Software Architecture Patterns for Distributed Embedded Control Systems

Veli-Pekka Eloranta, Johannes Koskinen, Marko Leppänen, and Ville Reijonen
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

Abstract. Embedded systems are often tightly coupled with their environment, implying specific requirements – such as distribution, real-time, and fault tolerance – to software, to be taken into account in the design of software architecture. Yet, there is little systematic support for designing such architectures. We argue that a collection of patterns specifically targeted for this domain would significantly assist software architects in designing high-quality systems. In this paper, we introduce a generative pattern language for embedded machine control systems. Furthermore, most interesting patterns of the language are presented in detail. These patterns were identified during architectural assessments carried out at several sites of Finnish machine industry.

1 Introduction

An embedded system is devised to control, monitor or assist the operation of equipment, machinery or plant [1]. In this context, an embedded control system is a software system that controls large machines such as harvesters and mining trucks. Such systems are often tightly coupled with their environment, implying specific requirements – such as distribution, real-time, and fault tolerance – to software, to be taken into account in the design of software architecture. As the embedded control systems have become larger, the software architecture of these systems plays a crucial role in the overall quality of the products. Yet, there is little systematic support for designing such architectures. We argue that a collection of patterns specifically targeted for this domain would significantly assist software architects in designing high-quality systems.

Unlike the conventional field of software architecture design, only a few patterns that are specifically geared towards embedded machine control systems have been identified. The identification of patterns is hardened due to several reasons. Typically, the software in many embedded systems has been poorly documented, the main architecture view of the embedded system is that of the hardware and machinery of the system rather than software. Engineers that implement such systems are very often from different area of expertise than software systems and therefore more familiar with the hardware technologies than with modern software engineering. However, there are proven solutions, which are communicated as folklore in the original Alexandrian sense [2]. Therefore, we feel that this particular field is a good target for patterns.

During years 2008 and 2009, we have visited several sites of Finnish machine industry to identify design patterns specific to this domain. The four target companies are global manufacturers of large machines and highly specialized vehicles intended for different branches of industry. During the visits, patterns were identified in the context of an architectural assessment of machine control systems provided by the companies. The process for architectural assessment was derived from ATAM [3]. Because the fundamental goal was to identify solutions that were regarded useful (and potentially reusable) in machine industry, we documented them as patterns ignoring any considerations on relative simplicity or prior existence of the pattern when an architectural solution seemed important or recurring.

In this paper, we introduce a pattern language for embedded control systems. The language is based on the found patterns. In addition, we describe in more detail seven patterns that we consider more interesting, typical for the domain and less familiar in the literature.

The patterns were formulated and written down by the members of the assessment team. As the pattern mining process was associated with an architectural assessment, quality attributes associated with the patterns were immediately available. This is visible in pattern descriptions in the *Forces* section. Each force is prefixed with the quality attribute it is related with. A figure is sketched to describe the idea of the pattern. These figures are intended to give a quick intuitive idea rather than a technically sound solution model. A more technical description of a sample application of the pattern, abstracted from the actual occurences, is also added to the pattern description.

2 The Pattern Language

Our pattern language consists of 35 patterns at this moment (Fig. 1). The language forms loose groups, based on the different aspects of the embedded control systems. For example, there is a group of patterns discussing distribution, another pattern group for safety through redundancy and third for achieving longer product lifetime in turbulent hardware environments. Some parts of the language are more vestigial, indicating that there might be undiscovered patterns in that area. For example, it is likely that more patterns can be found in the area of redundancy patterns.

When constructing the language, the patterns were at first grouped loosely together based on their area of effect. Patterns which were affecting the design of larger parts of the system were grouped together and patterns with smaller impact were put together depending on which part of the system they improve. This gave an overall view of the pattern relationships.

In the second phase we tried to connect the isolated groups with arrows. The semantics of an arrow pointing from pattern A to pattern B in our language is "pattern B refines pattern A". This means that if the architect has solved some design problems with pattern A, the design context is now one compatible with the required context of pattern B. The system may still have some problems to attain to, so the designer might look at all refining patterns if these solve the problem the designer is now facing.

The connecting of the patterns followed the thinking process of an imaginary architect, tackling the problems in the design one by one and advancing from more general

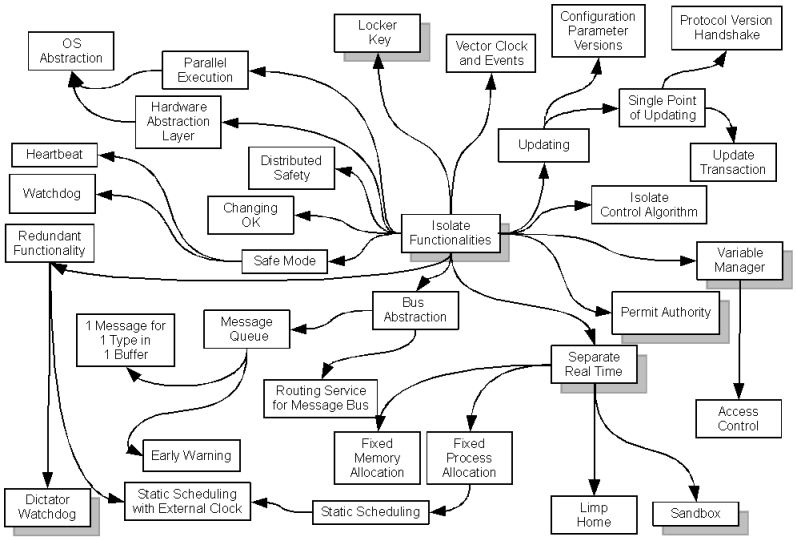


Fig. 1. The pattern language

problems towards more detailed design. The key question was to try to understand what problems the architect tries to solve and what patterns discuss this problem. Every solution brings new problems to the system, to be solved by more elaborate patterns.

This phase allowed us to see some "blind spots", solutions that emerged in the discussions but were not yet identified as patterns. Often we had separate groups of patterns that were impossible to connect together, but after adding a pattern depicting some more general principle, the orphaned pattern groups would nicely become children of this pattern. For example, UPDATING pattern was added to group all version management related patterns in one branch. The pattern connecting is therefore a very efficient tool to recognize patterns.

3 Patterns

In this section, we present the selected patterns that are most interesting and typical for the domain. The selected patterns are shadowed in Fig. 1.

We use the following template to describe our patterns. First, we present the design context where the pattern can be applied. The next section is the description of the problem that the pattern will solve in the given context. In forces section we describe the motivation or goals that one may want to solve by applying this pattern. Furthermore, we give a solution, argument it in the rationale section and give the resulting context that the pattern will result into. Finally, related patterns are presented and one known case of usage is given.

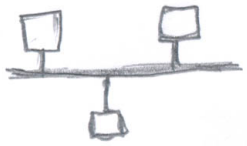
3.1 Isolate Functionalities

Context An architect has to design an embedded control system that has to control a large machine. Different sensors and actuators are needed to control the machine. Because of the limitations of the real world, controlling devices have to be close to the actuators.

Problem How to efficiently control a large machine without extensive wiring and monolithic software?

Forces Scalability: The system may incorporate many devices and a lot of communication may emerge between them.
Throughput: The system needs to provide reasonable throughput and response time.
Modifiability: The system configuration of devices changes, even during the run-time of the system.
Portability: During the life cycle of the product, the physical connections may change.
Cost efficiency: Wiring is expensive.
Fault tolerance: Extensive, thick cabling may break.
Understandability: A large and complex entity is hard to understand.

Solution The architect divides the system into functional entities so that they can be placed on separate devices. The devices are interconnected and they function as nodes on the bus. Communication takes place with a common message format.



Rationale Functional division makes system more understandable and manageable. Nodes can be added easily but the limiting factor is the capacity of the message bus. Throughput may be compromised due to heavy message traffic. Nodes do not depend statically on each other, but only on the message format, thus the system is easy to expand and modify, even at run-time. The bus presents an abstraction of the physical world, understandable to software developers. It may be difficult to know where desired functionality resides because the location is abstracted by the bus.

Related Patterns MESSAGE CHANNEL [4], MESSAGE BUS [5]

Resulting Context Distributed and scalable machine control system where nodes communicate with each other via the bus. This pattern forms the base for a distributed embedded control system for a large machine.

Known Usage The architect has decided to use Programmable Logic Controllers (PLC) as nodes. Nodes are connected with CAN bus. CANopen is used as communication protocol. The software is divided so, that engine controlling software resides on one controller and transmission controlling software on another.

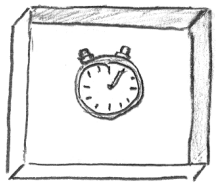
3.2 Separate Real Time

Context ISOLATE FUNCTIONALITIES has been applied and the system is now a distributed system. The system has both real time and non-real-time requirements. The architect needs to provide high end services and yet ensure safe operation in all situations.

Problem How to offer high end services without jeopardizing the safety of real time functionality?

Forces Safety: Real time functionality must not be interfered by other functionality of the system as it might cause the real time part to function improperly, i.e. deadlines are not met.
Testability: It should be possible to test real time and non-real-time parts separately.
Understandability: It is easier to develope software when the developer does not need to think about the real time requirements.
Response time: Real time part must meet its deadlines as it is used to control the machine.

Solution Architect divides the system into separate levels or layers: machine control level, machine operator level, and possibly emergency stop level. The machine control level handles all lower level operations. For example, steering should be implemented on this level. In this way, the strict response times of the machine controlling functionality can be met. Everything else is implemented in the machine operator level. For instance, if the machine needs to communicate with a system outside the machine, the communications should be implemented on this level as they usually do not require any time critical operations. Additionally, a third level, emergency stop, can be added to the system. This level should do nothing more but to implement emergency stop functionality. When an event that requires the system to stop all functions immediately, this level is used to stop all nodes or controllers in the system. In other words, the third level is an override system for the other two levels.



Rationale By isolating real time parts of the system, basic functionalities are easier to manage. It is easier to test if the real time requirements are met. Higher abstraction level languages and libraries can be used on the machine operator level. This makes the development faster and easier. This also allows use of the COTS software and operating systems.

Related patterns SANDBOX pattern refines this pattern by adding a sandbox to the machine operator level. LIMP HOME pattern can be applied to ensure that basic functionalities are available after failure in the high end service.

Resulting Context A system with distinct levels: machine control level that handles all machine control functionalities and machine operator level where other functionalities are implemented. After this pattern, FIXED PROCESS ALLOCATION and FIXED MEMORY ALLOCATION can be applied to the part of the system that has strict real-time requirements.

Known Usage Fig. 2 depicts division into layers. In the example, the system does not have local user interface at all. The system has a layer that has strict real time demands and is used to control the hardware. The machine control level receives sensor information from hardware and uses it to make control decisions that are realized by sending commands to hardware. Machine control level status is sent to machine operator level through message bus. Machine operator level is simple in this example as it just delivers the data through Ethernet to remote user interface. Machine operator level receives control commands from user through the same Ethernet connection. These user commands may change the machine operator level status. Machine operator level sends control commands forward to machine control level through message bus whenever it is necessary.

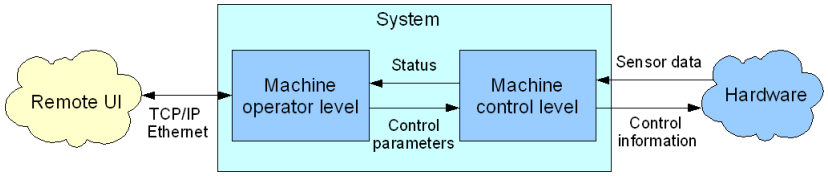


Fig. 2. *Example of responsibilities of levels.*

3.3 Permit Authority

Context A distributed embedded control system that has been divided with ISOLATE FUNCTIONALITIES. The system has multiple nodes that independently make decisions about their functionality. However, there may be situations where the functionality depends on the whole system state. The system state is too complex to be synchronized and stored on each node.

Problem How to ensure that an independent action of a node does not conflict with the system wide state or cause dangerous situations?

Forces Distribution: System is distributed.
Throughput: A node may require information from multiple nodes to make a decision. This may cause a lot of bus traffic.
Safety: Machine controlling should have short response time to ensure safe usage.

Solution The architect adds a dedicated component, an authority node, to the system that keeps track of the system state. This component makes decisions if something can or can not be done currently by the system. There are two possible solutions that the architect can use: other components of the system can ask permission from the authority node or authority node can tell the other nodes to take action.

Rationale Nodes on machine control level have strict real time demands and there may be not enough processing capability or processing time to make decisions. Therefore, we should have a component in the system that always knows the machine's whole situation and can immediately tell if an action is allowed. In this way, the state of the whole system can be stored only in one node.

Related patterns If SANDBOX pattern is applied, authority node can be located in the sandbox. This pattern can use VARIABLE MANAGER on authority node to store the system information.

Resulting Context A system where nodes can easily check from the authority node if they can perform requested action.

Known Usage One node of the system is a dedicated authority node. This node uses VARIABLE MANAGER to store system state. For example, the drive controller receives a command from the bus to move the machine. Before the controller executes the command, it checks from the authority node if the command can be executed. For example, if the machine operator has pressed emergency stop, the command can not be executed.

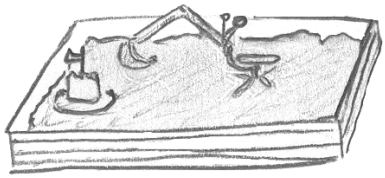
3.4 Sandbox

Context A distributed embedded control system with a SEPARATE REAL TIME. An architect needs to provide a platform for third-party applications, such as fleet management and navigation. It should be possible to use well-available hardware and software to provide a platform for third-party applications.

Problem How to cost-efficiently provide a generic and safe platform to run third-party applications?

Forces Efficiency: Third-party services may need a lot of processing power.
Safety: There is a need to run third-party applications that can not be trusted because they can cause dangerous situations if they interfere with the machine control.
Maintainability: Easy development and maintenance of third-party services on COTS hardware.

Solution The architect adds a component to the system that is used to offer a platform for the third-party software. This component is isolated from the basic machine control system, so it cannot interfere with the normal operation. The third-party applications communicate with the rest of the system via standard interfaces, that the machine vendor has made available. These interfaces must be proven and reliable. The application platform must have enough processing power to support possibly resource greedy third-party software. A common solution to provide this service is to have a PC node on board. If there is PERMIT AUTHORITY in the system it can be combined with the sandbox on the same node. Also if VARIABLE MANAGER pattern is applied there is a possibility to implement centralized system state on PC as there is a lot of resources available. This may make the development of other nodes easier. Remote access and remote diagnostics can be implemented on PC as there is a lot of tools available for that already.



Rationale There is usually a need to offer third-party applications to improve service quality for the machine operator. These applications rarely have real time demands but may require a lot of CPU time. Often their behaviour in all situations can not be guaranteed. Therefore, it is natural to have one centralized computing unit, the sandbox, for such applications. Sandboxing may cause some latencies that may not be acceptable for some third-party applications. Thus, all control functionalities which have real time demands must reside on the control nodes. In other words, architect should not violate the principles that are introduced by SEPARATE REAL TIME. The centralized PC creates a single point of failure and the system should be designed in a way that it is still operable with lower service quality if PC fails, for this see LIMP HOME.

Related patterns LIMP HOME, see Rationale. PERMIT AUTHORITY and VARIABLE MANAGER, see Solution.

Resulting Context A system with capability to offer third-party services safely for machine operator.

Known Usage A node implemented using PERMIT AUTHORITY pattern is replaced with a PC that has the same functionality. Additionally VARIABLE MANAGER is moved to the PC. Machine vendor offers a subcontractor an interface that is used to implement remote diagnostics application. Remote diagnostics application uses the interface to acquire diagnostics data, such as oil pressure and diesel consumption. Subcontractor creates an user interface that is implemented on PC using QT. This user interface can provide different sensor values and production information for the machine operator. Remote access on PC is used to provide work plans in the beginning of the shift and send production reports to organization's database after the operator's shift.

3.5 Variable Manager

Context A distributed embedded control system with several autonomous units. The units must share common state information about the system as a whole. ISOLATE FUNCTIONALITIES has been applied. The architect needs to abstract source and location of the information. There is a need to access all the data provider independently.

Problem How can you efficiently share system state in the distributed embedded system?

Forces Accuracy: Data is volatile.
Efficiency: Message traffic should be minimized.
Scalability: System must be scalable in terms of its units.
Extendability: It should be easy to add new units accessing the state.
Adaptability: It should be easy to change the way state is implemented.
Adaptability: It should be easy to change the location of origin of the state information.
Usability: It should be easy to find the desired state value.

Solution The architect adds a common variable manager module to every node that contains the shared state variables needed by the node. The local value of the remote variable can be updated when a changed value is noticed on the bus. The values of the variables are sent and updated using different strategies: by-request, periodically, as a side-effect for example when another variable is updated. A value can have an associated status or age. If SANDBOX pattern is applied to the system, variable manager can be implemented on the sandbox node. Then it can contain the shared state of the whole system.

Rationale Assuming that the updating frequency of the shared variables is high enough, each node gets sufficiently accurate state information concerning the other parts of the system. Nodes can change information location transparently. This solution does not prevent the nodes from modifying the common system state so that the information becomes inconsistent. Additionally, the solution may result in a large number of variable names that is difficult to manage.

Related patterns The pattern is a special kind of PROXY pattern [6]. BLACKBOARD uses similar kind of data sharing [7]. If variable manager is distributed, it can been seen as a BUS ABSTRACTION. On the other hand, implementation of the distributed variable manager can use BUS ABSTRACTION internally.

Resulting Context A distributed system with common state information.

Known Usage A heavy machinery system such as forestry machine uses multiple controllers to steer the machine. The controllers must share information in order to co-operate. This is achieved with every node keeping track of the needed parts of the system state information. The nodes update their system state information by receiving messages from the bus. Some information-carrying messages such as messages containing sensor data are sent periodically, some are sent when information is available.

3.6 Dictator Watchdog

Context A distributed embedded control system where hardware is replicated to ensure availability with REDUNDANT FUNCTIONALITY. The system needs to know which replicated hardware component is active and recognize that the active unit's control output is within acceptable limits.

Problem How to implement redundant hardware so that the backup unit is available within a short response time?

Forces Availability: A backup unit must take control in predetermined response time when the active unit malfunctions.
Reliability: Unit can not make sanity checks for itself.
Reliability: In two-unit system, a unit can not make reliable sanity checks for each others output.
Scalability: The same solution can be used for more than one backup unit.

Solution The architect adds a watchdog component to the system. The hardware is replicated using at least two redundant units. The watchdog device chooses the currently active unit according to HEARTBEAT or some other external information, for example, output control values. Only the active unit controls the process. If the active unit fails, the dictator watchdog switches the backup unit active and it will take charge of control.

Rationale Both of the units are operating normally, only the active unit's control signals are used. Redundant units share up-to-date state information and it allows the watchdog component to switch between active and backup unit within predetermined response time. As a downside of this solution, the watchdog component is a single point of failure.

Resulting Context A system where hardware is duplicated and the controlling active unit can be switched within predetermined response time.

Known Usage In an embedded control system, a redundancy control unit (RCU) is connected to two redundant control systems. The redundancy control unit is an intelligent switch, which chooses the currently active controller according to the operating statuses of the control systems. Alarm from the sanity checking system signals a malfunction and RCU is notified. RCU tells to backup unit to go active.

3.7 Locker Key

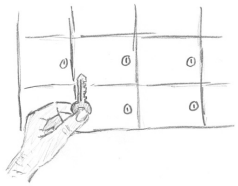
Context A system with fast shared memory and multiple processes or processing units communicating with large messages (ISOLATE FUNCTIONALITIES). The number of messages can be large and the architect wants to avoid memory allocation overhead.

Problem How to avoid dynamic memory allocation for messages and minimize transferred message size?

Forces Efficiency: Creating the message should be fast. Therefore, dynamic memory allocation is not plausible.
Safety: If dynamic memory allocation is used, there might not be free memory available for the message.
Efficiency: Sending the message should be fast.
Predictability: Message size should be constant.
Extendability: It should be easy to add new types of messages.
Extendability: It should be easy to add new senders and receivers.

Solution The architect designs a shared memory for the messages known as lockers. If there are multiple processing units, all the units must have an access to this space. Message sender requests a locker, a space in the shared memory, inserts data in the acquired locker and gives the key (index pointer) to the receiver. The receiver uses the key to access the message from the shared space. A utility function is used usually to

access the locker and the locker is normally freed after the access. The key can be index, Universally Unique ID (UUID) or result from a hash function when security is needed. Variable manager can be used as shared memory if the provided bandwidth is sufficient.



Rationale No dynamic memory allocation is needed for messages. The transferred message size is minimal as only an index is delivered. Solution offers a possibility for the receiver of the key to fetch the message data at suitable moment, for example to balance bus load. There is no protection for the memory space if indexes are used directly. Normal indexes are easy to guess and therefore may compromize locker integrity. By using UUID or hash, the key protection level is higher but also more processing is needed. In the situation where there is memory management provided by the operating system it may be costly and error-prone to map a single physical memory block to different logical addresses on different processes.

Resulting Context A system with fast and memory-saving messaging capabilities.

Known Usage The architect has reserved an array that is used as lockers. The array element size is determined from the largest possible message payload. The lockers are used through an interface. The message sender can store the message payload to the locker using an interface function. The interface then returns the key for the particular locker. The sender delivers the key to the message receiver. The receiver uses the key to retrieve the message payload through the interface. When the key is used the locker space is freed. In other words, the same key can be used only once.

4 Conclusions

In this paper, we have introduced a set of patterns that we have found during architectural evaluations in Finnish machine industry. They form a basis for an embedded control system pattern language. These patterns reflect the characteristics of evaluated embedded systems, namely distribution, real-time and fault-tolerance.

The pattern set is a promising start for a full-blown pattern language. Even though there might be some omissions, the discovered patterns cover the most important solutions identified during the software architecture assessments in the machine industry.

The work is expected to continue, leading to a more comprehensive and more systematically organized pattern language of embedded machine control systems. This language could serve as a toolbox for software designers in machine industry.

References

1. IEE: Embedded systems and the year 2000 problem guidance notes. Embedded Systems and the Year 2000 Problem: Guidance Notes., IEE Technical Guidelines 9:1997 (1997)
2. Alexander, C., Ishikawa, S., Silverstein, M.: A Pattern Language: Towns, Buildings, Construction (Center for Environmental Structure Series). Oxford University Press (August 1977)
3. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley Professional (January 2002)
4. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. Wiley (May 2007)
5. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
6. Vlissides, J.M., Coplien, J.O., Kerth, N.L.: Pattern languages of program design 2. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1996)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. John Wiley & Sons (August 1996)