Taming Multi-Paradigm Integration in a Software Architecture Description Language

Daniel Balasubramanian, Tihamer Levendovszky, Abhishek Dubey, and Gábor Karsai

Institute for Software Integrated Systems
Department of Electrical Engineering and Computer Science
Vanderbilt University, Nashville, TN 37235, USA
{daniel,tihamer,dabhishe,gabor}@isis.vanderbilt.edu
http://www.isis.vanderbilt.edu

Abstract. Software architecture description languages offer a convenient way of describing the high-level structure of a software system. Such descriptions facilitate rapid prototyping, code generation and automated analysis. One of the big challenges facing the software community is the design of architecture description languages that are general enough to describe a wide-range of systems, yet detailed enough to capture domain-specific properties and provide a high level of tool automation. This paper presents the multi-paradigm challenges we faced and solutions we built when creating a domain-specific modeling language for software architectures of distributed real-time systems.

1 Introduction

Software architecture description languages offer a convenient way of describing the high-level structure of a software system. An architecture combines the individual pieces of a system, such as software components and communication networks, into an integrated view. Combining what are normally considered "separate" pieces of the system into such an integrated view allows the relationships between the different elements to be explicitly represented.

Software architecture descriptions serve several purposes. Apart from providing documentation and a high-level view of a software system's design, they can serve as the basis for automated code generation. Automated tools can generate skeleton code for the software components based on their interfaces and communication links to other components. This ability to translate a high-level description into lower-level implementation artifacts makes architecture description languages ideal for rapidly prototyping applications. Automated analysis can use an architecture description at design-time to answer questions related to security, schedulability and resource utilization.

One of the big challenges facing the software community is the design of an architecture description language that is both general enough to apply to a wide range of systems, but at the same time expressive enough to describe problems

specific to individual systems and provide a high amount of automated tool support. For instance, AUTOSAR [2], a standard for defining software architectures in the automotive domain, is heavily tailored to concepts and standards found in the vehicle design domain. On the other hand, languages like AADL [8] can be too general to model software systems that contain concepts outside of its specification, such as custom security concepts.

One alternative to using standardized architecture languages is to create a custom language. This approach can work especially well when implemented with domain-specific modeling languages (DSMLs), which provide an intuitive syntax and ensure that only the relevant architectural concepts are captured. Metamodeling environments also facilitate rapid language development iterations. However, there are several multi-paradigm modeling challenges involved with this approach: architectures contain elements at different levels of abstraction, multiple formalisms and cross-cutting concerns such as security. A viable architecture description language must provide solutions to these challenges.

This paper presents our solutions to a series of multi-paradigm challenges we faced while building a domain-specific language for describing the software architecture of distributed embedded systems. The first challenge is combining multiple formalisms (textual code and block diagrams) to describe component interfaces. The second challenge is to transform high-level scheduling properties of individual elements into a combined temporal schedule. The third challenge is integrating different types of analyses into the modeling language. Even though our approach is tailored to our architecture language, the solutions are applicable across other architecture languages which face similar challenges on a similar level of abstraction, and thus want to provide a similar level of tool automation.

The rest of this paper is organized as follows. Section 2 provides an overview of the modeling language. Section 3 describes how we integrated a textual code formalism with graphical block diagrams. Section 4 describes how we transform high-level scheduling properties of individual elements into a low-level temporal schedule. Section 5 briefly illustrates the opportunities for design-time analysis. Section 6 presents related work, and we conclude in Section 7.

2 Overview

DREMS, Distributed Real-time Embedded Managed Systems [6], is a software infrastructure for designing, implementing, configuring, deploying and managing distributed real-time embedded systems. It consists of two major subsystems: (1) a design-time toolsuite for modeling, analysis, synthesis, implementation, debugging, testing and maintenance of application software built from reusable components, and (2) a run-time software platform for deploying, managing and operating application software on a network of computing nodes. DREMS is intended for platforms that provide a managed network of computers running distributed applications; in other words, a cluster of networked nodes.

The design-time toolsuite is underpinned by a DSML (an overview is presented in [6]) with tools and generators to automate the tedious parts of software

development and provide an analysis framework. The run-time software platform reduces the complexity and increases the reliability of software applications by providing reusable technological building blocks in the form of an operating system, middleware and application management services.

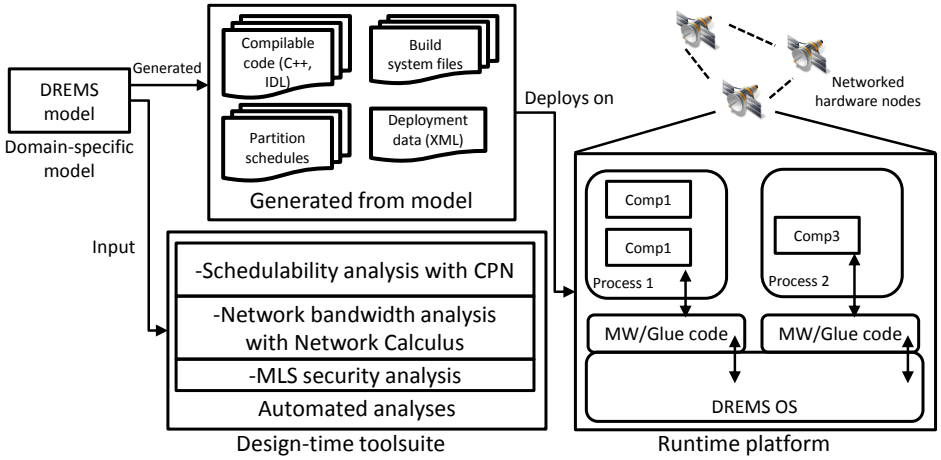

Fig. 1. The design-time toolsuite (left) and the run-time platform (right). The modeling language is used as the basis for both analysis and code generation.

Figure 1 shows a high-level overview of the development process. A model captures several aspects of the system in addition to structure. Models of the software components and interfaces are used to generate compilable skeleton code. Scheduling attributes of processes are used to generate OS schedules. The mapping of software to hardware, along with component interaction descriptions, generates deployment configuration files. Additionally, there exist design-time analysis tools; Section 5 contains more details.

The run-time system, shown on the right-hand side of Figure 1, includes a custom operating system (OS) and middleware (MW). Applications consist of some number of components, hosted inside processes, that communicate through the middleware. Each node in the system is expected to contain the full run-time infrastructure.

3 Integrating textual code with graphical block diagrams

The first integration challenge we describe is integrating textual code inside the graphical modeling language. The use of a textual language was motivated by the fact that the underlying platform uses a component-based software engineering (CBSE) methodology [10]. In CBSE, applications are built from reusable, communicating software components. For our underlying platform, these components are specified using a language called the Interface Definition Language

(IDL), an OMG standardized language to describe component interfaces. From an IDL specification of a component, code generators produce code stubs that are combined with user-written business logic and compiled to produce the actual component.

Thus, inside our DSML, we need abstractions for describing components, which include a suitable representation for IDL. The two main requirements for using IDL inside models were (1) other modeling elements should be able to refer to its properties and attributes, and (2) IDL developed independently from the model should be straightforward to use inside the model. Figure 2 shows the desired workflow.

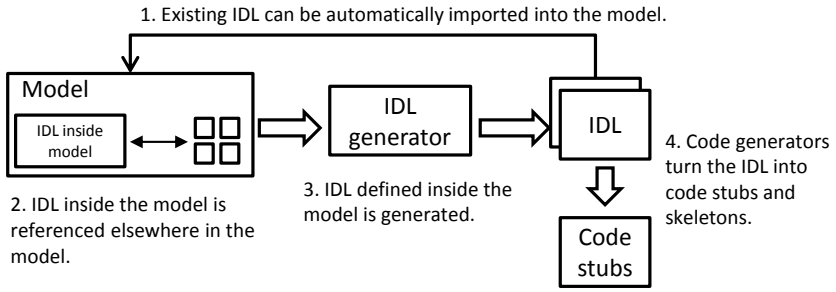

Fig. 2. Workflow for using IDL in the modeling language.

We had two main choices for how to represent IDL inside our models: a graphical representation or a textual representation. The drawback of using a graphical notation to represent IDL is that it is cumbersome for the user. Essentially, the user builds a graphical representation of the abstract syntax tree (AST) of their IDL. This is especially tedious for designers who already have some familiarity with IDL. The advantage of this approach is that it allows other modeling elements to easily refer to attributes and properties of the IDL.

On the other hand, the advantage of a textual notation is that it is compact, which makes it simple for users to write. It is also easy to import from existing IDL definitions. The drawback of a textual notation is that other elements in the modeling language cannot easily refer to its content. Also, the modeling language itself cannot be used to enforce that syntactically correct IDL is written by the user.

Ultimately, we decided on a combination of a graphical and textual representation. Figure 3 shows the portion of the metamodel (as a UML class diagram) defining the six types of IDL elements represented graphically inside the model; note that they are all subclasses of the abstract class *DataType* and inherit the *Definition* attribute. These are the graphical elements that appear in the modeling language.

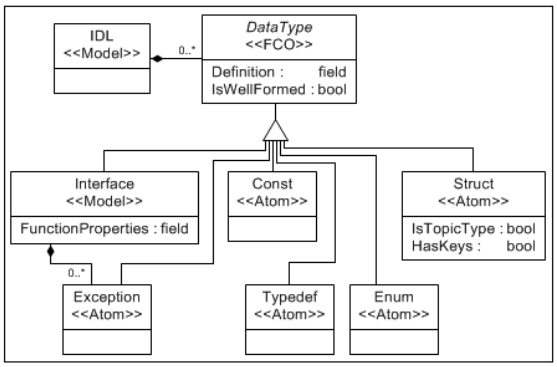

Fig. 3. The types of IDL elements represented graphically inside models.

The integration of the textual IDL language was accomplished by (1) generating an IDL parser using the ANTLR parser generator [12] and (2) creating an add-on to the modeling language using our modeling environment's extension API that uses the generated parser. The overall process is shown in Figure 5. The *Definition* attribute of each graphical IDL element contains its IDL code and is edited using a special code editing window that is provided by the add-on (see Figure 4). The code editor invokes the IDL parser and reports any syntax errors to the user, as shown in Figure 4.

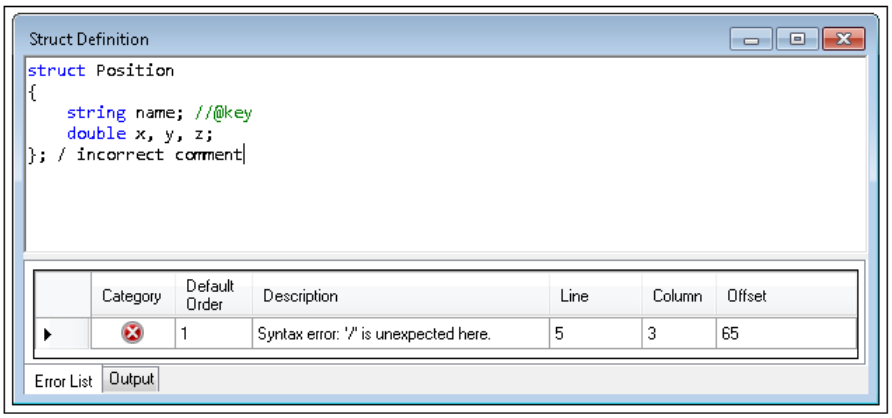

Fig. 4. Screenshot of the IDL code editing window, which was integrated into the modeling language. Errors reported by the parser are at the bottom. In this example, the IDL code is invalid because the comment on the last line begins with a single '/'.

If the IDL code is syntactically correct, then the integrated parser automatically sets attributes of the corresponding graphical IDL element in the model.

The key to the approach is the add-on that is able to (1) parse the textual language, and (2) based on the results of the parser, set multiple attributes on the graphical modeling elements. This is important because it enables attributes defined in the textual language to be integrated into the modeling language.

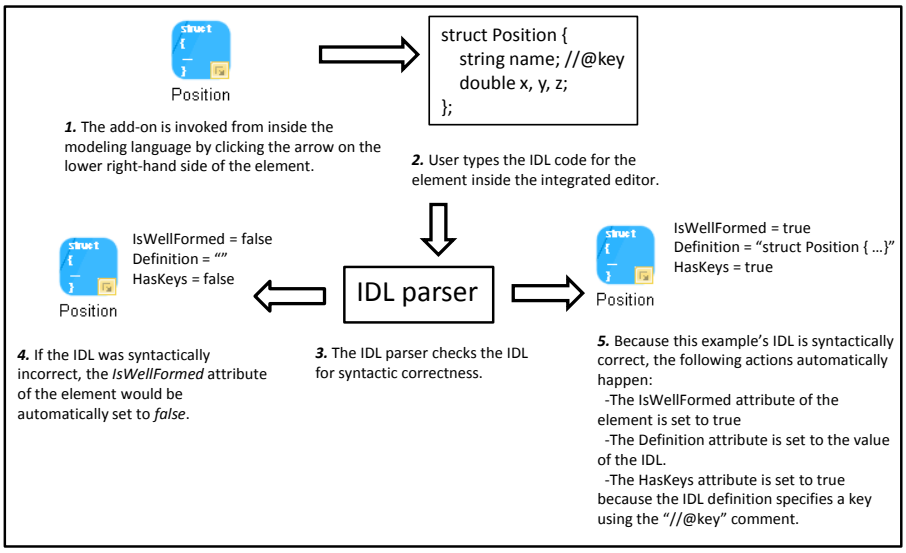

Fig. 5. Overview of how the textual IDL parser is integrated into the modeling language. Multiple attributes are set based on the value of the IDL in Step 5.

4 Schedule generation

The next challenge we describe deals with transforming high-level scheduling properties of individual elements into a combined, low-level schedule for the target platform. The motivation for this is that the target DREMS platform uses its own scheduler, namely, a temporal partition scheduler [1] to schedule processes. A partition is a logical group of executing processes in which all processes in the same partition are periodically given exclusive access to the CPU. Exclusive access means that no process from another partition is given access to the CPU during this time. Each partition can contain any number of processes and is defined by two attributes: a *duration* and a *period*. For instance, a partition with a duration of 4ms and a period of 10ms will run for 4ms every 10ms, and during these 4ms the processes it contains will have exclusive access to the CPU.

In order for the target platform to schedule the temporal partitions at runtime so that the period and duration constraints of each are satisfied, it must be given a valid partition schedule. This partition schedule is a periodically repeating

set of time slices called *hyperperiods*. The partition schedule specifies when to start each partition relative to the start of the hyperperiod. At the end of the hyperperiod, the same schedule is repeated again.

The multi-paradigm challenge with this is how to transform the process-partition assignment graph as well as the high-level scheduling attributes (a period and duration) of individual temporal partitions into a low-level partition schedule that can be used at runtime by the operating system. This schedule should be a part of the architecture description language because users must know at design-time whether a satisfying partition schedule exists and if so, what that schedule is. However, even for seemingly simple combinations of individual partitions, calculating a schedule by hand can be a non-trivial task. Thus the modeling language needs to hold enough information as well as provide a facility to calculate this schedule automatically and present it to users.

Figure 6 shows our solution using a small example. On the left side of Figure 6 are two processes (*P1* and *P2*), defined inside the software part of the modeling language. These two processes are assigned to two different partitions (*P1* is assigned to *T1* and *P2* is assigned to *T2*) in the software deployment portion of the language. The *schedule calculator* is the solution that provides a bridge between the individual temporal partitions and an integrated partition schedule. It is implemented as an add-on to the modeling language using our modeling environment's extension API and provides a bi-directional interface both to and from the modeling language.

When the schedule calculator is invoked, it queries the model and collects the individual temporal partitions. Next, it formulates the scheduling problem as a constraint satisfaction problem [13]. The constraint satisfaction problem is then given to an off-the-shelf solver (the Z3 SMT solver [3]). If the solver cannot find a solution, then it indicates a conflict with the temporal partitions and the schedule calculator then informs the user so that the temporal partitions can be modified.

If a solution to the constraint satisfaction problem is found, then the scheduling calculator must translate this solution into a partition schedule and insert it into the model. Creating the partition schedule from the scheduling calculator is done using the modeling environment's API which provides programmatic access for setting attributes.

5 Design-time analysis

This section describes three automated analyses that we built for the modeling language: security of communications, software component schedulability analysis and network quality of service (QoS) analysis. Due to space constraints, we only briefly describe each.

The operating system provides security to applications through spatial isolation (processes run in separate address spaces) and temporal isolation (temporal partitions guarantee that processes get a guaranteed portion of processor time). However, these two mechanisms alone cannot guarantee information flow iso-

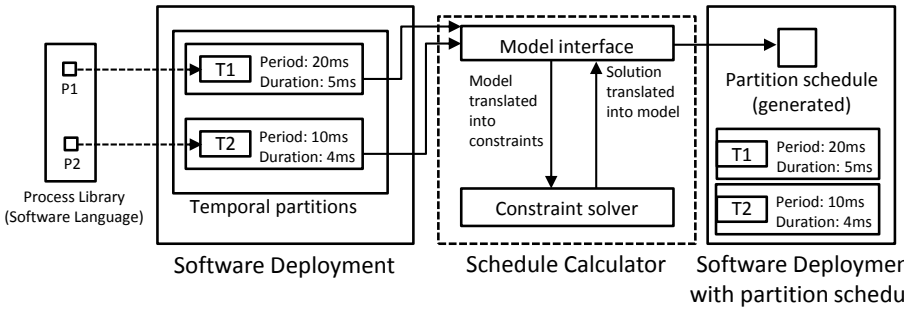

Fig. 6. The schedule calculator transforms individual partition attributes into a combined partition schedule. Processes (P1 and P2) from the software language are assigned to temporal partitions (T1 and T2) in the software deployment. The schedule calculator uses the attributes of T1 and T2 to formulate and solve a set of numerical constraints and generate a valid partition schedule, consisting of repeating sequences of the two partitions, into the model.

lation between applications. This is important because the runtime system is expected to host both trusted and untrusted (i.e., 3rd party) applications. Preventing covert channel communication between applications is also important. Our solution to this problem is a multilevel security (MLS) policy [6] that uses multi-domain *labels*. The modeling language supports the MLS policy with label elements that are attached to the communication endpoints and processes; an automated tool was built to check the compatibility of security labels between communicating processes at design time.

DREMS supports systems whose network quality and connectivity may vary over time. To analyze whether the QoS requirements of processes can be satisfied with time-varying networks, the modeling language supports QoS *profiles* describing the expected network parameters (e.g, bandwidth, latency) over time and QoS *requirements* that specify the network requirements of processes. An automated tool based on the network calculus [14] then analyzes whether the QoS requirements of all processes can be satisfied.

The third analysis relates to the verification of system properties, such as deadline violations by software components. This approach is based on modeling the abstract control flow and timing properties for component operations and then combining them with a formalized model of hierarchical schedulers in the system (the operating system scheduler and the component operation scheduler) to generate an integrated Colored Petri Net (CPN) [11] model. This model can be used to ensure that as long as the assumptions made about the system hold, the behavior of the system lies within the safe regions of operation. For example, the tool can verify that component operations will not cause deadline violations.

Overall, these design time tools provide analyses to both application developers and system integrators to ensure that the system meets the expected requirements and is robust to the runtime constraints imposed by the environment and the infrastructure.

6 Related work

There are several architectural description languages and standards that have similarities to ours, such as AADL [8], OMG's SysML [9], and AUTOSAR [2]. Both AADL and SysML are general-purpose architecture description languages that support abstractions such as software components, hardware and system integration. Our main reason for developing a custom architecture language was the need fo a language closely coupled to the semantics of our target platform to provide a sufficient level of tool automation. This required specific syntax and semantics for modeling language elements like scheduling, component interactions and security; using an AADL annex would have been too complex.

AUTOSAR [2] is a comprehensive standard for vehicle architectures that has a component model similar to ours. The drawback is that the AUTOSAR standard contains very little guidance on design-time analysis and tool support, both of which are crucial for architecture design languages.

There exist various approaches for integrating textual artifacts within graphical modeling languages. XText [7] is a framework integrated into the Eclipse Modeling Framework (EMF) for developing domain-specific languages. From a grammar in the XText grammar format, XText generates a parser that allows a graphical editor and text editor to be simultaneously used to edit a file, with the changes from each reflected in the other. The main difference to our approach is that our add-on sets multiple attributes based on the contents of the text. XText does offer an extension API that could be used to implement similar behavior.

The work in [5] considers the problem of using both textual and graphical notations to modify models of the same language. This is different from our work, which integrates textual code in one formalism with graphical models in another formalism. Language workbenches [15] also offer language composition features that can be used to combine textual and graphical languages for different formalisms.

Our temporal partition scheduler is based on the temporal partitioning method described in the ARINC-653 avionics standard [1]. ARINC-653 systems have been modeled using AADL in [4]. Unlike the ARINC-653 standard, which does not require address separation between processes of the same partition, DREMS allows system integrators to separate components into separate address spaces (called *actors*) within the same partition. The main advantage of our approach is that we can handle faults within related components (within the scope of an actor) with guaranteed isolation. This is not possible with the ARINC-653 standard.

7 Conclusions

This paper presented the multi-paradigm challenges we faced when building an architecture description language for distributed, real-time systems. The first challenge was integrating two different formalisms: textual code and graphical block diagrams. The second was computing a low-level runtime schedule from individual, high-level temporal partitions. The third challenge was integrating design-time analysis.

In cases such as ours, where a significant level of tool automation is required, we believe that a custom architecture language tailored to the semantics of the runtime platform provides a big advantage. The alternative, using a standard architecture language, can require adapations to code generators and automated analysis tools to account for semantic differences between the language and the runtime platform, such as scheduling or security.

Acknowledgments: This work was supported by the DARPA System F6 Program under contract NNA11AC08C and USAF/AFRL under Cooperative Agreement FA8750-13-2-0050. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA or USAF/AFRL.

References

1. ARINC Incorporated, Annapolis, Maryland, USA. *Document No. 653: Avionics Application Software Standard Inteface (Draft 15)*, January 1997.
2. Autosar GbR. AUTomotive Open System ARchitecture. `http://www.autosar.org/`.
3. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, pages 337–340, 2008.
4. Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement ARINC 653 systems using the AADL. In *SIGAda*, SIGAda '09, pages 31–44, New York, NY, USA, 2009. ACM.
5. Luc Engelen and Mark van den Brand. Integrating textual and graphical modelling languages. *Electron. Notes Theor. Comput. Sci.*, 253(7):105–120, September 2010.
6. Tihamer Levendovszky et al. Distributed real-time managed systems: A model-driven distributed secure information architecture platform for managed embedded systems. *IEEE Software*, 31(2):62–69, 2014.
7. Moritz Eysholdt and Heiko Behrens. Xtext: Implement your language faster than the quick and dirty way. In *SPLASH*, SPLASH '10, pages 307–309, New York, NY, USA, 2010. ACM.
8. P.H. Feiler, Bruce A. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In *Computer Aided Control System Design*, pages 1206–1211, 2006.
9. Matthew Hause et al. The SysML Modelling Language. In *Fifteenth European Systems Engineering Conference*, volume 9, 2006.
10. George T. Heineman and Bill T. Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.
11. Kurt Jensen and Lars Michael Kristensen. *Coloured Petri Nets - Modelling and Validation of Concurrent Systems*. Springer, 2009.
12. Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.
13. Klaus Schild and Jörg Würtz. Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357, 2000.
14. Lothar Thiele, Samarjit Chakraborty, and Martin Naedele. Real-time calculus for scheduling hard real-time systems. In *in ISCAS*, pages 101–104, 2000.
15. Markus Völter and Eelco Visser. Language extension and composition with language workbenches. In *SPLASH*, SPLASH '10, pages 301–304, New York, NY, USA, 2010. ACM.