Monaco: A DSL Approach for Programming Automation Systems[1]

Herbert Prähofer, Dominik Hurnaus, Roland Schatz,
Christian Wirth, Hanspeter Mössenböck

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University
A-4040 Linz / Austria
{praehofer, hurnaus, schatz, wirth, moessenboeck}@ase.jku.at

Abstract: In this paper we present the language Monaco, which is a DSL for programming event-based, reactive automation solutions. The main purpose of the language is to bring automation programming closer to the domain experts and end users. Important design goals therefore have been to keep the language simple and allow writing programs which are close to the perception of domain experts. The language Monaco is similar to Statecharts in its expressive power, however, adopts an imperative notation. Moreover, Monaco adopts a state-of-the-art component approach with interfaces and polymorphic implementations and it enforces strict hierarchical communication architectures which support the hierarchical abstraction of control tasks. We discuss the main design goals, the essential programming elements, and the visual program representation and illustrate how the language supports hierarchical abstraction of control functionality by an example application.

1. Introduction

Domain-specific languages (DSLs) are a proven approach to bring programming closer to application domains. DSLs focus on specific application domains, strive for presenting software in the notations of the domain experts, and allow a straightforward mapping of application concepts to software solutions. They have the capability to significantly improve the productivity and quality of software engineering in the focused domain. The ultimate goal of a DSL approach is to empower domain experts and end users by providing languages and tools that enable them to realize and adapt software systems by themselves.

In an ongoing project [20] we pursue a DSL approach in the domain of automation systems. We develop modeling and programming notations as well as tools that empower domain experts and end users to implement and adapt control programs in an intuitive and concise way. The background for this work is a cooperation with Keba AG

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

(www.keba.com), which is a medium-sized company developing and producing hardware and software platforms and solutions for industrial automation. Development of automation solutions is a multi-stage process involving different stakeholders at different stages of the automation process with different programming knowledge and capabilities as follows: Keba develops and produces a PC-based hardware and software platform with associated tool support, in particular a programming environment based on the IEC 61131-3 standard. The hardware and software platform enables the customers of Keba to realize automation solutions for their products. The customers of Keba are mainly OEMs of manufacturing solutions, like injection molding machines, painting robots for the automotive industry, material handling equipment, etc. Employees of OEMs, however, are often domain experts with limited software engineering capabilities. Moreover, an important part of the automation solutions are end user programming environments. In the domain, it is required that end users, which are the machine operators, are enabled to make adaptations to the control programs. This represents a particular challenge as the end users normally have no software development knowledge and, on the other side, have to make changes in a highly safety critical software system.

We have found that current modeling notations and languages in the automation domain, most notably the programming languages of the IEC 61131-3 standard [15] and others [5], do not satisfy the requirements of a programming language which can be used by domain experts or end users. Moreover, these languages lack many characteristics of state-of-the-art programming languages from a software engineering perspective. Other formalisms, in particular the widely adopted Statechart [13] formalism or the IEC 61499 standard [16], have the expressive power required and can be regarded as being up-to-date software engineering practices. However, those modeling approaches primarily target at software engineering experts. Moreover, in our investigations domain experts and end users clearly articulated a preference of flowchart-like notations over state-based notations.

Hence, we have defined a new DSL for event-based, reactive control programming – called *Monaco* (*MOdeling Notation for Automation COntrol*). The language Monaco is similar to Statecharts in its expressive power, however, adopts an imperative notation. It is designed with the goal that also domain experts and, in a limited way, end users are capable of reading, writing and adapting control programs. Monaco is specialized to a rather narrow sub-area of the automation domain, i.e., programming control sequence operations for manufacturing machines. The lower level continuous control layers and the higher manufacturing execution system (MES) layers are therefore out of scope. It is intended to cover the event-based, reactive control part of machine automation software only. Therefore, a continuous control system, typically realized in languages of the IEC 61131-3 standard or plain C, will form a lower layer which will be controlled, scheduled, and coordinated by the higher reactive layer implemented in Monaco. Its run-time infrastructure is a PC-based hardware with a real-time operating system.

In this paper we will present the main language concepts of the language Monaco. The outline of the paper is as follows. Section 2 presents the main features of the Monaco language as well as its visual representation. Moreover, equivalent Statechart structures

are presented. Section 3 demonstrates how hierarchical control programs can be realized using an example control program for an injection molding machine. Section 4 discusses differences of Monaco and Statecharts and UML-RT and other approaches. In Section 5 we present the current status of evaluation and Section 6 concludes with a summary and an outlook to future work.

2. The Monaco Language

In extensive discussions with domain experts of our industrial partners, we learned about the perception of domain experts and end users of automation machines as follows: (1) A domain expert perceives a machine as being assembled from a set of independent components working together in a coordinated fashion. (2) Each component normally undergoes a determined sequence of control operations. There are usually very few sequences which are considered to be the normal mode of operation, and those are usually quite simple. Complexity is introduced by the fact that those normal control cycles can be interrupted anytime by the occurrence of abnormal events, errors, and malfunctions. (3) Reactive behavior is intrinsically complex. Especially, realizing asynchronous event and exception handling in a concise way always represents a challenge. (4) The control sequences of the different machine components are coordinated at a higher level to fulfill a particular control task.

The design of Monaco is based on those findings and on the following ideas: (1) Although the behavioral model of the language is very close to Statecharts, an imperative style of programming is used. The language adopts proven concepts from imperative languages such as procedural abstraction, synchronous procedure calls, parameters, block structure, lexical scoping, and a Pascal-like syntax. (2) The main focus of the language is on event handling. Statements have been introduced to express reaction to asynchronous events, parallelism and synchronization, exception handling, and timeouts in a concise way. (3) Monaco pursues a component-based approach with strict modularization which allows a direct mapping of the machine structure to the software structure. (4) In contrast to many other component-based approaches in this domain, Monaco pursues hierarchical control architectures of subordinate and superordinate components where the superordinate is of full control over the tasks performed by its subcomponents. The superordinate component composes and coordinates the behavior of its subordinates and provides abstract and simplified views to its superordinate. Between sibling components only limited event signals for task synchronization purposes are allowed. How the sibling components can interact is again determined by the common superordinate. (5) The setup of Monaco programs is done in a separate configuration phase prior to execution, i.e., the entire system is statically configured. (6) Besides a text-based notation, there is a second, visual representation of Monaco programs, which is usually preferred by domain experts and end users.

In the following we present the main programming elements.

2.1. Component approach

Interface declarations (Figure 1) are used for defining the static contract between components and their clients and hence have a similar purpose as interfaces in modern object-oriented languages. However, interfaces in Monaco account for the hierarchical communication architecture of control programs. On the one hand, an interface defines the externally visible operations of a component in the form of *routine* declarations. Those represent the operations a superordinate will be able to perform. On the other hand, an interface defines how a component will provide feedback about the fulfillment of its control tasks by specifying *events* it will signal and *functions* it provides for accessing run-time state properties of the component.

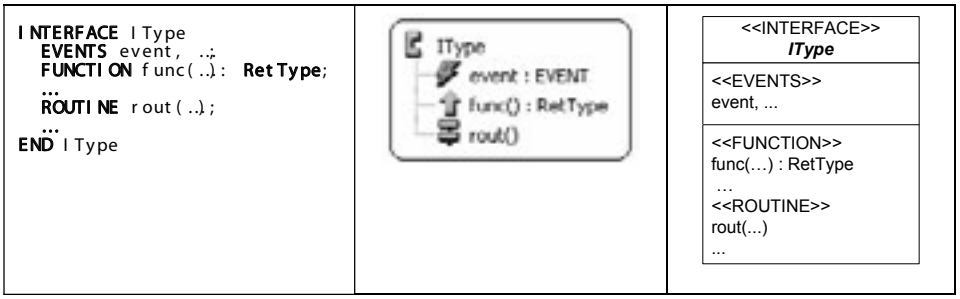

Figure 1. Interface declaration in Monaco (left), visual representation (middle), and UML (right)

Interfaces are implemented by *components* (Figure 2), i.e., components have to implement the routines, functions, and events defined in the interfaces. In addition, a component has *parameters*, which are run-time constants used to configure a component, and internal state *variables*. A component declares *subcomponent variables* which can hold references to subcomponent instances. Interface types are used in the subcomponent variable declarations. In a setup phase, Monaco components need to be instantiated and the component/subcomponent relation needs to be established. The component/subcomponent structure forms a tree-like hierarchy.

A function implementation in a component is similar to functions in procedural programming languages, e.g. Pascal. They return run-time state properties of components. Routines are used to implement control algorithms and therefore constitute the central programming elements of components. Established language constructs from structured programming languages like parameters, block structure, lexical scoping, loops, if-statements etc. are used. Additionally, special programming constructs for parallel execution tasks and reactive behavior with semantics similar to Statecharts are provided (see below).

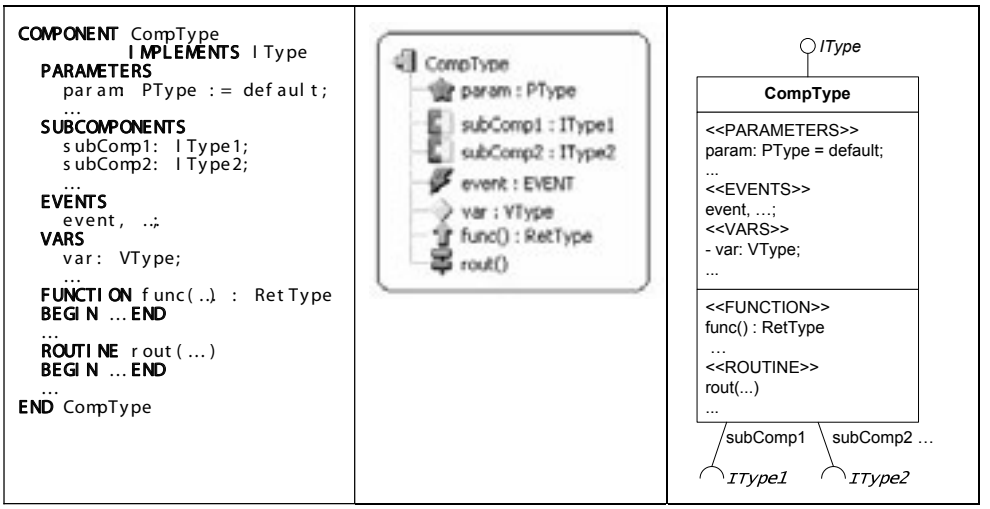

Figure 2. Component with provided and required interfaces in Monaco (left), visual representation (middle), and UML (right)

2.2. Reactive system programming

The WAIT statement is provided to suspend the execution of the current execution thread until a specified condition is satisfied. Any Boolean expression can be used. Compared to Statecharts, a WAIT corresponds to a state node with the condition as the triggering event (Figure 3).

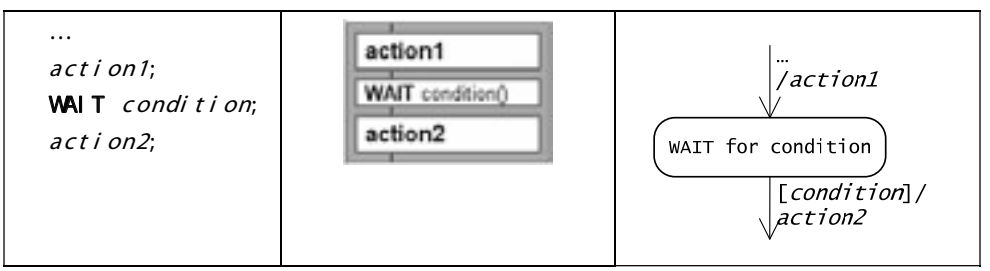

Figure 3. WAIT statement, visual representation, and equivalent Statechart model

ON handlers are used to handle events which can occur asynchronously to normal, sequential program execution. ON handlers specify an arbitrary event condition and are attached to BEGIN/END blocks (Figure 4). Their meaning is that, whenever the condition of the ON handler becomes true while program execution is within the BEGIN/END block, the block is left and the statement sequence of the ON handler is executed. For ON handlers to be meaningful, the guarded BEGIN/END block has to have blocking statements, i.e., WAIT statements, where program execution gets suspended and the asynchronous event handling can occur.

ON handlers in Monaco are analogous to OR states and their transitions in Statecharts. Figure 4 shows the relation. The OR state groups the states, e.g. the blocking WAIT state-

ments, and transitions within the BEGIN/END block. The transition leaving the OR state is labeled with the condition of the ON handler. An arbitrary sequence of statements can follow. ON handlers have interruptive behavior, therefore program execution continues with the first statement after the block.

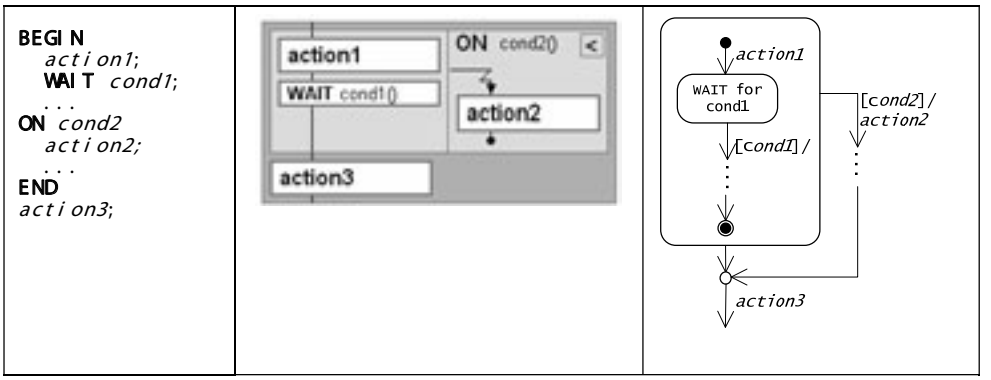

Figure 4. ON handler, visual representation, and equivalent Statechart model

The PARALLEL statement is used for creating multiple concurrent execution threads. Each parallel execution thread consists of a statement or a statement block. As soon as all parallel execution threads have terminated normally, program execution continues after the PARALLEL statement. The PARALLEL statement has the semantics of the AND state in Statecharts, see Figure 5.

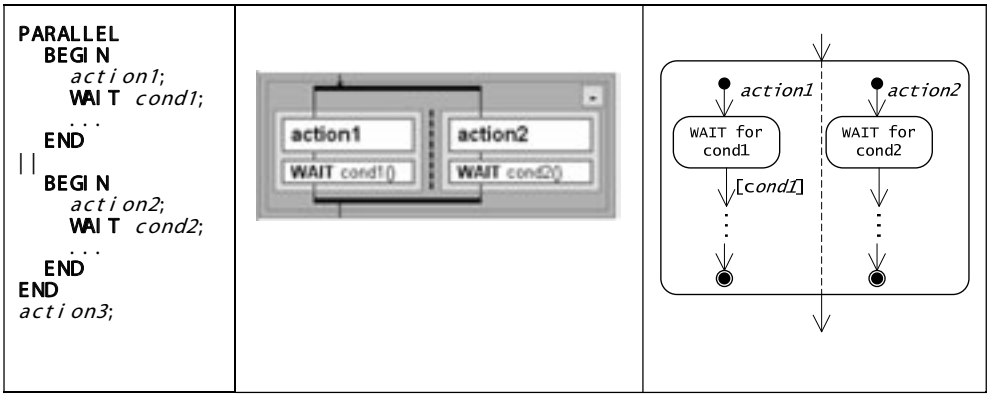

Figure 5. PARALLEL statement, visual representation, and equivalent Statechart model

Although Monaco allows using arbitrary Boolean conditions as event triggers, *event* signals are provided. Those are similar to the event triggers in Statecharts or the signal concept in Esterel [5].

3. Example Control Program

In this section we will demonstrate programming in Monaco with an example automation program for an injection molding machine. It is a reimplementation of the reactive part of an existing control program for an injection molding machine, originally implemented in the IEC 61131-3 [15] standard languages. This example should especially show how the language supports hierarchical abstraction of control functionality.

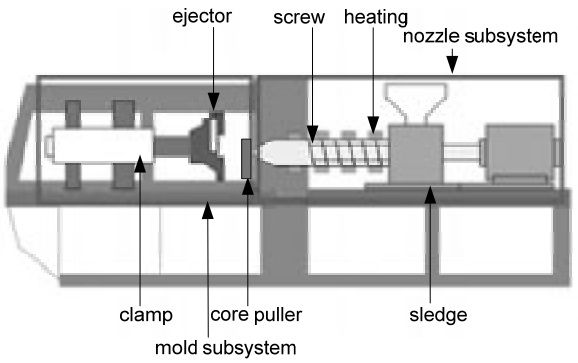

Figure 6. Injection molding machine

Figure 6 shows the structure of the sample molding machine. There are two main components in the machine: the mold subsystem with the clamp, the ejector, and a core puller; and the nozzle subsystem that is mounted on a sledge with the material funnel, the heating system, and the screw for injection. Finally, the ejector ejects the finished parts out of the mould.

Component hierarchy

The component hierarchy of the control program resembles the structure of the real machine (Figure 7). This leads to a direct mapping from the problem structure to the solution structure. On top, the Supervisor component is responsible for encoding the overall control cycles. It knows different operation modes, e.g. full automatic or half automatic. It relies on and coordinates several subcomponents corresponding to the different machine subsystems. The components for nozzle and mould are further decomposed according to the different parts of the subsystems. At the bottom of the hierarchy there are components for interfacing with the hardware or lower level control layers. Figure 7 shows the higher level components on the left hand side and the low-level components on the right hand side.

Components at different hierarchy levels typically serve different purposes as follows: (1) Components at the bottom are used for interfacing with the hardware or lower control layers. They usually read and write basic system variables. (2) Components at the first level compose those primitive operations into elementary control routines and supervise their execution. (3) Higher up in the hierarchy there are several coordination components which coordinate and supervise the operations of several subcomponents.

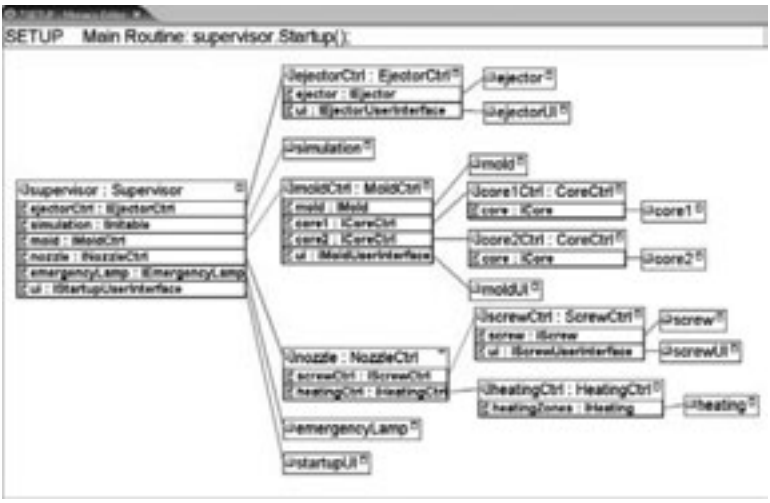

Figure 7: Component hierarchy of an injection molding program

Interface to hardware and continuous control layers

In the example program, the components forming the leaves of the component hierarchy are native Java classes building the interface to a simulator which simulates the real machine and the continuous control layer. The native components implement a Monaco interface which represents the interface for the superior components (there is a direct mapping of routines, functions and events to equally named Java methods). The following code snippet (Figure 8) shows the interface definition for the core puller component ICore. The interface defines elementary routines to set system variables to start and stop insertion and removal of the core and a function giving the current position of the core puller.

```
INTERFACE ICore
  FUNCTION position() : REAL;

  ROUTINE startInsert();
  ROUTINE stopInsert();
  ROUTINE startRemove();
  ROUTINE stopRemove();
END
```

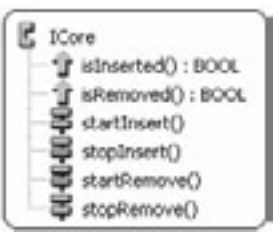

Figure 8: Interface ICore

First level control components

The components residing in the hierarchy level directly above the native components use those interfaces to compose elementary operations into basic task routines. For example, the CoreCtrl component has the native component as its single subcomponent (Figure 9). It defines routines to insert and remove the core and one to immediately stop all movements. However, besides defining the basic sequence of actions, those routines also

check for the correct execution of control tasks and correct reactions from the subordinate by ON handlers. In this way, the component provides routines to its superordinate having all possible errors already checked and reported as error events.

```
COMPONENT CoreCtrl IMPLEMENTS ICoreCtrl
    PARAMETERS
        coreStartedTimeout : INT := 200;
        coreInsertTimeout : INT := 1400;
        coreInsertPos : REAL := 0.6;
        coreRemovePos : REAL := 0.8;
    SUBCOMPONENTS
        core : ICore;
    EVENTS
        error;

    FUNCTION isRemoved() : BOOL
    BEGIN ... END isRemoved

    FUNCTION isInserted() : BOOL
    BEGIN ... END isInserted

    ROUTINE stop()
    BEGIN ... END stop

    ROUTINE insert()
    BEGIN ... END

    ROUTINE remove()
    BEGIN ... END
```

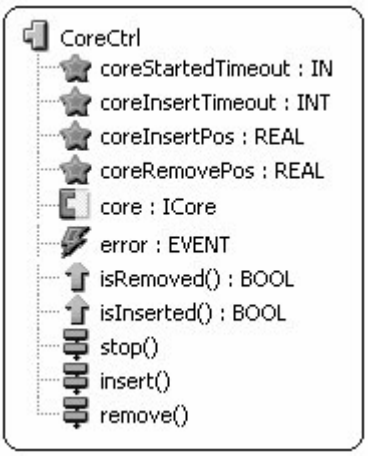

Figure 9: Component CoreCtrl

The following code snippet (Figure 10) demonstrates this approach with the insert routine (only the graphical representation is shown). First, startInsert is called from its subcomponent core which will set a hardware signal, thereby starting the insertion process. Next, a reaction from the isRemoved signal is expected. If this sensor does not go to false within a given (short) time period, a fault in the insertion process or a faulty sensor has to be assumed; so the process is stopped and an error event is fired. Next, execution waits for the isInserted signal to become true and then stops the insertion process. Again the process is supervised by two ON handlers. The first checks that the isRemoved signal does not switch to true again (which might be caused by a faulty sensor). The second checks that the reaction of the isInserted signal occurs in time. In both error cases the process is stopped and the error event is fired.

Figure 10: Routine insert of CoreCtrl

Coordination levels

As the next higher level component the MoldCtrl component is discussed. This component has to coordinate the operations of the core and the clamp subcomponents. The example routine shown in Figure 11 exemplifies this by the close routine. Its purpose is to control the process of closing the clamp and inserting two cores, all of which should occur in parallel.

Figure 11: Routine close of MoldCtrl

Finally, the routine automatic (Figure 12) represents the overall automatic control cycle of the machine. This is usually the level which is also presented to end users working directly at the machine. The operation cycle of the machine gets clearly represented in the code. In the inner control loop first the mold is closed. Then injection is performed and in parallel the cooling time is checked. Then, in parallel activities, the mold is opened, new material is inserted into the screw (nozzle.plasticize), and, after the mold has been opened to a determined point, the molded piece is ejected.

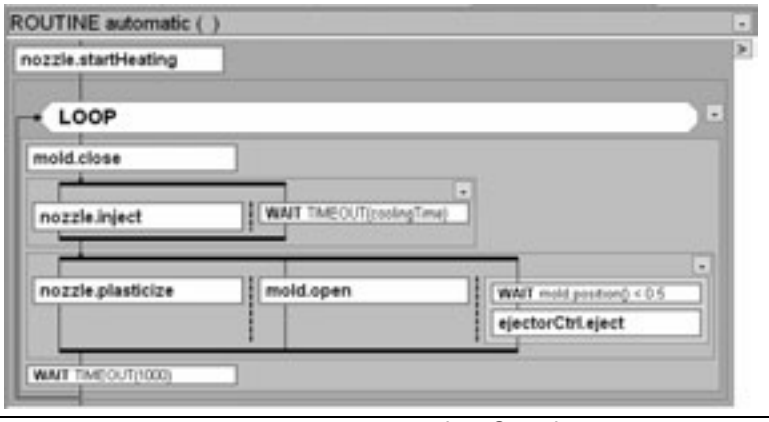

Figure 12: Routine automatic of Supervisor

4. Related Work

The design of the language Monaco has been influenced by several different languages and tools for reactive embedded systems modeling and programming, most notably by the synchronous languages Esterel [5] and its derivatives [7], [4], [8], the Statecharts formalism [13] and related systems [6], [2], and UML/RT [21] and other component-based modeling approaches [16], [17], [19], [23]. In the following we give a short comparison to Statecharts, Esterel and the UML/RT component approach.

4.1. Comparison to Statecharts

The Statecharts formalism [13] together with implementations like AnyLogic (www.xjtek.com) and Rhapsody (www.ilogix.com) have been the primary influencers on the Monaco language. As outlined in Section 2, Monaco has statements which are analogous to Statecharts' OR and AND states. However, Monaco is more restrictive in respect to control flow, i.e., Monaco enforces a hierarchical block structure with lexical scoping and arbitrary program jumps (*gotos*) are prohibited (which are possible in Statecharts because no restrictions are defined for source and destination states of transitions). The main difference between Monaco and Statecharts is the imperative notation. One could claim that Monaco is an imperative version of an essential subset of Statecharts. We argue that the imperative notation makes Monaco programs more suitable for domain experts. Our experiences with domain experts have shown that they are not used to thinking in states and state transitions, but usually perceive the control behavior as coordinated sequence of machine operations. Our visual representation of control routines compares to the Statecharts visual formalism. However, we argue that the structured approach of Monaco with block structure and a clear separation of normal control flow and asynchronous event handling results in better arranged diagrams.

4.2. Comparison to Esterel

A second important influencer has been Esterel [5]. Esterel is a synchronous language [3], [12] with an imperative syntax for describing event-based, reactive control of embedded systems. Esterel programs consist of collections of nested threads which communicate exclusively through event signals. As a novel feature, Esterel has introduced statements for pre-emption and exception handling.

Monaco relates to Esterel as it is also an imperative language for reactive control programming. The WAIT-, PARALLEL-, and ON-statements of Monaco have semantics similar as the await-, par- and abort-statements of Esterel. Moreover, the execution model is similar to Esterel; actually, it shows much similarity to the SugarCubes [8] and Junior systems [14], which are synchronous machine implementations strongly influenced by Esterel.

However, Monaco also differs significantly from Esterel and its derivatives in many respects. First, Monaco's event handling is much more general. In Esterel, thread communication and event synchronization is possible exclusively through signals

(signals are analogous to the EVENT construct in Monaco). In Monaco, any Boolean expression can be used to specify an event trigger. We argue that this is an important enhancement which makes control programming much simpler and more user-friendly. Moreover, Monaco syntax is simpler than Esterel. It has less control structures and shows stronger similarities to common imperative programming languages.

4.3. Comparison to component-based approaches

Our component approach has strongly been influenced by UML/RT [21] and comparable component approaches for embedded systems, e.g. [16], [17], [19], [23]. The main difference to those component approaches is that in Monaco communication is between superordinate and subordinate components only. In the more traditional approaches, components are coupled by connecting input and output ports, i.e., components at the same hierarchy level are connected. Although components can be arranged in a hierarchical manner, communication between components is not hierarchical. Moreover, in most approaches components can only communicate through simple signals whereas Monaco uses synchronous routine calls. We argue that the parent-child communication structures supported by the Monaco language are an important feature of our approach which facilitates the required hierarchical abstraction and simplification of control programs.

Moreover, our hierarchical communication scheme shows similarities to hierarchical scheduling schemes for real-time system recently emerged [18], [11], [24]. Similar to our approach, they employ event-based methods for modeling control of task execution in a hierarchical manner. However, their focus is on task scheduling issues and only limited language support is provided. Our approach concentrates on the language expressiveness and does not care for task scheduling in the lower control layers currently, which is considered to be taken care within the lower control layers themselves.

5. Evaluation

The language has been developed and tested relying on a set of typical examples, starting from simple machines to small manufacturing cells. In particular, we have re-implemented two existing automation solutions from our industrial partner, both implemented currently in the IEC-61131-3 languages Structured Text and Sequential Function Charts. The first is an automation solution for an injection molding machine. The example in Section 3 has been taken from this system. The other is an existing solution for the paint supply system of a painting robot used in the automotive industry. Both are typical solutions from the domain of our industrial partner, where end user programming environments are strongly needed. In both programs we have covered the event-based reactive part of the systems and tested them with a simulator.

Results of those two case studies were very encouraging. It was possible to reduce the code size of the systems to a fraction of the original one. The original injection molding

solution has in total more than 30 000 lines of code (Structured Text), where about 1/4 is for the pure reactive control part (which is a rather conservative approximation). Our solution has exactly 942 lines of code. The abstract control part for the paint supply system typically consists of over 20 medium-sized Sequential Function Charts plus several time-based programs (a proprietary programming model used in this type of solution). Our Monaco program has 1403 lines of code. However, it is difficult to compare SFCs to textual code, because of their different nature (SFCs are a graphical notation with actions coded in Structured Text).

As a next step in the project we plan detailed validation studies which should show two properties: (1) that the language is expressive to represent complex control programs in a concise way and (2) that the language is intuitive for domain experts and end users. To show the first we are looking for typical programming challenges in the domain and find patterns which show how to solve those. In particular, we want to show how the hierarchical component approach should be utilized. To show the second property we are planning to make usability studies, relying on manufacturing engineering students. In particular, we want to compare our visual representation to other visual languages in the domain, like Statecharts or Sequential Function Charts.

6. Summary and Outlook

In this paper we have presented the domain-specific language Monaco and its visual programming environment. We have discussed the objectives of the language, the main design decisions, the most important language features, and the visual representation scheme. The language Monaco is the core of a larger research project with the overall goal of making machine control programming more reliable and more adequate for the domain experts, and to give even end users limited programming capabilities. We regard the language features of Monaco, in particular its imperative notation, its hierarchical component and communication architecture for control abstraction, the asynchronous event handling mechanism as well as the static nature of Monaco programs as important features making such an approach feasible.

Also important to note is what has been omitted in the Monaco language. For example, Monaco language is intentionally *not* object-oriented. Language features like inheritance and dynamic binding are well accepted as powerful, high-level programming concepts. However, they introduce a complexity which stands in conflict with the requirements of bringing programming capabilities to domain experts. For similar reasons, Monaco does not support pointers, references, and a **new** operator (component instantiation only can be done at setup time).

We have developed an integrated development environment (Monaco IDE) based on the Eclipse RCP. The core of the Monaco IDE is a visual editor which displays control routines automatically from source code. Experience has shown that the visual representation is easily understood by domain experts. Currently we only have a virtual machine for Monaco programs in Java, which, of course, does not have the required real-time capabilities. An implementation of a compiler framework which allows translation

to different target platforms, in particular the IEC-61131-3 platform of our industrial partner and plain C, is underway.

Another important future extension of Monaco is the support of protocol contracts as well as their exploitation in guiding and constraining domain experts. The idea is simple: protocol contracts will define the valid sequences of operations and the legal feedback of a component defined in the form of finite state automata [10]. A component has to guarantee that it will not violate the contracts of its subcomponents. This will allow us to statically check that a sequence of operations is correct and will result in a semantically meaningful and complete program. Standard automata simulation and model checking methods [9] represent the underlying theoretical basis for this approach.

References

[1] Aldrich, J., Chambers, C., and Notkin, D. 2002. ArchJava: connecting software architecture to implementation. In Proc. of the 24th ICSE (Orlando, Florida, May 2002). ICSE '02. ACM Press, NY, pp. 187-197.

[2] André, C.: Representation and Analysis of Reactive Behaviors: A Synchronous Approach. CESA'96, IEEE-SMC, Lille(F), July 9-12, 1996.

[3] Beneviste, A., Caspi, P., Edwards, S., Halbwachs, N., Guernic, P., and De Simone, R.: Synchronous Languages 12 Years Later. Proc. of IEEE, 91 (1), Jan 2003.

[4] Benveniste, A., LeGuernic P., and Jacquemot Ch. Synchronous programming with events and relations: The SIGNAL language and its semantics. Science of Computer Programming, 16:103–149, 1991.

[5] Berry, G. and Gonthier G., The Esterel Synchronous Programming Language, Science of Computer Programming 19 (1992), pp. 87–152.

[6] Bond, G. W.: An Introduction to ECharts: The Concise User Manual. AT&T Labs–Research, August, 2006 http://echarts.org/.

[7] Boussinot, F.: Reactive C: An Extension of C to Program Reactive Systems. SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(4), 401–428, April 1991.

[8] Boussinot, F. and Susini, J-F.: The SugarCubes tool box - a reactive Java framework, Software Practice and Experience, 28(14), December, 1998, pp. 1531--1550.

[9] Clarke, E.M., Grumberg, O., and Peled, D.: Model Checking. MIT Press, 2000.

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

[11] Ghosal, A., Henzinger, T.A., Kirsch, C. M., Iercan, D. and Sangiovanni-Vincentelli, A.: A Hierarchical Coordination Language for Interacting Real-Time Tasks. Proceedings of the Sixth Annual Conference on Embedded Software (EMSOFT), ACM Press, 2006.

[12] Halbwachs, N.: Synchronous Programming of Reactive Systems. Kluwer Academic, 1993.

[13] Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8:231–274, July 1987.

[14] Hazard, L. and Susini, J-F. and Boussinot, F.: The Junior reactive kernel -- Inria Research Report, RR-3732, July, 1999.

[15] IEC, IEC 61331-3, Programmable controllers - Part 3: Programming languages. http://www.iec.ch/, 2003.

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

[17] Li, S., Wu, J., and Hu, Z.: A Contract-Based Component Model for Embedded Systems. In Proceedings of QSIC'04 (September 08 - 10, 2004). QSIC. IEEE Computer Society, Washington, DC, 232-239.

[18] Lipari, G., Gai, P., Trimarchi, M., Guidi, G., Ancilotti, P.: A hierarchical framework for component-based real-time systems, Electronic Notes in Theoretical Computer Science (ENTCS), vol. 116, 2005.
[19] Nierstrasz, O., et al.: A Component Model for Field Devices. Proc. First International IFIP/ACM Working Conf. on Component Deployment, ACM, Berlin, Germany, June 2002.
[20] Prähofer, H., Hurnaus, D., Mössenböck, H.: Building End-User Programming Systems Based on a Domain-Specific Language. 6th OOPSLA Workshop on Domain-Specific Modeling, Portland, Oregon, USA, October 2006.
[21] Selic, B. and Rumbaugh, J.: Using UML for Modeling Complex Real-Time Systems. ObjectTime Limited, 1998.
[22] Unified Modeling Language: Superstructure, version 2.0, http://www.omg.org, 2004.
[23] van Ommering, R.; van der Linden, F.; Kramer, J.; Magee, J.: The Koala component model for consumer electronics software. Computer 33, (3), Mar 2000 Page(s):78 – 85.
[24] Yang, G., Li. H., Wu, Z.: SmartC: A Component-Based Hierarchical Modeling Language for Automotive Electronics, dasc, pp. 203-210, 2006.