The Journal of Systems and Software 85 (2012) 932–943

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Debugging applications created by a Domain Specific Language: The IPAC case

Kostas Kolomvatsos*, George Valkanas, Stathes Hadjiefthymiades

Pervasive Computing Research Group, Department of Informatics and Telecommunications, National & Kapodistrian University of Athens, Greece

ARTICLE INFO

Article history:
Received 10 January 2011
Received in revised form 2 November 2011
Accepted 11 November 2011
Available online 26 November 2011

Keywords:
Debugger
Domain Specific Language
Software testing

ABSTRACT

Nowadays, software developers have created a large number of applications in various research domains of Computer Science. However, not all of them are familiar with the majority of the research domains. Hence, Domain Specific Languages (DSLs) can provide an abstract, concrete description of a domain in terms that can easily be managed by developers. The most important in such cases is the provision of a debugger for debugging the generated software based on a specific DSL. In this paper, we propose and present a simple but efficient debugger created for the needs of the IPAC system. The debugger is able to provide debugging facilities to developers that define applications for autonomous mobile nodes. The debugger can map code lines between the initial application workflow and the final code defined in a known programming language. Finally, we propose a logging server responsible to provide debugging facilities for the IPAC framework. The IPAC system is consisted of a number of middleware services for mobile nodes acting in a network. In this system a number of mobile nodes exchanged messages that are visualized for more efficient manipulation.

1. Introduction

A lot of research has been performed in various domains of Computer Science. In most of the cases, software components should be developed in order to provide more functionality in the produced systems. However, users, not having a lot of experience with programming languages, are not able to develop efficient software components. In this case, *Model Driven Engineering* (MDE) can provide a lot of advantages not only to underexperienced programmers but also to proficient ones that are unfamiliar with a specific domain. MDE is a software development methodology for creating models in a specific domain. MDE technologies offer a promising approach to address the inability of the third generation languages to express domain concepts effectively (Schmidt, 2006). The aim of MDE is to increase efficiency in developing applications for the specific domain. *Domain Specific Languages* (DSLs) follow the principles of the MDE development and can provide a number of advantages in cases where domain programming knowledge is limited (Mernik et al., 2005; Sprinkle et al., 2009).

A DSL is a language designed for a specific field of applications. Its aim is to solve problems related to a highly focused field of research. DSLs target to more specific tasks than classic programming languages. They provide expressions for describing parameters of a domain of interest and they have a concrete syntax. A number of semantics are used in order to lead to the automatic generation of specific tools important for the creation of the final code (Kelly and Tolvanen, 2008). In DSL tools, there are specific methodologies for the definition of the semantics of each language. The most significant advantage of the DSL usage is that they provide the opportunity to users to write more easily domain specific programs (Kosar et al., 2008). These programs are not dependent on the underlying platform, thus, providing an additional advantage.

From the above, it is obvious that there is a need for the development of debugging facilities for DSLs in order for the user to be able to debug code written in the specific language. The debugger should be oriented to the specific DSL covering all the elements of the language. However, this is a difficult task due to the fact that DSLs are oriented to specific domains and generic debuggers are not directly applicable. The debugging process should be error free and not time consuming otherwise the system will not be efficient.

In this paper, we present our proposal for debugging applications developed with a DSL. Furthermore, we present our system for a server responsible to translate debug messages sent by a number of mobile nodes. We shortly describe an *Application Description Language* (ADL) created in the framework of the *IPAC* (*Integrated Platform for Autonomic Computing*) project. The ADL is the basis for the creation of a number of utilities useful for the creation of applications. Our debugger builds on top of a logger messaging infrastructure, where messages follow a special format and are presented in a friendly user interface. The important is that these log messages are created in a known programming language and the debugger is capable of mapping the code lines of the generated code to the lines in the initial application workflow. Our proposal

* Corresponding author.
E-mail addresses: kostasks@di.uoa.gr (K. Kolomvatsos), gvalk@di.uoa.gr (G. Valkanas), shadj@di.uoa.gr (S. Hadjiefthymiades).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.11.1009

is characterized by simplicity as well as efficiency because it does not require a lot of experience and effort from the developer's side.

The paper is organized as follows. Section 2 is devoted to the description of the related work. We present research efforts in creating DSLs as well as some debugger proposals. Accordingly, in Section 3, we shortly describe the IPAC system. Section 4 describes the ADL, proposed for the creation of applications for autonomous mobile nodes. The IPAC code generation component is also presented in Section 4. This component is responsible to produce the final application code in a known programming language (i.e. Java). Specific templates are used for the mapping process between the final code lines and the initial application workflow. In Section 5, we describe the architecture of the proposed debugger. We present an *Open Service Gateway initiative* (OSGi) (OSGi) service created to manipulate log messages and depict the viewer that visualizes such messages. We devote Section 6 in describing the IPAC logging server responsible to visualize log messages sent by mobile nodes. This approach has the advantage that we can debug the whole system and the interconnection of nodes as they act in the network. Finally, a case study for two example applications is presented in Section 7 by describing step by step the debugging process and we conclude our paper in Section 8.

2. Related work

DSLs attracted a lot of attention due to the reason that they provide abstraction in the definition of applications oriented to a specific research field. In Wu et al. (2009), the authors demonstrate a framework to automate the generation of DSL testing tools. The presented framework utilizes Eclipse plug-ins for defining DSLs. Moreover, a set of tools concerning a translator, and an interface generator are responsible to map the DSL debugging perspective to the underlying *General Purpose Language* (GPL) debugging services. The aim is to present the feasibility and the applicability of debugging and testing information derived by a DSL in a friendly programming environment.

A program transformation engine supporting the debugging process written in a DSL is described in Rebernak et al. (2009), Wu et al. (2008), and Wu et al. (2009). The discussed approaches concern the methodology of generating a set of tools necessary to use a DSL from a language defined in a specific grammar. Such tools are: the editor, the compiler and the debugger (Henriques et al., 2005). This research effort focuses on issues related to the debugging support for a DSL development environment. The debugger is automatically generated by a language specification. Authors describe two approaches for weaving the debugger in conjunction with the *DSL Debugging Framework* (DDF) plug-in. The first approach is applicable when the aspect weaver is available for the generated GPL while the second approach involves the *Design Maintenance System* (DMS) (Baxter et al., 2004) transformation and is applied when the aspect weaver is not available.

In Sadilek and Wachsmuth (2008), the authors describe a prototyping methodology for of *Domain Specific Modeling Languages* (DSMLs) on an independent level of the MDE architecture. They argue that the prototyping method should describe the semantics of the DSML in an operational fashion. For this, they use standard modeling techniques i.e. *Meta Object Facility* (MOF) (Meta Object Facility) and *Query/View/Transformations* (QVT) Relations (Query-View-Transformation). By combining this approach with existing metamodel-based editor creation technologies they enable the rapid and cost free prototyping of visual interpreters and debuggers. Authors utilize the *Eclipse Modeling Framework* (EMF) which is similar to MOF and using the Ecore metamodel of a DSML they can generate the DSML plug-in with EMF. The created plug-in provides the basis for the creation, access, modification, and storage of models that are instances of the DSML.

A logic programming based framework for specification, efficient implementation, and automatic verification of DSLs, is presented in Gupta and Pontelli (2002). Their proposal is based on Horn logic and, eventually, constraints to specify semantics of DSLs. The semantic specification serves as an interpreter and more efficient implementations of the DSL, such as a compiler, can be automatically derived by partial evaluation. The executable specification can be used for automatic or semi-automatic verification of programs written in a DSL as well as for automatically obtaining conventional debuggers and profilers. The provided Horn logic syntax and semantics are executable leading to the automatic definition of an interpreter. Finally, the authors present their approach and give some examples indicating the efficiency of the discussed methodology.

The use of execution semantics in creating a debugger for a DSL is also discussed in Blunk et al. (2009). The authors present models for the debugging context, breakpoints and stepping of programs written in a voice control language. A model-to-model transformation approach is followed where a DSL model is translated to a debugging metamodel, e.g. in QVT Relations. The debugging metamodel describes concepts for visualizing threads, variables and their values. A specific user interface is used for depicting the variables.

In this paper, we propose a DSL for embedded systems and describe our approach for the debugging process. Our DSL is oriented to the definition of applications that rely on a number of middleware services in order to provide intelligent, context-aware services to final users. Our proposal for the debugging process involves the use of log messages sent to a central debugging service responsible for translating and visualizing them. Log messages contain references that indicate the mapping between the DSL code and the final code defined in a GPL. The proposed solution involves a visualization part where developers can easily identify the code components that are erroneous by observing the application variables values. The debugging facility is fully embedded in an *Application Creation Environment* (ACE) and, thus, its use is very simple. The applications built in IPAC framework include code for handling events when they run in the OSGi environment. The OSGi framework has its own event handling mechanism and the application cannot affect it. This imposes a difference between our framework and other proposed solutions. Our debugging mechanism should be able to react in such cases. When the application runs, it is registered in the OSGi environment to react to events. This means that the application is not mainly executed in a step wise manner but it is a combination of a step wise executed part and an event reacting part. Each application should be tested in the OSGi environment. Moreover, as the ACE is embedded in Eclipse but it acts as a separate system, we need a solution that will not utilize the *Eclipse Debugging Framework* (EDF). Therefore, we provide a separate debugging interface where the developer can see the reaction of the application to events. This is achieved by using the IPAC emulator. Comparing our work against the literature, we can denote the following differences:

- The work presented in Blunk et al. (2009) utilizes a meta-model description of the language model and operational semantics in order to map the language with the EDF which assumes the interfacing responsibility. A model to model transformation takes place. In our solution, we are not based on a meta-modeling approach as it adds extra overhead. Moreover, the meta-modeling approach has another disadvantage (Ehrig et al., 2008): it is not constructive, i.e., it does not offer direct means of generating instances of the language.
- Another difference of our proposed framework compared to Blunk et al. (2009), Wu et al. (2008), and Wu et al. (2009)