Control Engineering Practice 21 (2013) 1078–1096

Contents lists available at SciVerse ScienceDirect

Control Engineering Practice

journal homepage: www.elsevier.com/locate/conengprac

Model-driven engineering of process control software – beyond device-centric abstractions

Tomaž Lukman [a,*], Giovanni Godena [a], Jeff Gray [b], Marjan Heričko [c], Stanko Strmčnik [a]

[a] Jožef Stefan Institute, Department of Systems and Control, Jamova 39, 1000 Ljubljana, Slovenia
[b] University of Alabama, Department of Computer Science, Tuscaloosa, AL 35487-0290, USA
[c] University of Maribor, Faculty of Electrical Engineering and Computer Science, Smetanova ulica 17, 2000 Maribor, Slovenia

ARTICLE INFO

Article history:
Received 14 October 2011
Accepted 27 March 2013
Available online 12 May 2013

Keywords:
Model-driven engineering
Process control software
Programmable logic controllers
Modeling languages
Automatic code generation

ABSTRACT

This paper presents a new, two-level, model-driven engineering approach to industrial process control software. The first level (infrastructure engineering) is concerned with the following: the definition of the development process and guidelines, the definition of a domain-specific modeling language, the specification of the model transformations, and the development of a tool suite. This tool suite enables modeling of the process control software and the automatic code generation for programmable logic controllers. In the second level (application engineering), the process control software is engineered using the results of the infrastructure level. The approach is demonstrated on excerpts from an industrial project.

1. Introduction

Industrial process control systems are hardware–software systems that control and supervise the behavior of technological processes in order to achieve a process-oriented goal. These systems are used in almost all industrial sectors, because of their many positive effects, such as increased production rates, improved quality and flexibility, optimized production processes, reduced costs, and the reduced consumption of energy and raw material. Software is the central part of process control systems because it implements various kinds of complex control functions. However, the complexity of the development, operation and maintenance of the software for these systems is not so much associated with basic control (i.e., achieving and maintaining the desired state of the process variables), but more with procedural control (i.e., performing a sequence of activities that ensure the achievement of the goals of the system or process). According to the estimates of Boeing and Honeywell, software consumes 60–80% of engineering effort (Heck, Wills, & Vachtsevanos, 2009), and procedural control software represents the major and most complex part of this software. The focus of the research presented in this paper is the engineering of process control software that has a complex procedural control component and is not safety-critical (does not have to be certified i.e., there is no need for formal proof that the software is error-free). Process control software engineering is influenced by the hardware platform used. Although a variety of hardware platforms for industrial process control systems exist, the majority uses Programmable Logical Controllers (PLCs) (Colla, Leidi, & Semo, 2009). PLCs are programmed according to the widely adopted IEC 61131-3 standard (Lewis, 1998).

The software engineering practices in the process control domain are failing to address the steadily increasing complexity of control systems and the demands of the market (Maurmaier, 2008), such as short time-to-market periods, high-quality software, and an efficient development process. The causes for this are its many deficiencies (Friedrich & Vogel-Heuser, 2007), foremost its focus on the implementation phase with little or no activity in the earlier software lifecycle phases and an unawareness of advanced software engineering concepts and technologies that are successfully used in some other domains. One of these concepts is Model-Driven Engineering (MDE), which advocates the systematic and disciplined use of precise models throughout the software lifecycle (Schmidt, 2006). Another interesting concept, which complements MDE, is Domain-Specific Modeling Languages (DSMLs), which enables the modeling of software and systems using abstractions that are common to a specific domain (Sprinkle, Mernik, Tolvanen, & Spinellis, 2009). The shift from General-Purpose Modeling Languages (GPMLs), such as the general Unified Modeling Language (UML), to DSMLs has revealed several advantages, such as the inherent reuse of domain knowledge. However, the UML may also become a DSML when it is specialized for a specific domain through the definition of a UML profile.

* Corresponding author. Tel.: +386 1 477 37 30; fax: +386 1 477 39 94.
E-mail addresses: tomaz.lukman@ijs.si (T. Lukman), giovanni.godena@ijs.si (G. Godena), gray@cs.ua.edu (J. Gray), marjan.hericko@uni-mb.si (M. Heričko), stanko.strmcnik@ijs.si (S. Strmčnik).

0967-0661/$ - see front matter © 2013 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.conengprac.2013.03.013

The main research result presented in this paper is the Modeling and Automatic Generation of Industrial process Control Software (MAGICS) approach, which is a new development approach for process control software.

The paper starts by identifying the main issues of the process control domain in Section 2. The related work is described and analyzed in Section 3. Section 4 presents the two main concepts that were used during the development of MAGICS: MDE and the ProcGraph language (Godena, 2004) – an existing modeling language that was formalized to be used in MAGICS. Section 5 presents MAGICS, which consists of two engineering levels, each presented in its own subsection. Section 6 demonstrates MAGICS on an example project. A discussion about MAGICS is given in Section 7. The last section draws a conclusion and describes future work.

2. The issues of the industrial process control domain

Based on an extensive literature overview (Colla et al., 2009; Fischer, Hordys, & Vogel-Heuser, 2004; Godena, 2004; Streitferdt, Wendt, Nenninger, Nyβen, & Horst, 2008; Thramboulidis, 2004) and the experiences of the authors, the following main issues of the process control domain were identified:

i1. *The use of inadequate abstractions*: Most of the approaches found in the literature are based on modeling physical devices such as pumps and valves (Godena, 2004). These device-centric approaches do not expose the primary goals of process control systems (i.e., to control the process/es), but instead they expose the means to achieve these goals. Such a system should rather be modeled through process-centric (in other words, procedural-control-centric) abstractions that are goal-oriented, e.g., technological operations and activities. Device-centric models are complex, because of the high degree of coupling between the software modules corresponding to individual equipment entities (Godena, 2004). Therefore, such low abstraction level models violate the desirable design principles of low coupling and high cohesion that were described by Yourdon and Constantine (1979). The development of an optimal system based only on device-centric models is difficult to accomplish. It is important to realize that GPMLs, like the general UML, are not encouraging the use of adequate abstractions in the high-level models of the software, because they do not contain any knowledge about the process control domain.

i2. *The semantic distance between the early development phases and the implementation phase*: Several activities in the development lifecycle can be performed in a routine manner despite their complexity and volume. These activities, which are very likely to be subject to human error, are suitable for automation. An empirical study performed by Colla et al. (2009) has shown that process control software engineering organizations do not employ automation (e.g., code generation) during the development lifecycle. It is hard to transform the deliverables of early development phases into the implementation of a process control system (Colla et al., 2009) because of the wide semantic gap between the high-level constructs of the modeling languages and the low-level constructs of the procedural-imperative programming languages of the IEC 61131-3. This transformation cannot be automated if the specifications are defined in an informal way, which is predominant in practice (Colla et al., 2009).

i3. *The specific background of the developers*: Process control software developers are predominantly electrical engineers and mainly have experience in programming PLCs with low-level languages (Streitferdt et al., 2008). The problem is that these developers are reluctant to work in the early development activities and to use abstract high-level constructs, models and modeling languages, which are beneficial for taming the increased complexity of many process control systems. This issue is aggravated by the fact that the majority of the available modeling languages are GPMLs. The developers perceive GPMLs (e.g., general UML) as too vast, complex and inexpressive for the process control domain (Thramboulidis, 2004).

i4. *Lack of verification and validation*: The state-of-the-practice often relies on testing without verification and/or validation to address the need for high-quality process control systems (Fischer et al., 2004). Software and hardware are often tested and approved on-site; therefore, errors are often overlooked or even introduced with ad-hoc corrections. These errors may cause accidents that result in the discontinuance of production or may even endanger human lives or the environment. The downtime losses of large industrial factories are very high and can cost several million dollars per day (Streitferdt et al., 2008).

The MAGICS approach addresses the identified issues in various ways in order to increase the long-term possibilities of being adopted by process control software developers.

3. State-of-the-art process control software engineering approaches

Several approaches for the engineering of process control software or systems have been proposed. The existing approaches can be classified into three groups. The first one relies on the IEC 61499 standard; the second one on the UML; and the third one is independent of the UML and relies on MDE.

The main idea of the *first group* of approaches is to follow the IEC 61499 (Lewis, 2001) standard, which is the most recent standard in industrial process control and automation systems. The standard represents an extension of the ideas of IEC 61131-3 with support for the design phase and for distributed process control systems. However, IEC 61499 has not yet received broad adoption by industry (Thramboulidis, 2009) because of several deficiencies: a lack of tool support and reference implementations, lack of support by powerful market players, and lack of support for automatic PLC code generation. Also, IEC 61499 does not cover the analysis phase (Thramboulidis, 2005), which is its main deficiency, and it does not introduce any process-centric abstractions on its own.

The *second group* consists of various approaches that use the UML, or a combination of the UML and the IEC 61499 standard. The purely UML-based approaches are less suitable for the developers in the process control domain, because according to the literature (Friedrich, & Vogel-Heuser, 2007; Thramboulidis, 2004) the UML is not familiar to most process control practitioners and is perceived as too complex (see issue i3). The hybrid approaches try to overcome the drawbacks of the UML by using IEC 61499 in the later development phases. Vogel-Heuser, Witsch, and Katzke (2005) proposed a UML-based approach that enables automatic PLC code generation for the IDEs (Integrated Development Environments) of two vendors. Nevertheless, the information on when and how to use the combination of class, state machine and architecture diagrams (which are similar to UML deployment diagrams) is not provided. The UML-PA, which was introduced by Katzke and Vogel-Heuser (2005), is a UML profile developed for the modeling of real-time and distributed automation systems. The problems of UML-PA are its vastness, the lack of automatic code generation, and lack of development guidelines/process. Another approach (Estevez, Marcos, Sarachaga, & Orive, 2007) is based on three different UML profiles, which are used to model the functional, hardware architecture and the software views of the

control system under design. Its main drawback is the lack of automatic code generation. AUKOTON (Hästbacka, Vepsäläinen, & Kuikka, 2011) is an MDE approach based on a UML profile and a development process consisting of three main phases: functional requirements modeling, automation functions modeling (which is platform-independent) and automation components modeling (which is platform-specific). The AUKOTON tool support enables semi-automatic model transformations between the development phases and the automatic generation of PLC code. The main drawback of the approach is its relatively low abstraction level, because only device-centric abstractions are used throughout the whole development process. Tranoris and Thramboulidis (2006) proposed a hybrid MDE approach with a development process that uses the UML for the analysis model and IEC 61499 for the design model. The approach provides an automatic transformation from the analysis to the design model. A CASE (Computer-Aided Software Engineering) tool enables the editing of IEC 61499 models and device diagrams, which show the hardware of the control system, and enable the mapping of specific function blocks to specific computation devices. This approach does not provide automatic PLC code generation. Panjaitan and Frey (2007) introduced a hybrid approach that uses UML class, component and state machine diagrams, from which an IEC 61499 model can be generated automatically. This approach lacks development guidelines and does not support PLC code generation.

The *last group* of approaches is independent of the UML and is based on the idea of MDE. The Archimedes approach (Thramboulidis, Perdikis, & Kantas, 2007) covers the design and implementation phases. The design phase is supported by a CASE tool and uses IEC 61499 as the modeling language. This tool can generate CORBA (Common Object Requesting Broker Architecture) component models automatically, but ignores the need for PLC code generation. Estévez, Marcos, and Orive (2007) proposed an approach where the control system is modeled with an XML schema-based modeling language, which supports the modeling of three different views: the hardware architecture (controllers, I/O devices and communication busses), the software architecture (using IEC 61131 concepts) and the software-to-hardware mapping (distribution of the software over several computational nodes). This approach enables the generation of IEC 61131-3 code. The major drawback is that the models are expressed in XML, which is not suitable as a human-readable modeling language. Maurmaier (2008) proposed an MDE framework that is based on a chain of automatic model transformations that are customizable for every project. However, this approach is still a concept that does not define which specific DSML should be used. MEDEIA (Strasser et al., 2008) is a model-driven and component-based engineering approach, which is still a concept that promises to integrate diagnostics into the models and enable automatic code generation in the future. The systems will be modeled through a hierarchical plant structure model, where the main abstractions are automation components. Therefore, this approach is inherently device-centric.

Table 1 summarizes and compares the existing approaches using various criteria. The proposed approaches have not been widely adopted by industry (Colla et al., 2009) because of their weaknesses and particularly because of their inability to address the mentioned core issues of the process control domain properly. The majority of the reviewed approaches have a combination of the following weaknesses: lack of automatic PLC code generation, lack of development process definition and guidelines, nonexistent or immature tool support, and the use of device-centric abstractions (see issue i1). One of the goals of the MAGICS approach was to avoid these weaknesses. Since this goal was accomplished, these weaknesses are also the main differences between MAGICS and the reviewed approaches.

4. Relevant concepts and technologies

The MAGICS approach embodies the concept of MDE, which is presented in Section 4.1, and uses a formalized version of the ProcGraph language, which is presented in Section 4.2.

Table 1
A comparison of the available state-of-the-art process control engineering approaches.

Approach	Viewpoint					
	Process definition	Analysis phase	Design phase	Automatic model transformations	Tool support	Modeling of hardware
Vogel-Heuser, Witsch, and Katzke (2005)	/	/	UML (class & state diagram) +stereotypes	to IEC 61131-3	Artisan Real-time Studio	system architecture, I/O mapping
PA UML Katzke, & Vogel-Heuser (2005)	partial	/	UML profile (e.g., timed state machines)	/	unspecified UML tool(s)	system architecture, software-to-hardware-mapping
Estevez, Marcos, Sarachaga, & Orive (2007)	yes	UML (use case diagrams)	3 UML profiles (class diagrams)	/	Artisan Real-time Studio	system architecture, software-to-hardware-mapping
AUKOTON (Hästbacka, Vepsäläinen, & Kuikka (2011))	yes	UML profile (platform-independent models)	UML profile (platform-specific models)	UML to UML UML to IEC 61131-3	Eclipse-based (uses EMF and Topcased toolkit)	system architecture, software-to-hardware-mapping
Tranoris and Thramboulidis (2006)	yes	use cases+UML (collaboration, sequence & class diagrams)	IEC 61499+device diagrams	UML to IEC 61499	Rational Rose+CASE tool (developed from scratch)	system architecture, software-to-hardware-mapping
Panjaitan, and Frey (2007)	/	UML (class, component & state machine diagrams)	IEC 61499	UML to IEC 61499	Artisan Real-time Studio +Function Block Dev. Kit	/
Archimedes (Thramboulidis, Perdikis, & Kantas (2007))	/	/	IEC 61499	to CORBA component models	CASE tool	/
Estévez *et al.* (2007)	/	/	XML schema	to IEC 61131-3 (for two vendors)	tool(s) for XML technologies	system architecture, software-to-hardware-mapping
Maurmaier (2008)	yes	various (not specified which)	various (not specified which)	/	Eclipse-based (uses EMF and GMF)	/
MEDEIA (Strasser, et al., 2008)	/	/	various (e.g., Gantt chart)	/	/	system architecture, I/O mapping

4.1. Model-driven engineering

MDE is a software engineering paradigm that has the potential to sustainably raise productivity (Mohagheghi, & Dehlen, 2008) and to reduce the complexity of software and systems development (Schmidt, 2006). MDE relies on three main components, which are described in the following subsections: modeling languages, model transformations and software tools.

4.1.1. Modeling languages

Modeling languages for software and systems can be categorized in terms of GPMLs and DSMLs. The goal of GPMLs is to support modeling in a variety of domains; therefore, they tend to be extensive and complex. However, they are often standardized and have an already available infrastructure, such as tooling, training and documentation (Selic, 2012). An example is general UML 2.X (Object Management Group, 2011b), which consists of 14 diagram types and a very large set of modeling elements. On the other hand, DSMLs are designed to cover a specific application domain (e.g., manufacturing or petrochemical), which makes them compact and easier to understand for domain experts. A DSML can formalize the application structure, behavior and requirements within a particular domain (Schmidt, 2006). It is fairly obvious that a DSML will usually produce a more concise and direct specification and, therefore, more effective solutions for problems in its domain than a GPML (Selic, 2012). This means the real question is what is the most effective method for defining a DSML (Selic, 2012). DSMLs are usually defined in one of the following ways (Noyrit, Gérard, & Selic, 2012; Avila-Garcia & Garcia, 2008):

- *UML-based definition*. This approach specializes the UML for a specific domain through the definition of a UML profile.
- *Metamodel definition*. This approach defines the DSML "from scratch" with the use of a metamodeling language, e.g., the Meta Object Facility (MOF) (Object Management Group, 2006).

4.1.2. Model transformations

A model transformation generates one or more target models from one or more source models, based on a set of model transformation rules. With model transformations, MDE can automate many of the complex but routine development tasks that are often performed manually (Sendall, & Kozaczynski, 2003), such as model refactoring or the generation of lower level models and, eventually, the code from higher level models.

4.1.3. Tool support

To enable MDE in practice and fulfill its potential, sophisticated tools must be developed that support DSML(s) and the defined model transformations. A survey performed among industry participants showed that the availability of tool support is the most influential factor in the decision about whether to adopt a specific MDE approach (Mohagheghi, & Dehlen, 2008). Several alternatives exist for the development of the tool suite for the selected DSML. The development approach depends on the way the DSML was defined:

- *UML-based DSML*. The tool support may be provided with the use of existing modeling tools, which are parameterized by the UML profile definition, e.g., the Papyrus UML (Gérard, 2012). The UML standard is widely known and taught; therefore, an increasing base of developers may be familiar with this approach (Selic, 2012). It demands a relatively small effort, because the already available UML tooling can be used to provide the tool suite for the DSML (Noyrit et al., 2012). However, such tool suites are usually very generic, hard to customize and may not be able to support the functionalities that are needed by the users (Avila-Garcia, & Garcia, 2008).
- *UML-independent DSML*. In general, developing tool support for such DSMLs demands more effort and highly specialized expertise (Noyrit et al., 2012). On the other hand, it offers more flexibility than UML-based DSMLs. Two different development options are available:
  - *Manual development*. No specialized tools for MDE are used during the development of the tool support. The literature (Gronback, 2009; Kelly & Tolvanen, 2008) and the experience of the authors suggest that manual development is demanding and resource intensive (it may require several person-years) and also inflexible for maintenance. However, the main advantage of manual development is that it is not limited by the capabilities of the used specialized tools. Therefore, even very specific tool suite requirements can be realized.
  - *Development with metamodeling tools*. Metamodeling tools, as for example GME (Lédeczi et al., 2001), GMF (Gronback, 2009), and MetaEdit+ (Kelly, & Tolvanen, 2008), can automatically generate (portions of) the needed tool suite based on a formal definition of the DSML and the model transformations. Consequently, metamodeling tools have the potential to speed up the development and reduce the required effort and development complexity (Kelly, & Tolvanen, 2008). The main disadvantage of this approach is the relative immaturity of this technology.

4.2. ProcGraph language

The ProcGraph language (Godena, 2004) is a semi-formally defined process-centric language that was developed for the specification of process control software. The first subsection introduces the main elements of the ProcGraph language. The second subsection discusses the issue of the tool support for ProcGraph.

4.2.1. Elements of the language

The ProcGraph language consists of three different diagram types and a symbolic pseudo-language. The example excerpts of a ProcGraph model in Fig. 1 will be used as an aid to explain the elements of this language.

The root diagram type is the Entities Diagram (ED), which contains Procedural Control Entities (PCEs) and potential composite dependencies between them. A PCE is an abstraction of an operation, an activity or a sub-activity, which are the integral parts of the process control domain. A PCE can be either elementary or a super PCE. A super PCE is decomposed into another ED, which contains a set of PCEs. An example ED with two elementary PCEs and one composite dependency is shown in the center of Fig. 1.

The next diagram type is the State Transition Diagram (STD), which defines the behavior of a particular elementary PCE. Such a diagram consists of states and transitions, and super-state hierarchies. An active state is a state whose combination of action sequences is being executed. An STD is an extended finite state machine, which differs from other extended variants (e.g., Statecharts and a UML state diagram) in the following details:

(a) Two explicit types of states, i.e., durative and transient.
(b) The processing is organized into a richer set of action sequence types (entry, loop, exit, and always sequences for durative states, a transient sequence for transient states, and an action sequence of transitions).

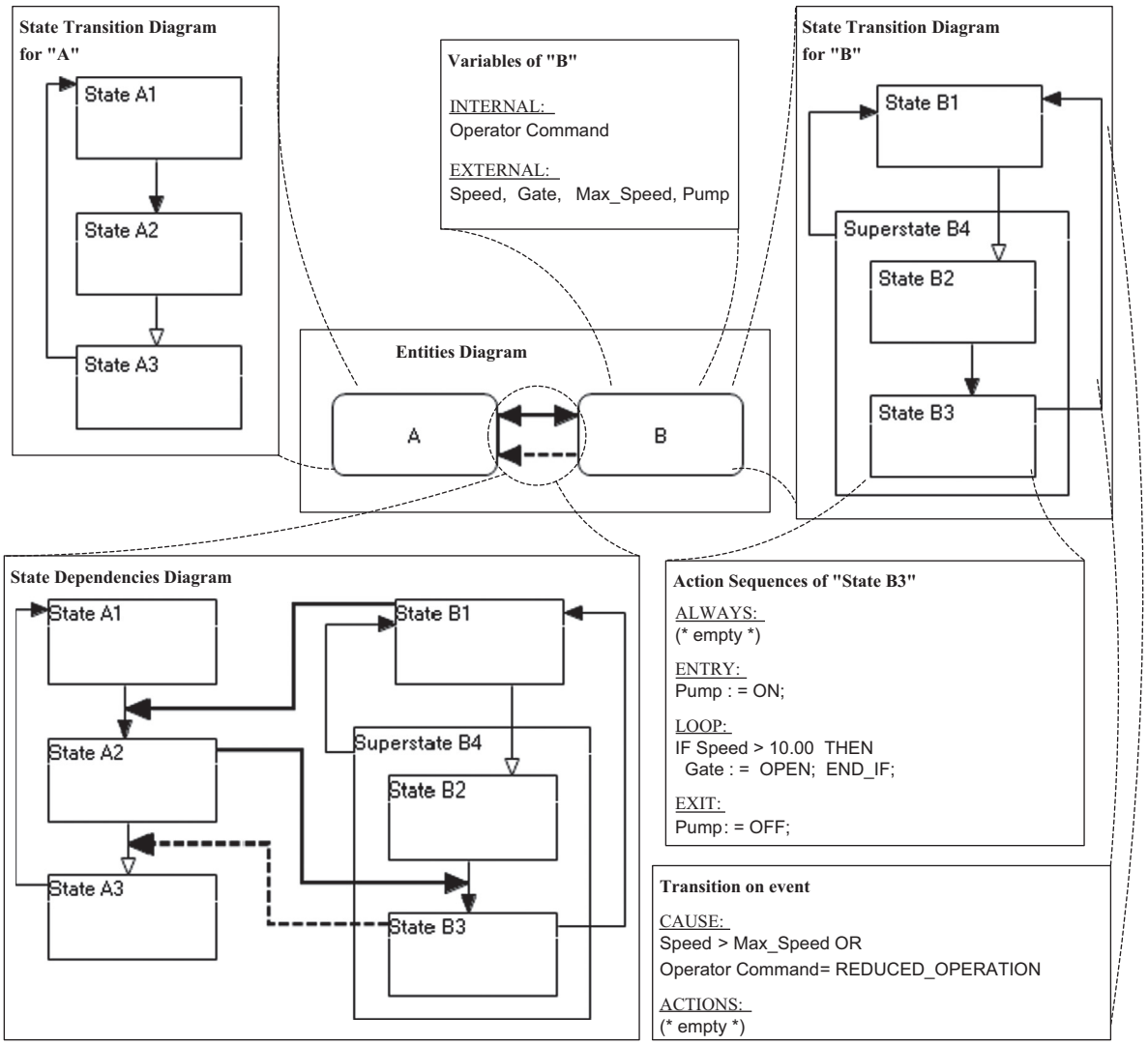

Fig. 1. A scheme that shows the structure of an example ProcGraph model.

(c) All action sequences have a duration.
(d) Overlapping super-states.
(e) Two explicit transition types (i.e., the transition on event and the transition on completion).

The transition on event, which is denoted by a filled arrowhead, is fired when the source state of the transition is one of the PCE's active states and the event (either an operator command or a process equipment signalization) occurs. The transition on completion, which is denoted by an empty arrowhead, is fired when the source state of the transition has finished its processing. An example STD is shown in the upper-left and another in the upper-right corner of Fig. 1.

All the action sequences that are the basis of the processing performed in states and transitions in a STD are defined either with ProcGraph's symbolic pseudo-language or with the ST language (which was used in Fig. 1) of the IEC 61131-3 standard.

The State Dependencies Diagram (SDD) is an explosion of a composite dependency, which exactly defines the mutual behavior dependencies between the two PCEs it connects. The lower-left corner of Fig. 1 shows an example SDD. An SDD consists of the STDs of two interdependent PCEs and a set of elementary dependencies. A dependency can be either a conditional dependency, which is denoted by a normal line with a filled arrowhead, or a propagational dependency, which is denoted by a dashed line with a filled arrowhead. The transition, which is the sink of a conditional dependency, can only be fired when the source state of this dependency is active. For example, in Fig. 1 the transition between 'State B2' and 'State B3' can only be fired if the 'State A2' is the active state of 'A'. The transition that is the sink of a propagational dependency is fired when the source state of this transition and the source state of the dependency are both active. For example, in Fig. 1 the transition between 'State A2' and 'State A3' is fired when 'State B3' is the active state of 'B' and 'State A2' is the active state of 'A'. All the behavior dependencies between two PCEs that are defined in a SDD are summarized with the shape of a composite dependency, which shows a union of the defined elementary dependencies.

4.2.2. Tool support

ProcGraph was used initially for the specification of control software "on paper" (i.e., specifications were either drawn with pen and paper, or with a general-purpose graphical tool). Therefore, the ProcGraph specifications had to be transformed manually into PLC code, which was resource-intensive and error-prone. Consequently, the idea arose to develop tool support for ProcGraph that would enable modeling and automatic code generation.

A previous development approach (Kandare, Strmčnik, & Godena, 2010) that attempted to develop tool support for ProcGraph following the manual way and ignoring the MDE paradigm

was discontinued. The reasons for this were the weaknesses of the manual development, the incomplete support for ProcGraph, and the lack of several required features. These drawbacks have been overcome in the MAGICS approach.

5. The MAGICS approach

The first stepping-stone towards the development of the MAGICS approach was the research presented in (Lukman, Godena, Gray, & Strmčnik, 2010). The MAGICS approach differs from other approaches in the process control domain by explicitly defining two distinct engineering levels, as illustrated in Fig. 2. The infrastructure engineering level is concerned with the development of the infrastructure, which currently consists of a DSML, an application development process definition, the model transformations and a software tool suite. The application engineering level uses the infrastructure for the development of process control software. The information flow from the application engineering level to the infrastructure engineering level incorporates the knowledge that is being accumulated during the application development process. This information can be potentially generalized and incorporated into new versions of the infrastructure, which would further improve the application engineering or extend the range (i.e., the variability) of the software that can be engineered.

The flexibility introduced by the separation into the mentioned levels is important because it enables a systematic and consistent evolution of particular components. Also, new tools like a documentation generator or model verification tool, which are planned to be developed in the future, can be integrated more easily by introducing another instance of the tool development activity at the infrastructure engineering level.

5.1. Infrastructure engineering level

The main infrastructure level deliverables are developed incrementally and iteratively in a separate activity, because they have their own concerns and require specific knowledge, skills and tools. Each of these activities is described in the next subsections. At the beginning of the infrastructure development, various requirements for the tool suite were defined, which influenced several development decisions. These requirements are described in (Lukman & Mernik, 2008). The current tool suite prototype components (a model repository, a model editor and two code generators) are shown in Fig. 2 as the inputs of the process control software engineering sub-process. Fig. 3 shows how these components interact during the application engineering. The sinks of the information flow chain are two external tools, which consume the generated code. The first enables the use of Mitsubishi PLCs and the second the use of PLCs that can be programmed with PLCOpen compatible IDEs, e.g., CoDeSys (3S-Smart Software Solutions, 2012).

5.1.1. The domain-specific modeling language

The initial motivation that led to the development of MAGICS was to develop a tool-supported and MDE-based development approach that would support the use of the in-house developed ProcGraph language. Therefore, the adoption of the ProcGraph language as the DSML of MAGICS was a natural choice. The use of ProcGraph is justifiable by its main features, which make it highly suitable for the process control domain and ensure that MAGICS addresses the i1 issue (i.e., the use of inadequate abstractions). These features are its process-centric abstractions, a unique behavioral model for these abstractions, and explicit modeling of the dependencies between them.

The adequacy of ProcGraph is substantiated by the positive experience from using it in over twenty industrial projects (ranging from a few man-months to several man-years of software development effort and from a few hundred pages to several thousand pages of listings in ladder diagram code). The experience suggests that ProcGraph leads to better solutions than traditional device-centric approaches, which often produce very limited solutions (e.g., a significantly low-level of automation, small conformance to the requirements and constraints of the customers). The constructed models contained only a small number of errors and the code that was manually generated from them approached the “near zero coding errors” goal.

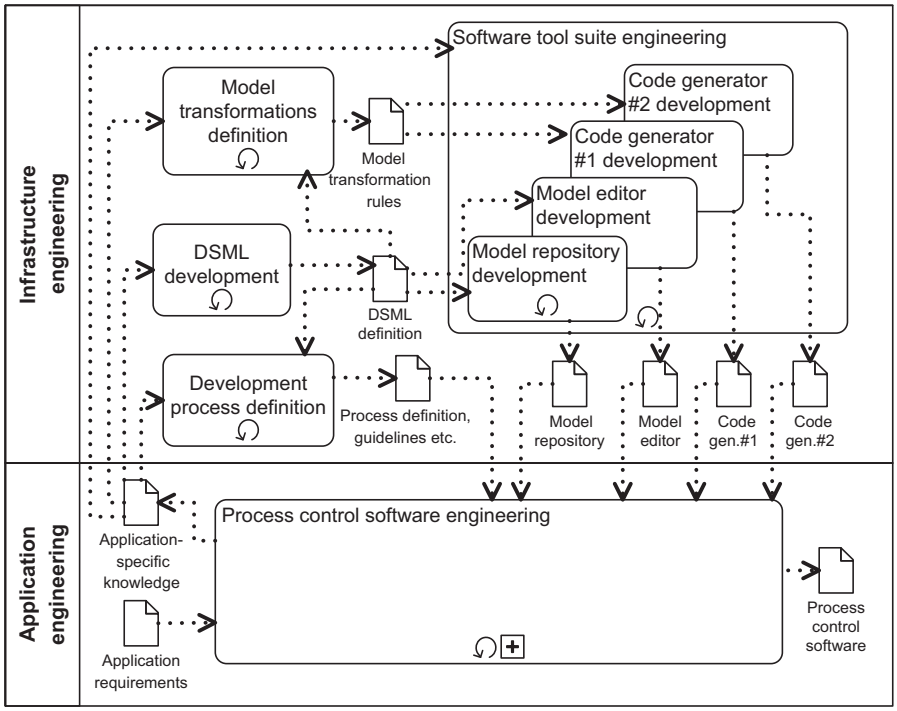

Fig. 2. The two distinct levels of the MAGICS approach.

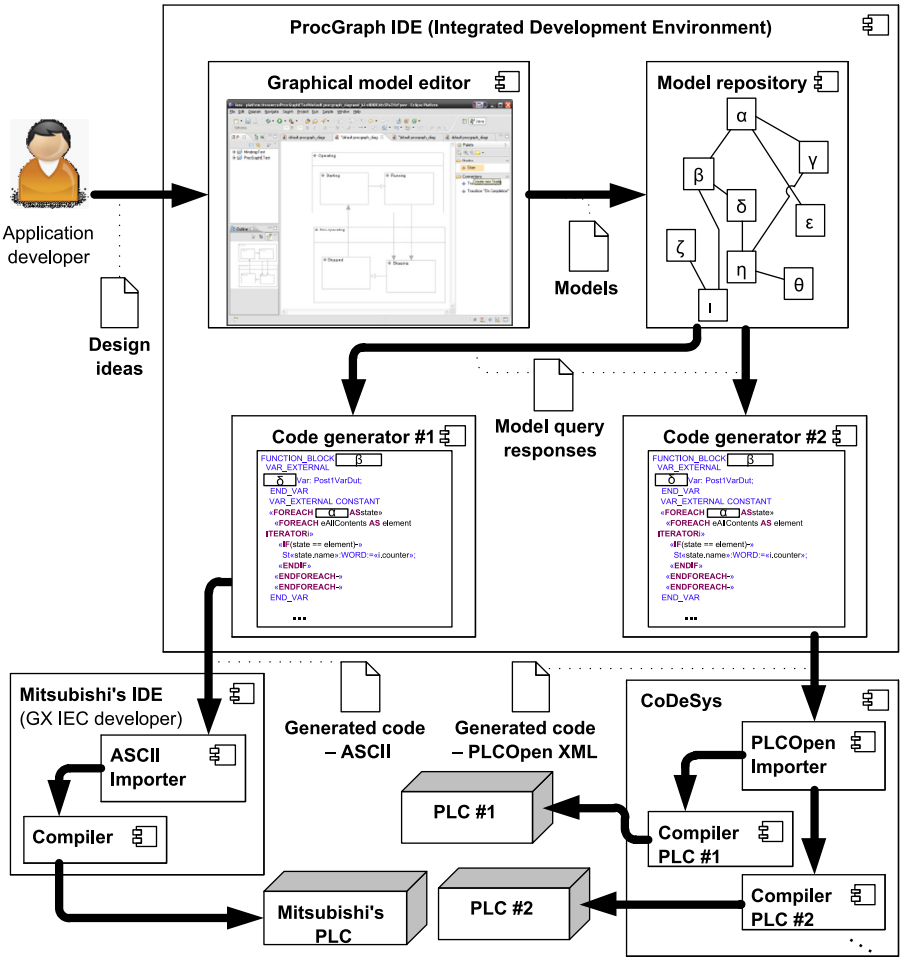

Fig. 3. The information flow between the tool suite components during the engineering of process control software.

A survey of alternative DSMLs for the process control domain revealed that none of them contain process-centric abstractions and consequently, none of them discourages the creation of device-centric models. This additionally justified the use of ProcGraph in MAGICS. Considering the criteria of familiarity to control engineers, the graphical languages in IEC 61499 are also a viable alternative, because they are based on function block abstractions. However, because of these low-level abstractions they are, unlike ProcGraph, not suitable for the system analysis phase.

ProcGraph was not directly usable in MAGICS because of its informal and partial definition. Therefore, an important design decision was concerned with how to formalize ProcGraph - either with a UML profile or by metamodeling (see Section 4.1.1). We have decided to use metamodeling, because of our familiarity with this approach and the goal that as many features of the original ProcGraph language as possible should be available in the formalized ProcGraph.

5.1.2. *The application development process definition*

The deliverables of this activity were developed based on the analysis of the selected DSML, past application projects, and domain expert knowledge. The initial application development activities covered the relevant dimensions of modeling process control software. As in the majority of modeling languages, the structure and behavior are the two main modeling perspectives. Therefore, two separate activities were introduced into the development process for these concerns. Also, the modeling of interdependent behavior activity was added, because the behavior of the software is defined through the behavioral dependencies between the previously defined PCEs. All of these three activities correspond to distinct diagram types of the ProcGraph language. The analysis of several process control software engineering projects that were realized with ProcGraph showed that these three viewpoints are modeled distinctively and that different concerns are important for them. Besides these modeling activities, the activity of transforming the models into code was introduced, which is automated in MAGICS. The last activity introduced was the testing activity.

In the analysis of past projects, several best practices were discovered that were generalized into development guidelines to assist the application developers. The produced development process definition and the development guidelines are presented in Section 5.2.

5.1.3. *Model repository*

The model repository was developed with the Eclipse Modeling Framework (EMF) (Budinsky, Steinberg, Merks, Ellersick, & Grose, 2003), which is the central tool for creating structural model repositories for the Eclipse platform. The EMF can generate the model repository automatically from the metamodel-based definition of the DSML. The EMF has several advantages (Hnetynka & Plasil, 2008): it is mature, well-supported and maintained, and very extensible (because a broad spectrum of Eclipse-based state-of-the-art MDE tools exist that can be integrated with EMF seamlessly). Also, it uses the standards MOF (its EMOF subset) and Object Constraint Language (OCL) rather than proprietary solutions, which are often used in other metamodeling tools (Kern & Kühne, 2009).

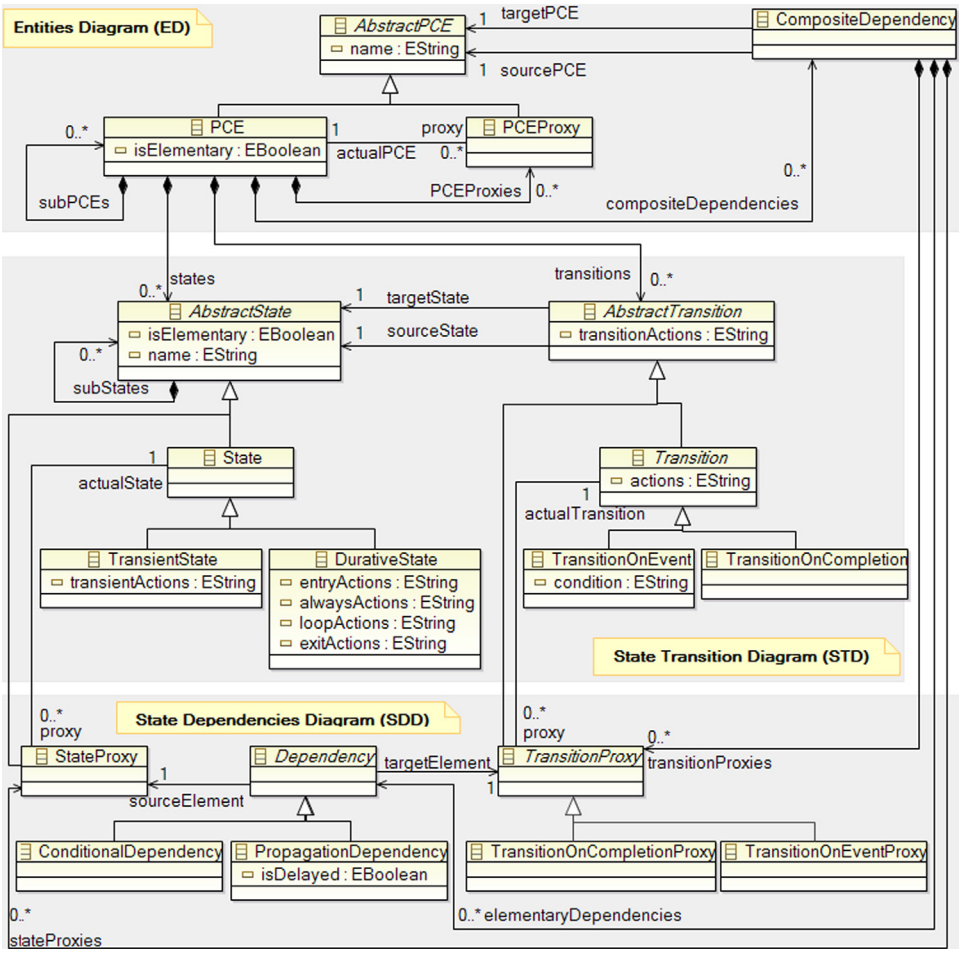

Fig. 4. The ProcGraph metamodel.

During the definition of the ProcGraph metamodel, which can be seen in Fig. 4, several obstacles originating from EMF's capabilities had to be overcome. In ProcGraph models, some metaclass instances should be present in two or more diagrams. Specifically, the same 'State' or 'Transition' should be contained in one STD and in several SDDs, and the 'PCE' should be contained in several EDs. In EMF the same instance of a metaclass can only be contained (through the composition relationship) in one metaclass. Therefore, the following proxy metaclasses had to be introduced: 'StateProxy', 'TransitionProxy', and 'PCEProxy'. The final metamodel was developed after a considerable amount of experimentation with EMF and the connected Eclipse modeling tools.

To ensure that the constructed ProcGraph models would not violate the important rules of the domain, several OCL constraints were defined. For example, a constraint for all transitions was defined by adding the OCL code in Listing 1 to the 'AbstractTransition' metaclass. The first line of the constraint disallows a transition from having the same source and target state and the second line ensures that super-states cannot be a target state of a transition.

Listing 1 An OCL constraint for the 'AbstractTransition' metaclass

```
self < > oppositeEnd
        and oppositeEnd.subStates - > size ()=0
```

5.1.4. Model editor

The Graphical Modeling Framework (GMF) metamodeling tool (Gronback, 2009) was selected for the development of the graphical model editor based on a systematic evaluation of the available metamodeling tools. This evaluation, which is reported in (Lukman & Mernik, 2008), was based on the requirements for the MAGICS tool suite. GMF is a powerful and complete framework for generating graphical model editors (Temate, Broto, Tchana, & Hagimont, 2011). However, this power comes at a price, because it is perceived as quite complex and less user friendly than desired (Evans, Fernández, & Mohagheghi, 2009). GMF is very flexible, because the generated editors can be specialized through custom code in a myriad of ways. This allows the realization of very specific editor requirements. However, this can often be labor-intensive and demanding, partially because GMF lacks complete documentation (Temate et al., 2011). The consequence of choosing GMF was that EMF, on which it relies, was chosen for the realization of the model repository.

GMF uses the following models of a DSML to generate automatically a basic model editor: a domain model (i.e., a metamodel), a notation model (to define the graphical symbols), a tooling model (to define the tools that manipulate the diagram), a mapping model (to map the elements of the domain, notation and tooling to each other) and an editor model (to allow the fine tuning of the generated visual model editor). The basic model editor that was automatically generated by GMF was not suitable for industrial development; therefore, it was specialized with a considerable amount of manual coding.

The screenshot of the current version of the model editor can be seen in Fig. 5, which shows the editing of one ED, two STDs and one SDD of the ProcGraph model from Fig. 1.

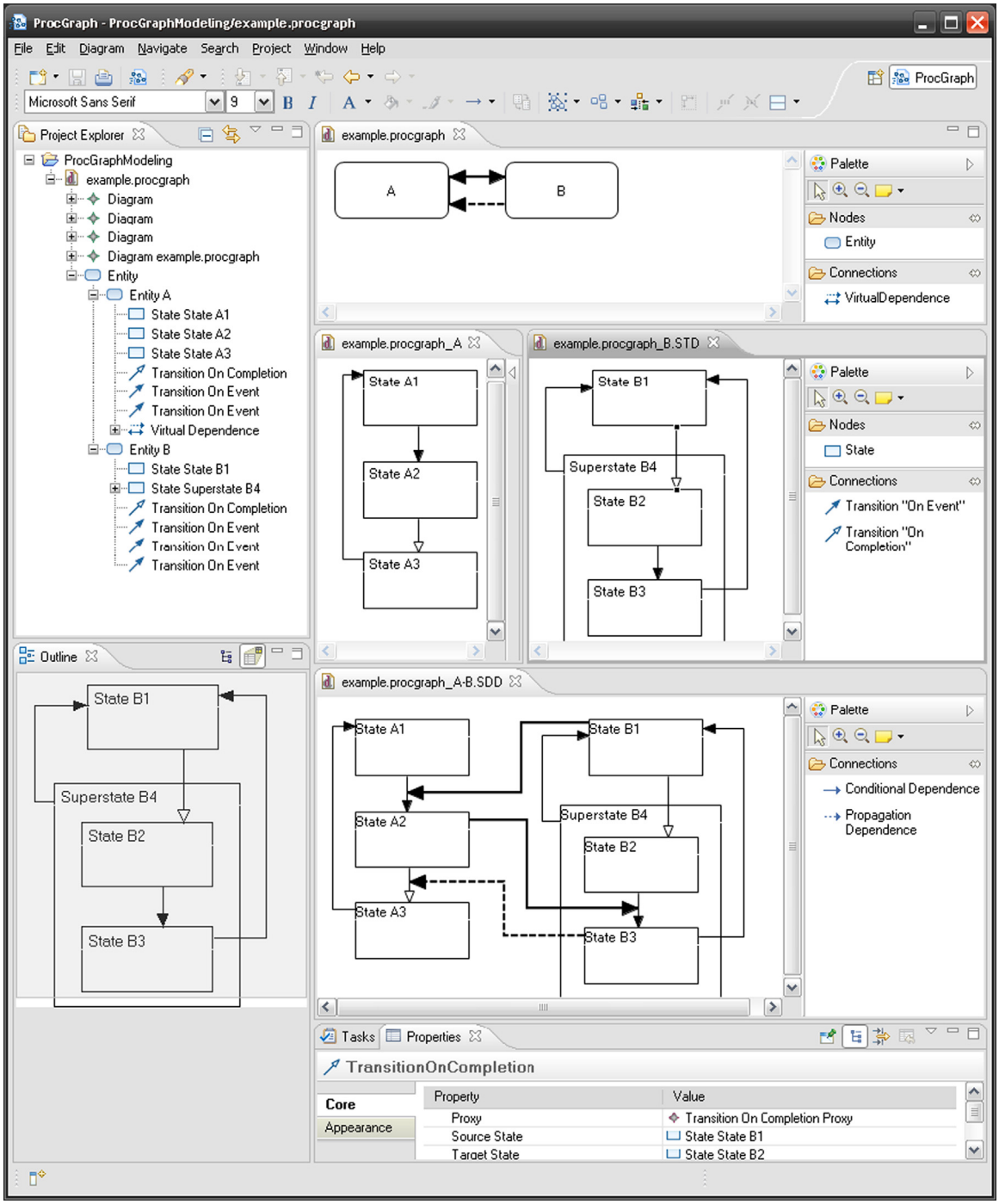

Fig. 5. A screenshot of the graphical ProcGraph model editor.

5.1.5. *Model transformation rules*

During the definition of the model transformation rules, it had to be determined into which of the IEC 61131-3 programming languages to transform the ProcGraph models: Instruction List (IL), Structured Text (ST), Ladder Diagram (LD), Function Block Diagram (FBD) and Sequential Function Chart (SFC). The FBD is semantically the most suitable, because it makes it possible to decompose PLC programs hierarchically. Therefore, it was selected together with the ST, which is used for the specification of the processing actions.

Before the model transformation from ProcGraph into the combination of the FBD and ST are presented, some relevant concepts from the target language are introduced. A *function block diagram* is a graphical definition of the algorithm of a *function block type* (which also has a definition of its variables). This algorithm can only be executed through a *function block*, which is essentially an instance of the *function block type* that defines this algorithm. A *function block diagram* can consist of *function blocks* and *functions*. Their inputs and outputs and also *variables* can be connected to each other with *links*, to create a signal flow that defines the algorithm. These elements are contained in *networks*, which visually and logically separate the contents of the *function block diagram*. Two common custom constructs (compounds of graphical elements with a certain purpose) are important for the

presented rules. A *checker* determines whether a specific variable is true and returns a corresponding output. A *setter* copies a variable to the value of another variable when the setter becomes explicitly enabled.

The semantics of the semi-formal ProcGraph were already informally described. During the development of MAGICS these semantics were defined explicitly through the transformation rules that are specified in Listing 2 and their implementation in the code generator. This is consistent with the common practice where the semantics of modeling languages are contained in model translators (Bryant et al., 2011).

Listing 2 The high-level transformation rules

r1 Each *ED* is transformed into a *function block diagram*, where each *PCE* of the *ED* is mapped into one *function block* in a separate *network*. The *function block type* of this *function block* is generated by r1. if it is a *super PCE* or by r2. if it is an *elementary PCE*.

r2 Each *STD* is transformed into a *function block diagram*, where each *top-level state* (i.e., a state that has no super-state) is mapped into a *function block* in a separate *network*. The *function block type* of this *function block* is generated by r3. or r4.
Because only one *elementary state* (together with all its *super-states*) of a *PCE* can be active, *checkers* are created and *linked* to the input of the corresponding *function block* (this ensures that only one *function block* is executed in a running cycle) the following way:
- If the *function block* is a transformed *elementary state*, only one *checker* is created (it checks whether the *elementary state* from which it was generated is currently the active one).
- If the *function block* is a transformed *super-state*, a *checker* for each of the *elementary sub-states* of that *super-state* is created. The outputs of these *checkers* are joined together with an *OR function*.

r3 Each *durative state* is transformed into a *function block diagram*, where each *action sequence* is transformed into a *function block* in a separate *network*.
When this *state* becomes active, the *function block* of the *entry sequence* becomes active and is executed only once. At the end of its execution, a *setter* makes sure that the *function block* of the *loop sequence* is executed in the next execution cycle. When its processing ends (either because of an internal reason, a fired transition or a propagation dependency) a *setter* makes sure that the next *function block* of the *exit sequence* is executed only once. The *function block* of the *always sequence* is active all the time its parent *function block* is active.

r4 Each *transient state* is transformed into a *function block diagram*, where the *transient sequence* is mapped into a *function block* in its own *network*, which is active all the time its parent *function block* is active.

r5 The *function block diagram* of a *super-state* gets an additional *function block* in its own *network* for each of its *immediate sub-states. Checkers* are added in the same way as in r2. This ensures that the *function block* of only one *elementary state* is active.

r6 For each *outgoing transition* of a *state*, a section of logic is added into the *function block diagram* of its *always sequence* or *transient sequence*. This logic first contains a *checker* for an event (an operator command, equipment signalization or the end of the processing for the current state). If an event occurs, then the active state of the *PCE* is set to the *sink state* of the *transition* and then goes into the *action sequence* of the *transition*.

r7 For each *dependency* having a *source state* that belongs to the *'A' PCE* and a *sink transition* that belongs to the *'B' PCE*, then the following applies. The *source state* of the *transition* that belongs to *'B'* will get an additional section of logic in the *function block diagram* of its *always sequence*:
- For a *conditional dependency*, an additional *transition condition* is created. It is fulfilled when the *source* state of the *conditional dependency* is an active state of *'A'*.
- For a *propagation dependency* this logic first checks if the *source state* of the *dependency* is an active *state* of *'A'* (before it checks the occurrence of the transition event). If it is, then the *function block* of the *transition* is executed and the *sink state* of the *transition* is set as the active state of *'B'*.

5.1.6. Code generator

The next design decision was to select the import format for the generated code. To generate the descriptions of the IEC 61131-3 code in a format that is importable into a specific PLC IDE, a new code generator has to be developed. First, a generator for the input format of the Mitsubishi GX IEC Developer (Mitsubishi Electric, 2011) was implemented. Additionally, a second generator for the PLCOpen format (PLCOpen international organization, 2010) is being developed, which is supported by an increasing number of PLC IDEs (e.g., CoDeSys).

To implement the code generators, the openArchitectureWare tool (Haase, Volter, Efftinge, & Kolb, 2007) was used, because it is a mature and widely used model-to-text engine that seamlessly works with EMF metamodels (Oliva, 2009). The code generation is based on templates, which are defined with the Xpand template language and consist of a static (i.e., invariant) and a dynamic part. The advantage of openArchitectureWare is its metamodel-aware editor for Xpand, which provides syntax highlighting and code completion. Its disadvantage compared to alternative frameworks is that it is more demanding to learn (Klatt, 2008).

During the development of the code generator the transformation rules had to be implemented in consideration of the syntax of the proprietary PLC IDE import format and the fact that FBD code is visual. This meant that the generation of both the content information (i.e., the elements and the connections between them) and the information about the topology of the diagram (i.e., the coordinates, width and height of the graphical elements) needed to be realized. This was the main challenge in the definition of the code generation templates.

The code generator and the model transformations it implements were verified through testing, which is common in practice (Sani, Polack, & Paige, 2011). Several past projects in which the code was manually transformed were used as test cases. They were remodeled in MAGICS and then the code was generated automatically. This code was compared to the correct, manually transformed code through code reviews.

The developed code generator automatically generates the whole application code from a complete and valid ProcGraph model, which consists of two parts. The first one is a high-level graphical part, which defines the PCEs, their behavior structure, and the behavioral dependencies between them. The second part is a lower-level textual content of the above-mentioned behavior structure, which defines the detailed behavior through the processing sequences. These processing sequences are currently defined with ST statements, which have to be written manually by the developers (in the model editor). During the automatic code generation, the high-level part of the model is transformed into a code skeleton, which is then filled in by automatically copying

the processing sequences from the lower-level part of the model. In contrast to possible expectations, this code skeleton sometimes represents a minor part of the automatically generated source code, while its major part consists of the processing sequences. Let us emphasize at this point that the share between the code generated from the low-level sequences and the code generated from the high-level part of the model is not very relevant. The relevant fact is that the latter part of the model provides a high-level abstract framework, which enables the engineers to manage the complexity of the development and is crucial for the quality of the control software.

An important thing to add regarding code generation is the fact that the processing sequences (i.e., ST statements) in the ProcGraph models do not have to be altered, when the same model should be used for generating the code for different target PLC IDEs (i.e., for different import formats). The reason for this is the wide adoption of the IEC 61131-3 standard in the existing PLC IDEs; therefore, the same ST statements can be used in them. Because the relevant difference between the PLC IDEs is the import format, the models do not have to be changed; only a code generator, which outputs the generated code in the desired import format, must be provided.

5.2. Application engineering level

The application engineering level uses the infrastructure, which enables MDE of the applications and facilitates it in several ways. The tool suite supports the modeling activities by allowing only the creation of syntactically correct and well-formed models. This way, errors that could occur easily with the use of pen and paper or unspecialized tools (e.g., general-purpose graphical editors) are avoided. The coding activity, which is error-prone and effort intensive when performed manually, is automated by the tool suite. The application process definition and the development guidelines help to ensure that the development is effective and that important concerns are not missed. The current application engineering process is composed of the main development activities that are shown in Fig. 6 and can be used in an iterative and incremental fashion. Other common activities, which will not be discussed in this paper, also should be performed (e.g., verification & validation in which model reviews are performed).

The inputs into the development process are the requirements for the process control system/software. Typically, these requirements consist of a description of the uncontrolled process and the requirements for the control system (Frey & Litz, 2000). The requirements that are needed by the MAGICS approach consist of a Piping and Instrumentation Diagram (P&ID) and supporting documents (e.g., informal functionality and safety requirements). A P&ID graphically represents the major equipment of a given process together with the associated instruments (Jones, 1996) in other words, it is a representation of the commonly used

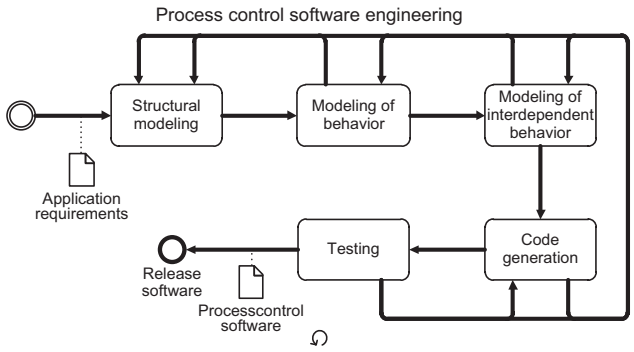

Fig. 6. The application engineering process of MAGICS.

technological object model. The next subsections describe each development activity, its deliverables and development guidelines.

5.2.1. Structural modeling

At the beginning of the structural modeling activity, the requirements are analyzed to identify the main operations that should control the studied technological process. Each operation should be named according to the goal it is pursuing. The next step is the device allocation for each operation. To avoid potential problems, each equipment entity should be controlled by only one operation. The initial PCEs list and their allocated equipment entities may change during the development process.

Essentially, the operations of the technological process were chosen well if they are highly cohesive and weakly coupled. The identification of the operations (and also the activities and sub-activities) can be aided by, but not limited to, the following specific guidelines:

(1) The equipment entities that must work together to achieve a partial operation of the technological process should be allocated to the same PCE.
(2) The equipment entities that must be started and/or stopped together should be allocated to the same PCE.
(3) The equipment entities that must not be started and/or stopped together should be allocated to separate PCEs.
(4) If particular parts of the processing of a PCE have different running attributes (priority, period, or deadline) then that PCE should be split into separate PCEs.

After the operations have been identified, the analyst should try to decompose the complex or extensive operations into activities. This enables scalability, so that even complex technological processes can be modeled by decompositions into activities and further into sub-activities that have a behavioral model (described in Section 5.2.2), which is moderately complex and consequently not hard to manage. As a specific guideline for when to create two sub PCEs, consider the case that a subset of the processing of a PCE has to be performed cyclically and the rest periodically. In this case, that PCE should be separated into the cyclic (usually main) and periodic (usually auxiliary) sub PCE. Because the auxiliary sub PCEs often perform a small amount of the processing using one or a few physical devices, it can exceptionally be named after that device(s).

The last step is to start modeling in ProcGraph. All the identified operations represented by top-level PCEs are placed in the root ED. In the beginning, it is often not clear which operations (if any at all) are interdependent. Therefore, some composite dependencies are drawn later, when these dependencies emerge during the modeling of the behavior. However, if it is already clear that a pair of operations is interdependent, they should already be connected with a composite dependency.

The result of this activity is a partial ProcGraph model, with one or more EDs.

5.2.2. Modeling of behavior

Each elementary PCE has to be exploded into a State Transition Diagram (STD), which defines the states of the PCE and the transitions between them. An important guideline is that each PCE (exceptions can be auxiliary PCEs) has the following set of typical states, which are always present: ‘Stopped’ (it defines what the PCE does when it is stopped), ‘Starting’ (it defines the starting sequence of the PCE), ‘Running’ (it defines what the PCE does when it is running), ‘Stopping’ (it defines the stopping sequence of the PCE), and ‘Fast stopping’ (it defines what the PCE should do when it has to shut down unconditionally as quickly as it

is safe to do so). Also, the following typical transitions should be drawn: a transition from 'Stopped' into 'Starting' caused by the operator command 'Start', a transition from 'Starting' into 'Running', a transition from 'Running' into 'Stopping' caused by the operator command 'Stop', a transition from 'Stopping' into 'Stopped', and a transition from 'Fast stopping' into 'Stopped'. Usually, all states except the 'Stopped' state must have a transition into 'Fast stopping', because of the operator command 'Abort' and/or the auto-stopping caused by the failures that are common to these states. A typical PCE failure occurs as a consequence of a PCE's inability to perform its processing, which is caused by a failure of one of the equipment entities used by that PCE. To avoid the repetition of information (i.e., multiple transitions), these states are enclosed into the 'Operating' super-state so that only one transition into 'Fast stopping' is needed.

The next guideline is to check if any state (or set of states) has additional specific auto-stopping causes. Such a state has to have its own transition into 'Fast stopping'.

Next is the investigation of the alarms, which are actually a set of exceptional conditions that are monitored and actions that are undertaken if these conditions are true (e.g., notify the operator). Often a state can have different alarms that cover different parts of that state. Consequently, this state should be split up into the needed number of states, which have different names that explain why they were split up.

When two or more elementary states have some common behavior (e.g., set of alarms) or outgoing transitions (e.g., due to common auto-stopping causes), the model engineer should introduce a super-state to avoid the repetition of information. Following this guideline improves the maintainability and comprehensibility of the STD. If the case occurs that a STD is hard to comprehend (e.g., because it has too many transitions), then the reason most likely lies in the failure to follow this guideline.

The next step is to define the detailed behavior through the action sequences of the states and transitions and the causes of the transitions. The action sequences define the processing needed to achieve a process-oriented goal, which is achieved by using the functions of individual devices as the means to achieve that goal. The current tool suite prototype supports the definition of the action sequences through the ST language. It is also possible that the system analyst would prefer to first go to the next development activity, and return to the definition of the detailed behavior later.

The deliverable of this development activity is a refined ProcGraph model, with information about the independent behavior of the identified software parts.

5.2.3. Modeling of interdependent behavior

The aim of this activity is to define the interdependent behavior between pairs of elementary PCEs that are behaviorally dependent, which is achieved by constructing SDDs. The system analyst first has to draw a composite dependency in the ED between the two interdependent elementary PCEs and then explode this dependency into a SDD. The model editor automatically copies the STD of both PCEs by creating proxies. The application engineer can only add elementary dependencies to the SDD. The number of elementary dependencies should be as low as possible, since (too) many dependencies are an indication that the decomposition into PCEs was not done well, as a consequence of failure in following the guideline of high cohesion and weak coupling (from Section 5.2.1). The visual appearance of the composite dependency that contains the SDD changes so that a union of the directions of the dependencies (both conditional and propagational) is visible.

The deliverable of this activity is a complete and syntactically correct model that is usable for code generation.

5.2.4. Code generation

In this development activity, the FBD and ST code for a specific platform is generated automatically from the complete ProcGraph model, which holds the main part of the application specification. First, a code generator that was already developed for a specific PLC (family or vendor) has to be selected. Currently, the tool suite prototype supports two formats through two code generators. MAGICS also enables the development of additional code generators.

An optional step before generating the code is to define the deployment of the software onto the hardware (i.e., the software-to-hardware mapping). Currently, MAGICS does not support the modeling of this aspect with a diagrammatic formalism. By default, all the generated code is deployed on one PLC in one task, which has turned out to be sufficient for the experimental projects that have been carried out using MAGICS. However, it is possible to redistribute the generated code to different processing nodes (PLCs) and tasks in two non-modeling ways. The first way is to alter the code generator to provide specific parts of the generated code with annotations, which will tell the target PLC IDE where to run it. The second way is "post generation", where the mapping is readjusted in the target PLC IDE after the generated code was imported into it. The main drawback of the second way is that the changes made are lost when the code is regenerated.

After all the mentioned aspects are defined, the deliverable of this activity (i.e., the code described in the import format of the selected target PLC IDE) can be automatically produced.

5.2.5. Testing

Because MAGICS does not provide a formal proof that the represented software is error-free, testing must be performed. However, the overall testing effort is reduced because MAGICS also relies on modeling, model reviews, automated well-formedness and syntactic correctness verification, and automatic code generation. According to our experience, these techniques significantly reduce the number of errors that exist before testing is undertaken. The testing activity can be started after the generated code is imported into the target PLC IDE. In general, an arbitrary testing technique for PLCs can be used with MAGICS. Because MAGICS does not yet provide tool support for testing, the testing procedure relies on the testing capabilities of the target PLC IDE.

Two different approaches to testing are possible with regard to the allocation of the testing code. One approach is to write the testing code into the ProcGraph model, generate the source code and perform the testing in the target PLC IDE. Because MAGICS currently lacks reverse engineering support, all the necessary code changes should be made in the model from which the updated source code is regenerated. This approach, where the testing code is intertwined with the code implementing the functionality, is most common in practice (Winkler, Hametner, & Biffl, 2009). Such code organization hinders efficient and systematic testing (Winkler et al., 2009). An alternative approach is to write and modify the testing code in the target PLC IDE and locate it in separate modules (FBDs) than the generated code. Thus, only the changes to the code that implements the functionality require the modification of the ProcGraph model and code regeneration.

6. An example process control software application

6.1. Introduction

This section presents the use of the MAGICS approach on an example that is a re-engineering of the software that was developed with the use of the semi-formally defined ProcGraph in a past industrial project. The first subsection introduces the technological process. The second subsection demonstrates how the

process control software was engineered with MAGICS to the source-code level by using the development process and guidelines that were defined in Section 5.2.

6.2. The technological process and application requirements

The aim of the project was to develop control software for the grinding of calcinated titanium dioxide (or calcinate, in short), which is a sub-process of titanium dioxide (TiO_2) production at the "Cinkarna" chemical works in Slovenia. This system contained around 50 devices that included nearly 400 signals. The requirements for this system were supplied in the form of a P&ID (its simplified version is shown in Fig. 7), a signal list, and functionality and safety requirements.

The calcinate grinding starts with the storage of the cooled calcinate in six silos, denoted as A1–A6. From each of these silos the material can be transported by means of vibrating sieves B1–B6, rotary valves C1–C6, screw-conveyors D1–D4, F, G, H, and I, and elevators E1 and E2 into the intermediate silo J. Any combination of the silos can be included in the calcinate dosing at a given time and the amount of material dosed from a particular silo can be controlled by the rotation speed of the rotary valve L. From the intermediate silo J, the calcinate is pneumatically transported (powered by the fan K) to the silo N. The air proceeds to the bag filter M, where it is cleaned and released into the atmosphere. The calcinate from the silo N is then dosed into the mill P by means of a vibrating sieve in the silo N and the screw-conveyor O. The mill crushes the material into a pigment. The air flow provided by the fan R transports the grounded pigment from the mill into the

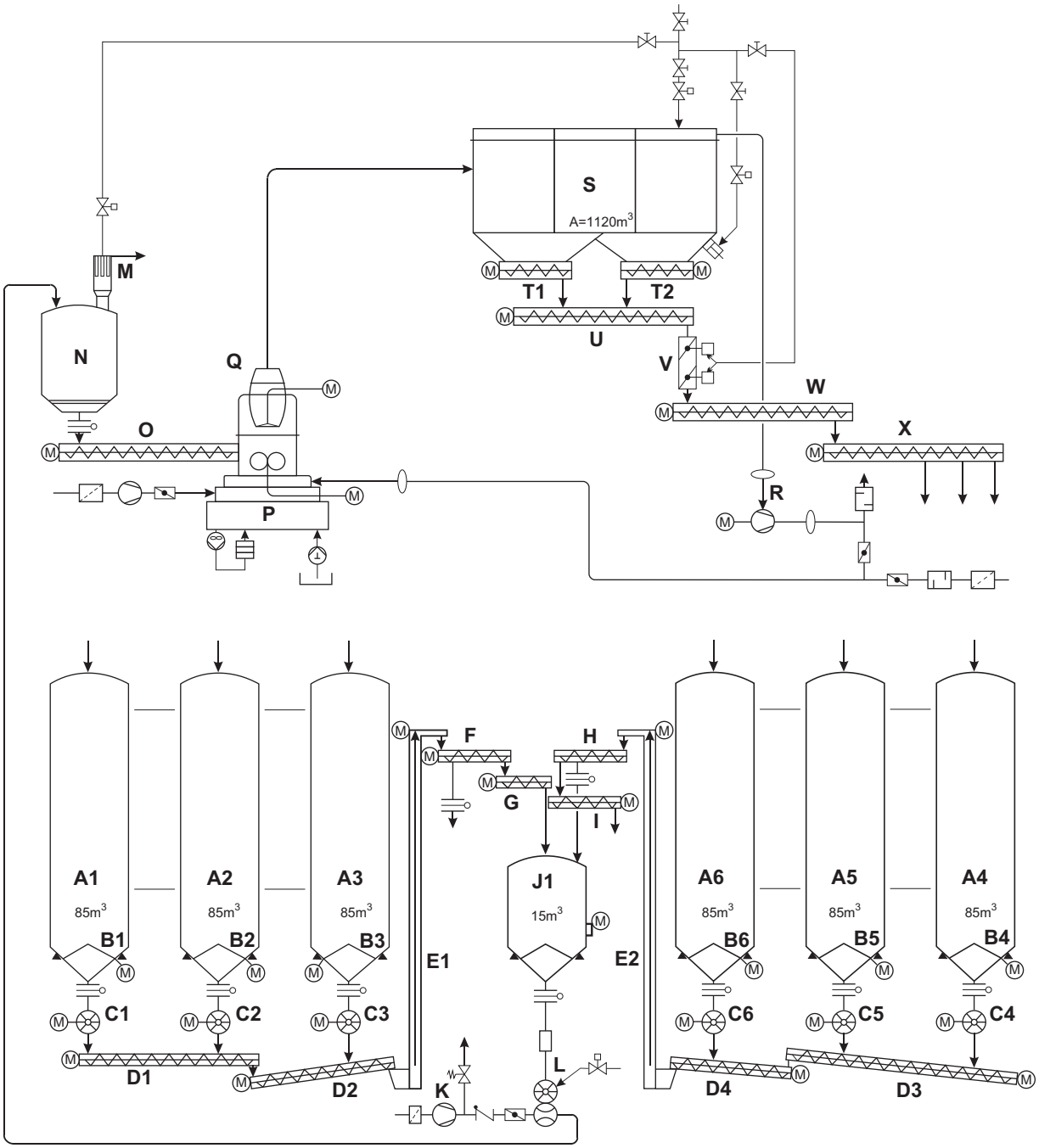

Fig. 7. Simplified technological scheme of the calcinate-grinding sub-process.

selector Q, where the coarse fraction of the pigment is separated and fed back into the mill. The ground pigment is pneumatically transported into the bag filter S, where it is separated from the air. The ground pigment that is caught in the bag filter S is transported with the screw-conveyors T1, T2, and U through a Double-Hatch Chamber (DHC) V to the screw-conveyors W and X, which transport it to the next sub-process of the TiO_2 production process.

6.3. Development with the MAGICS approach

First, the structural modeling activity (see Section 5.2.1) was undertaken in which the requirements were analyzed. Four operations of the calcinate-grinding process were identified and placed in the initial root ED, shown in Fig. 8: 'Input material transport' (using equipment A1 to I), 'Pneumatic transport' (using equipment J–M), 'Grinding' (using equipment N–R), and 'End product transport' (using equipment S–X). The 'Input material transport' and the 'End product transport' operations were decomposed into activities that are placed in their respective sub EDs. Due to the limited space available for this example, only the details of the 'Grinding' and 'End product transport' operations are presented.

The two activities into which the operation 'End product transport' is decomposed according to the guideline that suggested the separation of cyclical behavior and periodical behavior are shown in the initial sub ED in Fig. 9. Because the 'End product transport - Core' main activity controls the 'DHC control' auxiliary activity, they were connected with a composite dependency. However, currently it is not yet clear which states of the main activity will influence which transitions of the auxiliary activity, because the behavior of the activities has not been defined. Therefore, the composite dependency has an undefined shape (the thick gray connection in Fig. 9).

The next development activity (see Section 5.2.2) is to define the behavior of each elementary PCE in the emerging model. The final STD of the 'Grinding' operation in Fig. 10 was gradually constructed according to the guidelines described in Section 5.2.2. Initially, the typical states and transitions, including the 'Operating' super-state, were placed in the diagram. From the requirements, it was derived that the 'Grinding' operation should have two different running modes. Therefore, the 'Normal operation' and the 'Minimal load operation' sub-states were introduced. The latter handles the situation when an obstruction in the mill or in the pneumatic transport occurs, which requires the temporary stopping of the mill-feeding screw-conveyor. Another requirement was that this operation should not be turned on after a shutdown

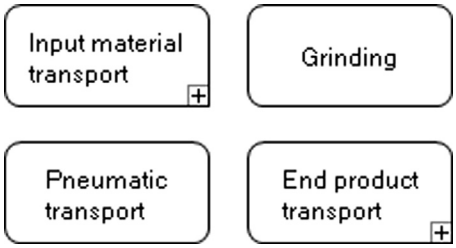

Fig. 8. The initial root ED for the example technological process.

Fig. 9. The initial sub ED of the 'End product transport' operation.

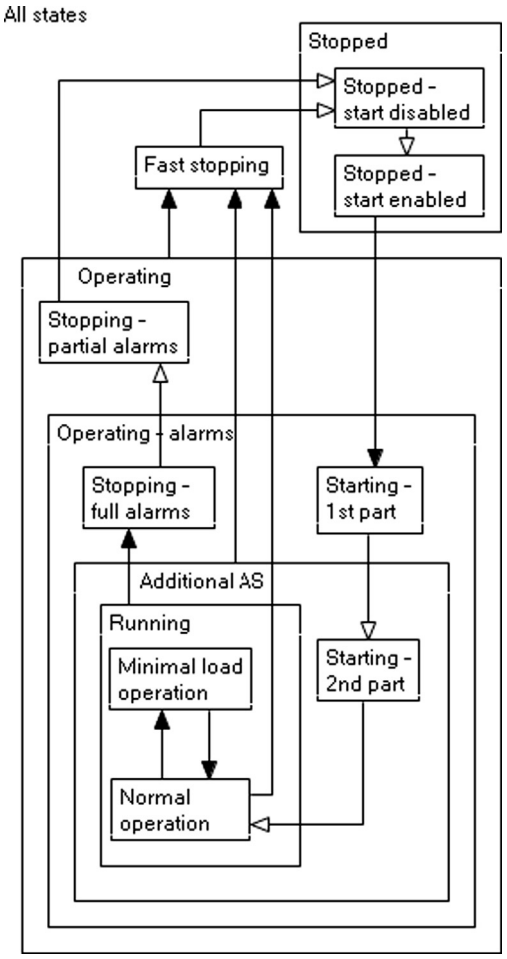

Fig. 10. The STD of the 'Grinding' operation.

for the time interval given by a parameter (around 2 min). The easiest way to ensure this was to introduce two 'Stopped' sub-states: 'Stopped-start disabled', which becomes active first and 'Stopped-start enabled', which becomes active automatically after the required delay has run out and from which the starting sequence can begin.

The next step was to look for specific auto-stopping causes. Some of the 'Operating' sub-states have common sets of additional failures; therefore, they were enclosed into the super-state 'Additional AS'. Notice that the 'Starting' state was split into the 'Starting 1st part' and 'Starting 2nd part', because the latter has additional auto-stopping causes. The 'Normal operation' state has even more additional auto-stopping causes. Therefore, it gets its own transition into the 'Fast stopping' state.

The next guideline suggests investigating the alarms. Consequently, the 'Stopping' state was split into two separate states. After looking at the commonality of the alarms, two super-states were introduced: 'All states', which means that there is a subset of alarms that have to be monitored in all states of the 'Grinding' operation, and 'Operating-alarms', which covers additional specific common alarms.

At this point, it is time to specify the detailed behavior. As an example, two state and two transition definitions are presented.

The processing of the 'Starting-2nd part' transient state (defined in Listing 3) sequentially turns on a set of devices and waits for the confirmation that they were turned on. These devices are the screw-conveyor O (M1354) and the rotary valve N (M1353). Then, it waits for the amount of time specified by the

'PP_G_DelayBeforeRunning', so that the operation's parameters can be stabilized in order to avoid unnecessary alarms.

Listing 3 The action sequence of the 'Starting-2nd part' state of 'Grinding'

```
TRANSIENT:
CASE StepCounter OF
 0: (* Turn on screw-conveyor O *)
    M1354_Command:=ON;
    (* Check if screw-conveyor O is turned on *)
    IF(M1354_State=ON)      THEN
          StepCounter:=StepCounter+1;
    END_IF;
 1: (* Turn on rotary valve N *)
    M1353_Command:=ON;
    (* Check if rotary valve N is turned on *)
    IF(M1353_State=ON)      THEN
          StepCounter:=StepCounter+1;
    END_IF;
 2: (* Start delay timer *)
    DelayTimer(IN:=TRUE, PT:=PP_G_DelayBeforeRunning, );
    (* Check if the timer ran out *)
    IF(DelayTimer.Q)      THEN
          StepCounter:=-1;
    END_IF;
END_CASE;
```

Listing 4 reveals that when the 'Minimal load operation' state is entered, the screw-conveyor O (M1354) is turned off to prevent the inflow of the calcinate into the mill. This then causes the mill to switch to the lowest operating speed (by means of the local automation, provided by the mill producer, which is therefore not specified as part of this project). Furthermore, the operator is notified that the 'NormalOperation' command is disabled. The loop sequence enables the return to the 'Normal operation' state through the 'NormalOperation' operator command, if the pressure value is below the return-to-normal value.

Listing 4 The action sequences of the 'Minimal load operation' state of 'Grinding'

```
ALWAYS:
(* empty *)
ENTRY:
(* Turn off screw-conveyor O *)
M1354_Command:=OFF;
PS_G_Message:='The command to return into the normal
  operation is DISABLED!';
NormalOperationEnabled:=FALSE;
LOOP:
(* Check if pressure has dropped below limit *)
IF(PT32831 <=(PP_G_PT32831_H_Pressure -
  PP_G_PT32831_H_DeltaPressure)) THEN
      PS_G_Message:='The command to return into the
                              normal operation is ENABLED!';
      NormalOperationEnabled:=TRUE;
ELSE
      NormalOperationEnabled:=FALSE;
END_IF;
EXIT:
(* Turn on screw-conveyor O *)
M1354_Command:=ON;
PS_G_Message:='';
```

Listing 5 specifies that the transition from the 'Normal operation' into the 'Minimal load operation' is fired when the air pressure at the input of the mill is too high.

Listing 5 The transition from the 'Normal operation' into the 'Minimal load operation' state of 'Grinding'

```
CAUSE:
PT32831 >=PP_G_PT32831_H_Pressure
ACTIONS:
(* empty *)
```

The return back to the 'Normal operation', which is defined in Listing 6, occurs when the operator issues the 'NormalOperation' command and the 'NormalOperationEnabled' is true.

Listing 6 The transition from the 'Minimal load operation' into the 'Normal operation' state of 'Grinding'

```
CAUSE:
(OC_G_Command=NormalOperation) AND
  NormalOperationEnabled
ACTIONS:
(* empty *)
```

The STDs of 'End product transport - Core' and of 'DHC control' are created in a straightforward way, due to following of the guidelines and are therefore not discussed.

The next development activity is the modeling of the inter-dependent behavior (see Section 5.2.3). The SDD in Fig. 11 defines that the 'End product transport - Core' main activity turns the 'DHC control' on and off through the three propagational dependencies.

The emergent model and the requirements revealed that 'Grinding' and 'End product transport - Core' are interdependent. Therefore, a composite dependency between them was added and the SDD in Fig. 12 was defined. The 'End product transport - Core' has to be running before 'Grinding' can begin its transition into its first starting state. On the other hand, the 'End product transport - Core' can only go into its first stopping state when 'Grinding' is stopped, except when 'End product transport - Core' goes into fast stopping, which must then be followed by fast stopping of the 'Grinding' operation.

After all the SDDs have been defined, the composite dependencies in the initial root ED (Fig. 8) and the initial sub ED of 'End product transport' (Fig. 9) change their appearance to show the union of the elementary dependencies, which were defined in the

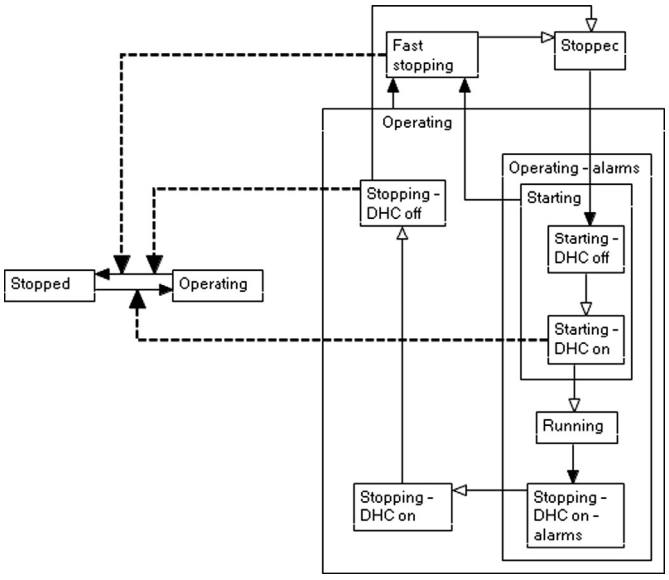

Fig. 11. The SDD of the 'DHC control' and 'End product transport - Core' activities

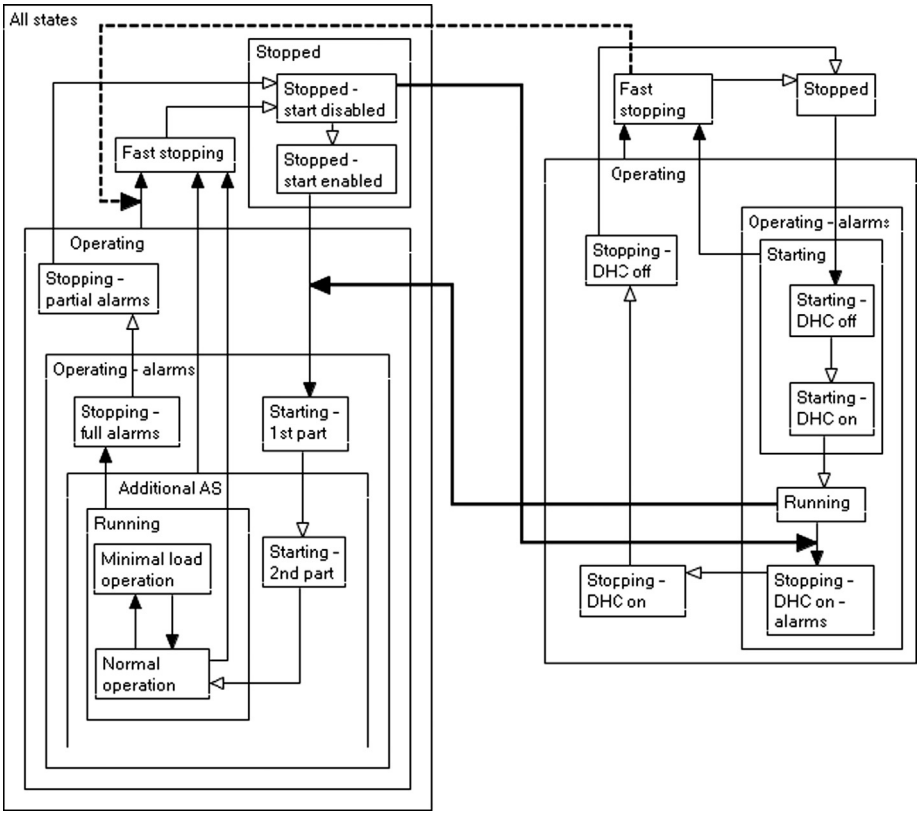

Fig. 12. The SDD of the 'Grinding' and 'End product transport - Core' PCEs.

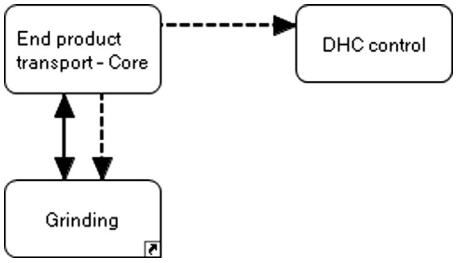

Fig. 13. The final sub ED of the he 'End product transport' ED.

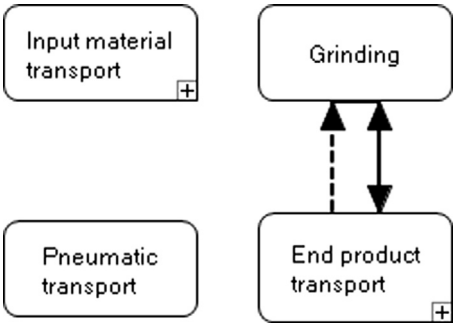

Fig. 14. The final root ED for the example process.

SDDs. The 'Grinding' is now present in the sub ED (in Fig. 13) as a PCE proxy that references the original, which was defined in the root ED (in Fig. 8). Consequently, the composite dependency between 'End product transport - Core' and 'Grinding', which actually spans across two hierarchy levels, also becomes visible in the root ED (in Fig. 14).

After the whole model is defined and validated by the modeling tools to ensure its completeness and conformity with the constraints, the code can be generated automatically (according to Section 5.2.4). For this example, the generator for Mitsubishi PLCs was selected. The original project, which was remodeled with MAGICS in the presented example, was used in the verification of the code generator. Therefore, the automatically generated code from the example is semantically equivalent to the manually transformed code from the pen-and-paper form of ProcGraph models in the original project. An excerpt of the automatically generated code is shown in Fig. 15. It shows the top-level code that was generated from the STD of the 'End product transport - Core'. The mentioned STD was transformed into an FBD according to the rules defined in the model transformation rules section (Section 5.1.5), particularly rule r2. In the FBD, three networks can be seen, each with one function block. This corresponds to the number of top-level states of the source STD. It is visible that the 'EPTC_Stopped' and 'EPTC_FastStopping' function blocks are active when the respective state from which they were transformed is the active state of the EPTC FBD (which is a transformation of the 'End product transport – Core' activity). The 'EPTC_Operating' function block is active when any of the elementary sub-states of the super-state from which it was transformed is the active state.

The generated code was compiled in the PLC IDE. The result was 118922 bytes of executable code, of which 80867 bytes (68%) were generated from the graphical high-level part and 38055 bytes (32%) were generated from the lower-level textual part.

7. Discussion

The aim of the presented example was to demonstrate the MAGICS approach and show its usability in practice. Apart from

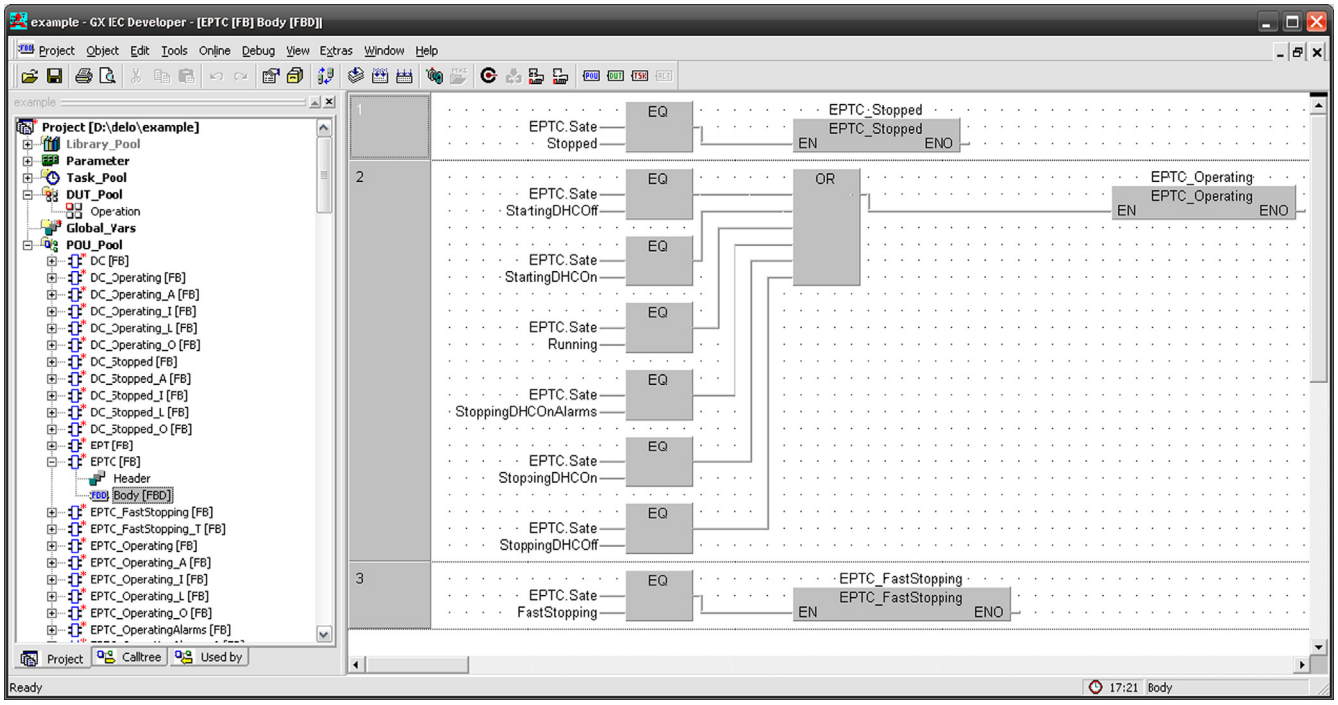

Fig. 15. A screenshot of the generated project that shows an open FBD.

this example, MAGICS has currently been used in laboratory experiments and for the re-engineering of other existing software. A qualitative evaluation of MAGICS with the presented example was performed based on the limited data about the original project and the collaboration with some of the engineers from the original project. This evaluation has revealed several benefits of MAGICS when it was compared to the original development approach of the project. The same benefits were reported by the end-users that participated in the testing of the MAGICS tool suite (this was based on their comparison to the development of process control software without MAGICS), which are as follows:

- *Improved software quality*. Due to the code generators, which were tested, and repeat the same transformations each time they are invoked, the human coding errors, which can occur with manual transformations, are eliminated. The ProcGraph language, which was selected and adopted for MAGICS through formalization and addresses issue i1 (the use of inadequate abstractions), is already associated with fewer software defects and a general increase in quality (Godena, 2004). The model editor limits the construction to only syntactically valid ProcGraph models and provides an OCL-based constraint checker, which prevents non-trivial structural model errors according to the well-formedness criteria. This is a small step toward addressing issue i4 related to the lack of validation or verification. A lower probability that errors are introduced in the models consequently lowers the probability of errors occurring in the generated software.
- *Improved productivity*. This improvement is due to: (a) the automated model-to-code transformations that are executed by the code generators, which addresses issue i2 (the semantic distance between the early development phases and the implementation phase); (b) the definition of the development process and guidelines, which guide the developers and reduce the possibility of undertaking the wrong development activities at the wrong time.
- *Improved communication and interaction between the development participants*. The code generators determine the semantics of the models. Therefore, ambiguous interpretations and misunderstandings can be avoided. The ProcGraph DSML has the potential to be adopted by developers in the control process domain, since it has a small set of expressive elements. Therefore, it addresses issue i3 (the developer specifics), with the prerequisite that the practitioners embrace the shift to the model-centric paradigm and receive the necessary training.

These benefits are consistent with the literature (Sprinkle et al., 2009; Kelly & Tolvanen, 2008), which reports that adopting an appropriate and already proven DSML into MDE (through formalizing the DSML and model transformations, and providing tool support with automatic code generation) will most likely bring productivity and quality gains.

The potential users of the MAGICS approach should be aware of its current limitations, which are:

- No support for the modeling of the application deployment. To mitigate this weakness, two non-modeling ways for addressing the deployment concern can be used (see Section 5.2.4).
- The tool suite is still a prototype.
- Code-to-model transformations, which are commonly known as reverse engineering, are not supported, which is also the case in all the reviewed state-of-the-art approaches.

The current version of MAGICS was developed specifically for the domain of industrial process control systems that are based on PLCs. Although the generalization of our work was not a current research objective, at least two possibilities for broadening the scope of this work can be realized. The first option would be to apply the clearly documented, major design decisions that were made through the development of MAGICS and the rationale behind them to the development of other MDE approaches. MDE practitioners may face the same questions for a related or even an unrelated domain. The second option would broaden the scope of MAGICS by supporting the automatic code generation for hardware platforms other than PLCs. This would be achieved through defining additional transformation rules and implementing them

in a new code generator. This option will be considered when these alternative platforms achieve a more significant market share.

8. Conclusions and future work

This paper presented MAGICS – a new MDE approach to process control software engineering – that enables automatic code generation for the PLC platform. MAGICS differs from most other approaches in that it uses process-centric instead of device-centric abstractions. The MAGICS approach is aligned with the issues of the process control domain and aims to overcome several weaknesses of the state-of-the-art approaches that were analyzed over various dimensions. It makes use of an existing process-centric modeling language, which was formalized and extended so it could be used as the main DSML of MAGICS. The MAGICS approach is illustrated using parts of an industrial project where the control software of a technological process of moderate complexity was developed. The applicability and usefulness of MAGICS was demonstrated using this example.

In the future, three main research directions will be followed to improve the MAGICS approach. The first research task will be to provide empirical data about MAGICS through extensive experimentation. The second direction is the development of an automatic verification tool for various properties of the modeled software (e.g., safety properties, liveliness or the absence of deadlocks). The integration with the MARTE profile (Object Management Group, 2011a), which also provides some support for the verification of real-time related properties, will also be considered. The third direction is to extend the MAGICS approach by adopting an existing formalism, or developing a new formalism, which will enable the modeling of the software-to-hardware mapping.

Acknowledgments

The support of the Ministry of Higher Education, Science and Technology of the Republic of Slovenia is gratefully acknowledged. This work is supported in part by NSF CAREER award CCF-1052616.

References

3S-Smart Software Solutions. CoDeSys. (2012). Available from ⟨http://www.3s-software.com/⟩ Retrieved 11.04.12.

Avila-Garcia, O., & Garcia, A. E. (2008). Providing MOF-based domain-specific languages with UML notation. In: *Proceedings of the 4th workshop on development of model-driven software*, San Sebastian, Spain.

Bryant, B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., & Karsai, G. (2011). Challenges and directions in formalizing the semantics of modeling languages. *Journal of Computer Science and Information Systems*, *8*(2), 225–253.

Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., & Grose, T. J. (2003). *Eclipse modeling framework*. MA, USA: Addison-Wesley Professional Reading.

Colla, M., Leidi, T., & Semo, M. (2009). Design and implementation of industrial automation control systems: a survey. In: *Proceedings of the IEEE international conference on industrial informatics* (INDIN'09), Cardiff, UK.

Estévez, E., Marcos, M., & Orive, D. (2007). Automatic generation of PLC automation projects from component-based models. *The International Journal of Advanced Manufacturing Technology*, *35*(5), 527–540.

Estevez, E., Marcos, M., Sarachaga, I., & Orive, D. (2007). A methodology for multidisciplinary modeling of industrial control systems using UML. In: *Proceedings of the IEEE international conference on industrial informatics* (INDIN'07), Vienna, Austria.

Evans, A., Fernández, M. A., & Mohagheghi, P. (2009). Experiences of developing a network modeling tool using the eclipse environment. In: *Proceedings of the 5th European conference on model driven architecture—foundations and applications* (ECMDA-FA'09), Enschede, The Netherlands.

Fischer, K., Hordys, G., & Vogel-Heuser, B. (2004). Evaluation of an UML software engineering tool by means of a distributed real time application in process automation. In: *Proceedings of the modellierung* 2004, Marburg, Germany.

Frey, G., & Litz, L. (2000). Formal methods in PLC programming. In: *Proceedings of the IEEE international conference on systems, man, and cybernetics* (SMC'00), Nashville, TN, USA.

Friedrich, D., & Vogel-Heuser, B. (2007). Benefit of system modeling in automation and control education. In: *Proceedings of the American control conference* (ACC'07), New York, NY, USA.

Gérard, S. (2012). Papyrus UML. Available from ⟨http://www.papyrusuml.org/⟩ Retrieved 11.04.12.

Godena, G. (2004). ProcGraph: a procedure-oriented graphical notation for process-control software specification. *Control Engineering Practice*, *12*(1), 99–111.

Gronback, R. C. (2009). *Eclipse modeling project: a domain-specific language (DSL) toolkit*. Netherlands: Addison-Wesley Professional Amsterdam.

Haase, A., Volter, M., Efftinge, S., & Kolb, B. (2007). Introduction to open architecture ware 4.1.2. In: *Proceedings of the model-driven development tool implementers forum* (MDD-TIF'07) *co-located with TOOLS* 2007, Zurich, Switzerland.

Hästbacka, D., Vepsäläinen, T., & Kuikka, S. (2011). Model-driven development of industrial process control applications. *Journal of Systems and Software*, *84*(7), 1100–1113.

Heck, B. S., Wills, L. M., & Vachtsevanos, G. J. (2009). Software technology for implementing reusable, distributed control systems. *IEEE Control Systems Magazine*, *23*(1), 267–293.

Hnetynka, P., & Plasil, F. (2008). The power of MOF-based meta-modeling of components. In: *Proceedings of the advanced software engineering and its applications* (ASEA'08), Hainan Island, China.

Jones, C. T. (1996). *Programmable logic controllers the complete guide to the technology: P*. Atlanta, GA, USA: Turner Publishing.

Kandare, G., Strmčnik, S., & Godena, G. (2010). Domain specific model-based development of software for programmable logic controllers. *Computers in Industry*, *61*(5), 419–431.

Katzke, U., & Vogel-Heuser, B. (2005). UML-PA as an engineering model for distributed process automation. In: *Proceedings of the 16th IFAC world conference*, Prague, Czech Republic.

Kelly, S., & Tolvanen, J.-P. (2008). *Domain-specific modeling*. Hoboken, NJ, USA: John Wiley & Sons.

Kern, H., & Kühne, S. (2009). Integration of microsoft visio and eclipse modeling framework using m3-level-based bridges. In: *Proceedings of the 2nd model driven tool and process integration* (MDTPI'09), Enschede, The Netherlands.

Klatt, B. (2008). Xpand: a closer look at the model2Text transformation language. In: *Proceedings of the 12th European conference on software maintenance and reengineering* (CSMR'08), Athens, Greece.

Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., et al. (2001). Composing domain-specific design environments. *IEEE Computer*, *34*(11), 44–51.

Lewis, R. W. (1998). *Programming industrial control systems using IEC 1131-3*. The Institution of Engineering and Technology, London, UK.

Lewis, R. W. (2001). *Modelling control systems using Iec 61499: applying function blocks to distributed systems*. The Institution of Engineering and Technology, London, UK.

Lukman, T., Godena, G., Gray, J. G., & Strmčnik, S. (2010). Model-driven engineering of industrial process control applications. In: *Proceedings of the IEEE international conference on emerging technologies and factory automation* (ETFA'10), Bilbao, Spain.

Lukman, T., & Mernik, M. (2008). Model-driven engineering and its introduction with metamodeling tools. In: *Proceedings of the international Ph.D. workshop on systems and control'08*, Izola, Slovenia.

Maurmaier, M. (2008). Leveraging model-driven development for automation systems development. In: *Proceedings of the IEEE international conference on emerging technologies and factory automation* (ETFA'08), Hamburg, Germany.

Mitsubishi Electric. GX IEC Developer. (2011). Available from ⟨http://www.mitsubishi-automation.com/products/software_gx_iec_developer.htm⟩ Retrieved 03.05.11.

Mohagheghi, P., & Dehlen, V. (2008). Where is the proof?-a review of experiences from applying MDE in industry. In: *Proceedings of the European conference on model driven architecture: foundations and applications* (ECMDA-FA'08), Berlin, Germany.

Noyrit, F., Gérard, S., & Selic, B. (2012). Facademetamodel: masking UML. In: R. B. France, J. Kazmeier, R. Breu, & C. Atkinson (Eds.), *Model driven engineering languages and systems* (pp. 20–35). Berlin, Germany: Springer.

Object Management Group. (2006). Meta object facility: MOF core specification, Version 2.0.

Object Management Group. (2011a). UML profile for modeling and analysis of real-time and embedded systems (MARTE),Version 1.1.

Object Management Group. (2011b). UML superstructure specification version 2.4.1.

Oliva, E. (2009). Interactive graphical maps for infocenter via model to model transformation. In: *Proceedings of the eclipse-IT 2009*, Bergamo, Italy.

Panjaitan, S., & Frey, G. (2007). Combination of UML modeling and the IEC 61499 function block concept for the development of distributed automation systems. In: *Proceedings of the IEEE international conference on emerging technologies and factory automation* (ETFA'06), Prague, Czech Republic.

PLCOpen international organization. PLCOpen. (2010). Available from ⟨http://www.plcopen.org/⟩ Retrieved 15.01.10.

Sani, A. A., Polack, F. A. C., & Paige, R. F. (2011). Model transformation specification for automated formal verification. In; *Proceedings of the 5th Malaysian conference in software engineering* (MySEC'11), Johor, Malaysia.

Schmidt, D. C. (2006). Model-driven engineering. *IEEE Computer*, *39*(2), 25–31.
Selic, B. (2012). The less well known UML. In: M. Bernardo, V. Cortellessa, & A. Pierantonio (Eds.), *Formal Methods for Model-Driven Engineering* (pp. 1–20). Berlin Heidelberg: Springer.
Sendall, S., & Kozaczynski, W. (2003). Model transformation: the heart and soul of model-driven software development. *IEEE Software*, *20*(5), 42–45.
Sprinkle, J., Mernik, M., Tolvanen, J.-P., & Spinellis, D. (2009). What kinds of nails need a domain-specific hammer? *IEEE Software*, *26*(4), 15–18.
Strasser, T., Rooker, M., Ebenhofer, G., Hegny, I., Wenger, M., Sunder, C., et al. (2008). Multi-domain model-driven design of industrial automation and control systems. In: *Proceedings of the IEEE international conference on emerging technologies and factory automation* (ETFA'08), Hamburg, Germany.
Streitferdt, D., Wendt, G., Nenninger, P., Nyβen, A., & Horst, L. (2008). Model driven development challenges in the automation domain. In: *Proceedings of the IEEE international computer software and applications conference* (COMPSAC'08), Turku, Finland.
Temate, S., Broto, L., Tchana, A., & Hagimont, D. (2011). A high level approach for generating model's graphical editors. In: *Proceedings of the 8th international conference on information technology: new generations* (ITNG'11), Las Vegas, NV, USA.
Thramboulidis, K. (2004). Using UML in control and automation: a model driven approach. In: *Proceedings of the IEEE international conference on industrial informatics* (INDIN'04), Berlin, Germany.
Thramboulidis, K. (2005). IEC 61499 in factory automation. In: *Proceedings of the IEEE international conference on industrial electronics, technology and automation* (IETA'05), Bridgeport, CT, USA.
Thramboulidis, K. (2009). The function block model in embedded control and automation from IEC61131 to IEC61499. *WSEAS Transactions on Computers*, *8*(9), 1597–1609.
Thramboulidis, K., Perdikis, D., & Kantas, S. (2007). Model driven development of distributed control applications. *The International Journal of Advanced Manufacturing Technology*, *33*(3), 233–242.
Tranoris, C., & Thramboulidis, K. (2006). A tool supported engineering process for developing control applications. *Computers in Industry*, *57*(5), 462–472.
Vogel-Heuser, B., Witsch, D., & Katzke, U. (2005). Automatic code generation from a UML model to IEC 61131-3 and system configuration tools. In: P*roceedings of the IEEE international conference on control and automation* (ICCA'05), Budapest, Hungary.
Winkler, D., Hametner, R., & Biffl, S. (2009). Automation component aspects for efficient unit testing. In: *Proceedings of the IEEE international conference on emerging technologies & factory automation* (ETFA'09), Mallorca, Spain.
Yourdon, E., & Constantine, L. L. (1979). *Structured design: fundamentals of a discipline of computer program and systems design*. Upper Saddle River, NJ, USA: Prentice Hall.