HAL
open science

DiaSuite:A Paradigm-Oriented Software Development Approach (invited paper)

Charles Consel

To cite this version:

Charles Consel. DiaSuite:A Paradigm-Oriented Software Development Approach (invited paper). PEPM'11 - 20th ACM SIGPLAN workshop on Partial evaluation and program manipulation, Jan 2011, Austin, TX, United States. pp.77-78, 10.1145/1929501.1929515 . inria-00581652

HAL Id: inria-00581652

https://inria.hal.science/inria-00581652v1

Submitted on 31 Mar 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

DiaSuite: A Paradigm-Oriented Software Development Approach

Charles Consel
University of Bordeaux / INRIA, France
Charles.Consel@inria.fr

Abstract

We present a software development approach, whose underlying paradigm goes beyond programming. This approach offers a language-based design framework, high-level programming support, a range of verifications, and an abstraction layer over low-level technologies. Our approach is instantiated with the Sense-Compute-Control paradigm, and uniformly integrated into a suite of declarative languages and tools.

Categories and Subject Descriptors D.2.11 [*Software Engineering*]: Software Architectures—Domain-specific architectures, Languages, Patterns; D.3.4 [*Programming Languages*]: Processors

General Terms Design, Language

Keywords Declarative Languages, Programming Support, Program Analysis, Program Generation

Introduction

A software development process inherently relies on some form of development paradigm to help structure, program and compose the building blocks of a software system. A development paradigm is most commonly concretized in the form of a programming language (*e.g.,* functional or object-oriented), associated with a range of techniques and tools to facilitate programming (*e.g.,* editing, verification, and debugging). However, a development paradigm involves multiple dimensions, besides programming, whose integration incurs both a conceptual and programming overhead. Further downstream from programming are dimensions such as middleware, which may impose a specific control flow on the program, as is done by event-based middlewares. Further upstream from programming are architectural styles and patterns [10] that guide the decomposition of a software system into components and define their interactions.

Interestingly, the partial evaluation community has been contributing to this multi-dimension approach by promoting the interpreter style [10] and introducing related techniques and tools. In the interpreter style, a program must implement a processing driven by the interpretation of a key input argument, whose binding time is assumed early. The structure of the resulting program allows a static analysis phase to determine binding-time invariants. These invariants are used by a transformation phase to specialize the program, given a value for the key input argument. The static analysis and the transformation phase form the partial evaluation process [5]. This process is most commonly used as an optimization strategy dedicated to removing the interpretation layer. The interpreter paradigm, coupled with partial evaluation, has been successfully applied to a wide range of areas, besides programming language implementations, including string matching [1], networking [2] and operating systems [7]. Although limited in the dimensions covered, this form of a paradigm-oriented software development approach has demonstrated its effectiveness for optimization purposes.

Realizing that the scope of development paradigm goes beyond programming is a key to improve the software production process, and it suggests to integrate the multiple dimensions of software development into a uniform approach.

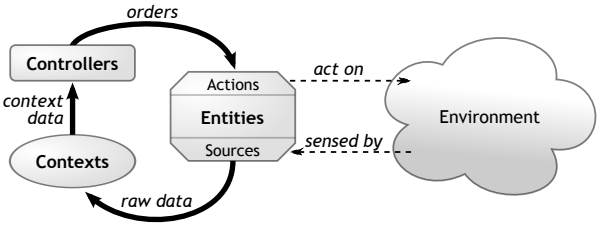

Figure 1. The architecture of an SCC software system

This talk. We report on research results showing that a paradigm-oriented approach covers many more dimensions than program optimization. We introduce a paradigm-oriented development approach that offers a language-based design framework, high-level programming support, a range of verifications, and an abstraction layer over underlying technologies, going beyond a contemplative approach. We have developed a paradigm inspired by the *Sense-Compute-Control* (SCC) architecture pattern [10]; it is realized by a suite of languages and tools, named DiaSuite [4].

Our Approach

In our approach, an SCC software system gathers data about an environment via sensors (whether hardware or software). Sensed data are used by *context components* to compute refined, application-specific values. These values are then provided to *control components*; they define the control logic aimed to issue orders to the actuators, impacting the environment. The architecture of an SCC software system is depicted in Figure 1.

The SCC paradigm has a wide spectrum of applicability; we have used it successfully in the domains of home/building automation, multimedia, avionics and networking.

Copyright is held by the author/owner(s).
PEPM'11, January 24–25, 2011, Austin, Texas, USA.
ACM 978-1-4503-0485-6/11/01.

Let us now examine how this paradigm guides the design of a software system, provides dedicated programming support, and enables verifications.

Software design. Our SCC-oriented development approach relies on an SCC-specific description language. This language provides syntax and semantics to define a conceptual framework, and is supported by processing tools.

Specifically, our approach revolves around a description language, named DiaSpec, dedicated to describing an SCC software system. To do so, DiaSpec consists of two language layers: (1) one layer is for declaring a software system at the functional level, decomposing it into context and control components, and defining how these components are connected to each other; (2) another layer is for declaring the sensors and actuators to be used by the SCC software system.

Programming support. A DiaSpec description is processed to generate a customized programming framework. This framework provides high-level support to implement sensors, actuators, and components, abstracting away from the underlying technologies (*e.g.,* hardware, networking and middleware). Furthermore, the generated programming support guides and constrains programming, leveraging the underlying programming language. For example, DiaSuite generates an abstract class for each DiaSpec declaration. An abstract class implements methods, hiding low-level mechanisms such as communications, and declares abstract methods, delimiting where the application logic is to be introduced.

Verification. At declaration time, a DiaSpec description is verified independently of an implementation for consistency properties. For example, every component, sensor and actuator must be connected. At development time, a DiaSpec description is used to reason about an implementation to check its conformance. For example, a component implementation only communicates directly with another component if they are connected in the DiaSpec description. In doing so, we ensure the *communication integrity* property [9]. At runtime, the generated programming framework includes code to preserve this conformance. For example, if a class of sensors has no running instance, an error can be raised and a repair strategy performed.

Non-Functional Properties

Beyond offering a conceptual framework, our language-based approach provides an ideal setting to address non-functional properties (*i.e.,* performance, reliability, security...). A description language can be extended with non-functional declarations, expanding further the type of conformance that can be checked between the description of a software system and its implementation, and enabling additional programming support and guidance.

We have investigated this idea by extending DiaSpec with non-functional declarations to address error handling [8], component flow behavior [3], and quality of service constraints [6].

Following our approach to paradigm-oriented software development, non-functional declarations are verified at declaration time, they generate support that guides and constrains programming, they produce a runtime system that preserves invariants.

Let us instantiate our approach with error handling [8].

Software design. First, declarations are introduced to provide a conceptual framework to express how errors should be treated. Specifically, declarations specify what errors are raised by sensors and actuators, and what types of treatment are provided by context and control components. For example, a declaration may require a component to fully handle sensor-related errors, shielding client components from these concerns.

Programming support. A DiaSpec description, extended with error-handling declarations, is used to generate a programming framework that guides and supports the implementation of error handling, besides the application logic. In doing so, the paradigm-oriented development makes the programming of error handling more rigorous and systematic.

Verification. Verification at declaration time aims to ensure that components define appropriate treatment types along the flow of errors. Conformance of the component implementation with respect to error-handling declarations is checked statically.

Conclusion

Our initial work on the paradigm-oriented software development approach suggests a number of future research directions. It would be insightful to go beyond the SCC paradigm. What other style could cover a range of application domains and give rise to a dedicated description language, and dedicated framework generation and verification phases? Another direction is to explore other non-functional properties of a software system, such as security and performance.

Acknowledgements

The author is indebted to the members of the Phoenix Group, who contributed to the research results that enabled this work. Special thanks to Emilie Balland for insightful discussions and helpful comments.

References

[1] T. Amtoft, C. Consel, O. Danvy, and C. Malmkjœr. The Abstraction and Instantiation of String-Matching Programs. In *The Essence of Computation*, number 2566, pages 367–390. Lecture Notes in Computer Science, 2002.

[2] S. Bhatia, C. Consel, A.-F. Le Meur, and C. Pu. Automatic Specialization of Protocol Stacks in OS kernels. In *Proceedings of the 29th Annual IEEE Conference on Local Computer Networks*, Tampa, Florida, Nov. 2004. Awarded best paper.

[3] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging Architectures to Guide and Verify Development of Sense/Compute/Control Applications. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 2011.

[4] D. Cassou, B. Bertran, N. Loriant, and C. Consel. A Generative Programming Approach to Developing Pervasive Computing Systems. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, pages 137–146. ACM, 2009.

[5] C. Consel and O. Danvy. Tutorial Notes on Partial Evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 493–501. ACM, 1993.

[6] S. Gatti, E. Balland, and C. Consel. A Step-wise Approach for Integrating QoS throughout Software Development. Technical report, INRIA, 2010.

[7] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wagle, C. Consel, G. Muller, and R. Marlet. Specialization Tools and Techniques for Systematic Optimization of System Software. *ACM Transactions on Computer Systems*, 19(2):217–251, May 2001.

[8] J. Mercadal, Q. Enard, C. Consel, and N. Loriant. A Domain-Specific Approach to Architecturing Error Handling in Pervasive Computing. In *Proceedings of the ACM international conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'10)*. ACM, 2010.

[9] M. Moriconi, X. Qian, and R. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineerin*, 21(4):356–372, 2002.

[10] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.