Domain-Oriented Architecture Design for Production Control Software

Rolf Engmann Rob van de Weg Roel Wieringa
Department of Computer Science,
University of Twente, the Netherlands[*]

May 28, 1998

1 Introduction

In this paper, we present domain-oriented architectural design heuristics for production control software. Our approach is based upon the following premisses. First, software design, like all other forms of design, consists of the reduction of uncertainty about a final product by making design decisions. These decisions should as much as possible be based upon information that *is* certain, either because they represent laws of nature or because they represent previously made design decisions. An import class of information concerns the domain of the software. The domain of control software is the part of the world monitored and controlled by the software; it is the larger system into which the software is embedded. The software engineer should exploit system-level domain knowledge in order to make software design decisions.

Second, in the case of production control software, using system-level knowledge is not only justified, it is also imposed on the software engineer by the necessity to cooperate with hardware engineers. These represent their designs by means of Process and Instrumentation Diagrams (PIDs) and Input-Output (IO) lists. They do not want to spend time, nor do they see the need, to duplicate the information represented by these diagrams by means of diagrams from software engineering methods. Such a duplication would be an occasion to introduce errors of omission (information lost during the translation process) or commission (misinterpretation, misguided but invisible design decisions made during the translation) anyway. We think it is up to the software engineer to adapt his or her notations to those of the system engineers he or she must work with.

Third, work in patterns and software architectures started from the programming-language level and is now moving towards the higher architectural and subsystem level. At the programming-language level, one is able to define domain-independent patterns such as adapter, facade and observer [3]. At higher levels, however, architectures get more domain-specific and we need to relate software architectures to domain architectures. In the case of production control, we should reflect the structure of the production process in the architecture of the software.

In this working paper, we apply these principles to the definition of a coordination architecture for production control software. In section 2, we look at the information contained in PIDs and IO lists for production systems and at the structure of a production process.

[*]Email (engmann | vandeweg | roelw)@cs.utwente.nl. http://wwwis.cs.utwente.nl:8080/maics/

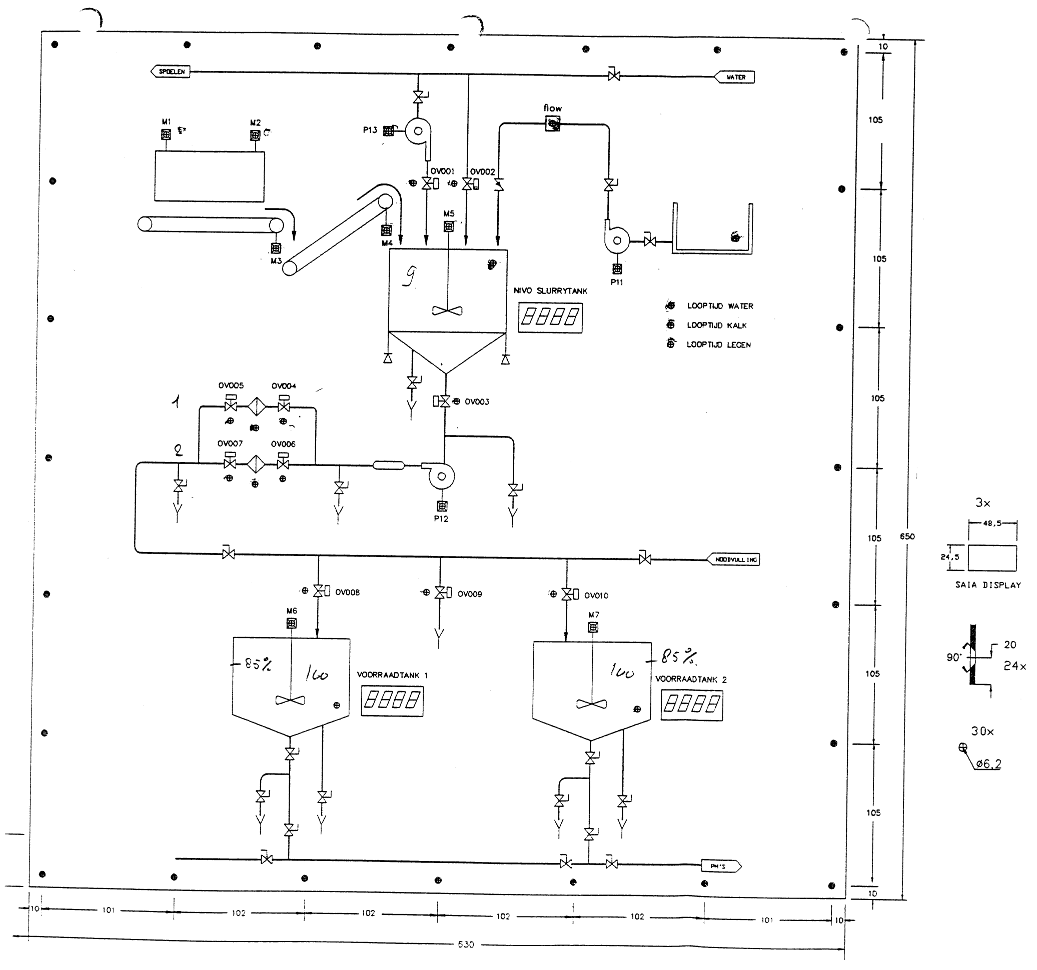

Figure 1: PID for lime slurry plant.

In section 3, we define a coordination architecture for production control software based upon the structure of production systems and processes, and in section 4 we illustrate the use of simple statecharts to represent control behavior. Section 5 briefly mentions a few architectural design heuristics for production control software and section 6 discusses how PLC code can be generated from architectural models. Section 7 ends this short paper with conclusions and a discussion. The results presented in this paper are based upon a cooperation with Moekotte B.V., Enschede, and have been validated in about a dozen commercial projects done by Moekotte.

2 Production Systems

Figure 1 gives an example of a PID for lime slurry plant production process in a paper factory. Inputs to the production process are lime, water and a chemical compound that causes lime to dissolve in water. Output is lime slurry, which is used in the paper production

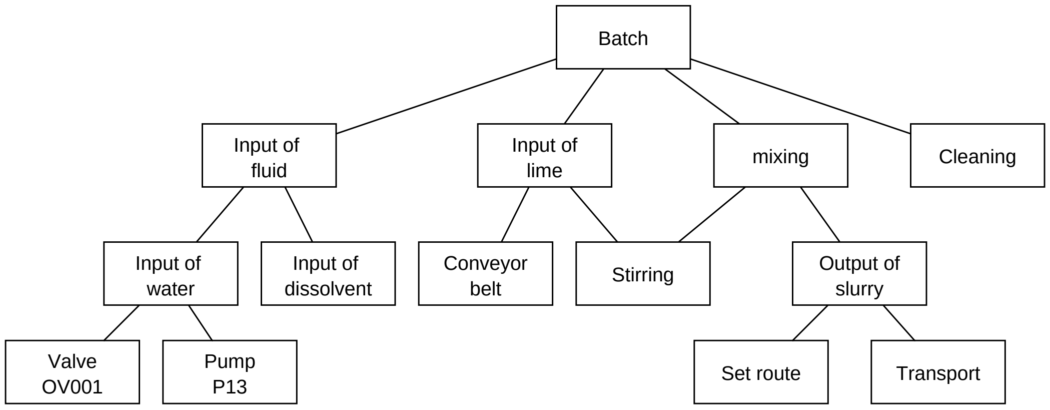

Figure 2: Hierarchical decomposition of the lime slurry production process.

process. The figure has been shrunk to a size where the text is barely readable, but the layout of the production process can be discerned: At the top there is a mixing tank into which lime is entered by conveyor belts on the left, water is entered from the top and a chemical solution is entered from a tank at the right. The pipes contain valves and pumps, all of which is represented by icons. The solution is stirred in the tank and then is transported from the tank by means of a system of pipes, to be stored in storage tanks. These slowly stir the solution during storage. The symbols in the PID are standardized icons known by all production system hardware engineers. The diagram shows the components of the production process and their physical relationships, i.e. which components are connected.

In addition to a PID, hardware engineers have an IO list containing the interfaces of the devices. For example, the IO list for a valve consists of the actions open and close. Detailed information about these actions is to be found in the technical documentation of the valve devices.

The production system designers also have a mental model of the production process that can be represented by a hierarchical diagram such as shown in figure 2. The vertical dimension of the diagrams represents decomposition of production steps. The horizontal dimension suggests the temporal ordering in the process. We return to this diagram when we define the coordination architecture of the control software. Here, we remark that the process diagram is an excellent means of communicating about the production process with the hardware engineers.

3 Coordination Architecture

The coordination architecture we propose is a specialization of the basic layered architecture found in many software systems [2]. We view production control as a **coordination problem**, because the software must coordinate the behavior of different devices so that they all contribute to a single, coherent production process. Coordination software for production control can be layered as follows (figure 3).

At the lowest level, **base objects** control hardware devices. These can be found in the PID of the production system — this is the first important domain-based uncertainty reduction. Base objects have interfaces consisting of atomic actions. These can be found from the IO list that defines the interfaces of the controlled devices — the second important

	Statics	Dynamics
External layer	Views	Transactions
Coordination layer	Scopes	Tasks
Relationship layer	Relationships	Common actions
Base object layer	Attributes	Actions

Figure 3: A layered coordination architecture for production control software.

uncertainty reduction. The state of a base object is represented by attribute values. (We require that there is an attribute that represents the status of the state machine of the device.) Attributes contain the knowledge that base objects (and hence the control software) has of the state of the controlled device. Attributes may be subject to integrity constraints. If an action changes the state of a base object, the object is responsible for maintaining its own integrity. Base objects are also responsible for dealing with exceptions that involve only their controlled device. (We define an exception as an external event that may indicate a failure in an external device.)

A PID contains information about physical relationships between devices in the production system. Relations can be represented in the software by means of **relationship objects**. Where each base object has its own atomic identifier, a relationship object has an identifier that consists of (the tuple of) the identifiers of its component base objects. This relationship concept is borrowed from database modeling. Representing device relationships by relationship software objects makes it possible to reuse the base object layer in different production system layouts. Relationships *must* be represented if the software must support the traceability of the finished product to the devices that cooperated in its production. Traceability is needed to answer product liability questions. The definition of relationship objects is the third domain-based design decision for control software. To keep the example simple, we do not include relationship objects in our example architecture.

Two related base objects may share actions in such a way that a shared action is performed simultaneously by the participating base objects. For example, a conveyor belt may drop items in a container, which can be modeled in the software by a shared action between the controlling software objects. These shared actions are allocated to the relationship object and not to the participating base objects. Only relationships can perform shared actions. The relationship layer represents a higher abstraction level than that of base objects, because relationships are defined in terms of base objects but not vice versa. Relationships can be made responsible for the global integrity of its component objects. Shared actions must maintain this integrity.

At the coordination layer, we find **coordination objects** whose purpose it is to coordinate the behavior of hardware devices in the production process. Here we use the fact that each production process can be decomposed into a number of steps, as represented in figure 2. We now use the diagram to represent the coordination architecture of the software: Nodes represent software objects, edges represent communication channels between software objects. For each coordination object, we define its **scope** as the set of objects at the coordination layer itself or in one of the lower layers that it must coordinate. There must be no loop in the subordination relationship. The coordination layer is thus itself partitioned into one or more sublayers. Coordination objects receive information about the environment (hardware devices as represented by the PID) only indirectly, viz. by receiving events from subordinate software objects all the way down to base objects. They respond to these events by sending actions to subordinate objects in its scope. Doing this, they trigger processes that consist of (possibly concurrent) actions of subordinate objects, which ultimately leads to signals sent by base objects to the hardware devices they control, thus enforcing coherent behavior on parts of the production process. Coordinator objects care also responsible for exceptions that could not be dealt with at a lower level in the hierarchy.

At the highest layer of our architecture, we find the **external layer**, which is similar to the view level in information systems. This level takes care of the interface with human operators. It accepts commands from operators to start and stop a production process. In many systems, it also contains functionality to report on errors and exceptions, produce statistical reports, etc. We ignore this level in our small example.

The advantages of this kind of structure are similar to those for any layered architecture [2, pages 48–50], with some extra benefits thrown in because we are now dealing with production control. The lower software layers can be reused in production control systems with different coordination regimes, and the impact of changes in hardware devices can be limited as much as possible to base objects. Exception-handling is simplified by pushing the responsibility for dealing with exceptions as low as possible in the hierarchy. At the same time, because the state machines structure belonging to this architecture (defined below) correspond closely to program structure for Programmable Logic Controllers (PLCs), development time is reduced and the resulting software is quite efficient.

4 Coordination behavior

All software objects in our architecture are reactive, which means that at any moment, they wait for an event and respond to this event in a way that depends upon their current state. As usual, there are two types of events, temporal events (deadlines) and signals received from hardware devices. Also, we distinguish active from passive states. In an **active state**, the object waits for an activity in the hardware environment of the production control software to be finished. In other words, the software objects knows that some hardware device is active and waits for this activity to finish. Because only the base objects of the software are connected to external devices, a higher-level object in an active state is waiting to receive a signal from a lower-level object, and a base-level object in an active state is waiting for an external device to terminate an activity.

In a **passive state**, an object is not waiting for an external device to terminate an activity. An object in a passive state can receive a signal from higher-level objects, a command from the operator, or a hardware failure event. A failure event always leads to a special type of passive state, called a **failure state**.

We use a simple version of statecharts to represent object behavior (figure 4). We use different state outlines to represent active, passive and failure states. Transitions in our statecharts must have simple triggers consisting, for each transition, of a single event. Statecharts allows the Moore convention of executing, upon entry of a state, an atomic **entry action**. This may start an external activity that causes the object to leave its state when it termninates. We will see below that by restricting oursel;ves to this simple use of statecharts, the architecture model can be translated into PLC code in a simple manner.

Due to our distinction between active and passive states, we can offer the following statechart design guidelines:

- Passive states may have atomic entry actions but have no activities.

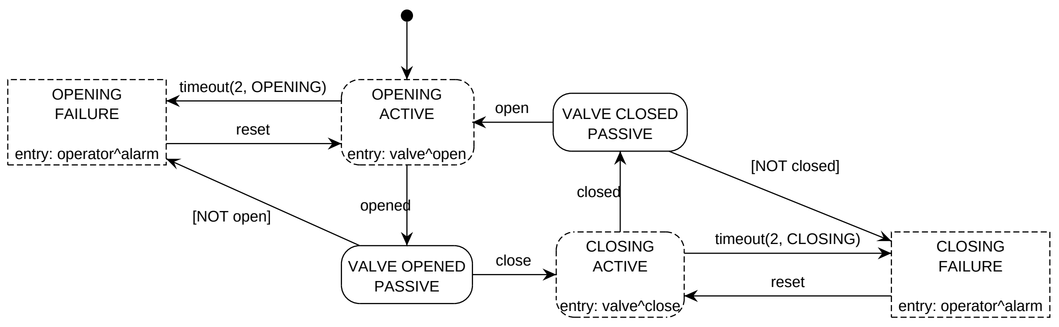

Figure 4: Simple statechart for the valve object of figure 2. We use different state outlines to represent active, passive and failure states.

- Active states have no entry actions but always have an external activity.
- From each active state, a timeout transition must depart that deals with the case that the external activity does not terminate in time. The timeout transition must enter a failure state.
- For each state, there must be at least one outgoing transition that leads to a non-failure state. This implies that there is no final state (production control software must always be able to reach the initial, idle state) and that there is always a way to leave a failure state.

5 Architecture Design Heuristics

Arentsen [1] gives an overview of coordination architectures of control regimes for production systems. The following architectures are in use.

- The **central control architecture** consists of just one coordinator object on top of a number of base objects. The coordinator performs all control tasks.
- A **proper hierarchy** consists of a master-slave relationship between coordinator objects and subordinated coordinator or base objects.
- A **modified hierarchy** extends the proper hierarchy with communication channels between coordinators at the same level.
- In **heterarchical control**, there is no control hierarchy, and all coordinator objects can communicate directly. This is a further degeneration of the hierarchical model, which is more difficult to manage and maintain.
- A **holarchical architecture** consists of holons. A **holon** is an autonomous, cooperative agent, that may consist of interacting subholons. Holons have goals and may negotiate about shared goals. The holarchical architecture is hierarchic, because we have a hierarchy of holons, but may involve complex communication patterns.

The last three architectures are used to improve the autonomy and the reactivity of the production process, at the cost of a performance penalty. As part of our current research, we are applying these architectures to case studies in order to analyze the costs and benefits associated with each architecture.

6 Implementation

Our proposed architecture is **essential** in the sense that it is defined only by reference to the environment of the architecture; it makes no assumptions about the underlying implementation platform [6, 8]. Nevertheless, the design translates in a simple manner in very efficient code for Programmable Logic Controllers (PLCs). This is one of the reasons our software designs are acceptable to the hardware engineers. A PLC is a microprocessor specifically designed for reliable industrial control even in harsh environments. Input signals are received from devices such as buttons, switches, and digital and analog sensors. Output signals of a PLC are used to control the operation of motors, valves, starters, etc. More sophisticated PLCs may include additional functionality such as a mathematics processor, a network interface, and a graphics display.

A statement in a PLC programming language consists of a Boolean expression of input signals and a set of output signals to be generated at the moment that the Boolean expression becomes true. Special purpose functions can be used in the evaluation of the Boolean expression such as timers, counters, and arithmetical operations. A PLC can evaluate several statements at the same time. This language structure allowed us to define a straightforward translation of our simple statecharts into programming code. Each state in the statechart corresponds to one statement in a PLC program. Incoming transitions of the state are translated into events and conditions that must evaluate to true. Output signals are then used to set conditions in the program and to activate hardware objects. The generation of PLC code from statecharts is done by means of a relatively simple program generator. This reduces errors in the resulting code, which is never touched by human programmers.

The structure of the PLC code provides a concurrent implementation of software objects concurrent. If all objects are implemented on one PLC, concurrency consists of interleaving the processes of different objects. Sometimes, however, more than one PLC is used, which then operate truly concurrently with respect to each other. We have used this possibility to allocate software objects such that PLCs are operating in a hierarchy reflecting the hierarchical coordination architecture [7].

7 Discussion and Conclusions

Software engineering for real-time, embedded and control software does not occur in isolation but takes place in close interaction with domain engineers. Software engineering is but one part in a very much larger engineering process. It has been argued by Jackson and Zave that domain analysis should be used to derive the software requirements [5, 4, 9]. We take this line of reasoning one step further and argue that at least in the case of production control software, the software architecture should be derived from the domain architecture. In this view, the role of software engineering and its notations is subordinate to that of system engineering.

We have shown that in the domain of production control, the structure of the domain dictates the architecture of the software. The PID gives us the architecture of the base objects and relationships, and the hierarchical decomposition of the production process gives us the desired coordination architecture of the software. During development, this approach facilitates communication with other engineers and it shortens development time. In addition, the resulting control software is well-structured because it contains a minimal degree of coupling between objects. It is therefore easier to maintain and it turns out to be more efficient than other designs.

Acknowledgements: Thanks are due to Raoul van der Hoef and Vincent ten Thij for their collaboration in the project and to Moekotte Enschede B.V. for providing the opportunity to do this research.

References

[1] A. L. Arentsen. *A Generic Architecture for Factory Activity Control*. PhD thesis, Faculty of Mechanical Engineering, University of Twente, 1995.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns: Patter-Oriented Software Architecture*. Wiley, 1996.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissideas. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] M. Jackson. *Software Requirements and Specifications: A lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[5] M. Jackson and P. Zave. Domain descriptions. In S. Fickas and A. Finkelstein, editors, *International Symposium on Requirements Engineering*, pages 56–64. IEEE Computer Science Press, 1993.

[6] S. M. McMenamin and J. F. Palmer. *Essential Systems Analysis*. Yourdon Press/Prentice Hall, 1984.

[7] R. van de Weg and R. Engmann. An environment for object-oriented real-time system design. In J. Ebert and C. Lewerentz, editors, *8th Conference on Software Engineering Environments*, pages 23–33. IEEE Computer Science Press, 1997.

[8] R.J. Wieringa. Postmodern software design with NYAM: Not yet another method. Technical report, Faculty of Mathematics and Computer Science, *Vrije Universiteit*, De Boelelaan 1081a, 1081 HV Amsterdam, January 1998. Earlier version presented at the NATO Workshop on Requirements Targeting Software and Systems Engineering, Bernried, Germany, October 1997.

[9] P. Zave. Formal methods are research, not development. *Computer*, 29(4):26–27, 1996.