural part of the V^{3}Studio meta-model, while Figure 4 shows a snapshot of the graphical tool representing this view.

As shown in Figure 1, the architectural view (represented by the `SimpleComponent` class) is coupled with the coordination view (represented by the `StatMachine` class) by means of an association named `behavior`. Using a simple association instead of a composition relationship achieves a loosely coupling between these two views. This allows designers to reuse their V^{3}Studio state-machine models for defining the behavior of different components, which is one of the main objectives of the design of V^{3}Studio. The same strategy has been applied to define those concepts which can be reused in different contexts or places. For instance, as it can be seen in Figure 1, the definition of the system `Interfaces` also follows this strategy, so different ports (even belonging to different components) can reuse the same interfaces.

Figure 1 shows that components can be simple (concept represented by the `SimpleComp` class) or complex (`ComplexComp` class) and that all components, either simple or complex, contain a series of `Ports`. These ports group together the component `Interfaces`, which define the `Services` the component provides (i.e. the services it offers to the rest of the components of the system), and those it requires (i.e. those required by the component and that must be provided by other components). Simple components represent the basic architectural and behavioral units of the system, while complex ones encapsulate and coordinate other components (either simple or complex, up to any level). As it can be seen in Figure 1, the `ComplexComp` class uses the *Composite* design pattern [8] to contain other components and encapsulate them. A complex component, thus, establishes a boundary and limits the access to its inner components. It is worth noting that V^{3}Studio defines the system architecture, that is, the application as a whole, as a reference to a complex component. This makes it possible to reuse any previously defined architecture design (in the form of a `ComplexComp`).

V^{3}Studio assumes that only simple components need to define their behavior using a state-machine. As shown in Figure 1, only the `SimpleComp` class is related, by means of the already mentioned association `behavior`, to the behavioral description core class `StateMachine`. Complex component behavior, on the other hand, is considered a derived property, which can be obtained from the combined behavior of the internal components. Unlike UML, which states that any component can have behavior on its own, the V^{3}Studio meta-model restricts the possibility of adding new extra behavior to complex components. This design decision considerably simplifies the semantics associated to this concept of the meta-model and it is not truly a restriction, as it is possible to add a new simple component which performs the corresponding wanted behavior.

The other big aspect of architectural description is related to component communication. Component communication is achieved by linking "compatible" ports of two components by means of the `PortLink` class (see Figure 1). Since there are two kinds of components, which play different roles in the system, there are two ways in which

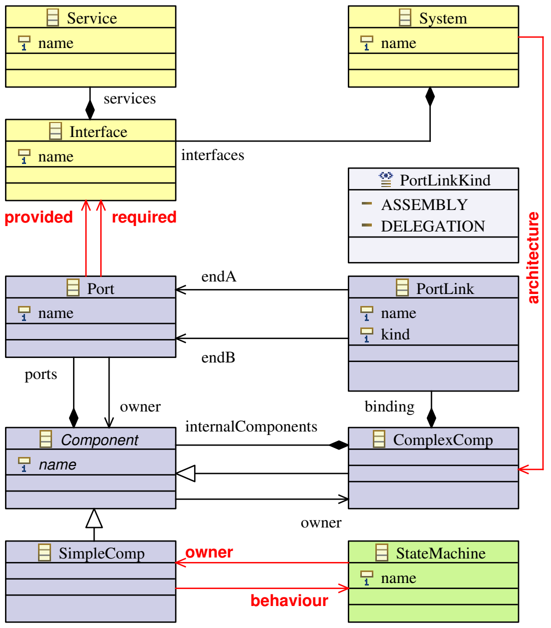

Figure 1. The V^{3}Studio structural view (component diagrams). The red association links define the relationships between the architectural and behavioral views and between the definition of components and interfaces. These associations make it possible to reuse previously defined models.

ports can be "compatible" and, thus, linked. V^{3}Studio takes them into account and defines an attribute that characterizes the `PortLink` being used:

DELEGATION `PortLink`. This kind of link connects a port defined in an inner component (whether simple or complex) to a compatible port defined in its container. In this case "compatible" means that these two ports can be linked if and only if they provide and require exactly the same interfaces. Thus, a complex component port is just an extension of the inner port, as the complex component will delegate every service requirement to it. Delegation links allows the creation of `ComplexComp` using a white-box view, as the developer can choose which inner component will fulfill part of the functionality offered by the complex component.

ASSEMBLY `PortLink`. This kind of link connects two ports defined in two components (whether simple or complex) that are created at the same level, that is, none of them contains the other. In this case, components are being assembled and "compatible" means that ports can be linked if and only if they are conjugated, that is, they must provide all the interfaces required by the other and vice versa. This kind of view enforces the use of components as black-boxes.

As a summary of the rules for connecting compatible ports, it can be stated that (1) components defined at the same level (contained in the same complex component) are always linked using `ASSEMBLY` port links, while (2) in order to connect a component to its container a `DELEGATION` port link is required. Thus, (3) simple components are always linked using `ASSEMBLY` port links, while (4) complex components support both types of connections (they can be internally linked by a `DELEGATION Portlink` but also assembled by an `ASSEMBLY Portlink`). Of course, these restrictions do not appear in the V^{3}Studio meta-model which is shown along this section, but it is necessary to add some OCL [14] constraints to verify them.

3.2. Component behavior description

As stated at the beginning of this section, in V^{3}Studio only simple components have their own behavior, which is modeled by means of a state-machine diagram in the coordination view. Figure 2 shows the coordination view of the V^{3}Studio meta-model together with the loosely coupled relationships with the structural view (represented by the class `SimpleComp`, red association link named `behavior`, see section 3.1), and the algorithmic view (represented by the abstract class `Activity`, red association links `fire`, `do`, `onEntry` and `onExit`, see section 3.3). In this case, a state-machine can reuse any predefined `Activity` model to complete the definition of their `States` and `Transitions`, stating which activity is going to be executed when a state is entered o exited, while staying at the state, or when a transition is successfully fired. Figures 5 and 6 (top figure) show snapshots of the graphical tool representing this view.

As shown in Figure 2, this part of the V^{3}Studio meta-model is a simplified version of its UML 2.x counterpart, and it follows the execution dynamics and semantics defined in UML 2.x. But, unlike UML 2.x, V^{3}Studio does not include either regions (to model macro-states, i.e., states that contain other states), or sub state-machines (to reuse a part of an already defined state-machine), or the always controversial Event. Besides these limitations, this view allows designers to model (1) the different states a component can go through their lifetime, and (2) the transitions that model the change of state. For every state and transition it is possible to select the algorithm to be executed from a set of already defined activities (see algorithmic view, section 3.3). Specifically, it is possible to:

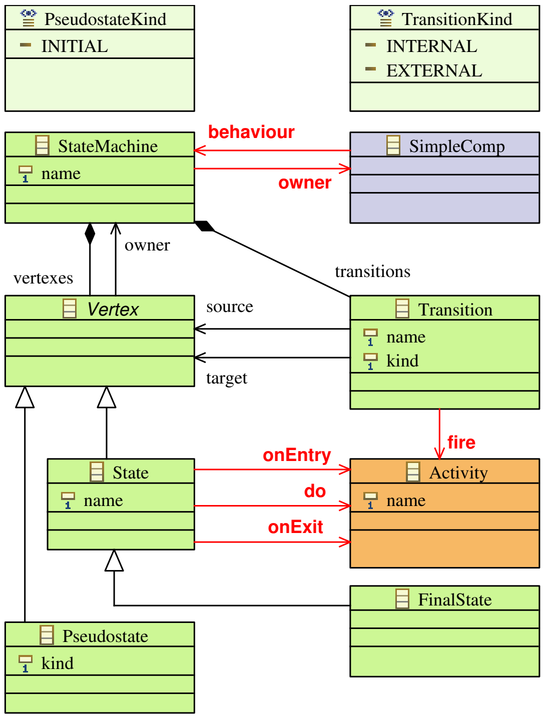

Figure 2. The V^{3}Studio coordination view (state-machine diagrams). Again, red association links show the relationships between the different views of the V^{3}Studio meta-model, specifically between the coordination and the architectural and algorithmic views.

- Model two types of `Vertex`: observable states of the state-machine (class `State`) and non-observable states (class `Pseudostate`). In the first case, it is possible to specify the activities that should be executed when the state is entered (`OnEntry` association), exited (`onExit` association) or when the state is active (`do` association). As can be seen, it is also possible to define a `FinalState` (which is an observable state) to mark the end of the state-machine.

- Model two kinds of `Transitions`: `INTERNAL` and `EXTERNAL`. A transition models the conditions that control the change of state in a state-machine. As UML 2.x does, V^{3}Studio allows designers to (1) specify whether the firing of the transition should really change state (`EXTERNAL` transition) or it is just a reaction, that only executes the fire activity and does not change the current state of the state-machine (`INTERNAL` transition); and (2) select one of the already defined activities to be executed as a consequence of this transition being fired (red association link named `fire`).

Finally, V^{3}Studio establishes that each and every state-machine must have an `INITIAL Pseudostate` that marks the state in which a component is initialized when created. This condition avoids the definition of what UML calles "ill-formed" state-machine diagrams (see [16], chapter 15). Of course, this condition has to be checked by defining additional OCL constraints.

3.3. Algorithmic description

Figure 3 shows the algorithmic view of the V^{3}Studio meta-model and its loosely coupled relationships with the coordination view (represented by classes `State` and `Transition`, see section 3.2). This final part of the meta-model is also a simplified version of the UML 2.x activity diagrams counterpart. Just like the coordination view is very similar to the UML 2.x state-machine diagram, the activity diagram shown in this section is a very simplified version of the UML 2.x activity diagram, as its UML counterpart has around a hundred classes available. Figures 5 and 6 (bottom figure) show a snapshot of the graphical tool representing this view.

As shown in Figure 3, the abstract class `ActivityVertex` constitutes the core class of this view, as the rest of the classes available to designers to model the different kind of activities inherit from this one. The following two classes directly inherit from `ActivityVertex`:

`Activity:` this abstract class models all the classes that represent the real execution of some code. As it can be seen in Figure 3, only the following two concrete classes represent an algorithm that models the behavior of a state or transition:

`SimpleActivity:` this class models a simple activity, that is, one that contains only executable code. Simple activities represent the description of a basic unit of behaviour. As next item points out, complex activities are just a medium for reusing and linking previously defined activities.

`ComplexActivity:` this class allows designers to group together and link activities (whether simple or complex) in order to create more complex activities for describing the behavior of a state or transition. Again, the *Composite* design pattern [8] is used to recursively contain all the `Activity` classes needed to describe a `ComplexActivity`. Besides, a complex activity contains `ActivityLinks` that allow designers to link `ActivityVertex` classes in order to establish the flow of execution inside a `ComplexActivity`.

`PseudoActivity:` this class models a kind of activities that do not represent the real execution of any code but that are necessary to correctly define the V^{3}Studio algorithmic view. Two different kind of pseudo-activities has been modeled: INITIAL, which points to the first activity that should be executed when executing a complex activity, and FINAL, which models the end of the flow of activities. V^{3}Studio establishes that every complex activity should have one `PseudoActivity` of each class to be correct. These restrictions have been added by means of two OCL constraints.

It is worth noting that, as shown in Figure 3, the addition of the `Activity` class prevents the use of a `PseudoActivity` as the associated behavior of a state or transition. This is an example of how a design decision can ease the definition of the semantics of the models. In this case, we could have removed the `Activity` class and make every activity class directly inherit from `ActivityVertex`. This decision would have eliminated one class from the meta-model but would also have forced us to add more additional constraints to check the rules that are naturally enforced by the actual V^{3}Studio meta-model. It can be generally said that the reduction of the number of classes in a meta-model forces the addition of OCL constraints to check model validity.

V^{3}Studio does not include any control-flow mechanism to allow designers modeling loops or bifurcations. We plan to add this mechanism in a future version of the meta-model, as it will be further explained in section 7.

4. The V^{3}Studio graphical modeling tools

As stated before, V^{3}Studio allows designers to model component-based software architectures using three complementary views. Such a meta-model partition exhibits several advantages, namely: (1) it eases the overall design process since designers only deal with a reduce set of concepts at a time; (2) the loosely coupling of the three views promotes model reuse and, last but not least, (3) the inter-model consistency is assured since all models are build from the same meta-model.

To keep all the benefits derived from the use of different views, we have implemented three graphical modeling tools (in fact four, but the interface definition tool is very simple), one for describing the system architecture (component view, see Figure 4), another for modeling state-machines (coordination view, see *Top* of Figures 5 and 6), and the

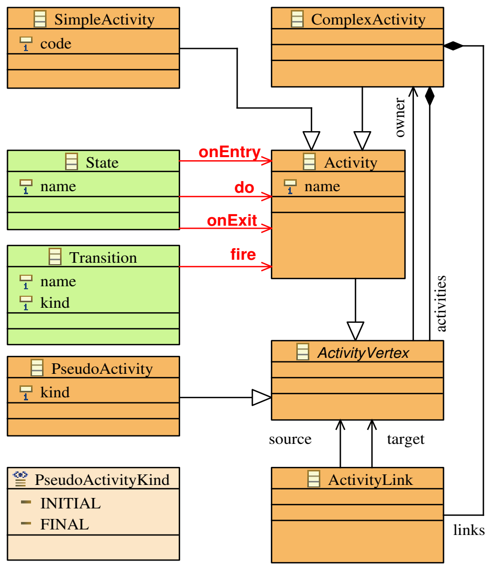

Figure 3. The V^{3}Studio algorithmic view (activity diagrams).

last one for depicting activity diagrams (algorithmic view, see *Bottom* of Figures 5 and 6) using the facilities provided by the Eclipse platform, namely the *Graphical Modeling Framework* (GMF) plug-in.

Each of these tools generates a model that includes only part of the concepts appearing in the complete V^{3}Studio meta-model. These models are fully reusable since they can be loaded from the other tools to complete other models. For instance, the activity diagram tool allows designers to build activities which then can be loaded by the state-machine tool to fill the `onEntry` property of some of its states. The resulting state-machine model can also be loaded by the component tool to fill the behavior of any of its simple components.

Some example diagrams, built using the implemented graphical modeling tools, will be shown in the following section, where a vision-guided robotic system case study is presented.

5. A case study on vision guided robotics

This section presents a simple but complete case study regarding the design of a vision guided robotic system and shows the definition of the system architecture, using components, and the definition of their behavior, using state-machines and activity diagrams. The purpose of the robotic system is to pick up objects that have a given characteristic (shape, color, etc). The vision system is in charge of identifying the object and generating the trajectory of the movement of the robot, which is finally executed and controlled by the Controller.

Figure 4 presents the component diagram and the interface design tools used to model the case study system architecture. The structure of this system, as shown in Figure 4, is defined as a complex component (as said in section 3.1, it is compulsory to define the architecture of the whole system as a complex component) comprising three internal simple components, namely: a *Computer_Vision_System*, a *Robot_Control_System* and a *Controller*, which plays the role of mediator between the other two components (it receives commands from the vision system and generates the appropriate movement commands).

Both, the computer vision and the robot control systems, are linked to their owner by means of a DELEGATION port link (discontinuous red line). These ports have a required interface used to request some functionality from the environment. More precisely, the computer vision system uses its port to send image request messages to the camera (that should be connected to the external port on the left), while the robot control system uses its port to send move commands to the robot (connected to the external port on the right).

The controller is connected to the other two components by means of ASSEMBLY port links (continuous black lines). It uses its left port to collect the visual inspection results obtained by the computer vision system. Then it decides whether the inspected object should be picked up by the robot or not. If the robot must pick up the object then it sends a move order using its right port.

The computer vision system behavior has been modeled as shown in Figure 5 (top). As expected, the result is a very simple state-machine and quite a complex activity diagram (see Figure 5 bottom), provided the data-flow nature of this component. Conversely, as shown in Figure 6, modeling the behavior of the robot control system required a more complex state-machine while the activity diagram associated to the most complex of its states (moveXYZ) is quite simple in this case.

6. Related work

The work presented in this paper draws from a number of related research works. Within the domain of software and system engineering, probably the most closely related technologies are those concerning *Architecture Description Languages* (ADLs) [10] and the modeling languages provided by the OMG, namely UML 2.x [16] (which was previously criticized in section 2) and SysML [15].

The *Systems Modeling Language* (SysML) was developed by the OMG to provide a modeling language for system engineering, which supports the specification, analy-

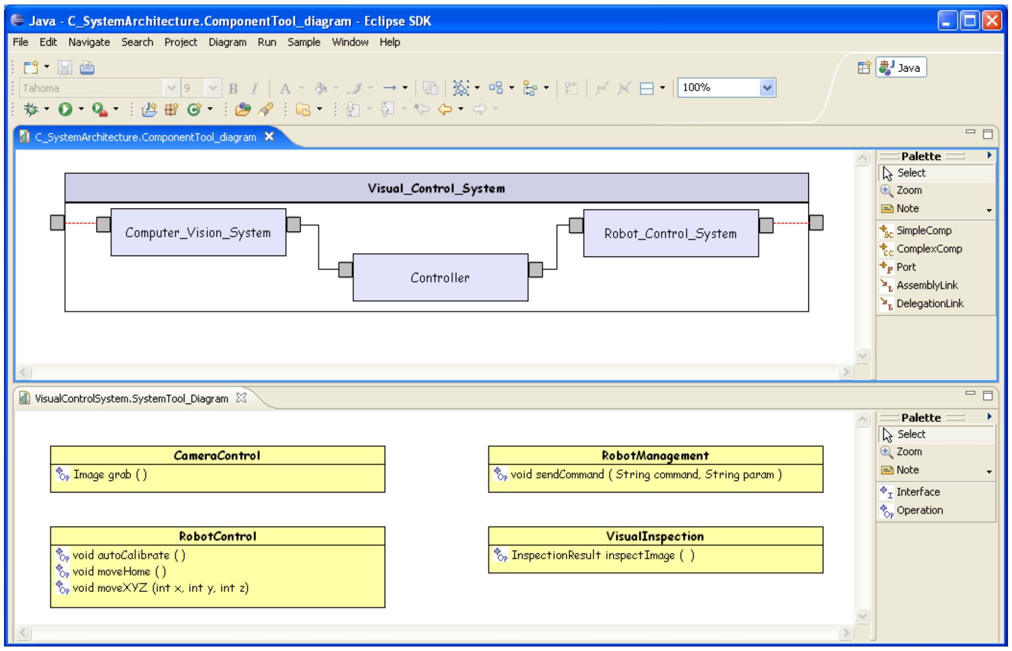
Java - C_SystemArchitecture.ComponentTool_diagram - Eclipse SDK
File Edit Navigate Search Project Diagram Run Sample Window Help
C_SystemArchitecture.ComponentTool_diagram
Visual_Control_System
Computer_Vision_System
Robot_Control_System
Controller
Palette
Select
Zoom
Note
SimpleComp
ComplexComp
Port
AssemblyLink
DelegationLink
VisualControlSystem.SystemTool_Diagram
CameraControl
Image grab ( )
RobotManagement
void sendCommand (String command, String param)
RobotControl
void autoCalibrate ( )
void moveHome ( )
void moveXYZ (int x, int y, int z)
VisualInspection
InspectionResult inspectImage ( )
Palette
Select
Zoom
Note
Interface
Operation

Figure 4. Models developed as part of the vision-guided robotic system case study. *Top*: **architecture view of the system,** *Bottom*: **interfaces and services definition tool.**

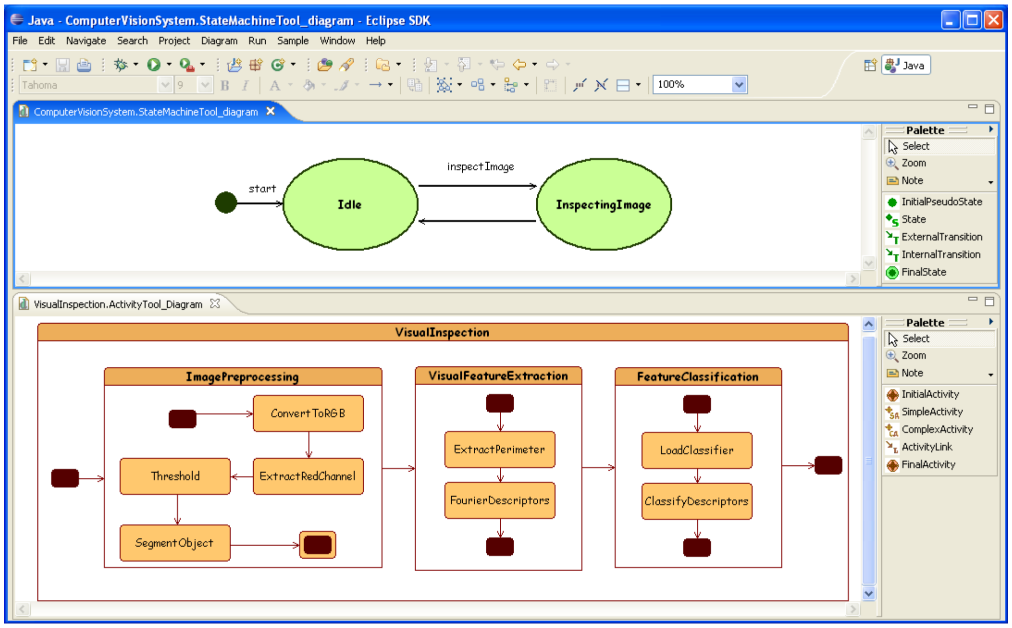

Figure 5. Models developed as part of the vision-guided robotic system case study. *Top*: **Computer_Vision_System coordination view,** *Bottom*: **Activity diagram associated to the InspectingImage state.**

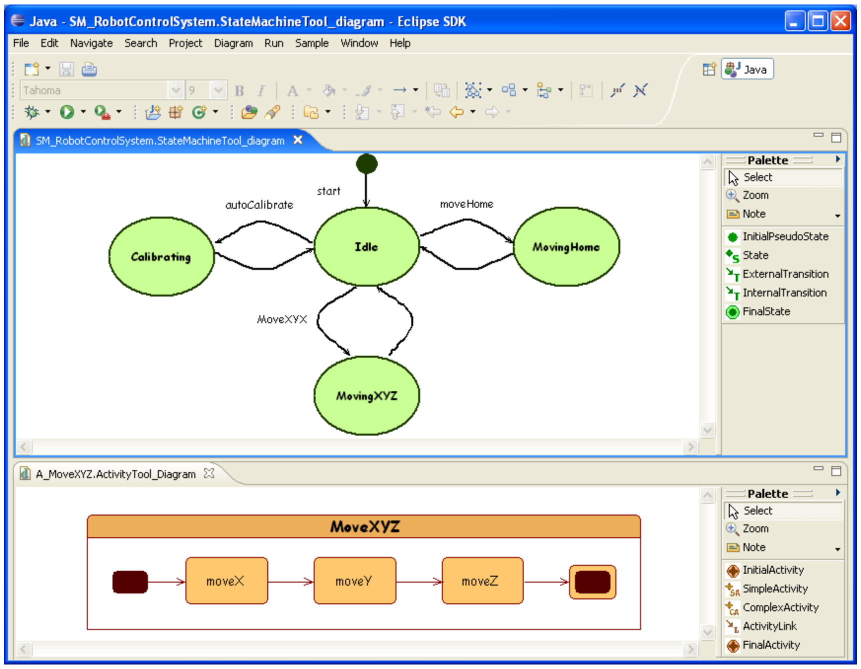

Figure 6. Models developed as part of the vision-guided robotic system case study. *Top*: **Robot_Control_System coordination view,** *Bottom*: **Activity diagram associated to the MovingXYZ state.**

sis, design, verification and validation of a broad range of systems and systems-of-system. SysML is defined as a UML *profile*, but it is actually an extension of a subset of UML. So, SysML is smaller, easier to use and learn than UML. In fact, SysML removes many of the software-centric constructs of UML, and thus the overall language is smaller (measured both in diagram types and total constructs). However, SysML is still difficult to understand and to use, and the limits between its different views are not clearly defined. Besides, the SysML `Block` concept, one of the core classes of this standard, cannot be (strictly) considered equivalent to the software concept "component".

Conversely, the software architecture design community focuses on building formal notations which enable the definition of the structure and behavior of software systems, and the non-software entities the system interfaces with. Recently, the *Architecture Analysis and Design Language* (AADL) [7] has been designed and standardized by the *Society of Automotive Engineers* (SAE), in collaboration with the *Software Engineering Institute* (SEI). AADL follows a MDE approach, and has a strong foundation on previous work regarding ADLs definition done by the SEI (e.g. ACME, Wright, or UniCon) [10]. However, most of these ADLs are focused on component interaction, because the separation between the data-flow and the control-flow is difficult to model.

The Cadena [4] platform also provides tools for building component-based software system, covering the whole application development life cycle. Cadena is also based on many of the concepts provided by traditional ADLs (actually, it provides its own ADL, called CALM) and relies on a middleware technology to achieve platform independence. However, Cadena has been specifically designed to build component-based software product lines, which is not the main objective of V^{3}Studio.

Finally, *openArchitectureWare* (oAW) provides a modular MDA/MDD generator framework. oAW supports arbitrary model parsing and provides a family of model checking and transformation languages, together with code generation facilities. However, oAW was still in an alpha state when we started developing V^{3}Studio, although we do not discard using it in a near future, as it provides a uniform environment for MDE.

7. Conclusions and future research

In this paper we have presented the V^{3}Studio component-based meta-model, which allows designers to model the structure and behavior of their systems, providing them with three highly cohesive and loosely coupled views. Although it was initially conceived to target the design of reactive systems, V^{3}Studio can be considered a general-purpose meta-model for describing a wide range of component-based applications. The V^{3}Studio meta-model allows designers (1) to design and validate component-based applications using customized tools, and (2) to reuse previously defined models in different contexts (for instance, a state-machine can be reused to describe the behavior of different components).

Three graphical modeling tools have been implemented on top of V^{3}Studio to support its three (in fact four) views. Models depicted using these tools can be easily loaded and reused to define new models. To probe the feasibility and the benefits of the proposed approach, a simplified vision guided robotic system has been presented.

Currently, the V^{3}Studio meta-model and the associated graphical modeling tools are still in an alpha state, although they are completely operative. In the future we plan to improve V^{3}Studio in the following directions:

1. Decouple component definitions from their implementations in order to allow designers to model completely reusable component definitions.
2. Include some mechanisms to enable the description of structural and behavioral variability.
3. Manage time constraints in the component behavior specification.
4. Use V^{3}Studio as an intermediate abstraction level for different DSLs created for the specific reactive system domains enumerated at the beginning of this paper (robotics, computer-vision, and WSAN applications). This intermediate level will play the role a MDA PIM meta-model, providing a common layer for each and every reactive system developed using those DSLs. As a PIM, V^{3}Studio will reduce the semantic distance between the very-high level models defined using the different DSLs and the final system implementation, and thus it will help reducing the complexity of the required model transformations.
5. Define a set of Model-to-Text transformations to automatically generate executable code from V^{3}Studio models. In this line, some results have already been achieved for the state-machine view [2].

Acknowledgement

This work has been partially funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02), and the PMPDI-UPCT-2006 program (Universidad Politécnica de Cartagena).

References

[1] A. Abouzahra et al. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the OOPSLA 2005 Conference*.

[2] D. Alonso et al. Automatic Ada code generation using a model-driven engineering approach. In volume 4498 of *LNCS*, pages 168–179, June 2007.

[3] J. Bézivin. On the unification power of models. *Journal of Software and Systems Modelling*, 4(2):171–188, May 2005.

[4] A. Childs et al. Calm and Cadena: Metamodeling for component-based product-line development. *IEEE Computer*, 39(2):42–50, 2006.

[5] I. Crnkovic and M. Larsson. *Building Reliable Component-Based Software Systems*. Artech House Publisher, 2002.

[6] S. Deelstra et al. Model driven architecture as approach to manage variability in software product families. In *Proceedings of the MDAFA 2003 Workshop*, pages 109–114.

[7] P. Feiler, D. Gluch, and J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, SEI (CMU), 2006.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[9] F. Losilla et al. Wireless sensor network application development: An architecture-centric mde approach. In volume 4758 of *LNCS*, pages 179–194, 2007.

[10] N. Medvidovic and R. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[11] T. Mens and P. van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[12] OMG. *Model Driven Architecture Guide Version v1.0.1, omg/2003-06-01*, June 2003.

[13] OMG. *Meta-Object Facility (MOF) Specification v2.0, ptc/04-10-15*, Oct. 2004.

[14] OMG. *Object Constraint Language (OCL) Specification v2.0, formal/06-05-01*, May 2006.

[15] OMG. *OMG Systems Modeling Language (OMG SysML) Specification v1.0, ptc/06-05-04*, May 2006.

[16] OMG. *Unified Modeling Language (UML) Superstructure Specification v2.1.1, formal/2007-02-05*, Feb. 2007.

[17] B. Selic. The pragmatics of model-driven development. *IEEE Trans. Software Eng.*, 20(5):19–25, 2003.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[19] ANSI-IEEE 1471-2000 Std. IEEE recommended practice for architectural description of software-intensive systems.

[20] A. van Deursen et al. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, 2000.

[21] C. Vicente-Chicote et al. Image processing application development: From rapid prototyping to SW/HW co-simulation and automated code generation. In volume 3522 of *LNCS*, pages 659–666, 2005.

[22] B. Álvarez et al. An architectural framework for modeling teleoperated service robots. *Robotica*, 24(4):411–418, 2006.