A Middleware for Software Evolution of Automation Software

Ingmar Kühl
Nordex Energy
Langenhorn Chaussee 600
D-22419 Hamburg
Tel.: +49 (0)40 300-30-1964
E-Mail: ikuehl@nordex-online.com

Alexander Fay
Faculty Machine Construction
Helmut-Schmidt-University
Holstenhofweg 85
D-22043 Hamburg
Tel.: +49 (0)40 6541-2254
E-Mail: alexander.fay@hsu-hh.de

Abstract

One of the key factors for automation systems in the domains smart grids, digital factories and modern process systems are the permanently shift of their requirements. Thus, software has to be evolve over time. This issue has been addressed by different solutions for evolving automation software. Though, they have their specific benefits, but lacks on deficiencies like a specific (non-standard) programming model, specific programming languages and a very narrow focus of a specific automation domain. In contrast to this a novel approach will be discussed to provide a more flexible and quite general solution, a common middleware, to reconfigure automation software dynamically with minimal downtime. A middleware to integrate different programming languages, container systems, frameworks, libraries, control applications and drivers as well.

1. Introduction

Programmable Logic Controllers (PLCs) and Industrial PCs (IPCs) are the corner stone of automation systems in factory and process automation, and play an important role in different other domains like for e.g. telecontrol. As the upcoming of PLCs some 40 years ago has been provided more flexibility to develop and maintain automation systems in terms of software in spite of hardware, the dynamic interchange of software components of a PLC with near-to-zero downtime some years ago has been provided the flexibility to alter automation systems while staying in full service. Because automation software evolves over time, this is a key requirement in the beginning age of decentralized energy generators and consumers prominently known as smart grid [1]. Automation systems have to be upgraded to new communication protocols and to provide new processing rules, if the interconnection of different grids will be forced in future, for instance to integrate the energy grids in middle and northern Europe. This is highly demanded because of a lack of energy storage.

The modification of an automation system has to be done in time; existing parts should not be affected. Another domain of interest is factory automation, for which the Iacocca Institute has provided twenty years ago an Agile Manufacturing [17] concept. Aim of this concept is a reconfiguration of all plant levels. This not only for software components, but also for the needed hardware. Such a reconfiguration for a working plant is needed to reduce costs because of downtimes due to modernization or to allow cost-effective productions of very small pieces.

A third domain of interest is the domain of enterprise computing software, where a reconfiguration of existing software is highly demanded, to lower maintenance costs. These software systems are very often backed by any kind of component container (with knowledge of the full lifecycle of the respective components). In the automotive domain where with OSGi a rather service-oriented storage has been successful used (though this services can rise up to very big data packages relating them to components). Due to the merging of automation and control and computer science the used technologies in the latter domain become more and more relevant from day to day for the first domain.

Common to all sketched solutions of the different domains is the need to minimize downtimes by dynamic reconfiguration of a system without a complete shutdown – the contrast to static configuration, which does need a complete shutdown of a system. Such a static reconfiguration is very costly and should be replace by a dynamic one. The yet known solutions are very specific: for dynamic reconfiguration of PLCs most technologies aren't based on IEC 61131-3 due to its execution model and because that there is no state information available. Furthermore, there is frequently usage of global variables which complicate the dynamic exchange due to increased dependencies. In contrast to that the Service Interface Function Blocks (SIFB) of IEC 61499, which define different services for the management of function blocks and their respective functions, allow a very modular software architecture and are therefore very often used for reconfiguration
purposes of industrial automation systems. In the domain of automotive OSGi [30] has been used, which is only available for Java. For enterprise computing (which is more and more connected with automation applications) often a component-based container solution to reconfigure software systems has been employed. Most of the proposed solutions of reconfiguring embedded software are also based on the concept of a component. Very different solutions have to be used to solve the task of reconfiguration with near-to-zero downtime. Very similar tasks are solved by very different technologies. Taking into account that the design knowledge of a reconfiguration solution is given by any kind of code representation, but not in any detail by documents and formal models, there is less reuse, if one solution is given in programming language X and another in programming language Y. For instance, if a dynamic reconfiguration approach has been used for IEC 61131-3, one can’t add programs with an OOP extension to the same system without a shutdown and redesign of the core system. Another example is a seamless integration of OSGi modules with PLC software components for instance in a SCADA system with web service support without foreign function interface (FFI) wrappers (which decrease the processing performance by a huge amount).

Viewing the sketched implementation technologies and domains more or less as parts of a big automation domain sheds light to the question whether one can find a far more generic solution, a common automation middleware for reconfiguration and software evolution, which is capable to be used in much more domains, with respective modifications? Such a common solution to the same task will allow reduce development costs in terms of a software product line, but also to ease development. An implementation for a specific scenario and domain needs then just an adoption of the given generic middleware. How such a common software basis can be found is the concern of this article.

The next section will derive needed properties for a quite reusable and adaptable reconfiguration process of automation software across different domains. Subsequently the state-of-the-art of reconfiguration approaches will be laid out and analysed in respect to the described properties. In addition a new concept fulfilling the derived properties will be proposed.

1.1. Needed properties for reconfiguration

Despite different domains with their respective different technologies to solve the problem of reconfiguration the underlying process is common to every automation device sketched in the introduction: one has to minimize downtime, one must store the state of processing, persist pending requests and blocking functions (or methods), which are directly affected by a reconfiguration, one needs to exchange functions or methods and classes of version i with their modified version i+1 and restore the state and processing the persisted requests. The needed commands, create, remove, link and unlink, already investigated for a reconfiguration process by Kramer and Magee [10] as well the required states to transform a configuration of an embedded system in a consistent manner are common to different domains, different hardware and different software technology. Thus, automation systems shouldn’t be viewed as single items of a uniquely development process, instead they have to be seen as adoptions of a software product line, i.e. a common technology foundation for adoption to different domains. To build such a product line one needs a common software technology as a platform which satisfies the needs of different domains. From a software engineering perspective it is required to support different programming languages. PLC controller are most often developed by one of the languages of the IEC 61131-3 (graphical- or text-based). Other automation systems need support for C and C++. In the last years there is some intense usage in the domain of factory and process automation of languages supporting virtual machines like JVM or Microsoft .NET Common Language Runtime (CLR) languages (for instance Java, Microsoft .NET C++ and C#). A virtual machine (VM) has some clear benefits: portability, security, Just-in-time-Compiler (JIT) to boost performance in time, ease of development in conjunction with a garbage collector, multi threading and others. Using a JIT one can achieve lifetime analysis which results in iteratively optimized code. Especially for use with embedded controllers a VM capable to do real-time is needed, i.e. to prioritize tasks like the run of the garbage collector. The mentioned languages must be supported by a common programming model for automation systems. Therefore, at least one needs a virtual machine with different frontends (programming languages) and a common virtual instruction set. Additionally, for a VM different backends can be engineered quite easily, i.e. driver support for different hardware modules. Moreover, to reconfigure automation systems with different frontends languages and with no a priori knowledge of which language to use, one needs a JIT Compiler, instead of an Ahead-Of-Time (AOT) Compiler. With an AOT Compiler one develops code in a classical manner by a develop-compile-link-deploy cycle, whereas a JIT enables to compile and recompile a running system. Using only an AOT Compiler one could provide a solution to known programming languages in advance, but for a modification of a language or a totally new one (used by a runtime environment which is up and running) this is a big limitation, i.e. the system has to be shutdown and replaced by one with support for the demanded programming language features. It has been pointed out by Magee and Kramer [15] more than 25 years ago that not every demanded feature can be designed in advance and thus a reconfiguration process is highly demanded. The JIT provides the capability to
compile demanded features instantly on a runtime environment.

To add new features to a running software system in a generic manner one has to rethink the cycle develop-compile-link-deploy. With this classical development approach one needs specific services to reconfigure compiled code, even in an intermediate language (like Java bytecode). But with specific services one can only do specific tasks, defined in advance. For instance most reconfiguration approaches are based on a component model. An infrastructure to manage these components is needed. Such a infrastructure isn’t interchangeable without a stop. The entities of reconfiguration are build by such a component model and have therefore an undesired dependency. If the requirements will evolve it is needed to reinstall the whole system. Therefore, to get the full power of a virtual machine with a JIT Compiler one needs a kind of generic interaction solution, i.e. a shell to read and eval commands. This shell must be connected to the engineer via a network connection. By utilizing such a solution one is independent of a fixed interface of a component or service container. Only a network channel must negotiated by client and server. This is a much lesser requirement as a rigid interface between component server and client. If it is demanded, a component container infrastructure can be used in conjunction with such a generic network interaction on top of a VM. To define new languages on top of a VM and a host language one needs a specific property, a language which is homoiconic, i.e. the same representation for functions, methods and processing on the one hand and data on the other hand. Bundled with a rewrite facility, one can define instantly new programming languages, without affecting existing software entities of a given programming language. With this feature whole new language abstractions can be designed, i.e. a feature which one has missed for a long period can be added without any redeploy, shutdown or restart. This is contrary to languages like IEC 61131-3 ST, C++ or Java which separate clearly between language keywords (like class, abstract or FUNCTION) and programs constructed by these keywords. To be exact, this is not a drawback of the programming languages, but of the used engineering tools and runtime environments. In a quite different environment (fulfilling the described requirements) it would be possible to add new non-standard features to IEC 61131-3 ST. This can be realized by defining IEC 61131-3 ST as a domain specific language (DSL) using a VM with a homoiconic host language. Whether this is desirable in any way is another question, but it’s for sure a useful feature for many scenarios.

The required features can be subsumed as follows:

• a VM with real-time capabilities,
• a generic (network-based) interaction channel in conjunction with a homoiconic host-language with a rewrite system.

In the following section the state-of-the-art of reconfiguration approaches for embedded devices will be presented and an evaluation of the proposed requirements will be given. Moreover, some interesting approaches of reconfiguration in other domains will also be described to present solutions of different domains and to provide the applicability of reconfiguration across domains.

1.2. State-of-the-art

One of the first solutions to change program data for standard and enterprise software while the system itself is in full service was the Java Hot Deployment [6]. A representative example is the deployment capability of the Jakarta tomcat web server [31]. Dropping a new web archive does deploy a new version of a web project. This reconfiguration has neither a systematic persisting concept for the actual state, nor a concept to store pending requests. Another approach of hot deployment is to use network or URL class loader, i.e. loading of needed classes not from the local file system, but via the network. Though, this deployment approach does allow altering existing software quite easily, it doesn’t do it in a very controlled way (no state information can be saved temporarily, no scheduled tasks will be dumped, if not specifically implemented). A more systematic approach is given by OSGi [19], which stood initially for Open Services Gateway initiative. OSGi is the approach to define a pool of services for the automotive domain by using a clear modularization of software components. Today OSGi has been successful used also in facility management, cell phones, home electronics and enterprise software and other domains. For instance, Dorminger [4] has presented a solution to the process domain, in which OSGi service-based components have been build the backbone for algorithmic computations of a steel making process. The components are stored in a container, which holds all information about the whole lifecycle of each component. In that feature OSGi is closely related to a huge class of components [22], i.e. to all components with a repository which is able to control the lifecycle of the respective components. A component is especially defined by its property not affecting other components. Thus, a component can be shut down and replaced by a newer one without affecting not related components. The state of processing will be stored and restored as the queued tasks, too. This is a quite systematic solution. Nonetheless, OSGi is very specific to Java (though there exist also a small solution for C++, too [29]) and lacks support for well known domain specific languages (DSLs) of PLCs, like IEC 61131-3 languages and software distribution with IEC 61499.

Inspired by container based component solutions to modify deployed modules using full lifecycle information researchers in the field of automation and
control have been investigated how to define concepts for interchanging data and program structures for PLCs. The general model of a reconfiguration process was defined by Kramer and Magee [9], which is pictured in Figure 1. Though, stated initial for embedded software systems this basic model was adapted to real-time control software by different authors. See for reference [28].

The authors have presented a method for reconfiguration of distributed software based on a DSL, which they named CONIC. The development language is clearly separated from the reconfiguration language. The reconfiguration objects, programs in CONIC, can be composed of components, which provide the functionality of a software system. Moreover, the authors have defined a process of reconfiguration, such that the state transformation results in a consistent state [10]. This process does consist of different states and needed commands to destroy connections to existing objects and to restore them. This concept has been validated in the same paper by a complex example, “the evolving philosophers”, which is an extension of the classical example of synchronization and parallelism in Computer Science, the “dining philosophers”. Philosophers can die or born, which is reflected by a sequence of commands. CONIC has been used by a bunch of universities and industrial companies as well.

Based upon the fundamental work of Kramer and Magee different authors have provided solutions for reconfigurations of industrial automation systems. Wermelinger [26] has investigated how to assure integrity of systems under change. He has analyzed two scenarios, the “passive” one, and the “blocking” one. At the passive scenario, a component is assumed not engaged to any transaction and does not start any. In the “blocking” scenario, a component is assumed consistent except during transactions. Utilizing these two scenarios the author has reduced the blackout time of components under reconfiguration opposed to the approach of Kramer and Magee. Furthermore, Wermelinger has extended the approach of CONIC to a hierarchical component system. This has been also stated formally. Rasche and Polze [20] have been defined a Microsoft NET framework, Adapt.NET, to improve reconfiguration concepts of Kramer and Magee. They have defined a component model to concentrate data and processing units under reconfiguration. Only those parts of a component which are directly connected to other components and are under reconfiguration will be freezed, as opposed to the model of Kramer and Magee. The authors addressed specifically the case of erroneous behavior of components. Brinkschulte et al. [2] have defined a service-based framework for real-time reconfiguration which allows the specification of deadlines and priorities for reconfiguration requests. The framework OSA+ is a microkernel architecture, i.e. with a minimal kernel of functionalities. Additive functionality can be added in terms of new services. For instance, the reconfiguration capability is defined in an optional service, the Configurator, which loads a new service, analyze the old one and assure system integrity while reconfiguring. The services are communicating via jobs. Stewart et al. have presented a framework to reconfigure real-time control software of modular and reconfigurable robots [21]. The software is defined in terms of components, i.e. by abstractions which are independent from each other. The main abstraction is the so called Port Based Object (PBO) which models certain functionality as a component, reads from input ports, calculates algorithms and writes results to output ports. The advantage of a PBO is the abstraction of real-time knowledge from the control tasks. Utilizing the component model it is required to synchronize they by global variable tables. The framework is part of the Chimera II Real-Time Operating System (RTOS) [21]. Another interesting approach is the Open Control Platform (OCP) [27], developed at the Georgia Institute of Technology. An evolution management unit guards the reconfiguration process. It has real-time capabilities to modify uninhabited autonomous vehicles (UAVs) in short time. The approach has two parts: a Core OCP API for distributed computing and a Controls API as a convenient interface for control engineers. One implementation of the Core OCP is based on real-time CORBA, a well-know middleware. Oreizy and Taylor [18] have provided an architecture-based approach to reconfiguration. The architecture style is given by a network of concurrent components. Furthermore, the architecture is implemented using a common component interface. The components can communicate in between by software connectors. The underlying framework has been developed for Ada, Java and C++. As a validation example the computer game KLAX has been reconfigured dynamically. Especially for small embedded controllers (16- and 32-bit) the project eCEDAC, evolution Control Environment for Distributed Automation Components, running 2005-2007, has been defined a concept based on IEC 61499 to reconfigure control software. In a bunch of papers different authors from company PROFAXCTOR and TU Vienna have developed a runtime environment, FORTE, and an Integrated Development Environment (IDE), 4DIAC, utilizing XML to request reconfigure processes, to
provide real-time specifications and to support the reconfiguration process. It has proven real-time capabilities and is based on modified function blocks of IEC 61499 to lower the memory requirement, which is needed to control the reconfiguration process systematically. See for reference of [28]. Garcia and Fröhlich [5] have provided a reconfiguration framework based on C++ static metaprogramming. The concept consists of a runtime, ELUS, the EPOS Live Update System which works on EPOS, the embedded parallel operating system. Reconfiguration commands can be send to ELUS using a specific communication protocol, the ELUS Transport Protocol. ELUS consists of a container system which maintains components. The system is capable to use with very small embedded devices (8-bit).

1.3. Analysis

Common to all described reconfiguration solutions is the classical development cycle. The software is engineered locally and later deployed to one or more targets. This requires to download older versions of a software entity, to alter and to upload a new instance. Integrating a compiler facility with a runtime does provide a far more agility, because compiling and linking can be done remotely on an automation device and thus lowering development costs. No system is reported providing an integrated runtime platform with compiler and linker facility.

As a technology either components or services are used (whereas services are related to components). In Computer Science there is a bunch of implementations of component containers. See for reference [14]. Every development team has its good reasons for a specific design, architecture and interface. Integrating one of the described approaches into another component or service container is due to its fixed interface not possible. Furthermore, no described concept is able to alter dynamically the employed component container interface, because no system has a generic network interaction for managing of the respective components and services.

Just three systems are based on a virtual machine. There is no specific usage of a JIT reported. Real-time support is given by two systems, one of them for Microsoft .NET and one of them supporting Java.

No system is able to modify the development language instantly, be it as a new feature or as a whole. This means, no used software system is programmed using a homoiconic language.

Thus, just the first requirement is partly fulfilled by the described concepts. Solutions with a VM have been provided, but without utilizing a JIT. For sure this is used by the VM itself, but not explicitly compiling instantly needed components into the memory. The third requirement isn't fulfilled by any approach. How to fulfill all requirements is the matter of the next section.

2. A generic middleware

As it has been described in section 1.1 there are some requirements to engineer flexible automation software, which can evolve over time. Especially the component and service interfaces require an assertion, which is a good design choice in many cases, but for reconfiguration it's not necessary. To reconfigure the systems with the analyzed methods it is a must have to model data and functionality with such a rigid interface, because the runtime system encompass a fixed infrastructure of managing services and components with a negotiated protocol between server and client, which is not modifiable without shutdown of the whole system. There is no way to integrate different component and service types in one system and there is no possibility to alter existing data and functionality modeling (including altering or defining new programming languages in part or as a whole). The concept of flexibility has to be raised one level higher not only enable to reconfigure predefined abstractions (as it is given by services and components), but rather to provide to modify the infrastructure of a runtime dynamically itself. This allows to define a true generic middleware for very diverse software designs. In section 1.2 it has been shown, that none of the sketched reconfiguration solutions fulfills just two of the described requirements. None of them allows to reconfigure the infrastructure and thus doesn't allow dynamically integrating different component and service types. Furthermore the development cycle of separating the engineering and the runtime is very inflexible and thus cost intensive. A more agile approach can reduce costs, because of minor workflow.

2.1. Foundation technology and architecture

To abstract the reconfiguration process from a specific programming language and to allow altering the whole infrastructure dynamically one need an environment in which programs of a programming language like IEC 61131-3 ST can be modified, possible while the system is in full service. For such a metaprogramming environment it is very important to assure that no performance penalties takes place, for real-time applications and non-real-time applications as well. A first naive idea is to use a sort of parser generator to provide different interpreters. Using a parser generator one can define the rules to analyze lexically and syntactically a code of interest. Having defined an interpreter framework parsing codes does allow to define instantly new programming languages, but by an enormous performance loss. Although this is practicable for off-line engineering systems like knowledge-based systems, this is already impracticable for telecontrol applications, e.g. via Internet, which are by far not (hard) real-time. For hard real-time applications, for instance in
factory automation this is not feasible. From this perspective one needs a technology stack which is able to perform real-time applications as well non-real-time applications with compiled code. But moreover, the environment must enable to compile just-in-time (JIT) as opposed to ahead-of-time (AOT) to provide an agile development environment which has also to be a runtime environment. With a JIT compilation no code has to be known in advance. This can be achieved in memory by so-called bytecode manipulation frameworks; no binary code has to be stored on the file system. Utilizing a JIT compiler enables the automation engineer to modify small parts of a program, even single bits, without a redeploy of huge bunches of components or services. Specifically for dynamic programming languages JIT compilers produce faster code than AOT compilers. This can be achieved by idle periods of a processor by different kinds of code optimization. Especially the Low Level Virtual Machine (LLVM) [12] has been designed for aggressive performance analysis and optimization. With a specific C compiler for LLVM it is possible to outperform by far the standard GNU C compiler. Thus a virtual environment does provide the performance as a technological foundation for automation systems.

From this perspective, but also to make applications most portable, with a common security model and to provide applications a common interface a virtualization framework like the Java Virtual Machine (JVM) or the Microsoft Common Language Infrastructure (CLI) is needed. Both technologies have proven real-time support, like jamaicaVM [32], AERO-VM [33] or Fiji [34] for Java and the .NET compact framework [16] for the CLI.

Returning to the approach of a parser generator as an environment for building interpreters for different languages, just a VM with a JIT is not capable to solve the problem of flexible reconfiguration. On top of a virtual environment (like a JVM) a metaprogramming environment must reside, which directly compiles to bytecode of the VM and is able to modify and extend its own scope of operation. Such a metaprogramming environment must have at least one very important feature, the homoiconic language feature. This is the necessary requirement to define a rewrite system which don’t result function values, but functions. If a clear separation exist between built-in program constructs on the one side and programs and data on the other side, one can build functions which use these built-in constructs, doing some processing, but there is no way to define instantly new constructs. The fundamental concept to come close to this, is a generic representation of computer programs (built-in constructs included): representing every code entity as a symbol or a collection of symbols. Every method, function or program construct can be reduced to its symbol representation. With a rewrite facility, a so-called macro system, one can create new symbols, which can be functions, methods, classes, program constructs, anything a runtime environment can execute. For this the macro system must be able to delay the evaluation of given parameters, such that, for instance, the process of building a function shouldn’t evaluate given parameters for the function, before the macro has created them. Most of programming languages don’t provide the homoiconic feature. Additive, with a symbolic representation of all code elements and a rewrite macro system there is only one big programming language family, LISP. Fulfilling the first requirement of section 1.1, the requirement of a running software on top of a VM there is need for a LISP dialect on top of a virtual environment (like JVM or CLI). LISP itself isn’t best known for its real-time capabilities (though there exist a report of successful use of LISP for hard real-time [23]), but on top of a VM with proven real-time functionalities and with directly compiling programs to VM bytecode there are no general caveats using LISP as an environment for automation systems, even for real-time applications. It’s just an application of a VM. Moreover there has been lately reported that in many applications a modern LISP compiler can outperform even handcrafted C code [25] [24], the same holds for modern Java virtual machines, because of their optimizing routines in used JITs. See for a Java-related discussion [3]. For a more general discussion in context of (generic) virtual machines not directly connected to Java or CLR, see for instance [11] [13].

Fig. 2 Technology stack

The benefit of LISP as a programming language and metaprogramming environment over rather more simple languages like Java or C#, which have been developed for use in big companies with less skilled programmers [7] is a sophisticated environment to adapt to every problem domain imaginable by implementing a DSL. This can be achieved with the metaphor of DSL stacking [8]: a DSL for an IEC 61131-3 language (like Structured Text or Instruction List), a DSL for IEC 61499, a DSL for reconfiguration, etc.

Figure 2 pictures the general technology stack. To communicate between the automation engineer, from its desktop PC, and the automation system there exist on top of the LISP system a system to read from a socket and write to him. Over this socket communication the automation engineer can interact with the automation system and has access to all features of LISP and the
VM. This can be used as a managing console to remove, update, stop or start software entities, for instance components. The work process is given in Figure 3.

This is a fundamental feature of every LISP environment. The so called read-eval-print loop (REPL) does allow a far more interactive programming style as the classical development cycle, because the software which should build the automation system isn't designed, programmed, linked and compiled on a local engineering PC, rather the source command is send via a REPL to a compiler (residing at the automation device), which compiles to the memory and returns a status. Such a REPL can work locally or via network. One can imagine this as a shell, in which commands are inserted and delegated to the execution unit (the compiler).

![Fig. 3 The work process](image)

As it can be seen in Figure 3 there exist no specific deployment processes, because programs are engineered on site. The local engineering Integrated Development Environment (IDE) is more or less an editor, which relays all requests and responses between the local engineering PC and a remote automation system (equipped with a runtime). Once an automation system has been successful engineered, it is executable instantly. No explicit deploy process must take place. This is a great benefit especially for a reconfiguration process, where otherwise existing software modules must be downloaded, reworked and redeployed. Moreover, there is no specific interface or protocol between engineering PC and automation system for the management of the respective automation system. The needed software infrastructure, be it a service container, be it a component container or just a function container can be redefined at any time. Furthermore, a testing of engineered modules can benefit greatly, because there is no need for a second environment equal to the productive one. Integration tests can be realized in a second memory block of the productive system with much more comparable conditions as locally at the engineering PC or a second system (full compatible to the productive site). This reduces testing costs.

2.2. The reconfiguration concept

The reconfiguration process has two aims: the reconfiguration of software entities and the interchanging of new DSLs. Due to its island position compared to existing reconfiguration approaches the latter will be described in this section. The first one has been already described in depth by Kramer and Magee in [10] [9] [15] and with formalization and for hierarchical components by Wermelinger [26]. The adoption between different components has been achieved by so called connectors, as it is described e.g. by Oreizy and Taylor [18]. This is well established in Software Engineering and is therefore not subject of this section.

To provide a new DSL for a LISP based metaprogramming system two things have to be accomplished:

- define a so called reader-macro to read programs with their respective syntax notation and
- define for each command of the targeted language a macro.

Every LISP system has a built-in reader macro. As most LISP systems uses a prefix notation, there is need for a reader-macro to handle infix notation (like used for ST). A reader macro is used in LISP languages to read a character stream and pass a lexical analyzer, i.e. as it is pictured in Figure 4 to provide symbols. A macro, instead a reader macro, can be understood as a term rewrite system to substitute input symbols to appropriate output symbols, i.e. symbols from a DSL to them of the language LISP. The evaluation of parameters of a macro is delayed until the macro has been rewritten, i.e. the last box in the Figure; this is a fundamental difference to parameters for a function, which are instantly evaluated. Using the requirement code is data one can define macros which write new code, which is directly executable.

![Fig. 4 A logical computation with ST](image)

An example should clarify this approach. A special reader macro for IEC 61131-3 ST reads the byte stream coming over the wire, analyze it lexically and returns a set of used symbols known to the selected input language (ST). By definition ST uses an infix notation, so the requested reader macro must check the input symbols against an infix notation. Subsequently the input symbols are transformed to a language construct, like FUNCTION or VAR_INPUT, for instance. The macro expansion returns LISP memory-based compiled functions directly executable on a JVM. The described process can be seen in Figure 4. The macro expansion,
i.e. the evaluation of a macro, has been to fulfill every
time a new program, function-block or function is added
or modified, but not at every execution of the resulted
LISP function. Thus, there is no general performance
caveat, but any new programming language or DSL for
automation devices can be added dynamically.

3. Benefit and Practical Example

The proposed solution can play an important role in
software systems which have a long run. All knowledge
of a system design cumulates into any kind of code
representation. This representation is always accessible,
but other types like informal descriptions, notes or
formal models (like UML) don’t represent in a realistic
scenario the complete set of informations. Therefore,
only the code files of an automation system provide the
required information. Due to a huge set of used
languages for the different levels of automation there is a
big resource wasting if a technology shift to be
imminent. A software technology allowing to reuse the
knowledge of different automation solutions with no a
priori requirements is a big step forward to reusability
across technology borders.

As an example the MicroCANopen framework can be
noticed. This framework provides a minimal CANopen
framework, used in automotive or factory automation.
Like many other frameworks for embedded computers
this code library is written in C. If one has written a
rather complex communication framework, say in Java, it
would be a complex task to adopt MicroCANopen to
integrate into a Java-based framework. This could be
achieved either by foreign function calls (like Java Native
Interface) or by reprogramming the C code in a OOP
fashion using Java. This is on the one side definitely very
costly, but on the other side not achievable by existing
dynamic reconfiguration frameworks. Because C-
functions resides inside executables outside of the scope
of a VM a dynamic reconfiguration without side effects
is rarely possible. For such a scenario a language
package C for the metaprogamming language allows to
reuse the MicroCANopen module as part of a rather
generic communication framework without use of
foreign functions. With a second reader macro it is even
possible to read binary files beneath the source files.

4. Conclusion

Existing solutions for reconfiguration of automation
software have been analyzed against needed properties to
provide more flexible solutions in spite of rigid
interfaces of service and component containers. A
middleware concept has been provided to modify
automation systems independent of rigid component
systems, able to alter even the infrastructure to integrate
new programming languages. Beneath this, the concept
allows to integrate different data types (like component
types) either in a binary version, if directly executable by
the selected VM (for instance as Java bytecode) or via
defining a DSL. Moreover, the presented approach is
very agile and don’t require to download old versions of
code entities and upload new ones.

5. Outlook

The basic concept has been described. Other parts,
like assuring consistence utilizing hierarchical states,
providing a solution to reconfigure low level components
(like hardware drivers) and the transaction system have
been recessed, which the authors will discuss in
forthcoming articles. A support for reconfiguring low
level components like device drivers will be achieved by
splitting the VM into a low-level and a high-level part.
The low-level part will be realized by a LLVM instance,
which builds the execution unit for all device drivers.
The high-level VM does execute the metaprogramming
system and required control applications. By this
separation of concerns it is possible to add and remove
also drivers. The only component which must be stable
in the long run is the low-level VM. The other parts can
be interchanged at any time.

One of the most required research is a sophisticated
performance analysis for real-time applications. This will
be also part of future work and is under development.
For this different communication protocols like
CANopen, MODBUS and I2C have been selected. For
the validation the PLC programming language
Structured Text has been chosen.

References


Real-time Reconfiguration in Distributed Systems:
Timing Issues and Solutions,” In Proceedings of the 8th
IEEE International Symposium on Object-Oriented Real-
Time Distributed Computing, IEEE, Seattle, IEEE Press,

“Benchmarking Java against C and Fortran for Scientific
Applications,” In In Proceedings of the ACM Java

based on OSGi,” In Proceedings of the 7th IEEE
International Conference on Industrial Informatics,

Dynamic Software Reconfiguration Infrastructure for
Embedded Systems;" In Proceedings of the 17th


