System level modeling of component based software systems

Aleksandar Dimov and Sylvia Ilieva

Abstract: *One of the main problems in the area of Component-Based Software Engineering (CBSE) is how to facilitate the integration of software components into large-scale and complex systems. A possible solution is to enforce on software architecture of the system. The increased importance of architecture in CBSE raises the necessity to explore and develop methods for its formal description. Several languages and notations, named architecture description languages (ADL), were developed for formal system level specification. This paper overviews them and outlines the main directions for their evolution.*

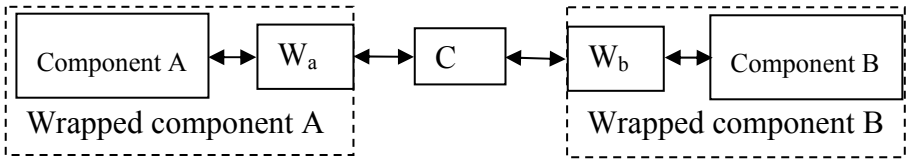
Key words: *Software Architecture, Component-Based Software Engineering, Architecture Description Language, COTS (Component-off-the-shelf).*

INTRODUCTION

Component-Based Software Engineering (CBSE) is an emerging discipline that represents a new look at the concept of source code reuse. Software components promise to provide a number of important benefits to development companies [2], like reduced development time and cost; opportunity to exchange information and experience between users of the same component product; sharing the responsibility for the quality between producers, etc.

The notion of software components is still new and there exist no comprehensive and single definition about the term component. We will consider that components are abstract black/gray box software entities, deployed in many copies and the users (typically software developers) have little or no control over their functionality and evolution. Such components are usually denoted as Commercial-Off-The-Shelf (COTS). We will follow the tendency of making the notion of COTS components broader, including in it also the so-called custom developed and open-source components.

Essential aim in the area of CBSE is to explore and develop techniques that facilitate the integration of software components into large-scale and complex systems. The design of a system, composed from COTS components requires an approach, which differs from the traditional development of a system from scratch. Possible solution is to enforce on software architecture – a level of design that goes beyond the algorithms and data structures of computation [8]. In [19] the authors argue that, straightforward integration of components may be reached, when the connections between them are treated as abstract entities, with first-class, equal to the components status. Fig.1 illustrates the architecture when applying this approach. The Connection between Components A and B is implemented is through the Connector C. The function of the additional modules, called wrappers is to adjust the component and this way it will better match to the requirements of the rest of the system [5].



W_a – wrapper for component A
W_b – wrapper for component B
C – Connector

Fig. 1: Architectural approach for component-based systems

The notion of software architecture is not new, but approaches to it were mainly a result of personal skills, or successful heuristics and not of strictly formulated rules for

integration. This fact and the increased importance of architecture with the appearance of component-based software engineering (CBSE) raise the necessity to explore and develop methods for formal description and specification of software architecture [9].

The aim of this paper is to survey and generalize the existing methodologies for formal architectural specification and modeling of software systems and make an analysis of their advantages and disadvantages. It is organized as follows: Section 2 motivates the need of software systems' formal descriptions. Section 3 makes an overview on first generation architecture description languages. Section 4 describes the recent advances in architectural description languages. Finally, section 5 concludes the paper and gives some directions for further research.

2. FORMALIZING THE DESCRIPTIONS OF SOFTWARE SYSTEMS

Formal specifications in software development originate from discrete mathematics. Early formal descriptions of software systems were the specifications of program algorithms and the use of numerical methods. The mathematical foundations of computer science are appropriate to specify the internal functionality of software components when constructing them from scratch. However, such descriptions violate the assumption that software components should be black/gray box entities. For this reason, we will concentrate at the system level modeling techniques – these, which stress on the software architecture and not on particular components. According to [17], software architecture has four levels of abstraction:

- Internal functionality of components
- Interfaces, exported by a component to the rest of the system
- Interconnection of the architectural elements (components, connectors and wrappers) in architecture
- Rules for the architectural styles

In fact, the first two of these levels concerns that part of the component that is relevant to the rest of the system – what does it requires and provides. The gray box notion of components supports descriptions for different properties of components like their behavior or semantics. The requirements for specification of the first two levels of software architecture lead to the appearance of process algebras such π-calculus [18], CSP (Communicating Sequential Processes) [12], Z notation [20], etc.

Our studies show that descriptions of software architecture at each level should incorporate some or all of the notational abstractions from the underlying layers. The most generalized notational form, in the context of the previously mentioned four levels, is the Architecture Description Language (ADL). It provides features for modeling a software system's conceptual architecture. The conceptual architecture reflects characteristics of the domain for which the ADL is intended and/or the architectural style and typically includes the ADL underlying semantic theory [16].

The existence of formal specification and modeling mechanisms for software architecture will lead to the following advantages [3, 9, 16]:

- Better, clear and unambiguous specifications, leading to improved understanding and mastering of the methods for integration of complex, large-scale software systems from preexisting components.
- Ability to involve algorithms for searching and identification of the component that optimally matches system's requirements. This process is also known as component evaluation.
- Opportunity to construct a toolset of code generators for construction of connectors and wrappers from their specification.
- Specification and assessment whether the system is capable to fulfill its nonfunctional requirements. Nonfunctional requirements are additional

constraints to the system, which concerns the amount of resource it should utilize, deadlines, for its tasks, etc.
- Possibility to enable the construction of dynamically changing software architectures.
- Ability to define and reuse successful architecture configurations as patterns and definition of component types and compiler-like checks of the conformance of component to the architecture.

In next sections, we will examine the concept of ADLs for modeling architectural configurations (compositions of components and connectors) and architectural styles – patterns of recurring successful configurations. By their maturity level these languages are divided into two main categories: First generation ADLs and Second Generation ADLs.

3. FIRST GENERATION ADLS

For the last decade, there appeared about a dozen of different ADLs. All of them aim at describing the architectural configuration, but they look at it from different point of view. Additionally, each of them had very specific syntax rules and was appropriate for describing some distinct properties of the architecture. Nowadays we call these notations first generation ADLs.

Some of the most representative examples are given in Table 1. For more detailed discussion of first generation ADLs refer to [16].

Table 1: First-generation ADLs

ADL	Underlying Formal Notation	Application area
Darwin [13]	π-calculus	Distributed systems
Wright [1]	CSP	Parallel and concurrent systems and the interactions and compositional properties between the component
Rapide [12]	Partially ordered sets of events, named "posets". Event patterns are used to recognize posets.	Simulation of architectural configurations
C2 [15]	Not explicitly supported. However, Z notation may be used to describe the semantics in C2	Construction of user interfaces and event based systems

It has been reached a consensus that every ADL should cover the description of at least three architectural features: (1) components and their interfaces, (2) connectors and (3) architectural configurations.

As an example, Fig. 2 presents the Wright ADL description of the system depicted on Fig. 1.

This is the basic specification, which shows the architectural elements (components and connectors) and their configuration (element instances and attachment between them). Components are presented by their interfaces and specifications of their functions. Component port is the only interaction point between the component and the rest of the system. The distinction between `provide` and `require` ports reflects correspondingly the component provision or requirement of services. Additionally it is allowed the presence of more than one port. Connector roles define the expected behaviour and obligations of each interacting component. Both ports and roles could be described with a sequence of alternating events in CSP notation. These descriptions are omitted, because they are not relevant for our survey. The specification of the glue process describes how the set of actions in the roles are coordinated.

Components should be interconnected with each other, using connectors. Ports are attached as roles in the attachments section. Thus, the connector coordinates the behavior of the component ports.

```
System Example
     Component CompA
          Port provide [provide protocol]
          Spec [ComponentA specification]
     Component CompB
          Port request [request protocol]
          Spec [ComponentB specification]
     Connector A_B_connector
          Role CompA [ComponentA protocol]
          Role CompB [ComponentB protocol]
          Glue [glue protocol]
Instances
     a: CompA;
     b: CompB;
     ab: A_B_connector;
Attachments
     a.provide as ab.CompA;
     b.request as ab.CompB;
end Example.
```

Fig. 2: Architecture specification in Wright ADL

A closer look at the example will show that it does not include declarations for the component wrappers from figure 1. In fact, Wright language does not explicitly support specification of component wrappers. This appears to be a common characteristic of all ADLs. Instead, wrappers are usually modeled as part of the connector.

4. SECOND GENERATION ADLS

Lessons learnt from investigating first generation ADLs, showed to researchers, that even different they share a similar conceptual basis or ontology that determines a common foundation of concepts and concerns for architectural description. This naturally led to the idea that it would be possible to create a generic ADL, which will combine the common properties and beneficial features of first generation ADLs. Such languages that try to take a broad view on the earlier attempts for architectural specification are marked as second generation ADLs. The most significant examples of second generation ADLs are ACME [7] and xADL [6].

ACME was designed to satisfy all the properties described in the previous section. It tries to generalize the existing ADLs, while concentrating on the definition of the essential aspects of software architecture. In addition, it provides the possibility to exchange descriptions between different ADLs. That is why ACME is also called an architecture interchange language. ACME supports the definition of four distinct aspects of the architecture:

- Structure: This is the organization of a system into its constituent elements – components and connectors.
- Properties of interest: This aspect gives information about a system or its parts. This way it will be possible to reason about the overall behavior.
- Constraints: This aspect supports for dynamic change of the architecture. ACME includes an underlying formal language, based on first-order predicate logic to define these constraints.
- Styles and classes: This aspect provides support for describing architectural patterns. This is an essential feature of ACME, as it makes possible to describe architectural styles.

One of the important drawbacks of first-generation ADLs is that they are not designed with the idea to be adaptable to change. We already stated the fact that they only stress on distinct facets of software systems. They also incorporate specific

underlying formal notations and tools, which make them hard to learn and understand. Hence, if one tries to explore their applicability in different domains of software engineering, he will face difficulties and probably will decide to develop a new ADL from scratch.

These facts were some of the major forces that drove the authors of xADL. It is based on XML (eXtensible Markup Language) Schemas [10], which is the key to its adaptability to change. xADL XML schemas support description not only of the architecture (Architecture modeling schema), but of the whole lifecycle of the system – architecture instantiation and configuration management schemas. Architecture instantiation schema provides refinement and transition from description to implementation to the system. Another significant benefit of xADL is that it explicitly separates the concepts of run-time (instances schema) and design-time architectures (structures and types schema). This language is not only useful for researchers, but also better suits to the needs of the practitioners, because they are not forced to learn all the features the ADL, but only those that they need.

5. CONCLUSION AND FURTHER WORK

The presented paper emphasizes on the growing need for formal description of software architecture from the viewpoint of CBSE. We have examined the facet of system level modeling with ADLs. Such description level of software systems enable the practitioners to successfully define and apply patterns of architectural configurations, called styles that allow the repetition of successful architectural designs. Examples of styles are Pipes-and-filters, Event-based architecture, Data repositories, RPC, etc [8].

We believe that the evolution of the notion of architectural styles will bring CBSE to a new level of maturity. Formal description of styles will make possible to define rules for checking the conformance of a component to specific architecture. The list of architectural styles is open for enhancement, because different domains of software engineering probably require different styles. For example, system's architecture will differ depending on the requirements: a database system, a graphical user interface or embedded control system. The difficulty is in the description of different architectural styles. As stated in [17], in the same way as there are multiple formalisms for describing interfaces and architectures, a single formal notation will be insufficient for modeling all styles. Second generation ADLs make a step in this direction, especially xADL with its extensibility based on XML. The descriptions of different architectural styles, with formal semantics such as Z, CSP or π-calculus are constrained by the application domain of these notations. Moreover, the complexity of notations is another serious disadvantage. In addition, a promising research area is to develop methods and tools to compose new styles form a set of predefined architectural primitives [14].

According to [16] ADLs insufficiently support dynamism, evolution, constraints and non-functional properties of software architecture. We believe that explicit support for component wrappers will help for more structured approach towards designing COTS-based systems [4]. There is also a lot of work to be done at the area of generalizing connectors. Hence, another direction for future research includes the investigation of alternative formal notations, which are capable to model all of the properties of different architectural styles, and sustain the enumerated poorly supported features, which an ADL should provide. Because of their generics, both ACME and xADL support a form of textual notation, but it lacks a rigid formal theory. Nevertheless, it will appear advantageous to rely on their scalability for building of an enhanced formal modeling language.

REFERENCES

[1] Allen, R. and D. Garlan: A formal basis for architectural connection; In ACM Transactions on Software Engineering and Methodology, 1997

[2] Brown, A.: Large-Scale Component-Based Development; Prentice Hall Inc. 2000.

[3] Crnkovic, I. and M. Larsson (Editors): Building Reliable Component Based Systems, ©2002, Artech House, Inc.

[4] Dimov, A. and S. Ilieva: Towards the distinction and classification of software component wrappers and connectors; In proceedings of SAER 2004, Bulgaria (to appear).

[5] Dean, J. and M. Vigder: System Implementation using Commercial-Off-The-Shelf (COTS) Software; 1997

[6] Dashofy, E., A. Hoek, R. Taylor: A highly exstensible, XML-Based Architecture Description Language; in Proceedings of the Working IEEE/IFIP Conference on Software Architectures, Netherlands, 2001, pp.103-113.

[7] Garlan, D., R. Monroe, D. Wile: Acme: Architectural Description of Component-Based Systems; Foundations of Component-Based Systems; Gary T. Leavens and Murali Sitaraman (eds); Cambridge University Press, 2000, pp. 47-68.

[8] Garlan D. and Mary Shaw: An introduction to Software Architecture, CMU SEI Technical Report, 1994.

[9] Garlan D. and Mary Shaw: Formulations and Formalisms in Software Architecture; Lecture Notes in Computer Science; Springer-Verlag 1996, pp. 307-323.

[10] Harold, E.: XML Bible, John Wiley & Sons; 2001

[11] Hoare, C.: Communicating Sequential Processes; Prentice Hall, 1985.

[12] Luckham D., et all; Specification and Analysis of System Architecture Using Rapide. IEEE Transactions on Software Engineering, 1995, vol. 21, No. 4, pp.336-355.

[13] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer; Specifying Distributed Software Architectures; In Proceedings of the Fifth European Software Engineering Conference Barcelona, 1995, pp. 137-153.

[14] Mehta N, and N Medvidovic: Composing Architectural Styles From Architectural Primitives; In Proceedings of the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE'03), Finland, 2003, pp 347-350.

[15] Medvidovic, N., P. Oreizy, and R. Taylor: Reuse of Off-the-Shelf Components in C2-Style Architectures; In Proceedings of the 19th international conference on Software engineering, Boston, 1997, pp. 692-700.

[16] Medvidovic. N and R Taylor: A classification and comparison framework for Software Architecture Description Languages; IEEE Transactions on Software Engineering, Vol. 26, No. 1, 2000, pp.70-93.

[17] N. Medvidovic, R. Taylor, and E. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In Proceedings of the California Software Symposium 1996, Los Angeles, CA, April 1996, pp. 28-40.

[18] Parrow, J.: An Introduction to π-calculus; In Handbook of Process Algebra, available at http://user.it.uu.se/~joachim/intro.ps.

[19] Shaw, M.: Procedure calls are the assembly language of system interconnection: Connectors deserve first-class status, In: Proceedings of the Workshop on Studies of Software Design, LNCS, Springer-Verlag, 1993, pp.17-32.

[20] J. M. Spivey. The Z notation: a reference manual. Prentice Hall, New York, 1989.

ABOUT THE AUTHORS

Aleksandar Dimov, PhD Student, Department of Information Technologies, Faculty of Mathematics and Informatics, Sofia University "St. Kliment Ohridski", Phone: +359 2 971 35 09, e-mail: aldi@fmi.uni-sofia.bg

Assoc. Prof Sylvia Ilieva, PhD, Department of Information Technologies, Faculty of Mathematics and Informatics, Sofia University "St. Kliment Ohridski", Phone: +359 2 71 71 27, e-mail: sylvia@acad.bg