Volume : 3 | Issue : 12 | Dec 2014 ISSN - 2250-1991

Research Paper Computer Science

Model-driven approach for component-based evaluation of metaheuristics

Dr. Carsten Mueller Faculty of Informatics and Statistics, University of Economics Nám. W. Churchilla 4, 130 67 Prague

ABSTRACT

In this paper a new approach for handling the problem of comparing different metaheuristics based on generated components is proposed. The main idea behind this research is to set up a model-driven framework that can be applied to the most well-known and widely used metaheuristics. The evaluation framework provides a flexible and standard testing methodology for comparing the accuracy and performance of selected algorithms.

KEYWORDS evaluation, metaheuristics, software architecture, model-driven architecture

Introduction

Based on the trend towards heterogeneous and distributed infrastructures, integration technology is increasingly used in order to exchange data between applications and systems as well as to provide uniform access to multiple data sources. In this research paper the integration concepts are classified into the following layers: information integration (data integration and function integration), application integration, process integration and GUI integration.

In the last 20 years, a new kind of approximate algorithm has emerged which basically tries to combine basic heuristic methods in higher level frameworks aimed at efficiently and effectively exploring a search space. These methods are called metaheuristics (Reeves, 1993). The term metaheuristic, first introduced in Glover (Glover, 1986), derives from the composition of two Greek words. Heuristic derives from the verb heuriskein (which means "to find", while the suffix meta means "beyond, in an upper level". This class of algorithms includes - but is not restricted to - Ant Colony Optimization (ACO), Evolutionary Computation (EA) including Genetic Algorithms (GA), Iterated Local Search, Simulated Annealing, and Tabu Search. In this research the class of metaheuristics belongs to the complex algorithms.

Component-based systems are made up of collection of interacting entities called components. The idea in component-based software engineering (CBSE) is to develop software applications not from scratch but by assembling various libraries of components. Therefore, one saves on the development costs and time.

CBSE shifts the development emphasis from programming software to composing software systems (Clements, 1996). In a component-based system, components and frameworks should have certified properties; and these certified properties should provide the basis for predicting properties relative to the whole system built out of those components. The widely accepted goal of component-based development is to build and maintain software systems by using existing software components (Reeves, 1993; Hybertson, 2001; Goessler & Sifakis, 2005).

To facilitate the software design and evolution, CBSE (Crnkovic, Sentilles, & Vulgarakis, 2011; Heineman & Councill, 2001) postulates a modular and systematic construction of software from reusable components that are extended, adapted and replaced. The cornerstone of CBSE is the underlying component model that defines how components are specified, constructed, assembled and deployed (Lau & Wang, 2007).

A component exhibits a set of functions and data through a well-defined interface and hides implementation details.

In (Crnkovic et al., 2011) several component definitions are discussed, this research is based on the definition given in (Bosch, Szyperski, & Weck, 2003). This definition implies that: i) a software component is a unit, ii) it specifies an interface (or interfaces) of services it provides, iii) it specifies context dependencies, and iv) it may be part of a larger composite component. A composite component is built from other components; a component that is not a composite is called a primitive component. As in (da Silva, de Castro, & Rubira, 2003), it is valuable to consider the notion of sub-typing as a formal way to organize types in the applications.

The UML component model represents software systems by means of components and their relationships. A component's behaviour is specified in terms of provided and required interfaces. Composition is realised by connecting components over required and provided interfaces that match. Components are realised through explicit classes defined in the scope of an associated class model that defines a component's inner workings, such as the implementations of the provided interfaces.

II. Approach

Services are reusable, self-contained, autonomous units with a well-defined interface, and are capable of communicating with each other via messages. Services are published to a repository by service providers and can be retrieved and consumed by service consumers.

Dynamic software adaptation addresses software systems that need to change their behaviour at run-time (Kramer & Magee, 1990). With model-based dynamic software adaptation, models are used to describe and sequence the adaptation of the software architecture and executable system at run-time (Gomaa, 2006). A model-based software adaptation pattern defines how the components that make up an architecture or design pattern dynamically cooperate to change the software configuration to a new configuration given a set of adaptation commands.

Mathematical models that capture the essence of a paradigm play an essential role as a foundation for methods, languages and support tools for that paradigm (Broy, 2007). Architectural models in particular contribute to the identification of abstractions that are useful for describing the architecture of software systems, including the architecture of specific classes of systems.

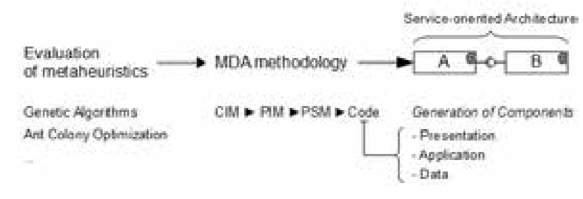

Figure 1: Overview
Model-Driven Architecture (MDA) is a paradigm for the model-driven development of software and hardware. The core approach leads from a computer-independent model (CIM) over the platform-independent model (PIM) to the platform-specific models (PSM). These PSMs, which are created in dependence on the platform models (PM), are used as input for template-based code generation.

PIM is a model of a software system that does not depend on the specific technologies or platforms used to implement it while a platform-specific model links to particular technologies or platforms. Leveraging this advantage of the MDA paradigm two basic layers are defined: abstract and technology-specific. The abstract layer includes the views without the technical details that the metaheuristics experts can understand and manipulate. Then, the IT experts refines or maps these abstract concepts into platform- and technology-specific views. The technology-specific layer contains the views that embody concrete information of technologies or platforms.

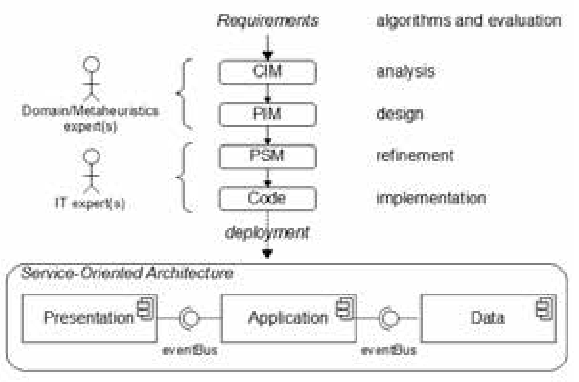

Figure 2: Architecture and responsibilities
As the first step targeting at model-based management, it is essential to have the proper model of service composition. The logic of the proposed approach to construct the model is: (1) build the meta-model of service composition; (2) specify the map between this metamodel and the running service composition by an access model (3) finally generate the source for the runtime model.

The implementation is based on the concept of a software factory including event-based communication and is adapted to the design and implementation of applications which rely on a data model. It provides the ability to define models and to also perform transformations on them in order to generate either refinements or platform-specific models. In order to achieve this, it uses source code generation as a transformation mechanism in order to produce one or many PSM from the PIM.

Modelling event-based communication at the architecture level requires new meta-model elements. Event-based systems have four core elements in common: Events, Sources, Sinks, and the Middleware (Carzaniga, Nitto, Rosenblum, & Wolf, 1998). Events are data elements asynchronously transferred between components to trigger a certain behaviour or to transfer data. They are instantiated within sources, which are responsible to publish and emit the event. The counterpart of a source is the sink, which receives and processes events. The communication between sources and sinks is enabled by a communication middleware supporting loosely-coupled communication among distributed software components. This allows a source to send an event and then continue working while the event is being delivered and processed.

III. Conclusions
In this paper a model-driven approach for component-based evaluation of metaheuristic was developed. This paper mainly realizes the model-based runtime management to provide a flexible and standard testing methodology for comparing the accuracy and performance of selected algorithms. This unique software architecture is highly flexible and offers standard component-based mechanisms to extend. Researchers worldwide are able to use this platform to evaluate existing and new complex algorithms against problems. Currently a component for basic statistics evaluation was developed. The further research includes the extension of additional algorithms and problems, standard public interfaces for the worldwide usage and an intelligent statistics engine.

REFERENCES

Bosch, J., Szyperski, C., & Weck, W. (2003). Component-Oriented Programming. European Conference on Object-Oriented Programming, Darmstadt, Germany, 34-49. | Broy, M. (2007). From "formal methods" to system modeling. In C. Jones, C. Liu, & J. Woodcock (Eds.), (Vol. 4700, p. 24-44). Springer. | Carzaniga, A., Nitto, E. D., Rosenblum, D., & Wolf, A. (1998). Issues in supporting event-based architectural styles. Proceedings of the third international workshop on Software architecture, New York, 17-20. | Clements, P. C. (1996). From subroutines to subsystems: Component-based software development. Component-Based Software Engineering: Putting Pieces Together. Addison-Wesley. | Crnkovic, I., Sentilles, S., & Vulgarakis, A. (2011). A Classification Framework for Software Component Models. IEEE Transactions on Software Engineering, 37 , 593-615. | da Silva, M., de Castro, P., & Rubira, C. (2003). A Java component Model for Evolving Software Systems. 18th IEEE International conference on Automated Software Engineering, Montreal, Canada, 327-330. | Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. Computers & Operations Research, 13 , 533-549. | Goessler, G., & Sifakis, J. (2005). Composition for component-based modeling. Science of Computer Programming, 55 , 1-3. | Gomaa, H. (2006). A Software Modeling Odyssey: Designing Evolutionary Architecture-centric Real-Time Systems and Product Lines. LNCS(4199), 1-15. | Heineman, G., & Councill, W. (2001). Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing. | Hybertson, D. (2001). A uniform component modeling space. Informatica, 25 , 475-482. | Kramer, J., & Magee, J. (1990). The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering, 16 (11), 1293-1306. | Lau, K., &Wang, Z. (2007). Software Component Models. IEEE Transactions on Software Engineering, 33 , 709-724. | Reeves, C. (1993). Modern Heuristic Techniques for Combinatorial Problems. Blackwell Scientific Publishing. |