Control Engineering Practice 21 (2013) 12–22

Contents lists available at SciVerse ScienceDirect

Control Engineering Practice

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

Developing control logic using aspect-oriented programming and sequence planning

Kristofer Bengtsson [a,*], Bengt Lennartson [b], Oscar Ljungkrantz [b], Chengyin Yuan [c]

[a] Sekvensa AB, Göteborg, Sweden
[b] Automation Research Group, Department of Signals and Systems, Chalmers University of Technology, Göteborg, Sweden
[c] Formerly, Research and Development, General Motors, Warren, MI, USA

ARTICLE INFO

Article history:
Received 8 May 2011
Accepted 1 September 2012
Available online 30 September 2012

Keywords:
IEC 61131-3
Reusability
Aspect-oriented programming

ABSTRACT

A fundamental functionality of a Programmable Logic Controller (PLC) is to control and execute a set of operations. But a large part of the program code is more involved in supporting the user with concerns like alarm, HMI, communication, safety and manual control. Code related to these supporting concerns is often tangled with operation execution code, the core concerns, which makes it hard to reuse.

This paper describes a method to reuse code and functionality when developing PLC programs and code libraries. The method proposes that core concerns are planned with a software tool called Sequence Planner, and the supporting concerns are integrated into the core concerns with a tool based on aspect-oriented programming.

1. Introduction

Programmable Logic Controllers (PLCs) have been used in industry for decades. They are used in various areas like nuclear power plants, farming, machinery, servo motion control, safety control and sorting tasks. In this paper, the focus is on the usage of PLCs in discrete manufacturing, where they for example are responsible for coordinating and executing robot and machine operations, handling production sequences, providing information to machine operators, logging production data, and communicating with other systems. A typical automated assembly plant can have several hundred PLCs, controlling the manufacturing process.

An important programming task for a PLC programmer is to design the sequential behavior of the automation system. This behavior can be specified as a set of operation sequences, which are unique for each program. The code that control these operation sequences is called *core concerns* in this paper. The operation sequences will also influence other parts of the program, like alarm, HMI, safety and manual control. These parts are called *support concerns*. A challenge in industry is to reuse both core and support concerns.

This paper proposes a method to help PLC programmers to reuse code. Specifications and requirements related to core concerns are reused from earlier stages of the design process by the tool Sequence Planner (Bengtsson, 2009). This tool specifies operation sequences with a set of self-contained operations, which include necessary requirements, like interlocking and resource usage. The operation specifications are verified by translating them to finite state automata models extended with variables (Lennartson et al., 2010). A supervisor can also be generated and converted back as new requirements on the operations (Miremadi, Åkesson, & Lennartson, 2011). Both the verification and synthesis are accomplished by the solver (Supremica). These operations can then be implemented as a PLC program.

Support concerns are more complicated to handle since they are located in many parts of the program and needs to be adopted to each operation. This paper therefore proposes automatic merging of supporting concerns into core concerns using a tool based on aspect-oriented programming (AOP). AOP is an emerging programming methodology (Filman, Tzilla, & Siobhàn, 2004) in computer science, and was proposed by Bengtsson, Lennartson, and Yuan (2009a) as a possible methodology to separate various concerns in a PLC program. That work is extended in this paper and a development method is presented, including an aspect weaver designed for the international PLC programming standard IEC 61131-3 (ISO/IEC, 2003), and a design method to plan and specify the core concerns.

This paper contributes in two ways; the first is the method to reuse both core and support concerns in a structured and unified way, which has not been fully accomplished in available tools and research. The other contribution is the introduction of AOP in 61131-3 languages and the specification of an aspect weaver for a non-object-oriented and graphical programming language.

In the next section, related work are presented and discussed. Aspect-oriented programming in computer science is introduced

* Corresponding author.
E-mail address: kristofer@sekvensa.se (K. Bengtsson).

0967-0661/$ - see front matter © 2012 Elsevier Ltd. All rights reserved.
http://dx.doi.org/10.1016/j.conengprac.2012.09.002

in Section 3. In Section 4, the various concerns in a PLC program are explained based on a small example. In Section 5, a prototype AOP weaver for 61131-3 is described, and in Section 6, the framework to develop a PLC program is presented.

2. Related work

The industrial practice when developing a PLC program has been studied by researchers, see for example Visser (1987), Lucas and Tilbury (2003), Richardsson and Fabian (2006), and Hajarnavis and Young (2008). These studies identify that the programming activity is quite complex, where the programmer creates the program based on various types of specifications and requirements, often with differing content. The complexity to create new programs makes it challenging to reuse information and code from earlier stages of a development process and in between various projects.

Several companies use function blocks to reuse and structure code (Ljungkrantz, Åkesson, & Fabian, 2010, Chap. 2). Function blocks are part of the widely used international standard for PLC programs, IEC 61131-3 (ISO/IEC, 2003). The standard also defines five inter-operable languages: Instruction List (IL), Ladder Diagram (LD), Function Block Diagram (FBD), Structured Text (ST), and Sequential Function Chart (SFC) (Lewis, 1998). Even companies that do not use function blocks structures and decomposes the code into reusable *components*, for example by copying from previous projects (Lucas & Tilbury, 2003) or using templates. The components include functionality related to for example standardized control of specific hardware, communication, production data, alarm or safety (Ljungkrantz, Åkesson, & Fabian, 2010, Chap. 2).

Only using components as a mechanism for reuse is not realistic. Usually a component is only possible to reuse efficiently, if it is highly specialized on one repeatedly occurring thing. When trying to reuse a larger code base in one component, that component tend to be harder to use in various places since it will be hard to adopt for each a specific place. It can for example be complicated to reuse code that *uses* reusable components.

To tackle this problem and to increase reusability of control code, some PLC vendors have introduced development environments with model and meta designing tools. Two examples are Aspect Objects technology from ABB (2001) and Simatic Automation Designer from Siemens (2011). These tools try to help the programmer and the end-user by allowing creation of a PLC program from high-level specifications and reusable templates.

Another problem to increase code reuse is that each PLC vendor has its own code standard. The organization PLCOpen (PLCopen, 2011) tries to tackle this by developing an independent standard based on IEC 61131-3 and XML. A vendor neutral XML-format approach is also proposed by other researchers, for example Estevez et al. (2008).

Other approaches include improvements on current control logic standards, like the OOONEIDA—Open Object-Oriented kNowledge Economy in Intelligent inDustrial Automation (Vyatkin, Christensen, & Lastra, 2005). This initiative aims at a highly integrated development process with standard building blocks and complete vendor neutral code, based on the industrial control standard IEC 61499 (Vyatkin, 2007). IEC 61499 is related to 61131 and is an event-driven distributed control architecture based on function blocks.

Another IEC standard is IEC 61512-1 (1997), also known as ISA-S88, addressing batch process control. This standard defines models and terminology and has various implementations like SattBatch from ABB and PackML from Organization for Machine Automation and Control. Both these implementations enable a higher degree of reuse by defining high-level model specifications in the form of operation recipes (the core concerns). These are then executed by a control system that implements the detailed execution code. Sequence Planner that will be presented later is actually influenced by these tools, but is a generalization of them together with a formal operation model. What is not handled by these tools is how to reuse the detailed implementation, which is unique for each installation.

Object-oriented languages were developed to increase the development efficiency with clear language syntax and well-defined structures and constructs, compared to procedural languages. The higher development efficiency results in better reusability, flexibility and expressiveness (Meyer, 1997). Object-orientation has also been proposed for PLC-programming by for example Speck (2003), Werner (2009), including the use of Unified Modeling Languages (UML) (Thramboulidis, 2004).

Industry has been slow to adopt other languages than IEC 61131-3. There are probably many reasons for this, but there are actually some benefits to use IEC 61131-3 languages. They are specialized and designed to address specific issues and challenges faced in automation manufacturing, and are therefore good at expressing automation manufacturing concerns. Most of the languages are easy to program and they give good comprehension for plant floor personals. It is also intuitive when performing maintenance and online troubleshooting, and finally the standard defines a robust and simple execution model.

Other methods to reuse code and to increase the code quality are based on formal modeling, to verify code and to synthesize controllers for complex control issues. One example is a framework based on 61131-3, proposed by Ljungkrantz, Åkesson, Fabian, and Yuan (2010), that defines reusable function blocks with a built-in formal specification; to increase the quality of reused code. Solutions based on formal methods are not discussed in this paper, but a comprehensive review has been written by Frey and Litz (2000).

Most of the published methods to increase code reuse for PLC programming have not fully addressed the root cause *why* it is hard to reuse larger parts of the code. One root cause is that various functionality or concerns are hard to separate and therefore complicated to reuse. Aspect-oriented programming tries to increase the code reusability by enabling separation of the concerns.

3. Aspect-oriented programming

Object-oriented programming increases program reusability by providing design and language constructs with features such as modularity, encapsulation, inheritance, and polymorphism (Meyer, 1997). Although object-orientation is widely used and very successful in modeling and implementing complex designs, it has its problems. Practical experience with large projects has shown that programmers may face some problems with maintaining their code, because it becomes increasingly difficult to cleanly separate various issues or concerns into modules (Kiczales et al., 1997). An attempt to do a minor change in the program design may require several updates to a large number of unrelated modules. This problem is what aspect-oriented programming tries to handle.

Aspect-oriented programming (AOP) is a methodology that considers how to separate various aspects of the system, when developing a software program. One challenge is to handle crosscutting concerns, such as data logging, synchronization and diagnostics, which are located across the entire software program. This is done in AOP by separating these crosscutting concerns into aspects, and then weave them into the base code (the normal concerns) at deployment or at runtime. The core functionality of AOP is *obliviousness*, which states that the base code is unaware of

the aspects, and *quantification*, which describes where and how the aspect code is weaved into the base code (Filman et al., 2004).

The most commonly used AOP tool is AspectJ, which is an AOP framework for Java. In AspectJ, the code is divided into base code and aspects. The base code is a normal Java program with the object-oriented structure and composition, and the aspects represent the concerns that are hard to separate cleanly. The aspects are automatically integrated into the base code by a process called weaving. The key parts of an aspect are

- Join point: A join point is a well-defined position in the structure of the program (in both the base code and in other aspects).
- Pointcut: The pointcut quantifies (or specifies) a set of join points.
- Advice: An advice defines the behavior that is weaved at the join points, which are picked out by a pointcut.

A join point defines a specific position in the program where additional behavior can be added. Some join points defined in AspectJ are: Method call, Method execution, Object initializing, Field reference and Field set. These join points define various positions in the source code or events during the execution of the program. The pointcut is used to pick out a set of join points in the base code, based on an expression. The expression can for example state: *Pick out every call to methods with the name foo** (where * is a wildcard), which targets every method name that begins with foo. For each picked out join point, an advice defines the behavior (the code) that is weaved at the join point. The advice also defines if the code should be added before (BEFORE), after (AFTER) or instead of (AROUND) each join point.

AOP has mainly been used for object-oriented software, but also for systems similar to PLC programming. For example Tesanovic, Nyström, Hansson, and Norström (2005) have been studying the use of components and AOP for real-time control systems. The use of AOP for graphical specification languages has been studied in the area of UML model weaving, see for example Atlas ModelWeaver (Bezivin, Joault, & Valduriez, 2004) and Motorola WEAVR (Cottenier, 2006). Especially, WEAVR tackles a similar problem as in this paper. The tool introduces an adopted weaver semantic to be able to weave advice models into state-based UML models, similar to the operation models used in this paper. But WEAVR is still adopted for a high-level language and object-oriented constructs.

The challenge addressed in this paper, compared to these otheer methods and tools, is that the 61131-3 languages are quite different from high-level languages since PLC-programs are completely static. This demands an adopted AOP-methodology. Another challenge in this work is that the final weaved result needs to be understandable and changeable by plant-floor personnel.

It is hard to understand what a crosscutting concern is and how AOP can be used in real applications without an example, hence, an example on how an operation could be implemented is presented in the next section.

4. Implementing an operation

The main functionality of a PLC program is to execute and coordinate various tasks or operations in the automation system and to manage the interaction among the resources. An important design activity when developing the control system is therefore to plan when and how the operations are to be executed. One example of an operation could be to close a clamp in a fixture, like the clamps in Fig. 1. To better understand what type of concerns that are typically implemented to control this operation, the CloseClamp operation will be studied. This example is based on a real programming standard from an automotive company.

4.1. Close clamp

The concerns in Table 1, can be divided into two groups, core and support concerns. The core concerns on the left constitute the core behavior of the operation, i.e when and how to execute the operation. The support concerns on the right, adds functionality to the operation for increased usability.

An operation must be executed or *realized* by one or more resources. For example, an operation that closes a fixation clamp is realized by a clamp in a fixture, which is closed by an actuator that drives a pneumatic cylinder. To execute the close action, the actuator's physical I/O is mapped to an internal variable used in the code.

Before the signal to close the clamp can be sent, the precondition must be satisfied. The precondition consists of three parts: *safety*, *allocation*, and *sequence* statements. The safety statements include guard predicates, to prevent dangerous or unwanted situations, that must be fulfilled before the clamp can close. If the operation needs to use shared resources such as a common working area or shared tools, they are checked for availability and booked by the allocation statement (including availability guards and booking actions). The trigger for when an operation should start its execution in automatic mode is controlled by the sequence statement.

When the precondition is satisfied, the action command will set the close output signal, which is mapped by the I/O mapping to the clamp actuator. To determine whether the operation is completed, a sensor checks that the clamp is in the closed position. This is defined by the postcondition.

The above core concerns are always specific to each installation and constitute the processing functionality of the cell. However, to have a functional manufacturing cell, other more user-oriented functions are needed, i.e., support concerns.

The human–machine interface (HMI) is the most important tool for users, as it allows them to close the clamp manually and presents status and diagnostic information. For diagnostics, the opening and closing times for the clamp are monitored. If the clamp cannot complete the close operation until the timeout is reached, an alarm will be raised and reported to the event handler.

Another useful support concern is the troubleshooting logic. When trying to solve a problem in the cell, it may, for example, be

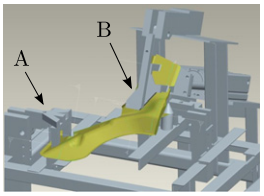

Fig. 1. A product part is fixated by the two clamps A and B.

Table 1
Operation concerns.

Core	Support
Realizing resources	Manual control
Precondition	Diagnostics
Postcondition	Troubleshoot logics
Action	HMI

important to execute an operation manually. In some cases, however, the preconditions are not fulfilled and the operation will not be executed, so it is useful to trace the conditions hindering the action. Another useful functionality is to bypass a sensor, to keep the production running until it is possible to change the failed resource.

All of these concerns are highly interrelated; for example, the postcondition influences the diagnostics, the troubleshooting logic uses the precondition and can change the I/O mapping, the manual condition is linked to the precondition and the action command, and the HMI interacts with most of the other concerns. This makes the various concerns of the operation entangled and difficult to separate cleanly. This is a problem, since these concerns cut across many operations and objects, making them more difficult to reuse completely.

4.2. Crosscutting and tangled concerns

If there is only a single operation or a single program developed, it is probably not efficient to separate the support concerns that are crosscutting the program. But in the automotive industry, there are hundreds of PLCs using the same or similar program architecture. If the support concerns are not separated from the rest of the code, it will be a tough challenge to develop and manage the code architecture.

The problems with existing PLC development approaches can be divided into the two areas: *standards and upgradeability*. A *standard* usually consists of a library of components and a framework on how to use these components. The problem of developing a standard for PLC programs is that each installation and program is unique and may require some special components. As the size of a library grows, it is hard to manage and handle all the variants, since the software components are not general enough due to tangled concerns.

The other problem, *upgradeability*, is related to the fact that a concern can be located in many places in the code, without the possibility to encapsulate them cleanly. This will make that concern hard to maintain and upgrade. To take a simple example, if the interface that sends status information to the HMI from an operation, must be updated, the same update must be applied to every operation component in the library and the running systems. It is the same with most interfaces, e.g. if a function block adds or removes an input, it must be changed at every instance of the function block.

These crosscutting and tangled concerns can be handled by aspect-oriented programming. In the next section, a 61131-3 aspect weaver is presented, and in Section 6, a framework that uses aspect-oriented programming for control logic development is proposed.

5. IEC 61131-3 aspect weaver

To use AOP when designing PLC-programs based on 61131-3 in real industrial projects, AOP needs to be fully integrated into the development environments of the vendors. But before that is possible, it is important to identify and describe how an aspect weaver for 61131-3 could work. This section specifies the core functionality of such a weaver.

5.1. Aspects and base code

AspectJ has almost become a standard for AOP, which usually makes it the starting point when new AOP frameworks are created. The suggested 61131-3 aspect weaver also tries to adopt the foundation of AspectJ, but the dissimilarity between 61131-3 and Java requires some differences in the weaver. One difference is that 61131-3 defines multiple languages that to some extent are interchangeable and combinable and some of them are graphical. Therefore the specification of the aspects will be different.

As in AspectJ, the aspects consist of advices, join points and pointcuts. The difference is that the behavior (the code) of the advice is located separated from the aspect code, since it should be possible to add code to all five 61131-3 languages. This separated code is called the advice code, and the main part of the aspect is called the aspect descriptor.

The structure of a program in 61131-3 is also different from Java. The 61131-3 standard allows a project to be broken down into functional elements, called program organization units (POUs), and tasks. POUs include functions, function blocks and programs. To allow full execution control, POUs are assigned to tasks to enable various scan-rates and parallel execution. The base code used by the aspect weaver is structured by multiple POUs, and each aspect is located in its own POU.

5.2. Aspect descriptor and advice code

The aspect descriptor consists of pointcuts and advices, which were defined in Section 3. It defines where and how the content of the advice code is weaved into the base code. The aspect descriptor is written in Structured Text with some additional features. The new features manage the dynamic functionality used to identify information from each identified join point, called context exposure, which is not possible to handle in a good way in Structured Text. There are also some new language constructs to define pointcuts and advices.

Aspect weavers can pick out various types of join points in the base code and in the aspects. However, since 61131-3 has a statical and cycle-based execution model, it is only possible to pick up statical join points. This is also a difference compared to AspectJ that has the possibility to pick out join points dynamically when the program is executing. The following join points can be identified in the base code and in the aspects:

- Call: When a call is made.
- Write: A value is written to a variable.
- Read: A variable is read.
- Declaration: A variable is declared.
- Execution: A section is executed.

These join points identify various locations in the code that fulfills a specific criterion. The *Call* join point identifies when a specific POU is called in the code, for example the call of a function block in a ladder rung. When a variable is used in the code it is identified by the *Write* and *Read* join points, and the *Declaration* join point identifies when a variable is declared in the VAR section. The *Execution* join point identifies a complete section of the code, for example the code in a function block.

Join points are picked out by pointcuts defined in the aspect descriptor. A pointcut identifies a set of join points by an expression. The expression consists of pointcut and attribute functions separated by AND, OR and NOT. The pointcut functions correspond to the possible join points and the attribute functions defines attributes related to join points.

In the weaver, three types of advice constructs are used: AFTER, BEFORE and AROUND. The advice defines what should happen at the identified join points in the pointcut. The AFTER advice will insert extra code after a join point, the BEFORE advice will insert the advice before and the AROUND advice will replace the join point.

5.3. Weaving

The input to the weaver is a base code and a list of aspects where the base code and the aspects are structured as 61131-3 *projects* stored in the PLCopen (2011) XML-format TC6. The weaver loads each aspect, one at the time, and weaves the aspect into the base code. The final result is saved as a new 61131-3 *project* and stored in the XML-format. To better understand how the AOP weaver works, let us study part of the diagnostic concern for the close clamp operation.

5.3.1. CloseClamp diagnostic example

The function block for the CloseClamp operation is shown in Fig. 2, where it is included in a ladder rung. This function block controls the core behavior of the operation and is initially in the init state where only the Init output is enabled. The PreCond input defines the operation precondition and is enabled when both the sequence and the safety contacts are true. The Sequence Condition defines when during the automatic execution of the manufacturing system the clamp should close, and the safety condition hinders the clamp to collide with other resources.

When PreCond is enabled, the operation can start its execution by setting the Action output and changing its state to execute (enabling Exec). The operation will be executing until the post-condition is satisfied, which happens when the Clamp.Closed sensor is active and the Clamp.Open sensor is inactive. When the operation has completed, the Fin output, i.e. the finished state, is enabled and the action output is deactivated. The operation can return to its initial state when the reset condition is enabled.

A common method to monitor and diagnose an operation is to measure the time between the start of execution until it completes. If the measured time exceeds a predefined time, the operation is assumed to be faulty and an alarm is raised. Most operations in the system will use this type of monitoring, which makes this functionality located at many places. The operation alarm is therefore suitable to implement as an aspect.

In traditional PLC programming, the alarm concern is often implemented for each individual operation. But when using AOP, the concern will only be implemented once as an aspect and then weaved into the code after each operation.

The aspect descriptor for the alarm aspect can be seen in Fig. 3. The first part of the aspect descriptor is the declaration section, where variables used in the aspect are declared. An advice code called Alarm_Advice is instantiated as AC, and will be described later. After the VAR section, the pointcut is defined with the `POINTCUT` *pointcut_name* `DO` construct, which is not standard ST syntax. The pointcut construct will be interpreted by the aspect weaver and a pointcut object, opCall, will be created. opCall picks out each join point in the base code that correspond to the pointcut expression.

In the given example, the pointcut opCall picks out each function block of type Operation, with the pointcut function CALL. The pointcut attribute HASOUTPUT identifies that the function

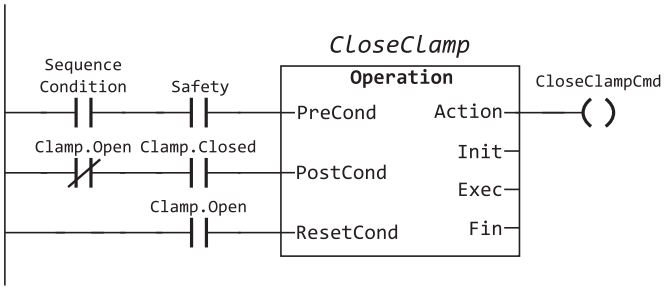

Fig. 2. Close clamp example.

```
VAR
  AC : Alarm_Advice;
END_VAR

POINTCUT opCall DO
  CALL(TypeName = 'Operation') AND
  HASOUTPUT('Exec' + 'Fin');
END_POINTCUT;

AFTER opCall DO
  AC(name:= opCall.getInstanceName,
     execCmd:=opCall.GetOutPutName('Exec'),
     finCmd:=opCall.GetOutPutName('Fin'));
END_AFTER;
```

Fig. 3. Aspect description for the alarm aspect.

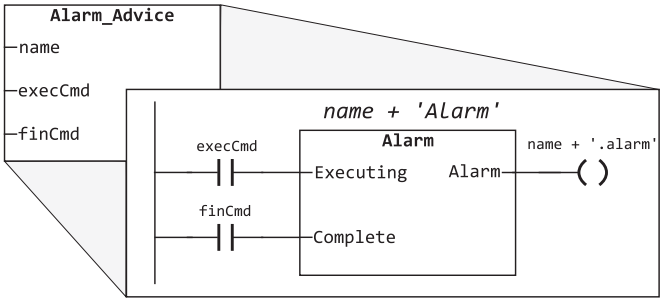

Fig. 4. The alarm advice code with its implementation. All inputs are of type string.

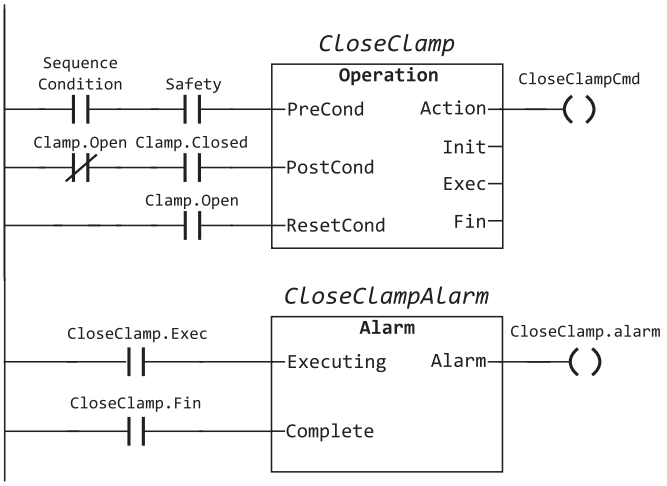

Fig. 5. The alarm weaved after the CloseClamp operation in the ladder code.

block also has outputs called Exec and Fin, which will be used by the advice.

The advice of the alarm aspect begins with the `AFTER` advice construct. This advice defines that the after each join point in the pointcut opCall, the advice will insert the advice code. The advice code, Alarm_Advice can be seen in Fig. 4, including its implementation in the box at the right. The three inputs to Alarm_Advice are string variables, which will be used inside the advice code implementation.

The weaver will take the input strings, which are unique for each operation function block, and construct the correct variable or instance names. In this example the weaver replaces the *name*, *execCmd* and *finCmd*, with the unique strings CloseClamp, CloseClamp.Exec and CloseClamp.Fin, respectively. The CloseClamp example including the inserted advice can be seen in Fig. 5. The new rung includes a function block called alarm that measures the time from when the operation enters the execution state until

it enters the finished state. If this time exceeds a specific time, an alarm will be raised.

5.3.2. The weaving process and its challenges

Let us study the functionality of the weaver in more detail. First the base code is loaded into memory as an XML Document Object Model. The weaver then takes one aspect at the time from the list of aspects and weaves the aspect into the document. It is important that the aspects are weaved in a specific order since a fundamental challenge in AOP is aspect interference (Durr, Staijen, Bergmans, & Aksit, 2005), where conflicts may occur when aspect interact. There exist techniques to resolve aspect conflicts, for example by defining precedence relations as in AspectJ, but for the 61131-3 weaver, only the order of the list is considered. For future weaver development, this issue should be studied in more detail.

When an aspect descriptor is loaded, the weaver first creates the variables defined in the VAR section. After that, each pointcut is created. A pointcut consists of one of the pointcut operator functions: *Call*, *Write*, *Read*, *Declaration* or *Execution*, and several attribute functions. It is a limitation that only one pointcut function is allowed in a pointcut, but that was a design decision taken to minimize the possibility to generate illegal code. This could probably be more general in the future but it was not an issue for the studied examples in this paper.

Each pointcut and attribute function has a built in attribute defining the type of base code element it can find. The functions also have a set of predefined input parameters that should be matched against these elements. These parameters are often defined as strings that can include wild card characters to better pinpoint a join point.

The weaver takes the pointcut function and add elements in the document that matches the built in element type and input parameter to a set. After that, the attribute function expression of the pointcut is checked against the children elements of each element in the set. If the expression is evaluated to false, the element is removed from the set. The result is a pointcut consisting of a set of matching elements from the base code document.

When the poincuts in the aspect descriptor are created, each advice is executed. The three advices used by the weaver, BEFORE, AFTER and AROUND, can be applied to multiple pointcuts. For each element in the pointcuts, the weaver will either place code before, after or instead of (around) the element. How exactly the code is inserted will vary depending on the pointcut function type, the language of the base code and advice code and what is defined in the advice code. For example if a Read pointcut function picks out a contact in a ladder rung, and the advice code only consists of a set of new contacts, these contacts will be inserted in the same rung. But as in the example above, a complete rung is defined in the advice code and therefore a new rung is inserted after the function block rung. This logic is built into the weaver, but could probably also be expressed in each aspect.

The advice can extract information about each pointcut, by calling various predefined methods defined by the pointcut and attribute functions. If specific information is required, it needs to be identified by the functions, even if the programmer knows that a join point has some specific parameters. The information from the pointcut is loaded into the advice code where the weaver will replace corresponding elements in its code. The final code is then inserted as a new element into the document.

Not all languages in 61131-3 have been studied in this research, for example sequential function charts (SFCs). One reason for this is that a SFC usually only describes core concerns. The join point model for SFCs may also be somewhat different, but this needs further investigations. SFC weaving is related to what Motorola WEAVR (Cottenier, 2006) is doing, which should be a starting point for the investigation.

This section shows how an AOP approach can work for PLC development. If the example above would have been programmed in a normal development approach, the alarm functionality is added to many places in the program. With the aspect weaver, the effort can be reduced. Obviously, this quite simple example can be managed by other methods like templates, since the interface between the operation and alarm function block are quite standardized. This however, will not be the case when more functionality is added to the alarm aspect, as we will see in the next section. That section will present a framework on how to manage both the core and support concerns during a PLC development project.

6. Developing PLC programs

To manage the development of PLC programs in industry, flexible and efficient programming standards and a libraries are crucial. These standards must accommodate a diversity of various manufacturing installations, and be easy for the plant personal to recognize and understand. However, crosscutting concerns are hard to fully integrate into a standard, as has been discussed in this paper. Therefore the standards and the libraries need to be flexible and adaptive to the diversity of various types of automation systems in a plant. This paper proposes a clear separation of core and support concerns, both during the development process and in the standards and the libraries.

The core concerns are unique for each PLC program and need to be specified and designed during the development. These concerns are related to the notion of operation, which defines the behavior of a system related to the products and the manufacturing resources (Bengtsson, Lennartson, & Yuan, 2009b). Planning these operations is often referred to as process planning or sequence planning.

6.1. Sequence planning

The challenge to plan sequences of operations can be found in many research areas, for example project management (Lee, Pena-Mora, & Park, 2005), product assembly planning (Abdullah, Popplewell, & Page, 2003), manufacturing task planning (Shabaka & Elmaraghy, 2008), computer aided manufacturing (Miao, Sridharan, & Shah, 2002), computer aided process planning (Marri, Gunasekaran, & Grieve, 1998), and control design (Shen, Wang, & Hao, 2006). Academics have mostly focused on the optimization based planning problem, but the industrial impact has been quite limited so far, probably due to the complexity to solve real problems. The industry focus has instead been to represent and visualize sequences and tasks, and to simulate them.

To develop the core concerns, a new tool called Sequence Planner (Bengtsson, 2009), is used by the framework in this paper. This tool does not only manage the complexity to develop the operations but also to represent and visualize them. Sequence Planner uses a new sequence planning approach, where sequences are based on the relations among operations instead of explicit manual sequence construction (Bengtsson et al., 2012). This is achieved using self-contained operation models that include only relevant conditions on when and how the operations can execute. The operation models are also represented by an automata model extended with variables. This model is for example used for formal verification, control synthesis and optimization (Lennartson et al., 2010).

Sequence Planner uses a graphical language called sequences of operations (SOP) introduced by Lennartson et al. (2010). Using various views or perspectives, the sequences of operations related to e.g the part flow, transport operations or workstation tasks can be visualized. The SOP language is based on operations where the execution of each operation consists of three states: initial state, execute state and finished state. If an operation is denoted O, the states are denoted O^i (initial), O^e (execute) and O^f (finished). The operation can start when its precondition $O^{\uparrow}$ is satisfied and stop when its postcondition $O^{\downarrow}$ is satisfied. To better understand the operations and the tool Sequence Planner, let us study an example.

6.2. An automation system example

This example is based on a real installation in the Robot and Automation Lab at Chalmers University of Technology. The cell is shown in Fig. 6. The cell receives two pieces of sheet metal, one from an Automated Guided Vehicle (AGV) (6), and the other one from a conveyor (1). One of the small robots (3), and the large robot (5), picks up the parts and put them in the fixture (2), which fixates the parts before the robots drill holes and rivet the plates together. The assembled part is then transported away by the AGV.

In the given example, let us focus on when the parts are loaded into the fixture by the robots. In Fig. 7, some of the operations executed by the fixture and the robots are shown in a SOP (sequences of operations). The SOP language visualizes the operations, denoted by boxes, and the relations among them by graphical lines and logical expressions. The basic assumption is that all operations are starting simultaneously, but where pre-, post- and reset-conditions influence the execution order. The relations among the operations due to these conditions can then be visualized by the SOP-language. The SOP in this example includes 10 operations structured in three sequences, where the left sequence describes the high-level behavior, and the two right sequences show the detailed sub-operations of the two load operations.

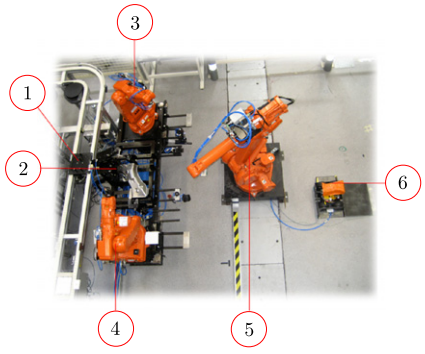

Fig. 6. The example cell.

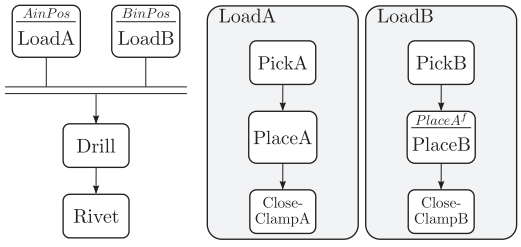

Fig. 7. Sequences of operations (SOP) describing the behavior in the fixture station.

The LoadA and LoadB operations load the parts into the fixture. These operations can start when the parts to be picked up are in the correct position. This is defined by the logical expressions *AinPos* and *BinPos* above the line in the operation boxes at the upper left in Fig. 7. Preconditions are shown at the top of the operation box and postconditions at the bottom.

Part B is picked up from the conveyor by Robot 3, executed by operation PickB. After that, the part is placed in the fixture by the PlaceB operation. The arrow inbetween PickB and PlaceB denotes that PickB must be completed before PlaceB can start, i.e. $PickB^f$ is included in the precondition of PlaceB. But part B cannot be placed in the fixture before part A, therefore the extra logical expression $PlaceA^f$ is also included in the precondition of PlaceB. The final precondition is: $PlaceB^{\uparrow} = PickB^f \wedge PlaceA^f$. This shows the strength of the SOP-language, since it would be messy to show both preconditions graphically.

When part B is placed in the fixture, the clamp that fixates B can be closed. After both parts are loaded, the robots can start to drill and rivet the parts together. The two parallel horizontal lines in the left sequence defines that the two operations LoadA and LoadB needs to be completed before the Drill operation can start its execution, i.e. they are included in the precondition for the Drill operation ($Drill^{\uparrow} = LoadA^f \wedge LoadB^f$). The SOP language also include alternatives, arbitrary order, loops etc, and is described in detail in Lennartson et al. (2010).

The operations are planned in more and more detail by adding new conditions or creating sub-operations. This operation specification together with the resource description constitutes the core specification needed to create the base code of the PLC program.

6.3. Creating the base code

The core specification from Sequence Planner can obviously be implemented in various ways. The proposed framework does not restrict how the base code is structured and implemented, but the chosen structure will influence the aspects. Therefore the base code needs to be implemented in a standardized way. In this example the base code is divided into a set of programs, one that coordinates the complete cell control and three station control programs. A station in the cell is an individual process area for a set of operations. Each station program controls the station resources and the execution of station sequences.

Some of the operations from the Sequence Planner example are shown in Table 2. In this example, these operations are implemented with the operation function block shown in Fig. 2 at various places in the program. Other parts of the base code handles the interaction with the resources, like the robots and fixture that are implemented from a library but is not further discussed in this paper.

In Fig. 8, an overview of the proposed development framework is shown. The base code is created based on the Sequence Planner core specification and the core libraries and standards. The program is structured as specified by the standards and contains the operations

Table 2
Operations with some of their core concerns.

Name	PreCond	PostCond	Resource
LoadA	AinPos	CloseClampA ^{f}	Robot5 $\wedge$ Fixture
LoadB	BinPos	CloseClampB ^{f}	Robot3 $\wedge$ Fixture
Drill	LoadA $^f \wedge$ LoadB ^{f}	DrillDone	Robot4
Rivet	Drill ^{f}	RivetDone	Robot3 $\wedge$ Robot4
PickA	LoadA ^{e}	APicked	Robot5
PlaceA	PickA ^{f}	APlaced	Robot5 $\wedge$ Fixture
CloseA	PlaceA $^f \wedge$ AinFix	APlaced	Fixture

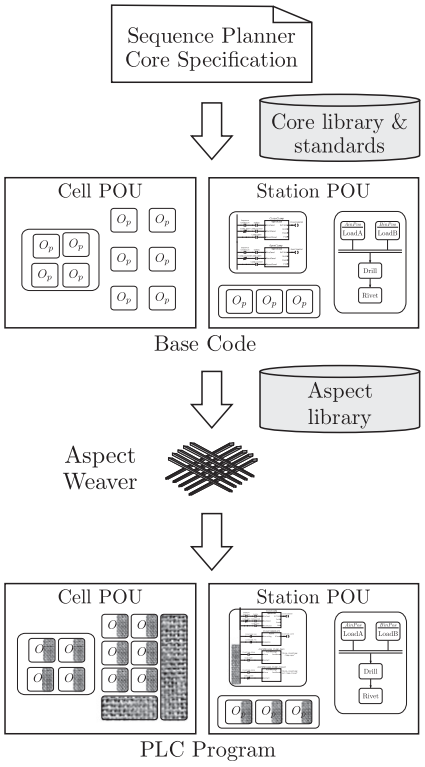

Fig. 8. AOP developing methodology.

and the resource control. However, a major part of a PLC program is about e.g. field bus control, HMI communication, general diagnostics, safety, cell monitoring, product information, and coordination between stations and other cells. These parts are implemented as aspects and are weaved into the base code by the aspect weaver.

6.4. Aspect weaving and changeability

One of the most important reasons to separate support concerns from core concerns is changeability. To change a cross-cutting concern without accidentally messing up other parts of the code is complicated in normal PLC programming. Therefore programmers tend to avoid changes in industry even if they are needed. Although the aspects are highly dependent on how the base code is structured, the AOP framework can hopefully give the programmers confidence to really do the needed changes.

In the example, the aspect weaver will take the base code and weave the alarm aspect into the code. Each operation will now have an alarm connected to it. But then someone "demands" that the code also should include alarm messages about why the alarm went off. This function can be implemented by checking which contact in the postcondition that is not fulfilled. In a normal PLC program these messages needs to be added to every single operation in the code. To implement this in the AOP example, only a few lines of code and an extra advice code is added to the aspect, which is shown in Fig. 9.

To identify the two contacts in the postcondition, *not Clamp.Open* and *Clamp.Closed*, the pointcut method *GetInputRung*, from attribute HAS_INPUT_RUNG, is used. This method extracts the rung connected to the input *PostCond* and places it in the variable *R* in the AFTER advice. The rung object include a translation of the rung to conjunctive normal form where a clause is represented by a set of contacts. This representation is used by the for loop where a new function block, for each extracted clause, will be created. The Advice Code *AlarmMessageAC* is not shown in the example but consists of one AlarmMessage function block.

```
VAR
  AC : Alarm_Advice;
  AlarmMessageAC : AlarmMessage_Advice
  R : Rung;
END_VAR

POINTCUT opCall DO
  CALL(TypeName = 'Operation') AND
  HASOUTPUT('Exec' + 'Fin') AND
  HAS_INPUT_RUNG('PostCond');
END_POINTCUT;

AFTER opCall DO
  AC(name:= opCall.getInstanceName,
     execCmd:=opCall.GetOutPutName('Exec'),
     finCmd:=opCall.GetOutPutName('Fin'));

  R := opCall.GetInputRung('PostCond');
  FOR i:=0 TO R.andClauseSize DO
     AlarmMessageAC(
         name:= opCall.getInstanceName,
         clause := R.andClauseArray[i]);
  END_FOR;
END_AFTER;
```

Fig. 9. Extended aspect description for the alarm aspect.

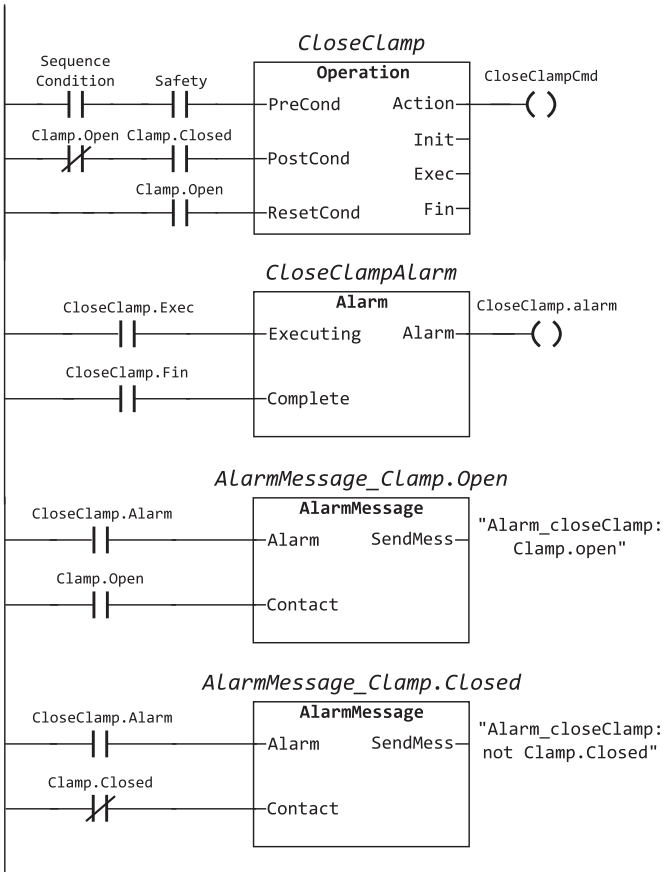

Fig. 10. The updated alarm aspect creates alarm messages.

The weaver result can be seen in Fig. 10, where the two AlarmMessage function blocks checks if the CloseClamp.Alarm has been enabled and if respective contact is true. If the signal

Clamp.Open still is true, the alarm message ''Alarm_CloseClamp: Clamp.Open'' will be sent to the HMI. If the signal Clamp.Closed has not been set, the alarm ''Alarm_CloseClamp: not Clamp.-Closed'' will be sent. In some cases both messages will be sent. This simple example shows how efficiently AOP can be used to include a new functionality by only changing at one place. But there are also some challenges that need to be handled when using the proposed development framework.

7. Evaluation

Aspect-oriented programming tries to modularize concerns in a better way than traditional language constructs. But it is still unclear if the required development effort with respect to development time, maintenance and learning curve, is worth the possible benefits. Some even state that there are no benefits, for example Steimann (2006).

Some studies have been conducted to study the benefits of AOP for Java programming. Hanenberg, Kleinschmager, and Josupeit-Walter (2009) suggest based on a small empirical study that the use of AOP is only beneficial if the crosscutting concern refers to a large number of places. Another experiment (Bartsch & Harrison, 2008) was not able to show any benefits but includes a summary of other studies that show both good and limited benefits using AOP.

Endrikat and Hanenberg (2011) argue that most empirical studies about AOP are missing the point of using AOP, since they only consider the initial development time, which is often found to increase when using AOP. Their empirical study focuses instead on the relationship between the initial design and future changes. They conclude that the strength of AOP is when frequent changes is required in the crosscutting code. On the other hand, if the base code needs frequent changes, the AOP approach results in higher development times, and poor reusability.

The major difference when considering AOP for Java programming compared to 6-1131, is that PLC-programs is only using statically defined variables and objects. This makes PLC code harder to make generic and flexible, but on the other hand robust and consistent. Challenges related to crosscutting concerns can in many cases be handled in Java or other high-level programs using dynamic language constructs, but for PLC-programming that option is not available.

The example presented in Section 6 can easily be created in Java without the use of AOP constructs, but in a PLC program, it is not obvious how to create the same reusable code. Even if the possible benefits using AOP for high-level programming is not clear, it seems to have better leverage for PLC-programs. Let us study the weaving example in Section 6 when used with a complete PLC-program.

7.1. Experiment setup

The studied PLC program contains the control code for 72 operations structured in several POUs. Each operation in the base code consists of 5–12 rungs. The PLC program also consists of code that for example handles the communication with other resources such as the robots and manage HMI interactions. All these parts form the base code where the operation code is approximately 40% of the base code.

When the aspect in Fig. 9 is weaved into the code, 3–7 new rungs are created for each operation. Based on this simple aspect, the weaver automatically adds almost 300 rungs of unique code. This can be compared to the standard method for these types of edits – copy and paste (Lucas & Tilbury, 2003) – which will be highly error prone due to the required small changes at each place.

Changes on the aspect can easily be managed by the weaver by reweaving the aspect code. But when a change is made to the base code it is not as straight forward, since it depends on what is changed. Only changes made on the structure of the join point structure, i.e. that the operation function block interface is changed, will require rework at each operation implementation and of the aspects. It is therefore important with well defined coding standards for the base code which is often the case for example in the automotive industry.

This research initially targeted large scale PLC development in the automotive industry, where a large number of PLC systems are using code standards and libraries. In these situations, the initial increased effort to use AOP, is probably well worth the investment. However, it may also be useful for single program development, since the number of possible join points seems to be large. But due to the requirements on a highly standardized based code structure, this needs to be further evaluated.

This example shows that the benefits of using AOP are greatest when a large number of join points can be targeted by few aspects and that these join points are not changed often.

7.2. Challenges

The evaluation shows a promising result, but to be able to use the AOP framework in industry, more work is needed. For one thing, the weaver must be able to understand vendor specific code files, since most vendors do not read PLC open files. Another important study is to see if normal PLC programmers can understand and create aspects.

A foundation of AOP is that the base code is unaware of the aspects. But the aspects on the other hand, are completely aware of the base code and in some cases also of other aspects. Changes in the base code structure can therefore demand large changes in the aspects. In PLC programming, many of the aspects are influenced by the core specification from Sequence Planner, but this information is retrieved indirectly from the base program. Further investigation must be done to analyze if it is better to retrieve some of the information directly from the specification. This is especially useful if the base code should be automatically generated.

This tight integration between base code and aspects was identified in the study by Endrikat and Hanenberg (2011) as a possible risk when using AOP. If changes are required on the structure of the base code, the aspects can be complicated to reuse. They do not however identify what types of changes that are problematic, but conclude that further research is required. PLC AOP development also needs to be further studied to increase the understanding of the impact of changes of the code structure.

Another challenge is how to handle manual changes in the generated and installed program. In other AOP approaches, changes are never made to the weaved code, but in a PLC program changes can be made after deployment of the code. These changes should be possible to retrieve back, to handle new updates, fixing errors in the libraries and to develop the framework. Obviously, it is complicated to feed back major changes in the code and in the structure of the PLC program. However, in reality, changes are mainly made on specification related parts, for example new interlocking condition or sequence changes. But since also the support concerns are tangled with the core concerns, they must be changed as well.

The suggested approach to handle changes, is to annotate the code where it is allowed to change and which parts are forbidden to change. Only parts that are related to operation specifications should be allowed to change. Then a separate program can extract changes and feed them back to Sequence Planner and a new code can be generated. It will be challenging to accomplish this in

practice, but a general trend in industry is to minimize online changes on the plant floor.

Even if online changes are not handled at the moment, the proposed framework can in its present form contribute to a better reuse of code compared to the common copy and paste development method. It also gives a better possibility to auto-generate code in a flexible way.

8. Conclusions and future research

A development method has been presented where core and support concerns are reused both when developing programming standards and during PLC programming. A software tool called Sequence Planner is used for planning core concerns, and the proposed IEC 61131-3 aspect weaver is used for implementing support concerns.

The presented method contributes by adapting AOP-methodology into the PLC programming domain, especially for the IEC 61131-3 languages. The method also unifies reuse of core and support concerns during the development process of a PLC program. When the proposed aspect-oriented programming for 6-1131 was tested using an example program, the reuse of one crosscutting aspect were able to create almost 300 rungs of unique code.

The paper presents a possible method for PLC programming using AOP. However, further work is required before it can be used in real industrial projects. For example, the proposed framework needs to be included in the integrated development environments of the PLC vendors. How PLC programmers will be able to understand and utilize programming by aspects must also be studied, as well as how PLC programs are currently changed online and if feedback functionality really is needed.

Not all languages in 61131-3 have been studied in this research, for example sequential function charts. Further investigation is needed to understand how to fully combine all languages during aspect weaving. There are also open issues how to handle aspect interference, more general advice insertion and more general pointcuts.

Acknowledgments

This work was carried out within the Wingquist Laboratory VINN Excellence Centre within the Area of Advance—Production at Chalmers, supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA) and the Knowledge Foundation.

References

ABB. (2001). 〈www02.abb.com/global/seitp/seitp161.nsf/viewunid/342d723e4a4d44c4c1256b580070ef69/$file/aspects+objects.pdf〉.

Abdullah, T., Popplewell, K., & Page, C. (2003). A review of the support tools for the process of assembly method selection and assembly planning. *International Journal of Production Research*, *41*, 2391–2410.

Bartsch, M., & Harrison, R. (2008). An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Journal*, *16*, 23–44, http://dx.doi.org/10.1007/s11219-007-9022-7.

Bengtsson, K. (2009). *Operation specification for sequence planning and automation design*. Licentiate thesis signals and systems. Sweden: Chalmers Göteborg.

Bengtsson, K., Bergagard, P., Thorstensson, C., Lennartson, B., Akesson, K., Yuan, C., et al. (2012). Sequence planning using multiple and coordinated sequences of operations. *IEEE Transactions on Automation Science and Engineering*, *9*, 308–319.

Bengtsson, K., Lennartson, B., & Yuan, C. (2009a). Aspect-oriented programming for manufacturing automation control systems. In *IFAC symposium on information control problems in manufacturing, INCOM*.

Bengtsson, K., Lennartson, B., & Yuan, C. (2009b). The origin of operations: Interactions between the product and the manufacturing automation control system. In *IFAC symposium on information control problems in manufacturing, INCOM*.

Bezivin, J., Joault, F., & Valduriez, P. (2004). First experiments with a modelweaver. In *Workshop on best practices for model driven software development held in conjunction with the 19th annual ACM conference on object-oriented programming, systems, languages, and applications*. Vancouver, Canada.

Cottenier, T. (2006). The motorola WEAVR: Model weaving in a large industrial context. In *Proceedings of the international conference on aspect oriented software development*. Industry Track.

Durr, P., Staijen, T., Bergmans, L., & Aksit, M. (2005). Reasoning about semantic conflicts between aspects. In *Proceedings of the European interactive workshop on aspects in software 2005*.

Endrikat, S., & Hanenberg, S. (2011). Is aspect-oriented programming a rewarding investment into future code changes? A socio-technical study on development and maintenance time. In *2011 IEEE 19th international conference on program comprehension (ICPC)* (pp. 51–60).

Estevez, E., Marcos, M., Irisarri, E., Lopez, F., Sarachaga, I., & Burgos, A. (2008). A novel approach to attain the true reusability of the code between different PLC programming tools. In *IEEE international workshop on factory communication systems, 2008 (WFCS 2008)* (pp. 315–322).

Filman, R., Tzilla, E., & Siobhàn, C. (2004). *Aspect-oriented software development*. Addison-Wesley ISBN: 0-321-21976-7.

Frey, G., & Litz, L. (2000). Formal methods in PLC programming. In *Proceedings of the IEEE SMC* (pp. 2431–2436, Vol. 4).

Hajarnavis, V., & Young, K. (2008). An investigation into programmable logic controller software design techniques in the automotive industry. *Assembly Automation*, *28*, 43–54.

Hanenberg, S., Kleinschmager, S., & Josupeit-Walter, M. (2009). Does aspect-oriented programming increase the development speed for crosscutting code? An empirical study. In *Proceedings of the 2009 3rd international symposium on empirical software engineering and measurement (ESEM '09)* (pp. 156–167). Washington, DC, USA: IEEE Computer Society.

IEC 61512-1. (1997). *Batch control Part 1: Models and terminology*. Technical report. International Electrotechnical Commission.

ISO/IEC. (2003). *Programmable controllers—Part 3* (2nd ed.). International standard IEC 61131-3. ISO/IEC.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., et al. (1997). Aspect oriented programming. In *Proceedings of the European conference in object-oriented programming*.

Lee, S. H., Pena-Mora, F., & Park, M. (2005). Dynamic planning and control methodology for strategic and operational construction project management. *Automation in Construction*, *15*, 84–97.

Lennartson, B., Bengtsson, K., Yuan, C., Andersson, K., Fabian, M., Falkman, P., et al. (2010). Sequence planning for integrated product, process and automation design. *IEEE Transactions on Automation Science and Engineering*, *7*, 791–802.

Lewis, R. (1998). *Programming industrial control systems using IEC 1131-3 Revised edition*. The Institution of Electrical Engineers.

Ljungkrantz, O., Åkesson, K., Fabian, M., & Yuan, C. (2010). Formal specification and verification of industrial control logic components. *IEEE Transactions on Automation Science and Engineering*, *7*, 538–548.

Ljungkrantz, O., Åkesson, K., & Fabian, M. (2010). Practice of industrial control logic programming using library components. In Guedes, L. A. (Ed.), *Programmable logic controller*. Intech.

Lucas, M., & Tilbury, D. (2003). A study of current logic design practices in the automotive manufacturing industry. *International journal of Human-Computer Studies*, *59*, 725–753.

Marri, H. B., Gunasekaran, A., & Grieve, R. J. (1998). Computer-aided process planning: *A state of art*. *The International Journal of Advanced Manufacturing Technology*, *14*, 261–268.

Meyer, B. (1997). *Object-oriented software construction* (2nd ed.). Prentice Hall professional technical reference. ISBN: 0-13-629155-4.

Miao, H., Sridharan, N., & Shah, J. (2002). CAD–CAM integration using machining features. *International Journal of Computer Integrated Manufacturing*, *15*, 296–318.

Miremadi, S., Åkesson, K., & Lennartson, B. (2011). Symbolic computation of reduced guards in supervisory control. *IEEE Transactions on Automation Science and Engineering*, *8*, 754–765.

PLCopen. (2011). 〈http://www.plcopen.org/〉.

Richardsson, J., & Fabian, M. (2006). Modeling the control of a flexible manufacturing cell for automatic verification and control program generation. *Journal of Flexible Service and Manufacturing*, *18*.

Shabaka, A., & Elmaraghy, H. (2008). A model for generating optimal process plans in rms. *International Journal of Computer Integrated Manufacturing*, *21*, 180–194.

Shen, W., Wang, L., & Hao, Q. (2006). Agent-based distributed manufacturing process planning and scheduling: *A state-of-the-art survey*. *IEEE Transactions on Systems, Man, and Cybernetics*, *36*, 563–577.

Siemens. (2011). 〈https://www.automation.siemens.com/mcms/automation-software/en/digital-engineering/pages/default.aspx〉.

Speck, A. (2003). Reusable industrial control systems. *IEEE Transactions on Industrial Electronics*, *50*, 412–418.

Steimann, F. (2006). The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st annual ACM SIGPLAN conference on object-oriented programming systems, languages, and applications (OOPSLA '06)* (pp. 481–497). New York, NY, USA: ACM.

Supremica. 〈http://supremica.org/〉.

Tesanovic, A., Nyström, D., Hansson, J., & Norström, C. (2005). Aspects and components in real-time system development: *Towards reconfigurable and reusable software*. *Journal of Embedded Computing*, *1*, 17–37.

Thramboulidis, K. (2004). Using UML in control and automation: A model driven approach. In *IEEE international conference on industrial informatics* (pp. 587–593).

Visser, W. (1987). Strategies in programming programmable controllers: A field study on professional programmers. In *Proceedings of the emperical studies of programmers: Second workshop* (pp. 217–230). Washington, DC.

Vyatkin, V. (2007). *IEC 61499 function blocks for embedded and distributed control systems design*. O3NEIDA—Instrumentation Society of America. ISBN: 978-0-9792343-0-9.

Vyatkin, V., Christensen, J., & Lastra, J. (2005). Oooneida: *An open object-oriented knowledge economy for intelligent industrial automation*. *IEEE Transaction on Industrial Informatics*, *1*, 4–17.

Werner, B. (2009). Object-oriented extensions for IEC 61131-3. *Industrial Electronics Magazine, IEEE*, *3*, 36–39.