A Domain-Specific Language for Industrial Automation[1]

Stefan Preuer
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University, Altenberger Straße 69, A-4040 Linz, Austria
preuer@ase.jku.at

Abstract: Software development is a complex task and therefore requires professional programming skills. Restricting the focus to a specific problem domain allows the application of domain-specific concepts and techniques that enable domain experts to develop software without being professional programmers. This paper describes such an approach in the area of industrial automation, where a new domain-specific language *Mocol*, together with a virtual machine and a development environment has been developed.

1 Introduction

General-purpose programming languages like C#, Java, or C++ are widely used today. They provide a variety of general concepts, which make them applicable to almost all kinds of problems. However, these languages require a deep understanding of their concepts to produce reliable and maintainable programs. These skills can only be expected from professional software developers. Specific problem domains often do not need this sophistication. On the other hand, these domains require specific concepts that are not directly provided by general-purpose languages. A language based on domain concepts can simplify software development. It does not require professional software developers, but allows domain experts to adequately develop software in an efficient way.

In my already finished diploma thesis a domain specific language for the automation and coordination of activities at a teeming platform of a continuous caster was developed. The goal of this language named *Mocol* (Motion Control Language) is to ease the development of programs in that area in order to allow people at the plant site to perform certain programming tasks without needing high-level programming skills. During my diploma thesis I was employed at the Christian-Doppler Laboratory for Automated Software Engineering, which is a research laboratory strongly focused on industrial cooperation. The industrial partner for my work was Siemens VAI, the world's leading engineering and plant-building company for the iron and steel industry.

[1] This work was performed in cooperation with Siemens VAI, Austria, and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

2 Approach

The diploma thesis had a fix deadline, which resulted from its involvement in a running project within Siemens VAI. Because of this deadline a total time of seven month was scheduled for the thesis. Since a very detailed draft for the language Mocol was proposed by Siemens VAI at the beginning of my thesis the most extensive part was the design and implementation of a compiler, virtual machine and integrated development environment for the language. Anyhow, the really crucial factor was the fit of the language Mocol itself. Therefore in a first step an evaluation of the proposed language took place. For this task experiences of colleagues researching in a similar area were very helpful. This potential for synergies was also an important reason why the work for the thesis was done within the context of the Christian Doppler Laboratories for Automated Software Engineering. Although some possible enhancements for the language were found, beside some minor changes the originally proposed language has been retained almost unchanged. The main reason for this decision was to keep the language as simple as possible but at the same time as complex as necessary. Based on the available domain knowledge at that time the proposed enhancements seemed to make the language unnecessarily complex. So the basic approach was to develop a prototypical implementation based on the draft of the language Mocol, and use this prototype to gain further knowledge about the requirements on the language. This approach required a strong willingness for change, which in general is a fundamental requirement for successful software engineering. The major driving force for this change was the gain of knowledge about the domain the language is targeted for. Also very important was the implementation of software components simulating the behaviour of real world components which activities the language should coordinate. This allowed a comfortable way of testing the system and especially to write automatically repeatable test cases. Such automatic tests are crucial to reach a high quality in a steady evolving software system, because it provides you with certainty that changes did not negatively influence already tested functionality. The following chapters very shortly explain the technical aspects of the developed software system.

3 Environment

The whole system responsible for the coordination of the activities at the teeming platform is called *Motion Controller*. To cope with the complexity of the system its architecture was designed with a clean layering. An important result of this layering is the abstraction from the hardware of the components to be coordinated. The pivotal component typically is an industrial robot. Other components are for example a repository for various tools, a positioning system, or a human machine interface. The Mocol VM is the heart of the Motion Controller. It is an interpreter for Mocol programs, which uses a component abstraction layer to interact with the hardware components and on the other hand acts as a server application allowing different clients to use its services. In productive mode such a client typically is an expert system, which decides what programs to start. Another client can be an integrated development environment for the development of Mocol programs. Figure 1 shows this layering.

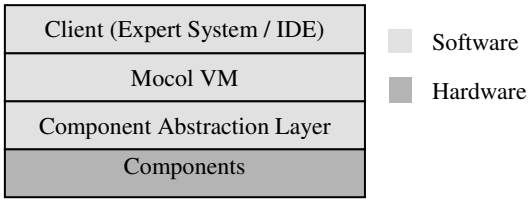

Figure 1: Basic layering of the Motion Controller

4 The Domain-specific Programming Language Mocol

Automating and coordinating activities on a teeming platform requires some fundamental concepts that a domain-specific language should be able to express in a simple way. In Mocol the most important concepts are:

- Interaction with the components abstracting from their real implementation. Therefore, the Mocol interface of each component consists of properties, commands and events or errors.
- Explicit and implicit parallelism.
- Waiting on state-specific conditions of the components.
- Expressing timing constraints.
- Handling of asynchronous and synchronous events and errors during the execution of a program.
- Managing data channels between components.
- Checking for preconditions which must hold before a program is executed.

Beside these specific requirements Mocol offers general concepts of structured programming. **Listing 1** shows an example of a Mocol program measuring the temperature of the liquid steel in a tundish.

```
PROGRAM TemperatureMeasurement
    PARAMETERS
        speed AS NUMBER
    USE Roboter, Repository, MeasureBox
    PRECONDITIONS
        Roboter.IsInHomePosition, NOT Repository.NoTempLance
    EVENTS
        ON Repository.Empty NAMED Handler: EXEC Repository.RequestLances
    ERRORS
        ON ANY: EXIT 100
BEGIN
    ACTIVATE Handler
    Roboter.Speed := speed
    CONNECT Roboter.Position TO Hmi.PositionMonitor
    PAR EXEC Roboter.MoveToRepository || EXEC Repository.RequestTempLance END
    WAIT Repository.TakeOverOk AND Repository.LanceOk MAX 10000
    EXEC Roboter.MoveToTakeOverPosition ( Repository.SelectedLanceSlot )
    WAIT Repository.TakeOverComplete MAX 5000
    EXEC Roboter.MoveToTundish
```

```
    START Roboter.DipLanceIntoLadle
    EXEC MeasureBox.PrepareProbeAdmission
    JOIN Roboter.DipLanceIntoLadle
    EXEC Roboter.DeliverProbe
    IF MeasureBox.MeasureOk THEN EXEC MeasureBox.DisposeProbe END
    EXEC Roboter.MoveToHome
END
```

Listing 1: Example of a Mocol program

5 The Mocol VM

The Mocol VM allows the interpreted execution of Mocol programs. Therefore a compiler generates an abstract syntax tree (AST) out of a Mocol source file. The compiler itself is generated by Coco/R [MLW06], a compiler generator using an attributed grammar (ATG) as input. Based on the AST the interpreter is responsible for the actual execution of Mocol programs and provides debugging functionality. **Figure 2** shows the generation of the compiler, and its embedding in the whole architecture of the Mocol VM.

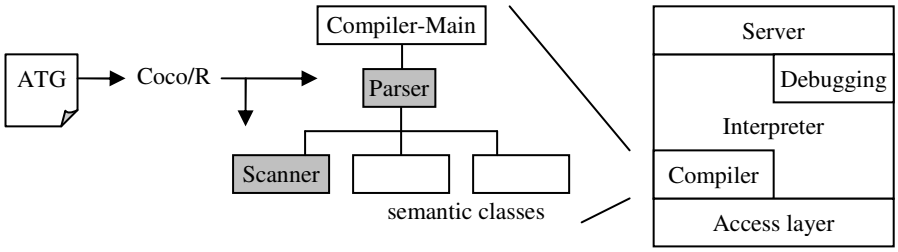

Figure 2: Mocol Compiler in context of the whole Mocol VM

6 Summary and Results

The implemented Motion Controller as a result of the diploma thesis allows the easy description and execution of coordination activities. Nevertheless, further enhancements are necessary to improve the applicability. In particular, a more elaborate model for the treatment of error situations is required. These enhancements are planned to get integrated in the current solution and therefore form a further evolutionary step in the development of the Motion Controller.

References

[MLW06] Mössenböck, H., Löberbauer, M., Wöß, A.: User Manual of the Compiler Generator Coco/R. http://ssw.jku.at/Coco/