Modeling and Executing Component-based Applications in C-Forge

Francisca Rosique, Diego Alonso, Juan Pastor and Francisco Ortiz
Division of Systems and Electronic Engineering (DSIE), Universidad Politécnica de Cartagena,
Campus Muralla del Mar, E-30202, Cartagena, Spain

Keywords: Software Engineering, Component-based Software Development, Model-driven Software Development, Framework, Software Design Patterns, Real-time.

Abstract: This paper describes a model-driven toolchain for developing component-based applications that enables users to use the same models that define their application to execute them. In this vein, models always remain true to the final application, unlike other approaches where a model tranformation generates a skeleton of the final application after the first steps of the development process. These kind of approaches normally end up with models that represent a different application than the one present in the code.

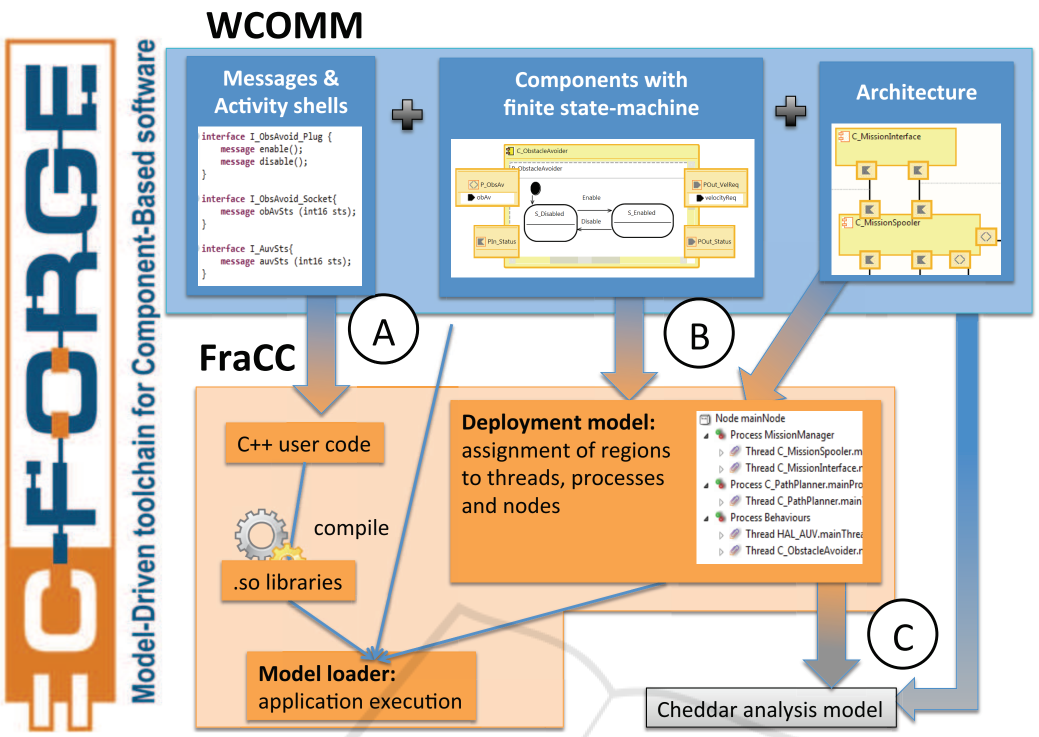
1 INTRODUCTION AND MOTIVATION

The Component Based Software Engineering (CBSE) (Crnkovic, 2011) paradigm promotes the encapsulation of proven solutions into reusable building blocks, considering components as something beyond a compiled module or library. CBSE approaches should consider concerns beyond purely structure/behavior modeling, such as component execution, configuration, deployment, analysis, etc. The Model-Driven Software Development (MDSD) (Schmidt, 2006) paradigm can help designing development processes for CBSE applications that take into account all these concerns, allowing users to concentrate on domain concepts rather than in code or configuration parameters. Model transformations (Malavolta, 2010) complement the approach by providing the means to automatically generate other representations, either in the form of models or text (source code).

Generally, model-driven processes for developing component-based (CB) applications only use models on the first steps of the project. Afterwards, a set of model transformations generate the code skeletons where the user fills-in the code and then the application is compiled. The problem with this approach is that when the user must make a change, he/she forgets about the original models of the application and now the user makes all the modifications directly on the code. But not all kind of changes should be made at the code level. For instance, if the application needs a new component, this component should be added at the model level and then the code should be regenerated in order to keep it true to the model. Otherwise, the model and the code end representing different applications. And this is an important problem not only in CBSE but in many other domains.

One of the possible solutions to this problem is to really have a model-driven process, from the beginning to the end, where all the involved software artefacts are either models themselves, or are aware of the presence of models and use them consistently and coherently. The goal of such a process would be to be able to use the same CB models that describe the application to execute it. In any case, code is necessary, but we should try to use models as long as possible. This paper describes how we have implemented these ideas in the context of the C-Forge toolchain. C-Forge is a development process for CBSE supported by a MDSD toolchain developed in the Eclipse IDE that is composed of three parts, shown in Fig. 1:

- WCOMM tool: provides two graphical editors and a textual one for modeling the structure and behavior of CB applications from a platform independent point of view.
- FraCC tool: provides a model loader, a deployment metamodel and a compiled C++ framework. The framework provides the run-time support required for executing WCOMM models with different deployment configurations (distribution of components to nodes and processes, and assignment of computational load to threads). FraCC also includes a model loader that instantiates the


WCOMM
C-FORGE
Model-Driven toolchain for Component-Based software
Messages & Activity shells
Components with finite state-machine
Architecture
FraCC
A
B
C
C++ user code
compile
.so libraries
Model loader: application execution
Deployment model: assignment of regions to threads, processes and nodes
Cheddar analysis model

```
interface I_ObsAvoid_Plug {
    message enable();
    message disable();
}

interface I_ObsAvoid_Socket{
    message obAvSts (int16 sts);
}

interface I_AuvSts{
    message auvSts (int16 sts);
}
```

```
Node mainNode
  Process MissionManager
    Thread C_MissionSpooler.m
    Thread C_MissionInterface.r
  Process C_PathPlanner.mainPro
    Thread C_PathPlanner.main
  Process Behaviours
    Thread HAL_AUV.mainThrea
    Thread C_ObstacleAvoider.r
```

Figure 1: The C-Forge development process comprises two tools (WCOMM and FraCC) and three model-transformations.

application from the WCOMM model and a deployment model (described in the following item). We said FraCC "executes" WCOMM models because, unlike other approaches, we do not generate implementation code. FraCC is already compiled and provides a model loader that parses WCOMM models and instantiates FraCC classes accordingly. It is not an interpreter, since C++ code is executed at runtime, and thus, the performance of the final application is not affected. More details about FraCC's design are provided in (Alonso, 2014).

- Three model transformations that generate the rest of the software artifacts required in different steps of the C-Forge development process. Model transformation (A) generates the *C++ code skeletons* where the user must add the application-specific code. The user code is then compiled into a dynamic library and loaded by the model loader. Model transformation (B) generates a *deployment model*, which describes how the application is executed by FraCC. Model transformation (C) exploits the information contained in the WCOMM and deployment models to generate an analysis file for performing a schedulability analysis with Cheddar, so that the user can check whether the final application will meet its real-time requirements.

A detailed description of C-Forge and its application in the context of an autonomous underwater vehicle, performed in collaboration with the Heriot-Watt University, is described in (Ortiz, 2014). The aim of C-Forge is to provide (i) full support for the development of component applications, from modeling to execution, (ii) strict control over the concurrency properties of the application, (iii) mechanisms that foster model reuse in all levels.

The rest of the paper is organized as follows. Section 2 describes some related works in the field of components, model-driven technologies and concurrency/real-time specification available in the literature. Section 3 contains the main part of the paper, where we describe with more detail the C-Forge toolchain. Finally, Section 4 outlines the conclusions and some future works.

2 RELATED WORK

Given the number of bibliographic references that review the state of the art of CBSD in the literature, the aim of this section is not to provide a new review, but to relate the work described in this article with the currently available component models. In order to this, we use the classification framework described in (Crnkovic, 2011), which is used by the authors to classify a large number of component models. Such classification framework proposes three main characteristics for studying component models, namely *Lifecycle*, *Construction* and *Extra-Functional Properties*. Table 1 classifies the elements that comprise C-Forge according to the aforementioned taxonomy.

C-Forge shares many similarities with approaches like ProCom (Vulgarakis, 2009), CHESS (Cicchetti, 2012), and RT-CCM (Lopez, 2013), since they are all component-based approaches supported by model-driven technologies, and they all provide certain degree of control over concurrency and real-time properties. The main differences with the aforementioned approaches are:

ProCom defines two component layers: the upper and the lower layer. The former allows developers to define large-grained active components, while the latter consists of basic functional passive components, which are interconnected inside subsystem. In C-Forge, all components are active, but this does not mean that they require their own thread.

CHESS considers four views: Requirements, Components, Deployment, and Analysis. Component behavior can be defined with state-machines, other standard UML diagrams, as well as with the *Action Language for Foundational UML* (ALF, http://www.omg.org/spec/ALF/). The Deployment view models the target execution platform, and software to hardware components allocations. CHESS provides generators to Ada/C/C++/Java source code. C-Forge focus on a single way to model component behavior and manage concurrency, which makes it easier to generate and compose C++ code only.

RT-CCM considers components as black-boxes, where the user models the concurrency needs (mainly threads and shared resources), so that the component implementation can provide them. The resulting models can be analyzed with the MAST (http://mast.unican.es) analysis tool. Compared to them, C-Forge considers that components are white-boxes.

3 MODELING AND EXECUTING COMPONENT-BASED APPLICATIONS IN C-FORGE

As shown in Fig. 1, the development process proposed by C-Forge is the following. The first step comprises the definition of the application datatypes, messages and activities by using WCOMM's textual editor. Afterwards, the user can select to either generate the code skeletons for the activities or continue modeling the application (i.e. defining simple components, modeling the application architecture and generating the deployment model). Once both parallel steps have been completed, the user can execute the application by passing the WCOMM and deployment models to the FraCC model loader. This process is iterative, in the sense that the user can add, modify and test components as needed. The following subsections describe the main highlights of WCOMM and FraCC.

3.1 Modeling Component-based Applications with WCOMM

An excerpt of the most important classes of the WCOMM metamodel is shown in Fig. 2. WCOMM comprises three complementary and loosely coupled sets of meta-classes that group together the elements related to modeling (i) the common application elements, namely data-types, messages and activities, (ii) simple components and their behavior, and (iii) complex components. As can be seen in the figure, there are several meta-classes devoted to model "definitions" (blue meta-classes, like ComponentDefinition, PortDefinition, ActivityDefinition, etc.) and their "instances" (orange meta-classes, like Component, Port, Activity, etc.) in order to foster model reuse. WCOMM defines simple and complex component definitions. Simple components are the basic building units, while complex components are built by assembling component *instances*, either from simple or complex components.

A simple component is an entity that encapsulates its internal state and that comprises both structural and behavioral parts. The structural part is defined by the component ports and the message *instances* flowing through them. Messages can carry data, and are sent between components following the asynchronous no-reply communication scheme, which makes possible to support any communication scheme and assures a low coupling between components. Component reactive behavior is defined by means of a finite-state machine (FSM), similar to those defined in UML 2.

FSMs only model the circumstances under which the component's activities are executed. Thus, the complexity of the application is mainly embedded in the C++ code executed by an activity. FSMs also define hierarchical and orthogonal regions, which allow users to model independent parts of the whole component behavior. FraCC then uses this potential concurrency modeling to organize the threading code and

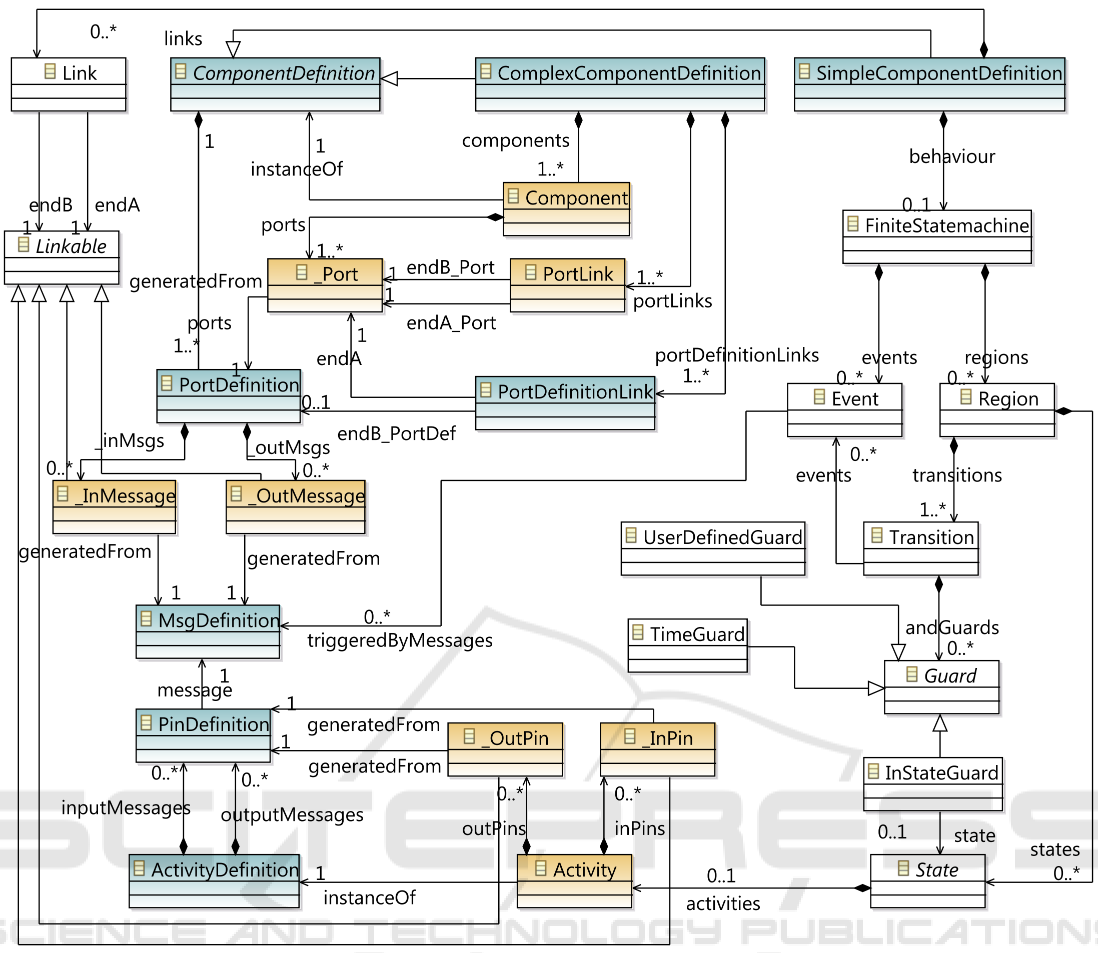


Figure 2: Excerpt of the WCOMM metamodel. Blue meta-classes model “definitions” while orange ones model “instances”.

execute the regions sequentially or concurrently, depending on the platform features (e.g., CPU characteristics) and the user desires. Transitions are triggered by events, which can be produced by message reception or by the results of an internal computation. States can contain one activity *instance*, which is a proxy of the concrete algorithm that will be executed when the state becomes the region’s active state.

3.2 Executing Component-based Applications with FraCC

In order to execute WCOMM components we developed FraCC, which comprises a deployment metamodel (shown in Fig. 3), a C++ framework and a model loader that instantiates the framework according to a WCOMM and deployment models, and a set of model transformations that generate a default deployment model and the skeletons of the C++ classes where the user must add the application specific code. The FraCC meta-model comprises three sets of classes, ordered vertically in the figure: (i) hardware definition, since we want to extend the approach to multicore and distributed systems; (ii) concurrency definition, defining the number of processes and threads where the regions that define the component behavior will be executed; and (iii) allocation of processes and threads to the hardware (nodes and CPU cores) that will execute them.

From the information contained in the WCOMM and deployment models, a model transformation generates an analysis file for the Cheddar schedulability tool, which checks whether the final application will meet its timing requirements or not. Appropriate changes can be made in the applications models as needed based on the outcomes of analysis. A detailed description and examples of the use of the deployment model and schedulability analysis are provided in (Ortiz, 2014). The deployment model provides C-Forge with flexibility since it can test components with various deployment configurations, e.g. same ar-

Table 1: Features of and MinFr component models according to the classification framework described in (Crnkovic, 2011).

A) LIFECYCLE DIMENSION	
	C-Forge: WCOMM+FraCC
Modeling	ADL-like language (datatypes, component structure, component behavior) plus deployment and concurrency view
Implementation	C++ framework based on pattern languages
Packaging	EMF files (stored in XML format) and zip files including code in binary format
Deployment	Set at compilation and at run-time

B) CONSTRUCTION DIMENSION: INTERFACE SPECIFICATION	
	C-Forge: WCOMM+FraCC
Interface Type	Port based (message passing)
Provided/Required interfaces	No, in/out messages
Interface Language	IDL-like and C++ header files.
Interface Levels (syntactic, semantic, behavior)	Syntactic only. However, behavioral (protocols) and semantic (pre, post conditions and invariants) levels are implicitly defined in the FSMs

C) CONSTRUCTION DIMENSION: BINDING AND INTERACTION	
	C-Forge: WCOMM+FraCC
Connectors are architectural elements	Not yet
Hierarchical composition of components	Delegation (internal components are not accessible from outside the composite)
Interaction Styles	Sender/receiver, message passing
Communication type	Asynchronous without response. Synchronous interactions should be defined in the FSM

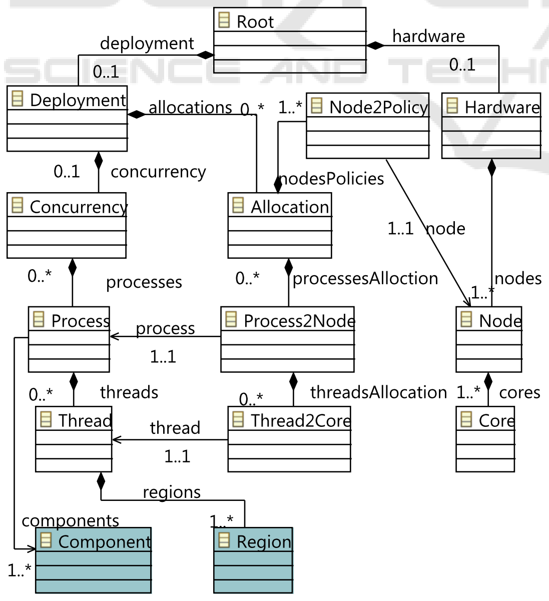


Figure 3: Deployment metamodel for FraCC. Related WCOMM classes are highlighted in orange.

chitecture; different platforms.

There are several changes the user can make in order to obtain an schedulable application. On the deployment model, he/she can reallocate regions to other threads or to new ones, grouping together regions with similar execution periods. On the WCOMM model side, he/she can (i) develop new algorithms, so that activities' execution times are lowered, (ii) relax the timing requirements imposed over the activities if possible, so that the load on the CPUs can be lowered; or (iii) change the FSM describing a component behavior, so that there are less regions to be allocated to threads.

4 CONCLUSIONS

This paper has described a model-driven toolchain for developing component-based application, entitled C-Forge, which main characteristics are that (i) the same models that are used to model the application are also used to execute it, and (ii) the execution framework provides the user with full control over its concurrency characteristics. Therefore, we take advantage of all the benefits provided by model-driven technologies. C-Forge also includes a separation of concerns, since the application architecture is separated from the way it will be executed and from the concrete code that will be executed by each component.

Reuse is also a key feature in C-Forge. We want to stress that reuse should be taken into account at all levels, and not only at the modeling one. In WCOMM, reuse is achieved by separating “definitions” from their “instances” in the meta-model, while FraCC design’s enable the reuse of previously compiled (i.e., in binary format) activities.

Finally, we also want to highlight again that, from our point of view, it is really important to consider that model-driven should permeate all the steps and tools involved in the development process. “Model-driven” is not just a matter of the tool (e.g. Eclipse), but rather a responsibility of the whole development process and all involved tools. In our case, though the FraCC framework has not been developed by using Eclipse-based model-driven facilities, but rather C++, it is nevertheless aware of the presence of models.

ACKNOWLEDGEMENTS

This work has been partially supported by the Spanish CICYT Project Visel-TR (ref. TIN2012-39279)and the Spanish Ministry of Economy and Competitiveness under the project cDrone (ref. TIN2013-45920-R).

REFERENCES

I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. IEEE Trans. Software Eng., 37(5):593–615.2011.

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

I. Malavolta, H. Muccini, P. Pelliccione, and D. Tamburri. Providing architectural languages and tools interoperability through model transformation technologies. IEEE Transactions on Software Engineering, 36:119 – 140, 2010.

F. Ortiz, C. Insaurralde, D. Alonso, F. Sánchez, and Y. Petillot. Model-driven analysis and design for software development of autonomous underwater vehicles. Robotica, pages 1–20, April 2014.

D. Alonso, F. Sánchez, J. Pastor, and B. Álvarez. Embedded and Real Time System Development: A Software Engineering Perspective, chapter A flexible framework for Component based Application with Real-Time Requirements and its Supporting Execution Framework, pages 3–22. Springer-Verlag, 2014.

A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the procom real-time component model. In Proc. of the 35th Euromicro Conference on Software Engineering and Advanced Applications, pages 478–485.IEEE, 2009.

A. Cicchetti, F. Ciccozzi, S. Mazzini, S. Puri, M. Panunzio, A. Zovi, and T. Vardanega.CHESS: a model-driven engineering tool environment for aiding the development of complex industrial systems. In Proc. of the 27th IEEE/ACM International Conference on Automated Software Engineering, pages 362–365. ACM Press, 2012.

Patricia López-Martínez, Laura Barros, and José Drake. Design of component-based real-time applications. Journal of Systems and Software, 86(2):449–467, 2013.