A Development Environment to Support Development with Reuse

Perla Velasco Elizondo

Centre for Mathematical Research, CIMAT.
Jalisco S/N Colonia Valenciana. Guanajuato, Guanajuato, 36240, México.
pvelasco@cimat.mx

Abstract. This paper describes a development environment to support the process of development with reuse. The objective of our environment is to automate this process within the context of an alternative component composition approach based on the semantics of a new component model.

1 Introduction

Component-based development (CBD) aims to develop software systems by *reusing* software components rather than coding the systems entirely from scratch. There is a general understanding of components as reusable building elements of computation that are *composed* together into larger blocks. We believe that such a scenario not only depends on the availability of the components, but also on how the components are assembled together and what kind of composition mechanisms are utilised.

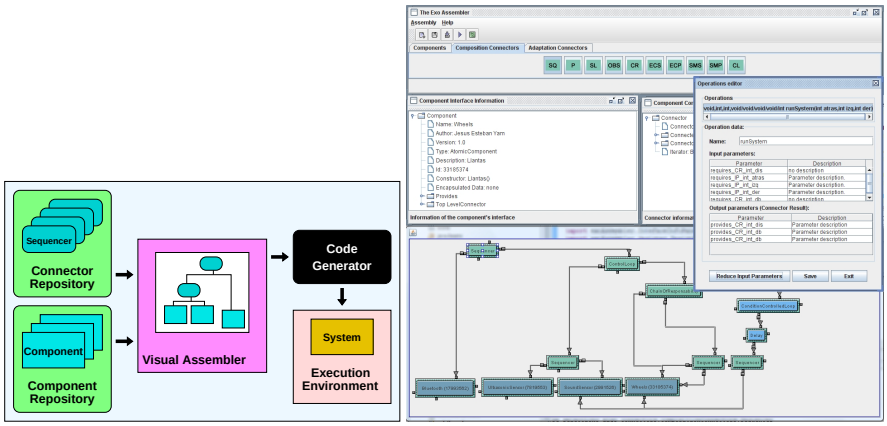
In current CBD approaches, components either are constructed from scratch or, because of the nature of the composition mechanisms utilised, components contain very specific information to the composition they are constituents. Constructing components from scratch goes against the approach of composing *pre-existing* components which is the basis of CBD. On the other hand, components containing very specific composition information makes them *highly coupled* with the components they communicate, which hinders their reuse for the construction of different systems. Additionally, components dealing with composition aspects makes them *more complex* and impacts on providing a clean separation of *computation* from *communication-coordination*, which is bad because reuse of computation is a main issue in CBD.

To tackle these shortcomings, we have introduced an alternative composition approach based on the semantics of a new component model [6, 4]. The approach is based on *components* –which encapsulate computation and *connectors* –which encapsulate well-known *communication* and *coordination patterns*. An assembly of components is built via connectors in a *hierarchical bottom-up* manner. An important characteristic in our approach is that both, components and connectors are not only *first-class architectural elements* but also *reusable compilation units* at implementation stage.

In preview work we have implemented components and connectors as well as generated some prototype systems from them, e.g. [5]. In [2] we have defined a catalogue of connectors to support the process of *development with reuse*; the process of constructing final systems by utilising *pre-existing* components in *binary* format. In this paper, we describe a development environment to support this process.

2 The Development Environment

As shown in Fig. 1 (a), the architecture of our development environment contains five main elements: (1) a *Component Repository*, (2) a *Connector Repository*, (3) a *Visual Assembler*, (4) a *Code Generator* and (5) an *Execution Environment*. All these elements as well as the GUI that integrates them (see Fig. 1 (b)) were developed in Java. Thus, any computer having Java Virtual Machine (JVM) installed will be able to run our environment as well as the generated systems.[1]



(a) (b)

Fig. 1. (a) The architecture and (b) GUI of the development environment.

The environment supports development with reuse by dragging, dropping and connecting in a hierarchical manner pre-existing components and pre-existing connectors from the *Component* and *Connector* repositories into the *Visual Assembler*. As components (and connectors) were implemented in Java (see [4, 5]), *reflection techniques* are utilised to perform structural introspection on components' *binaries* and their interfaces to retrieve the information about the services they offer.

Every time a connection between a component and connector is created, the *fine-grain* specification of such a connection is carried out via a composition wizard. According to the semantics of the connector utilised and the information known about the selected component, the wizard asks the system developer to provide specific data –e.g. the methods to execute in the connected component, the required parameters, etc. The tool performs the certain *syntactic* and *semantic checks* to ensure the proper definition of an assembly. Each assembly can be considered a subsystem and (if required) be generated as a identifiable compilation unit. Therefore, a subsystem can be tested and reused separately for further composition.

Once an architecture is fully defined, the *Code Generator* generates the source code and the binary file of the new Java class representing the system, which is meant to be deployed and executed on the *Execution Environment*, which is a JVM.

[1] Our development environment works on a JRE version 1.5 or higher and requires about 7.1MB of free disk space. There are not special memory/processor requirements above those recommended for the hosting operating system.

For space reasons, we do not provide examples of systems developed via our environment. However the details of some robotics systems, which were developed by using our environment, are presented at [1].

3 Discussion

As stated before, the aim of our environment is to automate development with reuse within the context a new component composition approach. Thus, this work shares many of the goals and principles of those concerned to define systems in terms of components and their interconnections, e.g. MILs [8], coordination languages [9] and ADLs [7]. The focus on conceptual architecture and explicit treatment of first-class connectors differentiate our approach from MILs. Connectors in coordination languages are very different in nature from ours because they do not behave like any pattern. Although some ADLs come with a set of predefined connector types and incorporate some support for system code generation, the "intrusive" nature of connector mappings in these ADLs (i.e. connectors are embedded in components code), results in neither first-class nor reusable architectural elements at implementation stage.

In our environment system construction is a pure *hierarchical bottom-up* process involving not only pre-existing components but also pre-existing connectors. The definition and generation of subsystems that are logically highly cohesive but low-coupled at the same time is supported too. If required, any generated subsystem can be tested and reused separately for further composition in the current or in a different system. Because of the former, in contrast to other approaches our development tool maximises both *external* and *internal reuse* [3].

Although the connectors in our catalogue offer a semantics that is *perse* a mechanism to enforce the correct construction of systems, in the current stage of this work it has been assumed that the composed components fully satisfy the requirements to the system to built and that they present compatible signature naming conventions, execution models and environmental dependencies among them and the execution environment into which are going to be deployed. These assumptions have to be revised, and ideally dropped, to fit a more realistic development context.

Finally, the process of system construction supported by our development environment differs from the processes adopted by current commercial component-based tools. This may limit its adoption in practice. However, experiences demonstrate that the manner in which components and connectors are treated in current approaches is not enough to achieve the CBD desiderata. That makes our environment worthy of consideration.

4 Conclusions and Future Work

This paper describes a development environment to support the process of development with reuse within the context of an alternative component composition approach based on the semantics of a new component model. The usefulness of the development environment has been demonstrated; the behaviour of some of the generated systems can be observed at [1].

As discussed before, in its current state our environment does not provide a full support to validate the correctness of systems architectures and the generated systems. Thus, planned future work include to tackle this aspect within a specific application domain so that domain specific components, connectors and analysis techniques can be developed and integrated to our environment. For example within the control systems domain, the period, deadline, computation time, maximum blocking time and worst-case execution time are examples of non-functional properties for which an analysis is compulsory. Then, components' interfaces containing these information can be defined so that they could make it possible to perform some checks during system construction to ensure the participant components are free of conflict before composing them.

Additionally, it may be also interesting to enhance the capabilities of development environment by considering tools to support other activities in the component-based development life cycle such as component development, component search and retrieval, (component and) system testing, etc. In this context, a more robust approach to component interface specification should be defined. Reflection techniques could be also utilised to inspect information in the resulting interfaces to support component search and retrieval as well as the automatic generation of test cases.

Acknowledgments

This work was supported by CONCyTEG under grant 08-02-K662-119.

References

1. P. Velasco Elizondo. Systematic and automated development with reuse. `http://www.cimat.mx/~pvelasco/exo/exotool_en.html`.
2. P. Velasco Elizondo and K.-K. Lau. A catalogue of component connectors to support development with reuse. Submitted to Journal of Systems and Software.
3. W. Frakes and C. Terry. Reuse level metrics. In *Proceedings of the 3rd International Conference on Software Reuse: Advances in Software Reusability*. IEEE, 1994.
4. K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In G.T. Heineman, I. Crnkovic, H. Schmidt, J. Stafford, C. Szyperski, and K. Wallnau, editors, *Proceedings of 8th International SIGSOFT Symposium on Component-based Software Engineering*, pages 90–106. Springer-Verlag Heidelberg, May 2005.
5. K.-K. Lau, L. Ling, P. Velasco Elizondo, and V. Ukis. Composite connectors for composing software components. In M. Lumpe and W. Vanderperren, editors, *Proceedings of the 6th International Symposium on Software Composition, LNCS 4829*, pages 266–280. Springer-Verlag, 2007.
6. K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F.S. de Boer *et al.*, editor, *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2006.
7. N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering*, 26(1):70–93, 2000.
8. R. Prieto-Diaz J.M. Neighbors. Module interconnection languages. *Journal of Systems and Software*, 6(4):307–334, 1986.
9. G.A. Papadopoulos and F. Arbab. *The Engineering of Large Systems*, volume 46, pages 329–400. Advances in Computers, Academic Press, September 1998.