An Architecture Description Language for Verification in Component-based Software

Ahcene Bouzoualegh, Dominique Marcadet, Frédéric Boulanger and Christophe Jacquet
SUPELEC, Department of Computer Science, France
<first_name>.<last_name>@supelec.fr

Abstract

In the context of component-based design, we propose ADLV[1], an architecture description language based on IDL3, which allows the specification of properties that should hold on the system. The joint description of both the structure of the application and the properties it should satisfy allows us to derive the properties that should be formally checked on the control component of the system. We focus here on the ADLV language and tool and on code generation for the CCM platform from ADLV descriptions. Code generation must preserve the semantics of special components that are in charge of interfacing the control and the processing parts of the application.

1. Introduction

An advantage of formal methods, in particular specification and programming languages with formal semantics, is to allow for automatic validation techniques which significantly reduce the validation phase of embedded systems. This is a key point in the success of synchronous languages such as Esterel [2].

In the context of component-based software engineering, Architecture Description Languages (ADL) are used to define an application as a set of interconnected components. The difficulty to apply model checking techniques with existing ADL is that model checkers work better with events or boolean values while the application designer desires to express the properties to be verified on the signals of the application, which often have more complex data types. The purpose of this paper is to describe a new ADL that integrates the classical component-based application description with properties to be verified by model checking. Using this language, called ADLV, the designer can express the properties to check using application inputs and outputs, whatever their data type, and independently of the model checking tool.

Section 2 is a brief overview of software architecture and CCM. Model checking is introduced in Section 3. Section 4 introduces the proposed ADL for verification and the associated tool. Before concluding, we present a case study in Section 5.

2. Overview of Software Architecture

Software architecture languages enable the precise definition of the overall system structure [6 and 10] as a set of interconnected components. There are several definitions of the term component, the most common being a software module with well defined interactions points called ports [5 and 8]. The architecture shows the intended correspondence between the system requirements and the elements of the constructed system. An architect must first define the contracts of the components required by the application, and then through what interfaces two components interact.

The developers of components construct their implementations from their definitions. Then, the integrators create the relationships between the components identified in the architecture and the implementations produced by the developers. Finally, the deployment of a component-based application corresponds to the instantiation of the components, followed by the initial configuration and the interconnection of these instances. In order to model the interactions between components, software architectural elements called connectors are introduced, therefore a software system is defined in terms of components and connectors [12].

In the Common Object Request Broker Architecture (CORBA) specification [9], "component" is a basic meta-type. To ease and improve the quality of the application production process, the OMG has defined the CORBA Component Model (CCM), which is an industrial model for distributed business components,

[1] This work has been performed in the context of the Usine Logicielle project (www.usine-logicielle.org) and is partially financed by the System@tic Paris-Région Competitiveness Cluster (www.systematic-paris-region.org).

in the context of heterogeneous programming languages [11].

This component model uses ports to represent connection points. These ports, as well as the component types, are defined with the Interface Definition Language (IDL), an implementation neutral language. There are two categories of ports: ports used for communication with defined CORBA interfaces and ports used to exchange events. An extension of IDL allows for dataflow ports.

3. Model Checking for Software Systems

Model checking is a systematic way for checking whether all behaviours of a system model fulfil their specifications [7]. Formal property verification consists in proving properties by combining properties formally defined or already proved.

Embedded software systems are often critical and thus require a high level of reliability and quality. This leads to lengthy and costly test phases. To ensure the reliability of such complex systems, verification methodologies become necessary in the process; one tries to construct a formal proof that shows that all executions of the program satisfy the desired properties. To perform verification, we need a modeling language to describe software architecture, a specification language for the formulation of properties to be checked, and a checking algorithm [4]. Many algorithms are based on graph exploration where the nodes are the states of the system and the edges are labeled by events that trigger the changes of state. Some recent ADLs can represent a system's structure and behavior together with its dynamic changes and evolutions [13].

4. Architecture Description Language for Verification (ADLV)

4.1. Overview

We propose ADLV, which tries to integrate both the scopes of ADLs and of model checking. It is designed as an IDL extension. In our methodology, we impose that an application has only one control component, and may have several processing or internal components. Processing components communicate through data flows and are activated and interconnected under the supervision of the control component which consumes and produces only pure events. Processing and control components are called external components because their behavior is specified using other tools. Internal components are small components whose behavior is specified in ADLV, and which are used to produce events from values and to create dynamic connections. The benefit of this explicit separation between processing and control is that it makes the control task explicit and verifiable.

The ADLV tool that we developed is used both to produce the application targeted at an OMG's Lightweight CCM (LwCCM) [3] implementation and to translate the application properties into control observers. The goal is to rely only on the description of the application structure and on the specifications of the internal components to transform the global properties to be proved into properties expressed in a form recognized by the checking tool used for the control specification.

4.2. Principles of Property Verification

The ADLV language allows the designer to specify properties that must be satisfied by the system. These properties must be expressed in linear temporal logic. Only the subset of *canonical safety formulae* is considered [1]. Such a formula specifies that some past temporal logic expression must *always* (or *never*) be true, and thus allows the designer to express the *desired* behaviour of the system. To ease the designer's task, safety properties may include conditions on dataflow values: this allows him/her to use a vocabulary 1) that he/she is familiar with and 2) that is well-suited for expressing general system properties.

We have proposed a method [14] that relies on the ADLV description of the application, mainly the specifications of the internal components, to transform conditions on dataflow values into temporal logic formulae involving only controller events. We are thus able to build a formula, called *intermediate form*, containing only events. Next, we translate this intermediate form into an observer in the target language of the controller. Finally, we can use the language-specific formal verification tools to *prove* that the controller, and thus the application, satisfies (or does not satisfy) the safety properties. If the properties are not satisfied, these tools can provide a *counterexample*.

4.3. Introduction to the ADLV Language

The `control`, `processing`, `internal` and `tool` specific new keywords have been added to the IDL syntax. They are used to distinguish the different component kinds and to specify the tools used to create the implementation code of the external components. The ADLV description of components follows this grammar:

```
control component <identifier> {
    tool <identifier>;
    <control_component_body>
};
processing component <identifier> {
    tool <identifier>;
    <processing_component_body>
};
internal component <identifier> {
    <internal_component_body>
};
```

The other syntactic aspects of external component bodies conform to the standard IDL syntax (`consumes`, `publishes`, `sink` and `source`). The bodies of internal components may specify which event to produce when some boolean expression, referring to dataflow values, becomes true; it may also define a dataflow value to store upon receipt of an event. An application is a kind of component with its own ports. It is defined by its component instances and by the connectors, which may be modified upon reception of an event, between component ports. Finally, extensions to the IDL language have been made to allow for expressing the properties to be verified.

4.4. Architecture of the ADLV Tool

The ADLV tool (see figure 1) is a set of Eclipse plugins. An abstract model of our language has been made and promoted as an eCore model to benefit from the Eclipse Modelling Framework services. A parser

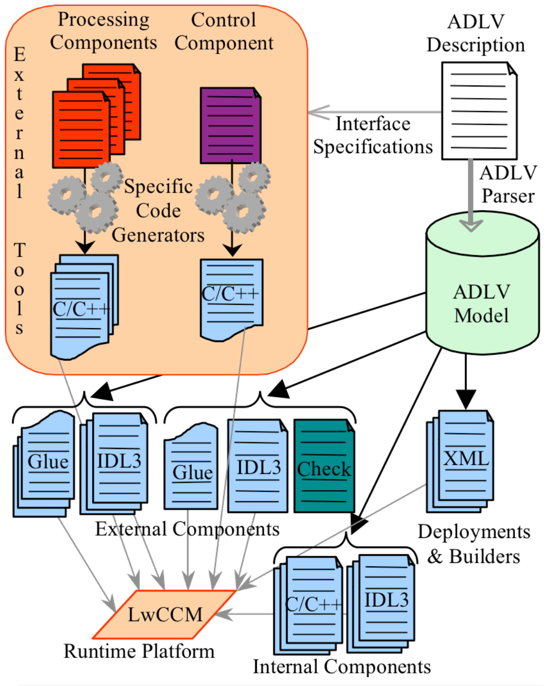

Figure 1. ADLV Workflow

for the ADLV textual syntax is used to populate an ADLV model; such a model could also be obtained using an UML2 profile and a model transformation.

The ADLV model is used by two tools: one is responsible for the generation of the observers in the controller language, the other is used for the generation of all the files needed for the chosen LwCCM implementation. These include the project build files, the standard CCM IDL files, the C++ implementations of the internal components, the C++ glue between the code generated by the tools used to design the control and processing components and the code expected by the containers of the LwCCM implementation, and finally the deployment XML files.

5. A Case Study in ADLV

Our case study (see figure 2) is a classical example of a car cruise control system; its main purpose is to maintain speed at a given value selected by the driver, which involves an automatic control system. The application has been described in ADLV and a simplified car simulator with a graphical interface has been created to demonstrate the generated application.

The cruise control system receives two events (start or stop the regulation) and three data flows (positions of the brakes and the accelerator pedal, current speed). The last one is used by three components: the duplication of this input value is automatically generated by the ADLV tool in the projection to LwCCM phase. The cruise regulator provides a command to the injection system that is either directly the one generated by the accelerator pedal, or the one calculated by an automatic control component to maintain an exact speed value when regulation is active.

The component in charge of the regulation, an external processing component in ADLV terminology, has been designed with Simulink while the other external component, the controller, is implemented in Esterel.

There are three internal components: two of them produce events from data flows (Brakes Check and Speed Check), the third is needed to provide the processing component with the target speed as a dataflow value using the current speed as input and an event from the controller as an order to memorize it. Finally, there is a dynamic connector driven by the controller to select the right output.

The designer of this system can write application properties to be verified such as: 1) regulation is deactivated when the driver depresses the brakes pedal, 2) the regulator is placed in standby mode when the driver depresses the accelerator, until the accelerator is released (provided that regulation was initially active), 3) the activation of regulation is prohibited if the speed is not in an allowed range, etc. These properties are

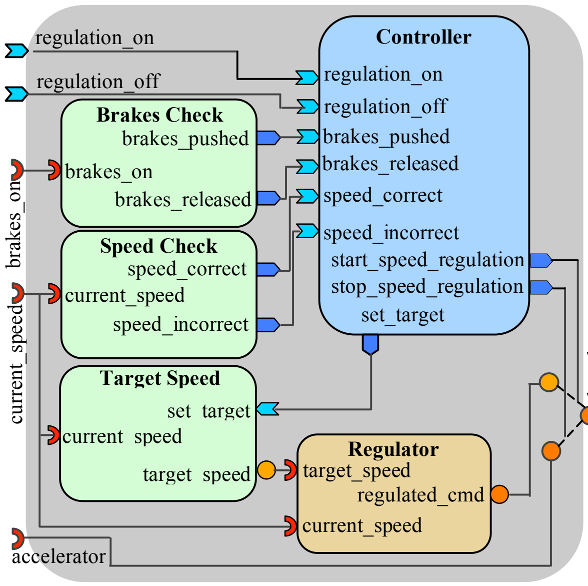

Figure 2. Architecture of the cruise controller

converted into Esterel observers and checked against the controller implementation, which has been realized independently of the whole system architecture (only its interface is imposed).

6. Conclusion

In this paper, we have presented our efforts to design an Application Description Language that can be used both for deploying and for verifying the application.

One of the goals of ADLV is to use the structure of the application to transform global properties on the application into properties on the control that, in turn, can be transformed into observers. These observers are recognized by the checking tools associated with the language used to specify the control.

To make this approach successful, we have limited the design choices of the developers: an application is made of only one control component which uses only pure events, of several processing components which consume and produce data flows, and of internal components which compute events from data flows and control the application in response to the events produced by the controller. As a counterpart to these restrictions, we provide a tool for generating all the needed files to build and deploy the application on a LwCCM execution environment, including the glue needed to integrate the code generated by the tools used for the external components.

7. References

[1] E.Y. Chang, Z. Manna and A. Pnueli, “Characterization of Temporal Property Classes”, *Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, Lecture Notes In Computer Science, Vol. 623, pp. 474–486. Springer-Verlag.

[2] G. Berry and G. Gonthier, “The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation”, *Science of Computer Programming*, 19(2):87--152, 1992.

[3] OMG: “Lightweight CORBA Component Model Revised Submission”, *Object Management Group, Inc.* May 2003, realtime/03-05-05

[4] S. Merz, “Model Checking: A Tutorial Overview”, *Proceedings of the 4th Summer School on Modeling and Verification of Parallel Processes,* volume 2067 of Lectures Notes in Computer Science, 2001, pp. 3-38. Springer-Verlag.

[5] C. Szyperski, “Component Software”, Addison-Wesley, 2nd edition, 2002.

[6] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Toung, and G. Zelesnik, “Abstraction for Software Architecture and Tools to Support Them”, *IEEE Transactions on Software Engineering*, April 1995.

[7] E. M. Clarke, J. Wing and Al, “Formal Methods: State of the Art and Futures Directions”, *ACM Computing Surveys*, 1999, pp. 626-643.

[8] I. Crnkovic, M. Larsson, “Building Reliable Component-Based Software Systems”, Artech House, 2000.

[9] OMG, Common Object Request Broker Architecture (CORBA/IIOP), formal/2008-01-08.

[10] IEEE Architecture Working Group, “IEEE Recommended Practice for Architectural Description of Software-intensive Systems*”*, Report IEEE Std 1471-2000

[11] OMG, “CORBA Component Model”, formal/2006-04-01.

[12] N. Medvidovic and R. N. Taylor, “A Classification and Comparison Framework for Software Architecture Description Languages”, *IEEE Transactions on Software Engineering*, Vol 26, no1, Jan. 2000, pp. 70-93.

[13] R. Mateescu. “Model Checking for Software Architectures”, *EWSA 2004*, 2004, pp. 219-224.

[14] C. Jacquet, F. Boulanger, D. Marcadet, “From Data to Events: Checking Properties on the Control of a System”, *in the Sixth ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2008)*, June 2008. Accepted for publication, to appear.