Building End-User Programming Systems Based on a Domain-Specific Language[1]

Herbert Prähofer, Dominik Hurnaus, Hanspeter Mössenböck

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University, 4040 Linz, Austria

{hurnaus,praehofer,moessenboeck}@ase.jku.at

Abstract. End-users of automation software systems – which are the machine operators – have the task to provide machine settings and program simple control algorithms to adapt and optimize the machine to the specific automation tasks at hand. End-user programming systems are therefore an intrinsic part of automation systems. In this paper we report on a project with the goal to build a software framework which allows realizing end-user programming systems with minimal effort. Our approach is based on a new component-based programming language Monaco for event-based machine control, a compiler-generator to realize a Monaco compiler, a virtual machine for execution of Monaco programs, and an Eclipse- and GEF-based modelling and program development environment.

1 Introduction

End-user programming as defined by [Nardi 1993] is the endeavour to give ordinary users some of the computational power enjoyed by professional programmers. The goal of end-user programming is that users are able to customize and adapt the software systems in use to their particular needs at hand, so that they can perform their work more efficiently and effectively. As we well know, programming is not a trivial task even for professionals. End-users, however, have in no way the capabilities of professional programmers and usually they are neither able nor willing to acquire such skills. Providing limited programming capabilities to users in a natural, intuitive manner is still an open issue and subject to active research [Myers and Ko 2003; Costabile, Piccinno, Fogli, Mussio 2003]. In contrast to ordinary programming environments, from an end-user programming system we expect much better user support, like domain-specific notations and presentations, user guidance, and, in addition to syntactic checks, also semantic integrity checks of user programs [Won 2002].

Our work is concerned with end-user programming in the automation domain. The end-user of the automation software system is the individual machine operator. His task is basic handling and supervision of the machine operations, usually with the help of an electronic control panel. Additionally, he is involved in configuring the machine to specific manufacturing tasks and tooling equipments. And he eventually has the task of adapting or arranging basic machine operations in defined sequences, so that a specific machining task can be performed in a most efficient way. Providing all the machine parameter settings and defining an operation sequence for a very specific manufacturing task usually requires deep knowledge of

[1] This work has been conducted in cooperation with Keba AG, Austria, and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

the domain and, due to its complexity, still represents a great challenge. Therefore, machine automation software systems usually have to come with elaborate machine configuration and end-user programming environments.

In this paper we present a project whose goal is the development of a software framework which allows building end-user programming systems in the automation domain. The background for this work is a cooperation with Keba AG (www.keba.com) which is a medium-sized company developing and producing automation solutions for industrial automation. Keba is mainly a platform provider and development of automation solutions based on the Keba platform is a multi-stage process involving several different stakeholders as follows:

- Keba develops and produces a hardware and software platform with associated tool support. This comprises a PC-based open hardware architecture, different programming languages with compilers and development environments and Java-based frameworks for building visualization systems and control panels.
- The hardware and software platform enables the customers of Keba (OEMs of automation machines) to realize automation systems for their products.
- The OEMs also build customized software systems and electronic control panels for the machine operators. Those also include customized, machine-specific configuration and end-user programming systems for machine operators.

Project experience showed that the effort for OEMs for building those end-user programming systems is tremendous. End-user systems are individually designed and always built from scratch. Keba observes increased customer demands for an easier development of customized end-user programming systems. A clear conception and a reusable software basis for building end-user programming systems are therefore heavily desirable. As a consequence, in this project we deal with the conception and the development of a software framework to enable customers of Keba to realize end-user programming systems with minimal effort. The approach, which is outlined in Section 2 in more detail, is based on a new domain-specific language for machine control, a compiler-generator framework to realize a compiler for the Monaco language, a virtual machine for execution of Monaco programs, a visual notation for the Monaco programs, and an Eclipse- and GEF-based integrated development environment (IDE).

In this presentation we show the current developments and we discuss some additional features which are required for a software platform facilitating realization of end-user programming systems. The rest of the presentation will be structured as follows: In Section 2 we present our approach in more detail. Section 3 gives a short introduction into the Monaco domain-specific language for automation control. Section 4 finally discusses additional features required for an end-user programming framework and shows how all the discussed concepts work together in the instantiation of a customized end-user programming system.

2 Approach

In this section we present the architecture and components of our software framework which together with some additional features as discussed in Section 4 will facilitate the realization of end-user programming systems.

The approach is based on the following concepts:

- A new domain-specific language for machine control – called *Monaco* (*Modelling NOtation for Automation COntrol*) – is defined which serves as a basis for building control flow programs. The language is similar to Statecharts in its expressive power, however, adopts an imperative notation which is much closer to the perception of the domain experts.

- A compiler-generator framework is used to realize a compiler for the Monaco language which constructs a parse tree object model (called *CodeDOM*) of the program as well as a writer program to write the object model back to source code.
- A virtual machine has been designed and a prototype is realized in Java for the execution of Monaco programs.
- A visual notation for the language has been defined which is usually preferred by domain experts and end-users.
- An Eclipse- and GEF-based integrated development environment (IDE) for the Monaco programming language is currently in development. The IDE allows a dual mode of programming, i.e., visual programming as well as text based programming.

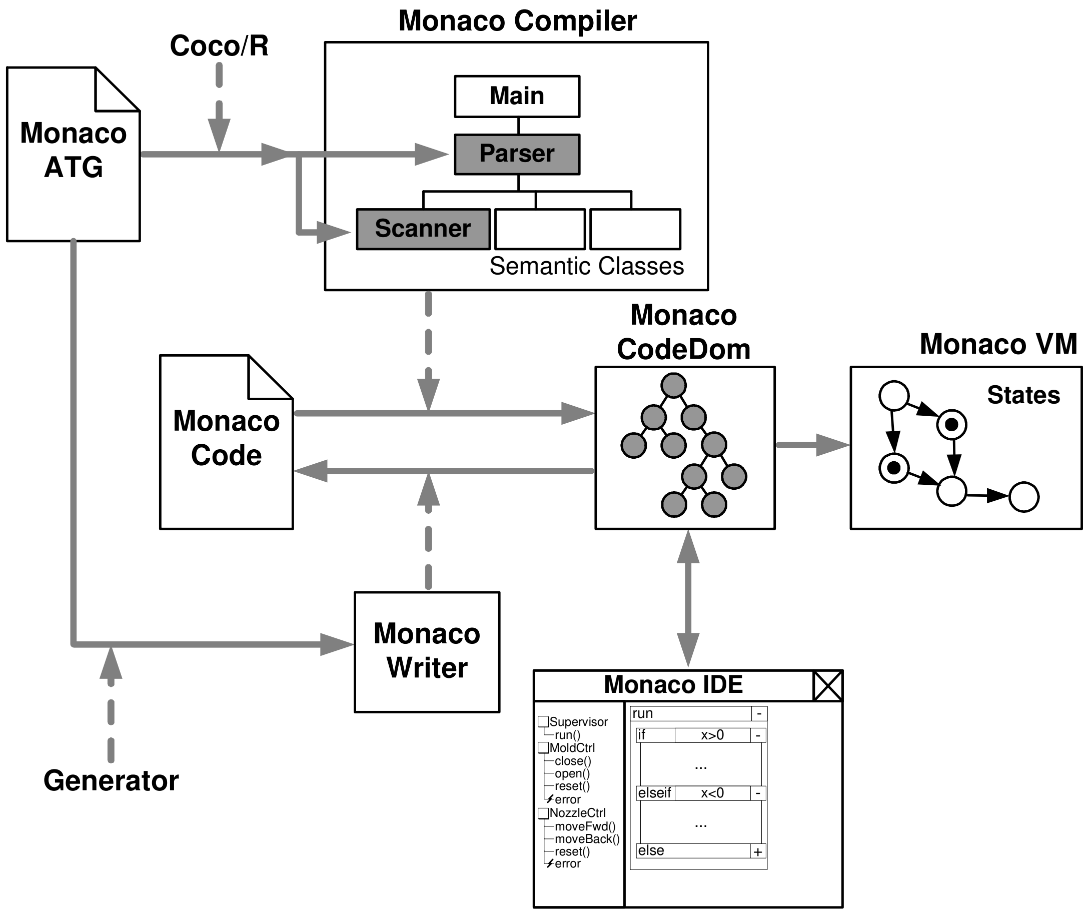

Figure 1: Architecture of domain-specific programming environment

Figure 1 gives an overview on the different components in the framework and their collaborations. The basis of the whole framework is the definition of the Monaco domain-specific language in an attributed grammar (*Monaco ATG*) which defines concrete as well as abstract syntax of the Monaco language. The attributed grammar serves as input to the Coco/R [Mössenböck 1990; Wöß, Löberbauer, Mössenböck 2003] compiler-generator which produces the *Monaco compiler* to read Monaco source code programs and generate parse tree object models (*Monaco code object model – Monaco CodeDOM*) representing the abstract syntax of Monaco programs in the form of a tree structure.

A Monaco CodeDOM then is input to the Monaco virtual machine (*Monaco VM*) which is capable of executing Monaco programs. Currently a purely interpreted version of the virtual machine in Java is available; however a version in C is envisioned to cope with real-time requirements. The execution of Monaco programs is based on a simple API to the virtual machine which basically allows parallel execution of threads, installation of (temporary) event handlers, handling of variable scopes, etc.

An Eclipse-based integrated development environment (*Monaco IDE*) is used to allow interactive development of Monaco programs. A visual program editor is used to allow visual interactive editing of Monaco programs. The visual editor operates directly on the Monaco CodeDOMs. In difference to many visual programming systems, the visual programming notation for Monaco is laid out automatically and the user has no control over positioning of program elements. In fact, the visual presentation directly reflects the source code structure, however, in a two-dimensional arrangement instead of a one dimensional as in the source code. Moreover, collapsing and expansion of block elements are heavily used (see more in Section 3.2).

Finally, a writer program (*Monaco Writer*) is generated from the Monaco attributed grammar definition which is capable of taking a Monaco CodeDOM and producing Monaco source code.

3 The Domain-Specific Programming Language Monaco

Monaco (Modelling NOtation for Automation COntrol) has been designed with the goal of bringing programming of machine control system closer to the domain experts and finally the end-users. It is not an end-user programming language itself, but is intended to form a basis to finally support end-user programming.

Current languages in the automation domain, most notably the programming languages of the IEC 61131-3 standard [IEC 2003], do not fulfill the requirements of a programming language which can be used by domain experts or end-users. Moreover, those languages do not show the characteristics of state-of-the-art programming languages from a software engineering perspective. Other formalisms, in particular the widely adopted Statechart [OMG 2004] formalism or the IEC 61499 standard [IEC 2005], would have the expressive power and can be regarded to be up-to-date regarding software engineering practices. However, the state machine notation seems to be too complex and cluttered for domain experts.

The language Monaco has been designed to overcome those shortcomings. The language is specialized to a rather narrow sub area of the automation domain, i.e., programming control sequence operations of single machines. This narrow domain includes any type of automated machines, but excludes bigger automation systems, like whole manufacturing systems. Within the multiple layers of automation systems it should cover the layer of event-based control of machine operations. Out of scope are therefore the continuous control and signal processing layer, which are supposed to form the layer below, and manufacturing execution layer, which could form the layer above.

Within this narrow domain, the language should allow expressing the desired control sequences in a natural and concise way. It has to allow expressing sequences of machine operations but also allow handling asynchronous events, exceptions, and errors. The language should allow writing reliable programs, which are easy to comprehend and maintain also by domain experts. With respective tool support, the language is supposed to form a basis for end-user programming systems.

The language design has been driven by the following main assumptions on the perception of the domain experts of automation machines. Those are:

- A domain expert views a machine as being assembled from a set of independent components working together in a coordinated fashion.
- Each component normally undergoes a determined sequence of control operations. There are usually one or several sequences which are considered to be the normal mode of operation. Those are usually quite simple. Complexity is introduced by the fact, that those

normal modes of operation can be interrupted anytime by the occurrence of abnormal events, errors and malfunctions.

- The control sequences of the different machine components are coordinated at a higher level in fulfillment of a particular control task.

The language design reflects just those assumptions by pursuing the following language concepts:

- Monaco pursues a strict component-based approach. Components are modular units (black boxes) which exclusively communicate over defined interfaces.
- The language supports hierarchical abstraction of control functionality. The component structure forms a strict hierarchy in that interaction of components only occurs with its subordinate and superordinate components. A component relies on the operations, state properties, and events from its subordinate components. It composes and coordinates the behavior of its subordinates and provides abstract and simplified views to its superordinates.
- Although the behavioral model of the language is very close to Statecharts, an imperative style of programming is used. This, in particular, allows programming of control sequences as a sequence of statements. Moreover, the language supports procedural abstraction.
- Focus of the language is on event behavior. Statements have been introduced to express reaction to asynchronous events, parallelism and synchronization, exception handling and timeouts in a concise way.

In the following section we will briefly depict the language by an example control program.

3.1 Example Monaco program

This example introduces some of the language concepts of the Monaco programming language. The example is part of a larger application in the domain of injection molding. The application is hierarchically composed of various components (Figure 2). Each of these components represents either the control unit for one part of the real machine or a control unit that aggregates the functionality of other components. This view of a machine exactly matches the end-user perception of the automation system.

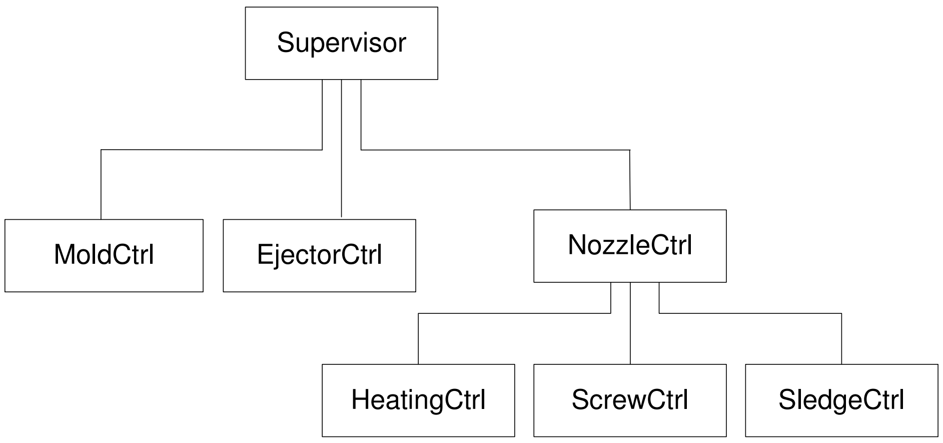

Figure 2: Component hierarchy

Each of these components implements a special interface to facilitate the assembly of those components to a system of components. The code sample shows the implementation of the upmost `Supervisor` component (Figure 3). This usually represents the level an end-user would be involved with. This component implements the interface for Supervisor controls (ISupervisor) and can be parameterized by two parameters which have to be set at start-up time. All subcomponents of the Supervisor are only defined by their interfaces to allow easy exchange of components. The component has two routines run and stop. In the following we will take a closer look at the run routine.

```
COMPONENT Supervisor IMPLEMENTS ISupervisor
  PARAMETERS
    coolingTime : INT := 1000;
    plastFirst : BOOL := TRUE;

  SUBCOMPONENTS
    ejector : IEjectorCtrl;
    mold : IMoldCtrl;
    nozzle : INozzleCtrl;

  ROUTINE stop()
  BEGIN
    mold.reset();
    nozzle.reset();
    ejector.reset();
  END stop

  ROUTINE run()
  BEGIN
    mold.open();
    ejector.moveBackward();
    nozzle.moveForward();
    LOOP
    BEGIN
      mold.close();
      IF plastFirst THEN
        nozzle.plasticize(100);

      PARALLEL
        BEGIN
          nozzle.inject();

          IF NOT plastFirst THEN
            nozzle.plasticize(100);
        END
        BEGIN
          WAIT coolingTime;
        END
      END

      mold.open();
      ejector.moveForward();
      ejector.moveBackward();
    END

  ON mold.error.FIRED OR nozzle.error.FIRED OR ejector.error.FIRED
    stop();
  END run
END Supervisor
```

Figure 3: Monaco source code sample – the `Supervisor` component

This routine describes the main control cycle of the automated machine. It orchestrates the subcomponents by calling routines of those subcomponents, which in turn call routines on their subcomponents or directly manipulate the machine by setting signals. Events/errors are handled at the end of the routine prefaced by the keyword ON. Event handlers must be declared using an event condition that consists of Boolean expressions and/or events. In our example any error event fired by one of the subcomponents invokes the `stop()` routine that stops all subcomponents.

This representation of a Monaco component is already quite straight-forward and, to some extent, already understandable by end-users that are not familiar with programming languages. As a next step, a visual representation of this component can help end-users not only to understand existing programs, but also to independently adapt existing Monaco components or even create new components. The next section introduces ideas about a visual notation of Monaco programs.

3.2 Visual notation of Monaco programs

Domain experts and end-users prefer visual representations of control programs. We therefore have defined a visual representation for Monaco programs and are in the process of developing an integrated development environment (IDE) based on the Eclipse RCP (www.eclipse.org) and the Graphical Editor Framework (GEF, www.eclipse.org/gef). In this section we describe the ideas behind this visual notation.

The visual notation of Monaco programs directly reflects the structure of Monaco programs. Rectangular visual elements are used for each statement and in particular reflect the block structure of the program. Each visual element can be collapsed and expanded in order to define the level of detail the user wants to see (Figure 4). Expansion of elements is not restricted to vertical expansion, but can also be used horizontally to, for example, expand event handlers defined for a block.

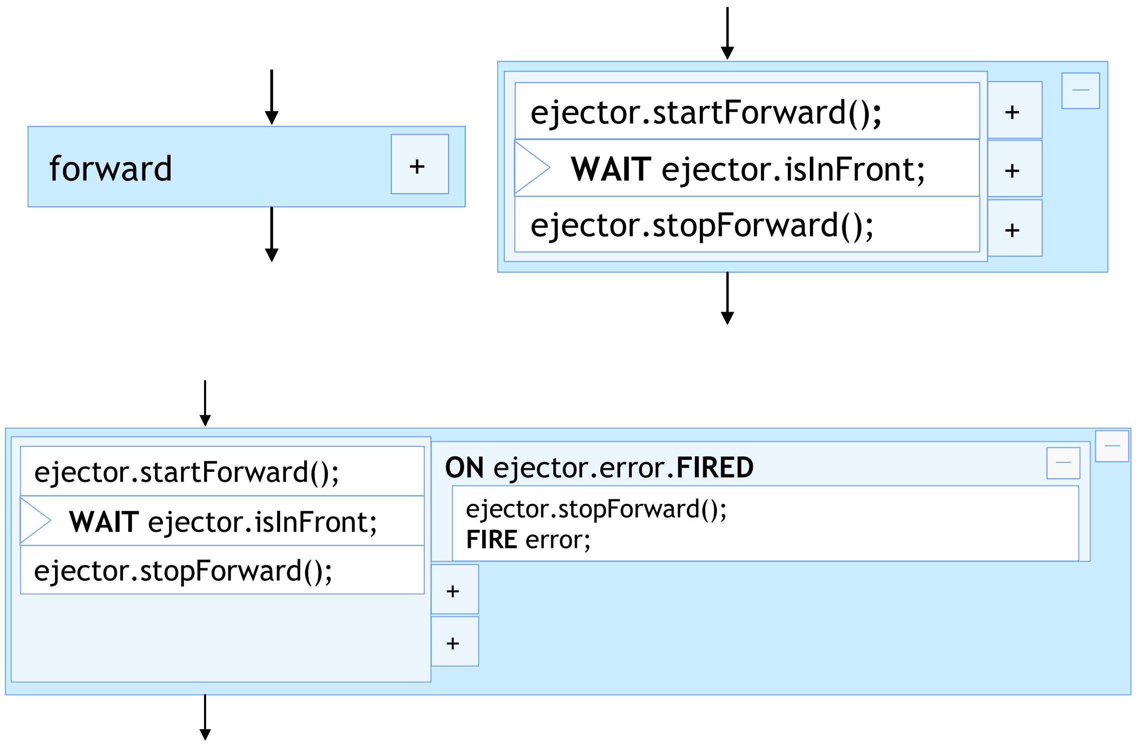

Figure 4: Visual representation of programming elements (collapsed and expanded)

A visual notation of a programming language is only meaningful if there is appropriate tool support. The Monaco IDE is capable of automatically creating a visual program editor from the Monaco CodeDOM without requiring any further positioning from the user, i.e., layouting of elements is done automatically. The duality of the language representations (textual and visual) will also be reflected in the IDE. Changes to the visual representation directly change the underlying Monaco CodeDOM which in turn updates the textual representation of the program.

4 Towards End-User Programming Systems

In Section 2 we have shown our approach for realizing a programming environment for the Monaco domain-specific language. However, for a real end-user programming system for machine control as envisioned in section 1, several features are still missing. In this section we will discuss additional features, we think are required for an end-user programming framework, and show ways for realizing those.

To support end-user programming the following features are required:

Configurable end-user presentation

Control programs should be presented in domain-specific terms and notations familiar to end-users. This includes the use of icons for program elements [Bischoff and Seyfarth 2002], e.g. components and control operations, the use of customized interactive dialogs for settings and configurations, the use of technical terms and physical units instead of program data types, and the support of the native language of the end-user.

How to tackle those issues actually seems quite straight-forward. Meta-modelling environments [Zhu, Grundy, Hosking 2004; Luoma, Kelly, Tolvanen 2004; Greenfield and Short 2004] have shown how to customize the visual presentation of modelling elements. In distinction to those systems, our approach additionally supports layout management which can be customized by the adaptation and introduction of layout managers, which are supported by the GEF framework. From interactive GUI builders and GUI components, like JavaBeans [Hamilton 1997] and Delphi [Mitchell 2002], we know how to introduce individual property editors and customization dialogs for component properties. In [Praehofer and Kerschbaummayr 1999] we have shown how a model of physical properties and units can be used advantageously in the realization of domain modelling languages.

User roles and permissions

There usually exists not only one particular type of user of a machine. User types differ in the permissions they have for changing machine settings and control algorithms. The everyday operator of a machine might be allowed to watch the operation and react to malfunctions. An advanced operator might have the permission to reconfigure the machine and make operation-specific settings. A machine expert might be permitted to reprogram the machine in a limited way. Maintenance staff then will have to change the control programs themselves, but may be prohibited to change safety critical program parts.

To cope with those different user types, one requires a user administration and permission concept and a means to clearly specify the permitted interventions. This has to go hand in hand with strong program variability mechanisms as discussed next.

Program variability

A control program for an automation machine cannot be a rigid unit but has to show considerable flexibility and variability with respect to reconfiguration, adaptation and reprogramming. Moreover, a complex control program is usually not realized for one individual machine type but for a family of similar machines. As there exists usually a product family for the machines themselves, the control programs have the characteristic of a software product line [Clements and Northrop, 2002]. Expressive concepts for modelling program variability are strongly required.

The Monaco language already supports some limited forms of variability. First, subcomponents of components are polymorphic, i.e., they are declared by an interface and a subcomponent can be replaced by another component implementing the same interface. Second, components have parameters which allow the adaptation of components in a defined and limited way. However, stronger concepts to model program variability are needed. We need means to identify program parts which can be changed and to declare the constraints thereupon.

We intend to adopt mechanisms to represent code variability as introduced by the product line engineering community [Bosch 2000; Bosch et al. 2001]. Together with the user role and permission model as discussed above, this should give the information what a particular user should be able to see from a control program, what parts the user should be able to change, and how the user can change those parts.

User guidance and semantic integrity checks

End-user programming is mainly concerned with programming the up-most sequence of control operations. An end-user, however, normally has not the capabilities to check that a sequence of operations is correct and results in a semantically meaningful and complete control program. A system which guides the user and checks the correct assembling of operations into semantically meaningful, correct, complete, and secure control algorithm would be heavily desirable.

Such an approach does not exist yet. However, recent work in formal software specification and verification [de Alfaro and Henzinger 2001, 2005] provides promising results in that direction. The application of such theoretical results in the realm of end-user programming, therefore, will be an important research direction in the future.

5 Conclusion

In this paper we have presented a framework for domain-specific programming in the automation domain. The approach is based on a new domain-specific programming language Monaco for event-based machine control. Monaco emphasized a component-based approach with hierarchical structuring of control programs and mechanisms for programming with events. It shows strong similarities to Statecharts with respect to its behavioural model, but adopts a notion similar to classical imperative programming languages. Then a visual notation for Monaco programs has been defined and a visual editor is in development.

In section 4 we have discussed additional features which we consider to be required for an end-user programming framework. Backed with the Monaco programming system and with those additional features, an end-user programming system for a particular machine is instantiated as follows:

- The basis is a Monaco program for a particular automation machine.
- User types and associated permissions are set up.
- This Monaco program is augmented with a variability model which precisely specifies the ways the program can be changed, adapted, and extended by the different types of users.
- Configuration files are introduced which define domain-specific pictures, icons, physical units, and language-specific settings to be used by the end-user.
- With all those configurations, a specific, individual end-user programming system for a particular automated machine can be instantiated without further programming.

References

[Bischoff and Seyfarth 2002] Bischoff, R. Kazi, A. Seyfarth, M.: The MORPHA Style Guide for Icon-Based Programming. *Proc. of the 11th IEEE Int. Workshop on Robot and Human interactive Communication*, ROMAN2002, Berlin, Germany, September 25-27, 2002, pp. 482-487.

[Bosch, 2000] Bosch, J.: Design and Use of Software Architectures, Adopting and Evolving a Product Line Approach. Addison Wesley, 2000.

[Bosch et al., 2001] Bosch, J., et al.: Variability Issues in Software Product Lines, Proc. *Of the 4th International Workshop on Product Family Engineering*, 2001.

[Clements and Northrop, 2002] Clements, P. and Northrop, L.: *Software Product Lines: Practice and Patterns*. Addison-Wesley 2002.

[Costabile, Piccinno, Fogli, Mussio 2003] M.F.Costabile, A. Piccinno, D. Fogli and P. Mussio. "Software Shaping Workshops: Environments to Support End-User Development", For the: CHI 2003 Workshop on Perspectives in End User Development

[de Alfaro and Henzinger 2001] Luca de Alfaro and Thomas A. Henzinger. Interface automata. Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE), ACM Press, 2001, pp. 109-120.

[de Alfaro and Henzinger 2005] Luca de Alfaro and Thomas A. Henzinger. Interface-based desig. In Engineering Theories of Software-intensive Systems (M. Broy, J. Gruenbauer, D. Harel, and C.A.R. Hoare, eds.), NATO Science Series: Mathematics, Physics, and Chemistry, Vol. 195, Springer, 2005, pp. 83-104.

[Greenfield and Short 2004] Jack Greenfield and Keith Short. Software Factories. Wiley, 2004.

[Hamilton 1997] Graham Hamilton (Editor). JavaBeans. Sun Microsystems. 1997, http://java.sun.com/products/javabeans/docs/spec.html.

[IEC 2003] IEC, Programmable controllers-Part 3:Programming languages. http://www.iec.ch/, 2003.

[IEC 2005] IEC, IEC 61499-1, Function blocks - Part 1: Architecture. http://www.iec.ch/, 2005.

[Luoma, Kelly, Tolvanen 2004] Janne Luoma, Steven Kelly, Juha-Pekka Tolvanen. Defining Domain-Specific Modeling Languages: Collected Experiences. OOPSLA Workshop on DSM 2004, http://www.dsmforum.org/Events/DSM04/luoma.pdf.

[Mitchell 2002] Mitchell C. Kerman, Programming & Problem Solving with Delphi, Addison Wesley, 2002.

[Mössenböck 1990] Mössenböck, H.: A Generator for Production Quality Compilers. 3rd intl. workshop on compiler compilers (CC'90), Schwerin, Lecture Notes in Computer Science 477, Springer-Verlag, 1990, pp. 42-55.

[Myers and Ko 2003] Brad Myers and Andrew Ko. "Studying Development and Debugging to Help Create a Better Programming Environment", For the: CHI 2003 Workshop on Perspectives in End User Development

[Nardi, 1993] Bonnie A. Nardi, A Small Matter of Programming: Perspectives on End User Computing, MIT Press, 1993.

[OMG 2004] Unified Modeling Language: Superstructure, version 2.0, http://www.omg.org, 2004.

[Praehofer and Kerschbaummayr, 1999] Praehofer, H., Kerschbaummayr, J.: Development and Application of Case-Based Reasoning Techniques to Support Reusability in a Requirement Engineering and System Design Tool. *Engineering Applications of Artificial Intelligence*, 12, 1999, pp 717-731.

[Won, 2003] Won, Markus: "Supporting End-User Development of Component-Based Software by Checking Semantic Integrity", in: ASERC Workshop on Software Testing, 19.2.2003, Banff, Canada, 2003.

[Wöß, Löberbauer, Mössenböck 2003] Wöß, A., Löberbauer, M., and Mössenböck, H.: LL(1) Conflict Resolution in a Recursive Descent Compiler Generator. Proceedings of the Joint Modular Languages Conference (JMLC'03), Klagenfurt, August 2003, Lecture Notes in Computer Science.

[Zhu, Grundy, Hosking 2004] Zhu, N., Grundy, J.C. and Hosking, J.G., Pounamu: a meta-tool for multi-view visual language environment construction, In Proceedings of the 2004 International Conference on Visual Languages and Human-Centric Computing, Rome, Italy, 25-29 September 2004, IEEE CS Press, pp. 254-256.