V^3STUDIO: A COMPONENT-BASED ARCHITECTURE DESCRIPTION META-MODEL
Extensions to Model Component Behaviour Variability

Cristina Vicente-Chicote, Diego Alonso
División de Sistemas e Ingeniería Electrónica, Universidad Politécnica de Cartagena, 30202 Cartagena, Spain

Franck Chauvel
IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

Keywords: Model-Driven Engineering, Component-Based Architecture Description Language, Variability Modelling, COTS Product Integration.

Abstract: This paper presents a Model-Driven Engineering approach to component-based architecture description, which provides designers with two variability modelling mechanisms, both of them regarding component behaviour. The first one deals with how components perform their activities (the algorithm they follow), and the second one deals with how these activities are implemented, for instance, using different Commercial Off-The-Shelf (**COTS**) products. To achieve this, the basic V^3Studio meta-model, which allows designers to model both the structure and behaviour of component-based software systems, is presented. V^3Studio takes many of its elements from the **UML 2.0** meta-model and offers three loosely coupled views of the system under development, namely: a structural view (component diagrams), a coordination view (state-machine diagrams), and a data-flow view (activity diagrams). The last two of them, concerning component behaviour, are then extended in this paper to incorporate the two variability mechanisms previously mentioned.

1 INTRODUCTION

Component-Based Software Development (**CBSD**) has proven to obtain highly reusable, extensible and evolvable designs (Szyperski, 2002). 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**) approach (Selic, 2003) offers an effective solution for bridging the gap between architecture design and **CBSD**. This approach revolves around the definition of models and model transformations. Models represent part of the function, structure and/or behaviour of a system (Deelstra et al., 2003), and they are defined in terms of formal meta-models. Each meta-model includes the set of concepts needed to describe the system at a certain level of abstraction together with the relationships existing between these concepts. Model transformations, commonly described as meta-model mappings, enable to automatically refine abstract models into more concrete ones. This process concludes when the final application code is automatically obtained from the lowest level models.

As stated in (Abouzahra et al., 2005), nowadays there are two main trends in **MDE**. The first one promotes the use of standard modelling languages like **UML 2.0** (OMG, 2005) or **SysML** (OMG, 2006), while the second one advocates the benefits of Domain-Specific Languages (**DSLs**). The inclusion of variability mechanisms into meta-models to enable, for instance, the description of software product lines, is currently one of the most active topics in **MDE**. However, neither **UML 2.0** nor **SysML** natively support variability modelling, although some **UML** profiles have been developed to support this aspect (Ziadi et al., 2003). Nevertheless, as stated in (Bézivin, 2005), commonly it is more difficult to work by restriction (i.e. profiling **UML**) than by extension (i.e. defining a new **DSL**). This is why we decided to define V^3Studio as a new meta-model (not from scratch but incorporating many of the elements already defined in **UML**), instead of defining *yet another UML profile*. Thus, V^3Studio is a

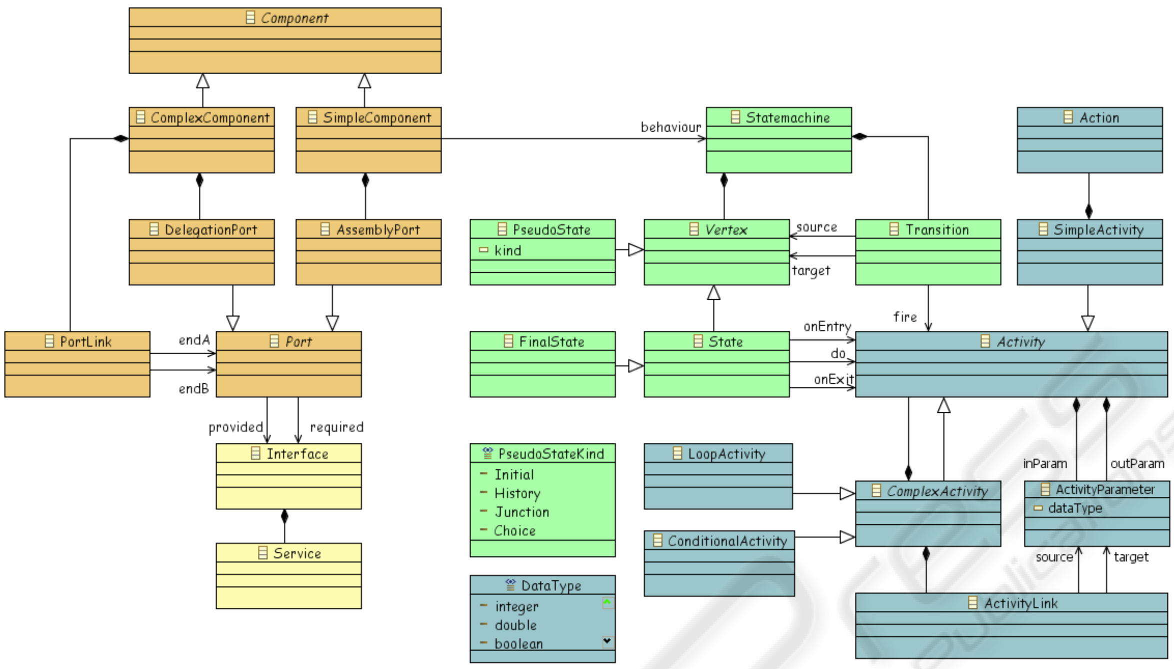

Figure 1: The basic V^{3}Studio meta-model.

general-purpose component-based meta-model which can deal with behavioural variability, and which can not be considered a **DSL**, since it does not include any domain-specific concept.

The rest of the paper is organized as follows. Firstly, the component-based V^{3}Studio meta-model is presented in section 2. Secondly, the behavioural variability extensions introduced in this meta-model are described in section 3. Finally, section 4 outlines the conclusions and futures research lines.

2 THE *V^{3}Studio* META-MODEL

The variability mechanisms presented in this paper are built as an extension to a previously designed meta-model called V^{3}Studio (see figure 1). This meta-model, which takes many of its elements from **UML 2.0** (OMG, 2005), enables a precise description of component-based software architectures, including both structural and behavioural features. Actually, the meta-model encompasses three different views, namely: a *structural* view, a *coordination* view, and a *data-flow* view.

As we suggest a component-based approach, the structural view defines the system in terms of components (either simple or complex) and the interconnections between them. Components provide (require) services to (from) other components by means of interfaces, grouped into ports. Component behaviour, defined as the set of reactions derived from its relationship with other components, is described in terms of a state-machine. Finally, the activities performed in each of the states defined in the component state-machine are specified using an activity diagram. These activities are defined in terms of some internal actions or making use of the services provided by other components.

Next, the three views the V^{3}Studio meta-model has been divided into, are briefly described:

The structural view. System software architecture can be defined using either a top-down or a bottom-up approach. In the first case, the software architecture is defined by grouping the main system functionalities into high-level components. Thus, the system is initially considered a complex component built from very high level components, assembled using connectors to link their ports. These high level components are successively decomposed into simpler ones, until no more refinements are possible. Conversely, a bottom-up approach starts from simple components, selected to fulfil part of the system functionality. These components are linked together to build more complex functional units, and this process is incrementally repeated until the final system functionality is achieved.

The data-flow view. Once the system architecture has been structurally defined, the behaviour of its components must be described. Each service provided by a component is defined in terms of activities. Activities represent functional units, such as functions or blocks in imperative programming languages. Activities can have input and output parameters, each one associated to a certain data type. Activities can be linked together to define the execution path, and they can also be grouped into complex structures to ease their reuse.

The coordination view. Finally, the designer has to specify how components collaborate with each other. This collaboration requires specifying both (1) how components react to messages sent by other components depending on their current state, and (2) the (optional) communication protocol for sending messages or calling services.

The following section presents the modifications introduced in the V^{3}Studio meta-model to enable designers to include both data-flow and implementation variability into their component models.

3 VARIABILITY MECHANISMS

This section presents the data-flow and implementation variability mechanisms introduced in the V^{3}Studio meta-model. For the sake of clarity, two figures containing the modified and newly added elements in the affected views of V^{3}Studio are included in each of the following sections.

3.1 Data-Flow Variability

As explained in the introduction, designers often need to add some variability to the behaviour of components. It would be desirable to specify that a service can be performed in various ways, using different algorithms. To achieve these objectives, the state-machine part of the V^{3}Studio meta-model is extended with some new concepts, as shown in figure 2. As it can be observed, the new meta-model enforces the separation between a *StatemachineDefinition* and the real *Statemachine*. It also adds two parallel concepts: *VariableActivity* and *ActivityBinding*. The first one is aimed at defining which activities can vary in those components that share the same component definition. The second one specifies the particular activity executed by the component. All these concepts are highlighted in figure 2.

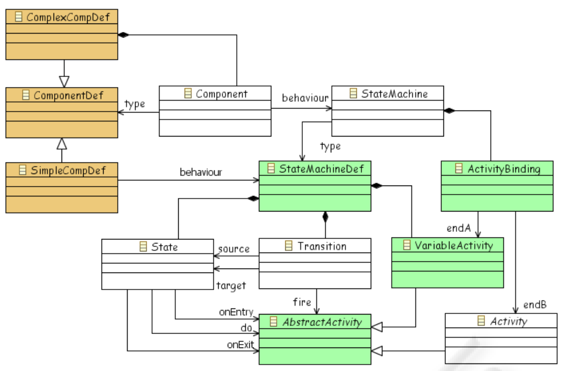

Figure 2: Extension to model data-flow variability.

With this mechanism, variants of the same behaviour can be designed by modelling some activities as parameters. As activities are used to describe the reactions of components, the behaviour described by the state-machine is preserved but these reactions can be customised (Harel, 1987). Modelling components this way allows designers to have various components that, considered as black boxes, are all exactly the same (so they can be replaced), but considered as white boxes they are all different, since they use different algorithms.

3.2 Implementation Variability

Currently, the marketplace offers a wide variety of **COTS** products that provide many of the functionalities typically required when developing new applications. However, generally each of these products uses its own defined data types, function calling conventions and error handling mechanisms, making it difficult to use them together. In order to allow designers to incorporate the functionality provided by all these **COTS** products, the V^{3}Studio meta-model includes a wrapping mechanism aimed at encapsulating heterogeneous library functions to build homogeneous and thus inter-connectable implementation units.

This variability mechanism has been added to the data-flow view of the V^{3}Studio meta-model (see figure 3). Although the multiplicity of the relationships is not shown in the diagrams because of a limitation of the tool, in this case all simple activities contain only one action. Each action represents a functional block which might have been imported from one of the existing **COTS** libraries. To allow designers to join activities containing actions from different **COTS**, the *PinParamLink* connection between activity parameters and action pins must (1) perform the corresponding data type conversion, (2) pass the data to the ac-

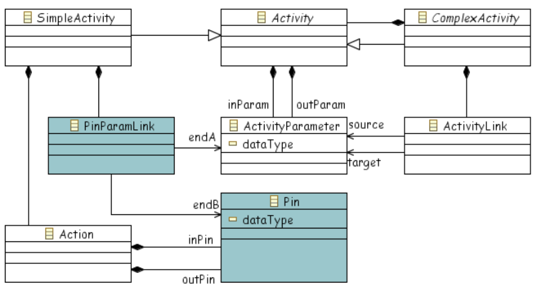

Figure 3: Extension to model implementation variability.

tion using the correct calling convention, and (3) correctly interpret and handle all possible errors during the execution.

Allowing only one action into each simple activity could seem quite restrictive and inefficient (i.e. linking activities containing actions from the same **COTS** will require unnecessary data conversions). However, designers are free to use actions as complex as they wish into *Activities* (i.e. complex algorithms implemented using several functions from a single **COTS**). Regarding efficiency, activity models built using V^{3}Studio can be easily optimized using a simple model transformation which avoids unnecessary data type conversions when possible.

4 CONCLUSIONS

In this paper we propose an extension to the V^{3}Studio meta-model, which allows designers to model component behaviour variability at early design stages. The major contribution of this proposal is to enable the inclusion of variability aspects both into design and implementation models. Several variants of the same component can be easily built, while most of the existing approaches focus on component substitutability to support product line development.

This approach allows designers to build complete component definitions (in which both structural and behavioural characteristics are fixed) and, optionally, they can define the activities that may change between different components of the same type, by means of parameterised state-machines. The extended V^{3}Studio meta-model also considers implementation variability, that is, how to deal with different implementations of the same functionality provided, for instance, by different **COTS** products available in the marketplace. In this sense, the meta-model reinforces black-box component reuse.

The work presented in this paper has considerably enriched the previous versions of the V^{3}Studio meta-model. However, there are still some open issues which must be addressed. Currently we are working in two main directions, namely: (1) defining structural variability mechanisms, similar to those defined in this paper regarding component behaviour, and (2) including some operational semantics in the meta-model using *KerMeta* (Muller et al., 2005).

ACKNOWLEDGEMENTS

This research has been partially funded by the Spanish CICYT project MEDWSA (TIN2006-15175-C05-02) and the PMPDI-UPCT-2006 program (Technical University of Cartagena, Spain).

REFERENCES

Abouzahra, A., Bézivin, J., Didonet, M., and Jouault, F. (2005). A practical approach to bridging domain specific languages with uml profiles. In *Proceedings of the OOPSLA 2005*.

Bézivin, J. (2005). On the unification power of models. *Journal of SoSyM*, 4(2):171–188.

Deelstra, S., Sinnema, M., van Gurp, J., and Bosch, J. (2003). Model driven architecture as approach to manage variability in software product families. In *MDAFA 2003*, pages 109–114.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Proceedings of UML MoDELs 2005*, volume 3713 of *LNCS*. Springer.

OMG (2005). Unified modeling language 2.0: Superstructure specification. Official specification formal/05-07-04, Object Management Group, Needham, MA, USA.

OMG (2006). Systems modeling language (sysml^{TM}) specification. Final Adopted Specification ptc/06-05-04, Object Management Group, Needham, MA, USA.

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

Szyperski, C. (2002). *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley.

Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2003). Towards a UML profile for software product lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, volume 3014 of *Lecture Notes in Computer Science*, pages 129–139. Springer.