2013 European Control Conference (ECC)
July 17-19, 2013, Zürich, Switzerland.

Paradigms for Unified Runtime Systems in Industrial Automation

Sten Grüner* and Ulrich Epple

Abstract— **Borders between different levels of the industrial automation pyramid become more and more blurry because of the increasing production flexibility and plant complexity. The arising challenges can be overcome by the use of the same runtime systems and programming paradigms for distributed automation at the manufacturing execution system, the process control and the field device levels. Paradigms for unified runtime system are proposed involving message-based communication, co-existence of cyclic/event-based execution semantics of function block diagrams, a flexible hierarchical tasking concept, development workflow and non-functional system properties.**

I. INTRODUCTION

The plant automation pyramid [1] depicted in Figure 1 classifies tasks of the industrial automation into different hierarchical levels. We assume a four level pyramid: starting from the field device level, the pyramid contains the process control level, the manufacturing execution system (MES) level, and, on its top, the enterprise resource planning (ERP) level. On the field device level there are devices with restricted hardware. The process control level is usually represented by programmable logic controllers (PLCs), while the MES and ERP levels are governed by PC-based systems.

Different programming paradigms and communication protocols are used for the automation tasks on different pyramid levels. The field devices typically have proprietary firmware with fixed functionality and are connected via field buses to the process control system (PCS). These devices have to be parameterized manually either directly at the controller's embedded interface or remotely via the field bus. The process control level of the pyramid is the focus of the classical industrial automation. The programming languages from the IEC 61131-3 [2] are mainly used at this level and the inter-PCS communication is possible by using protocols by different vendors (including the communication via field buses). The lion's share of the available MES systems is also proprietary and composed of different packages providing functions like product tracking or collection of the production data. The de-facto standard for the communication between PCS and MES levels is still OPC, providing interoperability between MES systems and PCSs of different vendors.

The tendency towards system openness, interoperability and advanced user-configurable functions can be observed on every level of the pyramid. Just an example: modern field devices, referred to as "smart-sensors", offer a possibility to execute user-defined functions. The process control level historically makes use of programmable logic controllers that are shifting from closed proprietary systems to open information servers. Simultaneously, the administrative functions of the MES system get a real-time production role that is relevant for the planning and optimization of the production across the PLC and package unit boundaries.

Additionally, the need for portable, dynamically reconfigurable and interoperable automation solutions across the pyramid levels emerges due to a more flexible production, the growing quality requirements and the significant increase of the plant's complexity. In order to close this gap, a concept of a software system architecture for user-defined automation functions is proposed that matches the mentioned challenges. The applications of the architecture incorporate the upper part of the field device level (the user-definable functions of smart-sensors), the whole process control level, and reach the lower part of the MES level where production-relevant user-defined control or data collection takes place. The hardware-specific parts of the sensor firmware, as well as the higher MES functions, including the connection to the ERP system, are out of the scope of the presented concept.

A unified runtime has the following advantages when compared to the state-of-the-art solutions.

- Previously defined functionality of one automation level can be encapsulated and directly reused on a different level, e.g. a time-proven controller from a PLC can be embedded into a smart actuator or an optimization function from the process control level can be lifted up to the MES level to optimize distributed applications.
- Software components can not only cross the boundaries of automation levels, but can also be moved/distributed between devices of different vendors enabling cloud services in the automation domain.

The contribution of this work is a runtime system architecture that makes use of function blocks (FB) as ba-

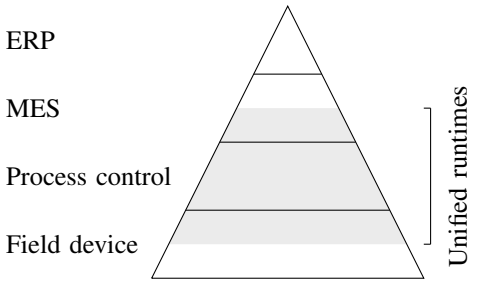

Fig. 1. Automation pyramid and the scope of unified runtimes.

*This author was supported by the DFG Research Training Group "AlgoSyn" (Algorithmic Synthesis of Reactive and Discrete-Continuous Systems), German Research Foundation grant DFG GRK 1298

Authors are with the Chair of Process Control Engineering, RWTH Aachen University, 52056 Aachen, Germany
{s.gruener,epple}@plt.rwth-aachen.de

sic program organization units. In contrast to IEC 61131-3 runtime systems, the control flow of included elements is maintained solely by the encapsulating block ensuring deterministic migration of FB-based applications and their nesting. Furthermore, we address distributability of applications by introducing the concept of concurrency-safe standalone components that define predetermined breaking points for application distribution and an optional message-based communication model. The combination of the mentioned techniques enables consistent software development and deployment across levels of the automation pyramid and borders of physical systems.

The remainder of the paper is structured as follows: in Section II an overview of the proposed system components is given, in Section III the different principles of flow control and tasking are introduced. Section IV reviews the proposed inter-component communication, Section V describes the development process and non-functional aspects of components. Finally, Sections VI and VII give an overview over the related work and an outlook, respectively.

II. COMPONENT ARCHITECTURE

The proposed architecture uses different techniques for the system level (programming in the large) and the level of single components (programming in the small). The system architecture is introduced in a bottom-up fashion at first.

A. Function Block

The presented system makes use of the data flow oriented block-diagram systems that are widely accepted in the automation community and used in both the IEC 61131-3 and the IEC 61499 [3] programing standards. The core element of this concept is the FB, including a number of input and output ports and connections between the ports that transfer the data between function blocks.

A function block is a generic logical encapsulation unit from the perspective of the component-based software development and may be used in different ways for a black or a white box abstraction of a software component. A native FB is a black box that has been instantiated from a predefined FB type. A function block may have an internal state or not, the latter function blocks are also denoted as "functions" (cf. [2]). An aggregated FB is a white box encapsulating either a function block diagram (FBD, also referred to as function block network or continuous function chart), describing a continuous function, or enclosing a sequential state chart (SSC) which is a finite state machine-like structure.

Sequential state charts are particularly worth mentioning due to the key role of discrete procedures at the MES level. A language for this purpose called sequential function charts (SFC) was included in the IEC 61131-3. Unfortunately, there is no standardized semantics of SFC because of the standard ambiguity [4]. The semantics depends on the particular runtime vendor. Another disadvantage of SFC is its relatively complex syntax (e.g. action qualifiers). These disadvantages promote the development of alternative languages for describing discrete procedures. These languages are mostly inspired by the UML-Statecharts modeling framework [5]. In addition to "syntactic sugar" for SFC, these alternatives provide rigorously defined semantics which enables application migration and the use of formal methods for the program analysis [6].

The core requirement for a function block is to have no implicit data dependencies besides the links going from and to its data ports. This requirement is the key for the portability and the distribution of FB-based applications. FBs of the IEC 61131-3 that are programmed in structured text (ST) fail to fulfill this requirement if VAR_EXTERNAL and VAR_GLOBAL language constructs are used allowing data manipulation to bypass block's ports.

B. Function Block Diagram

Function block diagrams are a composition of FBs via signal-based connections. FBDs are the basic building blocks for programming in the small. The execution order of included elements is governed by the FBD itself. The different execution methods are discussed in Section III-A. Since any FBD can be considered as an aggregated function block, it has to provide a "frame" with data inputs and outputs that define the FBD's interface.

C. Standalone Component

Standalone components (SCs) are the basic building blocks for programming in the large. The following properties hold for SCs: they consist of aggregated function blocks (FBDs and SSCs) and further standalone components, they are triggered cyclically in a concurrency-safe context and they have full control over the tasking of the included components. SCs provide a well-defined set of input and output ports for signal-based communication as well as mailboxes for the message-based communication. From the system design point of view, standalone components may encapsulate agents of a multi-agent system [7] that have a special task and aim for special goals.

D. Application

An application provides a logical name space for a group of standalone components fulfilling a higher-level goal. Our concept of an application is very close to the IEC 61499 application except for the distribution policy. The standard allows separation of FBDs on arbitrary points and the distribution of these parts. Our approach reduces this separation to predetermined breaking points around concurrency-safe standalone components.

In contrast to a standalone component, an application does not have a "frame" with connection ports. However, a dedicated SC may exist that represents the whole application (component group). The interface of this representative component defines the interface of the application in this case. Otherwise the union of the interfaces of the contained SCs is considered as application's interface.

Hierarchical application composition with name space nesting becomes interesting in the context of virtualization which is a topic of future research.

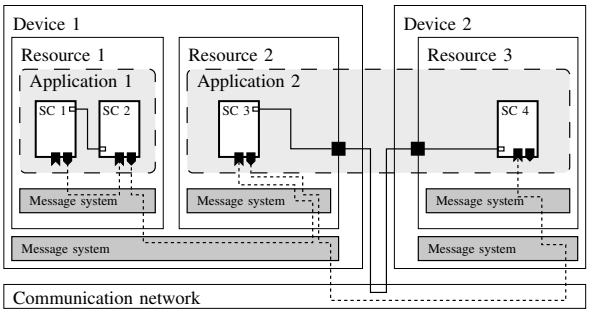

Fig. 2. Architecture component overview including signal and message-based communication.

E. Resource

Resources are the next aggregation level representing a thread-safe execution environment for standalone components including logical infrastructure for the communication with the process and other resources. Different resources are assumed to run concurrently.

F. Device

A device provides a hardware interface between the physical and the virtual world and may encapsulate several resources. The encapsulated resources run in parallel either on the software (scheduling of processes) or on the hardware level (different CPUs or CPU cores).

A component overview can be found in Figure 2. Application 2 is distributed between different resources and devices. Solid and dashed lines denote signal communication between ports and message oriented communication between incoming and outgoing mailboxes, respectively (for description of the communication model consider Section IV).

III. RUNTIME CONCEPT

The history of automation goes back to standalone analogous devices performing continuous data processing. In fact, cyclic scan implemented in classical PLCs brings this philosophy onto digital hardware. The data processing function of a cyclic block can be considered continuous if the cycle time is much smaller than the time constants of the controlled physical process and communication delays. Cyclic scan is a classical concept of FB and FBD semantics from the IEC 61131-3 and became discredited in recent times most notably for its low performance compared to event-based concepts and the problems arising from different block scan order in different runtime implementations. However, strong sides of the cyclic scan approach are determinism and the ability to keep constant response times even in the worst case scenario. Determinism is a main argument to use cyclic execution semantics for the IEC 61499 [8] too.

A paradigm of intelligent, standalone components at the level of programming in the large is postulated. These components have inner states and should therefore not only triggered at external calls. Thus, cyclic execution is chosen at this program organization level. It is important to note, that this decision does not contradict the event-based philosophy, since no restrictions on the internal implementation of a component are posed. Below we discuss the different control flow concepts inside of a cyclically triggered component.

A. Control Flow Types of SCs and FBDs

FBDs and SCs maintain the control flow of the nested sub-components. Achieving this increases the portability and enables migration of components by making them independent of the executing resource. The following control flow types are supported.

1) Call-Based Control Flow: Calling functions from another library is common in general-purpose programming languages. If a block calls a function, its own execution is paused until the function provides a return value. While safely usable inside of one FB, function calls create implicit data dependencies when used between different FBs. For this reason the usage of function calls is only allowed inside of SSC-based function blocks describing discrete procedures.

2) Task List-Based Control Flow: Contained components of a component are triggered according to their order in the component's task list. After a sub-component has finished its execution, it returns the control back to the task list. In contrast to the IEC 61131-3 the task list is an artifact belonging to and encapsulated by the SC. This encapsulation allows the engineer to control the exact execution order of the components.

3) Event-Based Control Flow: In the event-based model a component triggers further component(s) after finishing its own execution by propagating an event through the fixed event-connections. While SCs incorporate the cyclic execution model with a fixed task list, the included aggregated FBs do not necessarily have to follow this philosophy. It means that although an FBD is triggered by the higher-level component cyclically, its own control flow may indeed follow an event-based model and e.g. skip the execution of some included blocks. Task-list-based and event-based execution may also be combined the other way round when an event-triggered FBD triggers components including a task tree.

Computing the worst case execution time of an FBD composed of FBs with defined maximal response times is simple if the task-based control flow is used. This task is more involving when using the event-based control flow semantics. Still, formal estimation of maximal execution paths in such FBDs is possible [9][10].

B. Real-Time Capability, Stability and Concurrency

A characterization of a real-time system system can be stated as follows "in a [...] real-time [...] environment, each task must be completed within a specified period of time after being requested" [11]. We assume that every component of an industrial automation system has a real-time capability. Certainly, different response times are required for different automation domains (plant vs. factory automation), automation levels (field vs. MES level) and even applications (chemical reactions vs. drives). Therefore, we propose a separation of the runtime into different zones where specified response bound times have to be satisfied. Different programming and

communication methods are suited for different response-time zones e.g. rigid signal-based communication might be unavoidable for "quick" tasks with low response bound while more flexible message-based communication can be used for tasks at the MES level (cf. Section IV).

Another requirement for industrial automation (especially for drive control applications) is the bounded jitter of execution triggers that is needed e.g. for equidistant signal sampling. While cyclic scan philosophy minimizes jitter (since the outputs are read and written during short periods at the begin and the end of the cycle, respectively), it is unknown to the authors whether there is some work on jitter analysis for event-triggered environments.

Response time and jitter are not the only dimensions to classify different programming philosophies. The treatment of concurrency is of paramount importance in the world of distributed automation. In the world of the IEC 61131-3 concurrency issues may only occur at the level of inter-system communication, since different PLCs can either be not time-synchronized or have variable/different cycle times. However, these issues are neither considered to be relevant in the practice nor addressed in the standard.

The design of the IEC 61499 is event-driven and compliant programs are therefore concurrency-safe by definition. An 61499 FBD is distributable to different physical devices while preserving its semantics and response time (which is only slightly affected by communication delays). Unfortunately, there is no standardized definition of the event-processing model of the IEC 61499 [12], therefore the event-processing depends on the runtime system vendor.

In order to preserve the advantages of both worlds, we propose a clear separation of program units in concurrency-safe and unsafe ones. For example, IEC 61131-3 programs have to be encapsulated into relatively small units running in a concurrency-safe environment. Such units can be distributed among concurrent systems if they are interconnected by a concurrency-safe communication system. Such a separation provides a trade-off between habitual programming and physical or logical distribution of the control functions.

IV. INTER-COMPONENT COMMUNICATION

Standalone components should be interconnected in a concurrency-safe way since they may be executed concurrently. Semantics of an application containing different standalone components should therefore not depend on their execution order. Although communication blocks introduced in the IEC 61131-5 [13] offer a possibility for such an interconnection in a signal-based way, they still require manual (re-)configuration. Only a loose, primarily message-based coupling of functional components allows the dynamic reconfiguration and distribution of industrial automation systems that is required for production systems of the future.

The accepted notion of data propagation in an FBD are the data links going from an output of a block to an input of another block. This communication model is called signal-based. The overall understanding of such a connection is an almost continuous, long-term data transfer between the data or event ports (although differences exist between different runtime implementations e.g. whether the data is pushed or pulled between the source and the target of a signal connection). The characteristic features of signal-based communication are the fixed source and target of a signal connection. Using signal-based communication together with distributed software is possible by using connection blocks defined in the IEC 61131-5. However, the distribution still requires explicit configuration of the communication channel.

The second communication type is the message-based communication. In contrast to long-term continuous signal-based data exchange, message-based systems are usually used to exchange non-periodical messages. Participants of a message system are equipped with an incoming and an outgoing mailboxes storing messages that are delivered and collected by a message service. Rather than having a fixed communication relationship between participants, messages can be addressed to an arbitrary participant of the system. Message-based communication involves a naming concept for participants like Domain Name Service (DNS) and a delivery/collection system transferring messages between different hardware devices over available networks transparently. The main advantage of the message-based communication is its suitability for distributed and dynamically reconfigurable systems due to a transparent communication over the borders of logical and physical components. The still unsolved problem of coupling flexible recipe execution to the basic automation functionality can be solved by the usage of message-based "control" requests in a syntactically correct and standardized way.

In the architecture presented in Section II, SCs have access to the message-based communication system. The collection and delivery service is available in every resource. The technical issues are beyond the scope of this work.

The signal-based communication between the SCs of Application 1 and 2 is depicted by solid lines in Figure 2. This communication link has to cross the borders of the resource in the second case and hence, the IEC 61131-5 communication blocks have to be created explicitly (black squares on the resource borders). Message-based communication is depicted by the dashed lines between the mailboxes of SCs. Message transport is handled by a message system depicted by a dark-gray box. This service delivers messages transparently and hence no additional configuration is needed when comparing Application 1 and 2.

V. COMPONENT DEVELOPMENT PROCESS

A. Development Phases

The automation process is clearly separated in two phases: (type-)development and engineering. Development is the creation of new function block types (classes). Engineering is the process of FB instantiation, parametrization and composition of the instances into FBDs and sequential state charts. An overview of this process is shown in Figure 3 where a wall separates the two phases. The lower part of the figure depicts native FBs (black boxes), sequential state charts, and nested FBDs inside of the function block diagram.

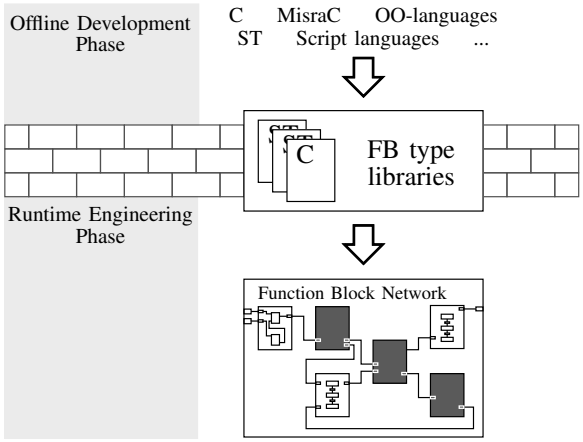

Fig. 3. Component development process with a clear separation between the development and the engineering.

Developing a complex function block type usually requires a deep expert knowledge depending on the block's function e.g. some advanced control algorithms as well as knowledge of programming languages for the implementation. Types are usually developed in general-propose programming languages depending on the purpose of the block and its non-functional requirements (cf. Section III-B). A function block type offers a suitable possibility to hide expert know-how due to its data encapsulation principles.

Composing a system from available components is a task of a system engineer who needs a broader knowledge about blocks from different domains and their interaction. From system's perspective, FB instances fulfill roles with certain requirements and can be handled without an understanding of their implementation as a result (black box components).

An abstraction mechanism is also possible at the engineering level defining white box components. In this case, an engineer reuses a manually-defined FBD as an FB type during system composition. SSCs can be used for defining discrete procedures that call functions from other FBs.

While adding new FB types or replacing a block type by a different implementation may be connected with a downtime of the runtime, the engineering process should be clearly performed during the runtime. This requires a far more flexible runtime system compared to the case of classical PLCs and includes dynamic management of tasks, dynamic reconfiguration of the communication up to a hot-swap of function blocks without a halt of production. On the inter-system and inter-level scope the transfer of software components between hardware devices and levels should be as transparent as possible. Dynamic reconfigurability is an essential requirement for flexible industrial automation systems ("cloudification" of automation).

In the engineering phase, function block types at hand are considered to be safe and reliable. The most harmful side-effect of composing function blocks is overloading the CPU or the memory of the runtime platform.

B. FB Programming and Verification

Development is considered to be far more "dangerous" (with respect to its impact on the runtime system) than engineering. There are many potential pitfalls in the type development due to degrees of freedom when using general-purpose programming languages. These include among others segmentation fails, memory leaks and potential infinite loops. Using a new function block type in the runtime system can be compared with installation of a new program in the world of personal computing. However, automation runtime systems are usually far less protected from malicious code than modern operating systems. Therefore, some trust and verification methods like program signing and program review with different compliance levels for the non-functional FB requirements are needed at this point.

Depending on the language in which a function block is developed, its semantics and worst case execution time can be verified by using formal methods e.g. model checking [14] [15]. Less costly possibilities for worst time and stability analysis are extensive testing or code reviews of the encapsulated code. A further, more pragmatic approach of solving these problems is the restriction of programming languages that can be used for developing FBs. One example of such a restriction is the MISRA C subset of ANSI C programming language that is widely spread in the automotive industry. For example, MISRA C minimizes the risk of segmentation faults by forbidding the usage of dynamic memory allocation. Another proposed pragmatic approach is the usage of sandbox-environments in which particular blocks are executed. As a proof of concept an open-source interpreter for JavaScript was embedded into an ANSI C runtime system. JavaScript code inside of a block has no direct access to the host memory and cannot produce any segmentation faults. Memory leaks are avoided by the built-in garbage collector of the interpreter. Maximal execution time is guaranteed by a watchdog that interrupts the interpreter after a predefined time. In case of the watchdog activation, predefined safe values are returned at block's outputs.

Another approach of developing stable FBs is the extensive usage of FBDs containing only basic operation blocks. Such aggregated blocks can be considered stable as long as basic FBs are verified or at least time-proven. One possibility to improve the performance of an aggregated block is a so called recompilation. An FBD consisting of FBs with a known source code, data links and a task list can be automatically transformed into a semantically equivalent native block, consisting of the code of single FBs that is executed in the correct order. The same holds true for an SSC block. Recompilation does not only increase the performance of an aggregated block. Due to the detour via the source code it provides a possibility to transform a white box into a black box and can be used for protecting the know-how.

Clearly, the overheads implicated by the sandbox approach or the extensive aggregated FB usage without recompilation are not suitable for hardware at the lowest levels of the automation pyramid. However, the tendency towards more

powerful hardware can compensate these in the near future.

VI. RELATED WORK

The related work on the runtime architectures can be divided into work regarding the aspects of programming in the large and aspects programming in the small.

Regarding programming in the small, the weak and the strong points of existing standards, the IEC 61131-3 and the IEC 61449 have been identified by the automation community. There are efforts to eliminate the disadvantages of both standards, e.g. object oriented extensions in the 3rd edition of 61131-3 and the work towards a standardized event model in 61499.

Simultaneously, runtime frameworks enabling the coexistence of the IEC 61131-3 and the IEC 61499 function blocks in one runtime system on the level of process control have been introduced in [16]. In the taxonomy of this paper our system is an IEC 61131-3 based solution.

A further research topic regarding the component architecture is the standardization of a language for the definition of discrete procedures cf. [4] and [5].

On the level of system architecture, there are proposals for introducing service oriented architectures (SOA) for distributable inter-component and inter-application communication in the industrial automation domain [17]. The message-based communication of introduced runtime can and should be used by SOA implementations in industrial automation.

A further approach for programming in the large are the multi-agent control systems. The usage of mailboxes for a loose inter-agent communication was presented in [18][19]. However, the implementation and evaluation were based on existing IEC 61131-3 PLC runtimes.

VII. CONCLUSION AND OUTLOOK

A runtime architecture enabling the integration of industrial automation applications across the levels of the plant automation pyramid was introduced. Different techniques are used when programming in the large (system architecture level) or programming in the small (component level). The proposed concept for the first category is the usage of cyclically triggered standalone components that are coupled by the means of message-based communication. This concurrency-safe communication philosophy creates the prerequisites for massive distribution of industrial automation applications. The proposed approach for the programming in the small is an extensive usage of function blocks/function block diagrams with nested control flow management for program organization. Further techniques ranging from the co-existence of task and event-based control flow concepts to the acceleration of the execution of function block diagrams by recompilation were discussed.

Flexible combination of proposed practices on the system and component level enables the usage of a unified runtime architecture across different levels of the automation pyramid and hardware of different capabilities and vendors. The key elements of this system are: a clear separation of concurrency safe and unsafe components, a possibility of loose coupling of components by the means of message-based communication, co-existence of IEC 61131-3 and IEC 61499 based execution semantics within of a flexible hierarchical tasking.

There is ongoing work on a runtime implementation realizing all the mentioned aspects. This runtime will be utilized in an experimental modular process plant with dynamic pipe and instrumentation structure that depends on the recipe/process and can be changed at runtime.

REFERENCES

[1] I. Harjunkoski, R. Nystrm, and A. Horch, "Integration of scheduling and controltheory or practice?" *Computers & Chemical Engineering*, vol. 33, no. 12, pp. 1909 – 1918, 2009.

[2] IEC 61131-3, *Programmable controllers - Part 3: Programming languages*. International Electrotechnical Commission, 2003.

[3] IEC 61499, *Function blocks*. International Electrotechnical Commission, 2005.

[4] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell, "A Unifying Semantics for Sequential Function Charts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2004, vol. 3147, pp. 400–418.

[5] D. Witsch, M. Ricken, B. Kormann, and B. Vogel-Heuser, "PLC-statecharts: An approach to integrate UML-statecharts in open-loop control engineering," in *8th IEEE International Conference on Industrial Informatics (INDIN 10)*, July 2010, pp. 915–920.

[6] D. Witsch and B. Vogel-Heuser, "PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering - Aspects on Behavioral Semantics and Model-Checking," in *Proc. of 18th IFAC World Congress*, 2011, pp. 7866–7872.

[7] VDI guidline 2653, "Multi-agent systems in industrial automation," 2010.

[8] P. Tata and V. Vyatkin, "Proposing a novel IEC61499 runtime framework implementing the Cyclic Execution semantics," in *7th IEEE International Conference on Industrial Informatics (INDIN 09)*, June 2009, pp. 416–421.

[9] A. Zoitl, *Real-Time Execution for IEC 61499*. ISA, 2009.

[10] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop, " Determining the worst-case reaction time of IEC 61499 function blocks," in *8th IEEE International Conference on Industrial Informatics (INDIN 10)*, July 2010, pp. 1104–1109.

[11] D. W. Leinbaugh, "Guaranteed response times in a hard-real-time environment," *IEEE Transactions on Software Engineering*, vol. 6, no. 1, pp. 85–91, 1980.

[12] K. Thramboulidis, "IEC 61499: Back to the well Proven Practice of IEC 61131?" in *17th IEEE International Conference on Emerging Technologies and Factory Automation, (ETFA 12)*, September 2012, pp. 1–8.

[13] IEC 61131-5, *Programmable controllers - Part 5: Communications* . International Electrotechnical Commission, 2000.

[14] S. von Styp, G. Quirós, and L. Yu, "Automatic test-case derivation and execution in industrial control," in *Proceedings of the Workshop on Industrial Automation Tool Integration for Engineering Project Automation (iATPA 2011)*, September 2011, pp. 7–12.

[15] J. Carlson, "Timing analysis of component-based embedded systems," in *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering (CBSE 12)*, 2012, pp. 151–156.

[16] A. Zoitl, T. Strasser, C. Sunder, and T. Baier, "Is IEC 61499 in harmony with IEC 61131-3?" *Industrial Electronics Magazine, IEEE*, vol. 3, no. 4, pp. 49–55, December 2009.

[17] H. Mersch, M. Schlütter, and U. Epple, "Classifying services for the automation environment," in *15th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 10)*, September 2010, pp. 1–7.

[18] A. Wannagat, *Entwicklung und Evaluation agentenorientierter Automatisierungssysteme zur Erhöhung der Flexibilität und Zuverlässigkeit von Produktionsanlagen, PhD Thesis*. Sierke Verlag, 2010.

[19] A. Wannagat and B. Vogel-Heuser, "Agent oriented software-development for networked embedded systems with real time and dependability requirements the domain of automation," in *Proc. 17th IFAC World Congress*, 2008, pp. 4144–4149.