Patterns for Distributed Embedded Control System Software Architecture

Veli-Pekka Eloranta, Vesa-Matti Hartikainen, Marko Leppänen, Ville Reijonen, Ilkka Haikala, Kai Koskimies, and Tommi Mikkonen

Tampere University of Technology
Department of Software Systems
{firstname.lastname}@tut.fi

Abstract. In this paper, a set of 11 embedded machine control system patterns are presented. These patterns were identified during architectural assessments carried out at several sites of Finnish machine industry. The pattern set is categorized according to three major characteristics of such software: distribution, real-time, and fault tolerance. The pattern set does not yet constitute a full-blown generative pattern language.

1 Introduction

By one definition, embedded system is a devised to control, monitor or assist the operation of equipment, machinery or plant. Embedded reflects the fact that they are integral part of the system. [IEE 1997] In this context, we mean by embedded control system a system that controls large machines such as harvesters and mining trucks. Such systems are often tightly coupled with their environment, implying new 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 architecture has grown in importance but there is not much literature on embedded control system area. Therefore, we need pattern set which helps to design such 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 device is that of the hardware and machinery of the system and not 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 [Alexander et al. 1977]. Therefore, we feel that this particular field is a good target for patterns.

During spring 2008, 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, we have aimed at finding patterns by integrating pattern mining with an architectural assessment we per-

formed to different embedded software systems provided by the companies. The patterns we have recognized reflect embedded system characteristics of machine industry: distribution, real-time, and fault tolerance. In this paper, we introduce a core subset of the found patterns organized in accordance to these characteristics. The work is expected to continue, leading to a more comprehensive and more systematically organized pattern language of embedded machine control systems.

The paper is structured as follows. Section 2 introduces our pattern mining process and provides an overview to the different pattern categories. Section 3 gives the actual pattern descriptions. Section 4 discusses related work, and Section 5 concludes the paper.

2 Overview on patterns

Even though patterns presented in this paper are new in this context, a subset of them can be found in different domains in literature. For example, fault tolerance patterns are well covered by [Hanmer 2007]. Additionally, other sources present patterns that are applicable in this context. Still, even when using existing patterns it takes some effort to find suitable patterns and transform the context to achieve a consistent set.

We integrated our pattern mining with architectural assessment. The process for architectural assessment was derived from ATAM [Clements et al. 2002]. Because our 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.

As the outcome of the visit to the different sites, we had a catalog of 21 pattern candidates, from which 11 were selected as they were discovered in multiple systems and they form one consistent set. In the future, it is possible that the pattern candidates that were excluded could be a part of the final set, a language, if they are encountered again.

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 section where we discuss forces that are goals and constraints. 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.

The categories we found appropriate for the purposes of machine industry are the following:

1. *Distribution*, i.e., patterns that are associated with the fact that the system consists of several nodes each of which may contain several processes running in parallel.

2. *Real-time*, i.e., patterns that aim at satisfaction of given real-time restrictions, or enhance predictability of the system.
3. *Fault tolerance*, i.e., patterns that serve in identification, isolation or recovery of error situations.

Individual patterns that fall to these categories are listed in Table 1.

Table 1. Patterns, pattern categories, and pattern introductions

Category	*Pattern*	*Description*
Distribution	MESSAGE BUS	Nodes communicate through a message-passing bus. Application logic is distributed to different nodes.
	HARDWARE ABSTRACTION LAYER	The hardware is abstracted by a driver (or manager) with an abstract interface.
	COMMON SYSTEM STATE	Predefined named slots in shared memory are allocated for maintaining the state of the entire system.
	SINGLE POINT OF UPDATING	A higher-level subsystem can update lower-level subsystems.
Real-time	FIXED PROCESS ALLOCATION AND SCHEDULING	All processes are created and initialized when the system starts. Schedule is fixed during the compile time.
	STATIC SCHEDULING WITH EXTERNAL CLOCK	Program is scheduled statically during compilation time. At runtime, the system monitors the schedule with an external clock.
	FIXED MEMORY ALLOCATION	Memory is allocated when the application starts. The maximum used memory is calculated beforehand.
	LOCKER KEY	A fixed buffer is used for messages. The sender requests space, insert data in the acquired space, and gives a reference to the message to the receiver, thus avoiding memory allocation.
	MANAGERIAL BOSS	One unit has the overall control of the system, but all hard real-time actions are performed by low-end embedded controllers having only partial information on the system state.
Fault tolerance	DISTRIBUTED SAFETY	Possibly physically dangerous functionality is divided into multiple nodes to prevent actions in case of a failure in a node.
	VIRTUAL TIMESTAMPS	Events in different processes are time-stamped to create ordering for e.g. logging.

3 Patterns

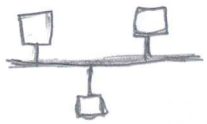
3.1 MESSAGE BUS

Context
A distributed embedded control system managing multiple device nodes. There is a physical connection between the nodes.

Problem
How to communicate in an abstract way between the nodes in the system?

Forces
Scalability: The system may incorporate many nodes 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 nodes changes, even during the run-time of the system.
Reusability: The same bus abstraction can be used by other nodes.
Portability: During the life cycle of the product, the physical bus may change.

Solution
The architect introduces a dedicated message dispatching component, which receives message dispatching requests, and registers nodes interested in certain kinds of messages. The message dispatcher takes care of the routing of the message to the appropriate receivers, without the sender knowing the identity of the receiver. This requires a common message format.



Rationale
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.

Related Patterns
The pattern can be viewed as a MESSAGE DISPATCHER, applied to an embedded machine control system with a physical bus or connectivity.

Resulting Context
Distributed and scalable machine control system where nodes communicate with each other via the bus. It may be difficult to know if requested service is local or remote.

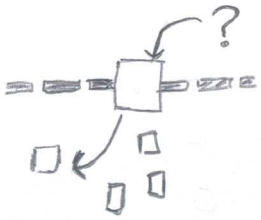
3.2 HARDWARE ABSTRACTION LAYER

Context
An embedded control system controls hardware devices (e.g. sensors and actuators).

Problem
How can you control different kinds of hardware within application, without needing to know the details of the hardware devices?

Forces
Scalability: System must be scalable in terms of the number and type of hardware devices.
Extendability: It should be easy to add new hardware devices.
Adaptability: It should be easy to change the hardware.
Usability: It should be easy to write new software controlling the devices.

Solution
The architect designs a hardware abstraction layer interface containing functions to control all the hardware devices. Device interface is generic so that multiple, similar devices can use the same interface functions.



Rationale
Similar hardware devices can be used in the same way through the common interface. Hardware can be changed under the abstraction without changes to the interface. Adding a new device type is possible by implementing a new interface. If the underlying device changes drastically, a new API is required for the device. The abstraction layer is not violated if design-by-contract [Meyer 1986] is used. If there is a need to pass the abstraction layer, it will result in abstraction leaks.

Related Patterns
LAYERED ARCHITECTURE [Buschmann et al. 1996]

Resulting Context
Devices in the same hardware category can be accessed in the same way.

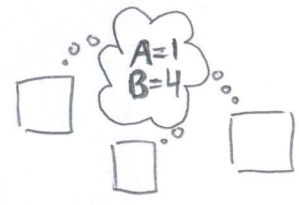
3.3 COMMON SYSTEM STATE

Context
Several autonomous units in a system must share common state information about the system as a whole.

Problem
How can you efficiently share sufficiently accurate, consistent system state between different parts of 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 to a central node a common variable manager module that contains the variables constituting the shared state. The other nodes access the variables by static names using MESSAGE BUS. The values of the variables are 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.



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.

Related patterns
The pattern is a special kind of PROXY PATTERN [Vlissides et al. 1996]. BLACKBOARD uses similar kind of data sharing [Buschmann et al. 1996].

Resulting Context
A distributed system with common state information. The solution results in a large number of names that may be difficult to manage.

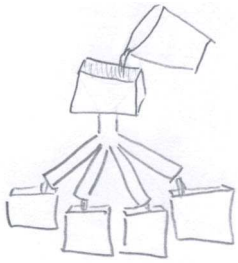
3.4 SINGLE POINT OF UPDATING

Context
A distributed embedded control system. Several autonomous units in a system have individual software installations that communicate with the rest of the system. The MESSAGE BUS pattern has been used to achieve this.

Problem
How can you be sure that the system has compatible software versions on every node?

Forces
Reliability: Different software versions of a node can be incompatible with other nodes. This may lead to bugs that are difficult to trace.
Usability: Version updates should be done easily and quickly.

Solution
The architect organizes software version updates so that a single node handles all software packages in the distributed embedded system. A complete update package is delivered to every other node through the updating node. A single node is never updated from another update package than the other nodes or bypassing the updating mechanism. Version package can be stored in a version repository. Updates may be done either in push or pull strategy. Existing configuration may be updated with push strategy where updating unit sends updates to every unit. When a new hardware is added, pull strategy can be used where the added unit asks for a software update from the updating unit. Updates are stored in a version repository, so that an older update combination can be used if necessary.



Rationale
Since the updating is carried out through a single point in a single updating package, the system is always in a consistent state as far as versions are concerned if the whole update is successful. A great care must be taken when creating installation package to be sure that it is applicable in all existing environments.

Resulting Context
The embedded system can be updated so that the new version is guaranteed to be consistent if the whole update is successful.

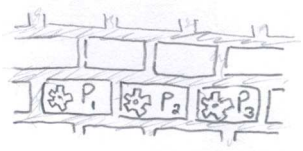
3.5 FIXED PROCESS ALLOCATION AND SCHEDULING

Context
An embedded control system requiring several processes. The scheduling is not pre-emptive because there are no priorities between the processes. However, some tasks may need more frequent repetition and some tasks may need more processing time.

Problem
How to create processes with predictable time behavior?

Forces
Accuracy: Startup time of a process is not predictable.
Accuracy: Startup sequence is not predictable.
Accuracy: Scheduling dynamic loads is unpredictable.
Efficiency: Limited amount of CPU time to use.
Efficiency: There must always be enough processing power for the main functionality.

Solution
Initialization time is not known beforehand for all tasks (processes, threads, etc.) and therefore all resources are reserved during system startup. The system designers divide the system into executable blocks, which have their own execution time. During the compilation, the worst-case scenario is calculated for every block using known instruction execution times. The fixed scheduling is adjusted based on the calculations. There are no other external interrupts except synchronization pulse. If there is a need for system maintenance tasks, those can run as long as there is free time available between the scheduling cycles.



Rationale
When all processes are allocated in the beginning, you do not have to implement safeguards, and execution time is predictable. Implementing safeguards for unsuccessful process creations would require more memory space and processing power. This would be away from the data. Because of the seldom-occurring worst-case scenarios are used for time slot reservations there is inherent overhead in scheduling. As there are no interrupts, events must be polled.

Related patterns
POOLED ALLOCATION [Noble, Weir 2001]. ABACABAD scheduling algorithm.

Resulting Context
All the resources must be allocated at startup. STATIC SCHEDULING WITH EXTERNAL CLOCK can be used to synchronize execution for increased accuracy.

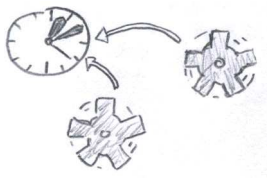
3.6 STATIC SCHEDULING WITH EXTERNAL CLOCK

Context
This pattern refines FIXED PROCESS ALLOCATION AND SCHEDULING. Embedded control system requiring several tasks.

Problem
How to make sure that a node running hard real-time processes stays synchronized internally or with other nodes?

Forces
Accuracy: Separate task units must stay in sync.
Accuracy: The local clock might not be accurate enough in all conditions.

Solution
The architect creates static scheduling based on the predetermined execution times. After a timing signal arrives from an external clock, a new scheduling cycle is initiated. The external clock can be for example GPS or an atom clock. After all tasks have been allowed to execute, the system waits for the next timing signal. If the processing is unfinished when the next scheduling cycle starts, a fault is generated unless it is system maintenance task, which is allowed to run as long as there is free time.



Rationale
External clock is not affected by the operation of the system and therefore it provides precise timing. Since execution times of all operations are known, static scheduling is possible. Scheduling fault should never occur, as the scheduling is predetermined. Tasks with no strict real-time requirements can be run in MANAGERIAL BOSS.

Resulting Context
Different units of the system are scheduled synchronously. The deadlines of the tasks are always met.

3.7 STATIC MEMORY ALLOCATION

Context
Embedded control system with limited amount of memory. The maximum need for stored data is known.

Problem
How to get rid of the risks caused by failed memory allocation and variance in memory allocation time in a processing and memory limited environment?

Forces
Reliability: Memory allocation must not lead to a failure.
Efficiency: Memory allocation time may vary.
Predictability: Maximum memory consumption at run-time must be known.
Efficiency: Limited amount of available CPU time.
Efficiency: There must always be enough processing power for the main functionality.
Efficiency: The needed memory space cannot be reduced by means of compression.

Solution
The architect calculates the memory budget beforehand. All memory is allocated at system startup even if it is not needed yet.

Rationale
When all memory is allocated at the startup, you do not have to implement safeguards, execution time is predictable, and the memory usage is known. Implementing safeguards for unsuccessful memory allocation would require space for code from data, and extra processing power. Dynamic memory reservation is not possible. At run-time, there is no possibility to use new data with unknown size. Pre-allocation may consume too much memory. Because the memory budget is for worst-case situation, the application has unused allocated memory most of the time.

Related patterns
FIXED ALLOCATION [Noble, Weir 2001]

Resulting Context
A system where no memory is allocated after startup.

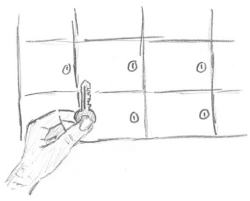
3.8 LOCKER KEY

Context
A system with shared memory and multiple processes or processing units communicating with messages or relaying data.

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

Forces
Efficiency: Creating the message should be fast. 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 an index, a random UUID or a result from a hash function when security is needed.



Rationale
No dynamic memory allocation is needed for messages. The transferred message size is minimal as only an index is delivered. Understanding the locker content is on the responsibility of the receiver. There is no protection for the memory space if indexes are used directly. By using UUID or hash, the 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.

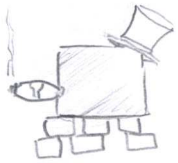
3.9 MANAGERIAL BOSS

Context
A distributed embedded control system with a MESSAGE BUS.

Problem
How to offer high-end services for the machine operator and execute applications that have strict real-time requirements, so that these two do not interfere with each other?

Forces
Efficiency: High-end services need a lot of CPU time.
Efficiency: Machine control has strict real-time requirements.
Testability: Real-time requirements must be validated.
Safety: Machine controlling should have short response time to ensure safe usage.
Safety: There is a need to run third party applications that cannot be trusted because they can cause dangerous situations.
Maintainability: Easy development and maintenance for high-end services.

Solution
The architect adds a dedicated component to the system that runs applications that do not have strict real-time requirements. Time-critical operations reside in controllers that execute real-time operations.



Rationale
Not all operations can be implemented on a centralized PC, as the bus and distribution can cause latencies, and some operations need real-time execution. There are usually services in the control system that offer improved service quality for the machine operator. These services rarely have real-time demands but may require a lot of CPU time. Therefore, it is natural to have one centralized computing unit, the managerial boss, for such services. This separates third-party applications that may cause dangerous situations when interfering with the machine control functionality. The system allows safe execution of real-time operations. Processing-intensive operations can be run without need to worry about performance issues. The centralized PC might fail and the system should be still operable with lower service quality.

Resulting Context
This results in a system with capability for both processing-intensive high-level operations and time-critical real-time operations. After this pattern, FIXED PROCESS ALLOCATION and STATIC SCHEDULING WITH AN EXTERNAL CLOCK can be applied to the part of the system that has strict real-time requirements.

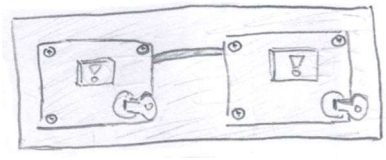
3.10 DISTRIBUTED SAFETY

Context
A distributed embedded control system.

Problem
How can you prevent possible dangerous situations in case of either hardware or software failure in some unit?

Forces
Testability: Testing of all possible situations in all possible hardware configurations is not possible.
Safety: System should not cause danger when a single unit fails.
Redundancy: Duplication of components is not economically plausible and might not solve all risks.
Efficiency: Safety checks should not add overhead to code or communication over network.
Understandability: Safety checks should not make code too hard to understand.

Solution
The architect divides functionality into multiple nodes communicating with each other. An action is only executed if all nodes work harmoniously together. When communication fails for one reason or another, the system is set or stays in a safe state. In this way, a misbehaving node will not cause any dangerous situations. An example: the steering controller asks power from the motor. If the motor cannot give the power, no actions are taken. If the motor gives too much power, the steering can stop the movement.



Rationale
A single node cannot cause dangerous situations because actions need co-operation with other nodes. If some operation is divided into two nodes when it normally would only use one node, the risk of losing functionality doubles because the hardware failure probability increases. The system is somewhat less efficient and understandable because of the added complexity.

Related patterns
WATCHDOG, HEARTBEAT [Hanmer 2007]

Resulting Context
A distributed system with improved safety level where a dangerous situation cannot occur because of a single failure.

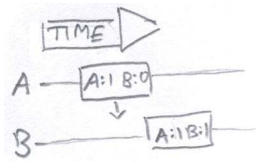
3.11 VIRTUAL TIMESTAMPS

Context
Embedded distributed control system.

Problem
How to know the order of events in a distributed embedded system?

Forces
Testability: System execution history has to be logged.
Accuracy: The order in which events are triggered should be traceable.
Accuracy: It is impossible to have an (absolutely) synchronous clock in a distributed system.

Solution
The architect designs a system that uses virtual time provided by a vector clock such as Lamport timestamps [Lamport 1978] to stamp events. For example, an event counter can be used for the timekeeping. The event counter is synchronized between the nodes every time the communication occurs.



Rationale
With vector clock, partial order of events in a distributed system is known. If there is no communication between nodes the internal order of events in a node cannot be compared with some other node's internal event order.

Resulting Context
The order of events can be determined when communication occurs between nodes in distributed system.

4 Related work

In this paper, we focus on embedded machine control system patterns concerning distribution, real-time efficiency and fault tolerance issues. In related pattern work, there are some same and similar patterns concerning some of the same issues but are not presented in machine control system domain.

Low level embedded system patterns are presented in [Pont 2001]. The patterns are more hardware oriented than our abstract system architecture patterns. For example, Pont also discusses the scheduling methods. Nevertheless, Pont focuses on single microprocessor family and gives some pure hardware solutions.

Telecommunication, distribution and fault-tolerance patterns are discussed in [Rising 2001]. Nevertheless, there is no complete generative pattern language for embedded machine control system presented. Some of the patterns are similar or same and can be applied to this paper's domain.

Small memory patterns are presented in [Noble, Weir 2001]. Unlike in our work, distribution and large memory solutions are not discussed. There are similar or same patterns for memory and process allocation. Additionally, Noble presents set of compression patterns, which we have not encountered in our focus area.

Patterns for fault tolerant system are discussed in [Hanmer 2007]. There are similar patterns to ours that could be incorporated to our pattern set but we have not encountered them yet. On the other hand, some of the patterns presented in this paper cannot be found from Hanmer's book.

Buschmann et al. [1996] discusses mainly architectural styles and distribution. For example, blackboard architecture is presented in the book. To some extent, these patterns can be applied to our domain.

Related pattern publications mainly discuss about only one specific aspect or hardware system. Our patterns emphasize the architectural structure of the embedded control system and purely hardware level patterns are ignored, but the special nature of the embedded environment is taken into account.

5 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. Naturally, there are some omissions, but some of the discovered unpublished pattern drafts will fill some of these gaps. Presented patterns cover the most important solutions identified during our software architecture assessment project in the machine industry.

Our future goal is to complete the pattern set and create a generative pattern language for machine control systems. This language could serve as a comprehensive toolbox for software designers in machine industry. This is achieved by evaluating more machine control system architectures.

References

Alexander, C. et al.: A Pattern Language: Towns, Buildings, Construction. Oxford University Press, 1977.

Buschmann, F. et al.: Pattern-Oriented Software Architecture: Volume 1, A System of Patterns. Wiley, 1996.

Clements, P. et al.: Evaluating Software Architectures. Addison-Wesley, 2002.

Hanmer, R. S.: Patterns for Fault Tolerant Software. John Wiley & Sons, 2007.

IEE, Embedded Systems and the Year 2000 Problem – Guidance Notes. Institution of Electrical Engineers, 1997.

Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of the ACM, 1978.

Noble, J., Weir, C.: Small Memory Software. Patterns for Systems with Limited Memory. Addison-Wesley, 2001.

Meyer, B: Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986

Pont, M. J.: Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers. Addison-Wesley, 2001.

Rising, L.: Design Patterns in Communication Software. Cambridge University Press, 2001.

Vlissides, J. et al: Pattern Languages of Program Design 2. Addison-Wesley, 1996.