Patterns for Distributed Machine Control System Data Sharing

Marko Leppänen and Veli-Pekka Eloranta
{firstname.lastname}@tut.fi

Department of Software Systems
Tampere University of Technology
Finland

1 Introduction

In this paper we will present three patterns for sharing sensory data and other information in distributed machine control systems. A distributed machine control system is a software entity that is specifically designed to control a certain hardware system. This special hardware is a part of a work machine, which can be a forest harvester, a drilling machine, elevator system etc. or some process automation system. Some of the key attributes of such software systems are their close relation to the hardware, strict real-time requirements, functional safety, fault tolerance, high availability and long life cycle.

Distribution plays a major part in the control systems. Different functional hardware parts of the machine are physically apart from each other and their corresponding control software is usually located in a embedded controller node near the controlled hardware. The nodes must communicate with each other in order to perform their functionalities. It is also common that the system nodes have very wide variety in their computational capabilities. Usually the system has several simple embedded controllers with limited computational abilities also known as low-end nodes. In addition to these embedded controllers the system may contain one high-end node that has processing power that is comparable to a common desktop PC. Due to these facts, a distributed control system needs to distribute information between different parts of the system. The information-sharing capabilities of such systems is discussed in these patterns in more detail.

The patterns in this paper were collected during years 2008-2011 in collaboration with industrial partners. Real products by these companies were inspected during architectural evaluations and whenever a pattern idea was recognized, the initial pattern drafts were written down. These draft patterns were then reviewed by industrial experts, who had design experience from such systems. After these additional insights, and iterative repetitions of the previous phases, the current patterns were written down. We hope that the final pattern language can be tested on implementation of some real system after all patterns in the language are published.

The published patterns are a part of a larger body of literature, which is not yet publicly available. All these patterns together form a pattern language, which consists of more than 70 patterns at the moment. A part of the pattern language in this paper is presented in a pattern graph (Fig. 1) to give reader an idea of how these selected patterns fit in the language. These three patterns are closely related in the pattern language and therefore are ideal to be submitted together as a whole. In the following sections, all the pattern names are written in SMALL CAPS.

In the second section, we will first introduce our pattern language and the pattern format. Following this, the selected three patterns are presented in detail. Finally, the last sections contain the acknowledgments and references.

2 Patterns

In this section, a set of three patterns is presented. Together, these patterns form a sublanguage in the pattern language in Fig. 1. The pattern graph is read so, that a pattern is presented as a box in the graph and an arrow presents a connection between the patterns. The connection means that the pattern from which the arrow emerges is refined by the pattern that the arrow points to. In other words, if the designed system still has some unresolved problems even after some pattern is applied, the designer can look to the refining patterns for yet another solution if they want solve the current design issues. The patterns refine each other extending the original design with other solutions.

For example, the CONTROL SYSTEM pattern is the root of the whole pattern language and it is referenced in the following patterns. So, the CONTROL SYSTEM is the central pattern in designing distributed control systems. It presents the first design problem the system architect will face: Is a control system needed in this context? Table 1 presents all patterns that are shown in Fig. 1 and all the patterns that are referenced later on in this paper.

Table 1: Patlets

Pattern Name	**Description**
CONTROL SYSTEM	**Implement control system software that controls the machine and can communicate with other machines and systems.**
ISOLATE FUNCTIONALITIES	**Distribute the system into subsystems according to their functionalities. Interconnect these subsystems with the bus. Use multiple interconnections between subsystems if necessary.**
SEPARATE REAL-TIME	**Divide the system into separate levels according to real-time requirements: e.g. machine control and machine operator level. Real-time functionalities are located on the machine control level and non real-time functionality on the machine operator level. Levels are not directly connected, they use bus or other medium to communicate with each other.**
DIAGNOSTICS	**Collect such data from a system, which allows to notice if some subsystem starts to operate poorly or produces erroneous data. Usually all data values have limits where they should operate and a deviation from this indicates a risk of breakdown.**
REMOTE ACCESS	**Add an operator level component which allows at least sending of the machine data to the operational command level.**
VARIABLE MANAGER	**For each node, add a component, which contains all the information that is relevant to operation of the corresponding node. This information is presented as state variables. The value of a variable is updated every time when a message containing the information is received.**
STATE VARIABLE GUARD	**Design a mechanism to guard the state variables. It must demand authorization for other system parts to submit their own state changes to system state information.**
VARIABLE VALUE TRANSLATOR	**Add a converter layer on the top of the variable manager. This layer converts the data to the correct unit, e.g. from mph to km/h or from inches to cm, when data is requested.**
SNAPSHOT	**Implement a mechanism to save the current state information (e.g. from VARIABLE MANAGER) as a snapshot. This mechanism should also be able to restore system-wide state from the snapshot.**
CHECKPOINT	**Describe the properties which may or will change during the life cycle of the machine as parameters. The parameters can be altered from the UI when necessary.**
PARAMETERIZABLE VALUES	**Describe the properties which may or will change during the life cycle of the machine as parameters. The parameters can be altered from the UI when necessary.**
DATA STATUS	**Add status information to each data nugget or variable. Status information tells the age and/or state of the information (OK, fault, invalid, etc).**
COUNTERS	**Create a counters service that provides counting functionality for different purposes. This makes it easy to have centralized diagnostics and logging of different kinds of information.**

Continued on next page...

Table 1: (Continued)

Pattern Name	Description
ERROR COUNTER	**Create a counter which threshold can be set to certain value. Once the threshold is met, an error is triggered. The error counter is increased every time a fault is reported. The counter is decreased or reseted after certain time from the last fault report has elapsed.**
VARIABLE CACHING	**Store variables retrieved from other nodes locally for a certain period. After the variable spesific predetermined period has elapsed, cached value can not be used any more. A new value should be retrieved from the source node.**

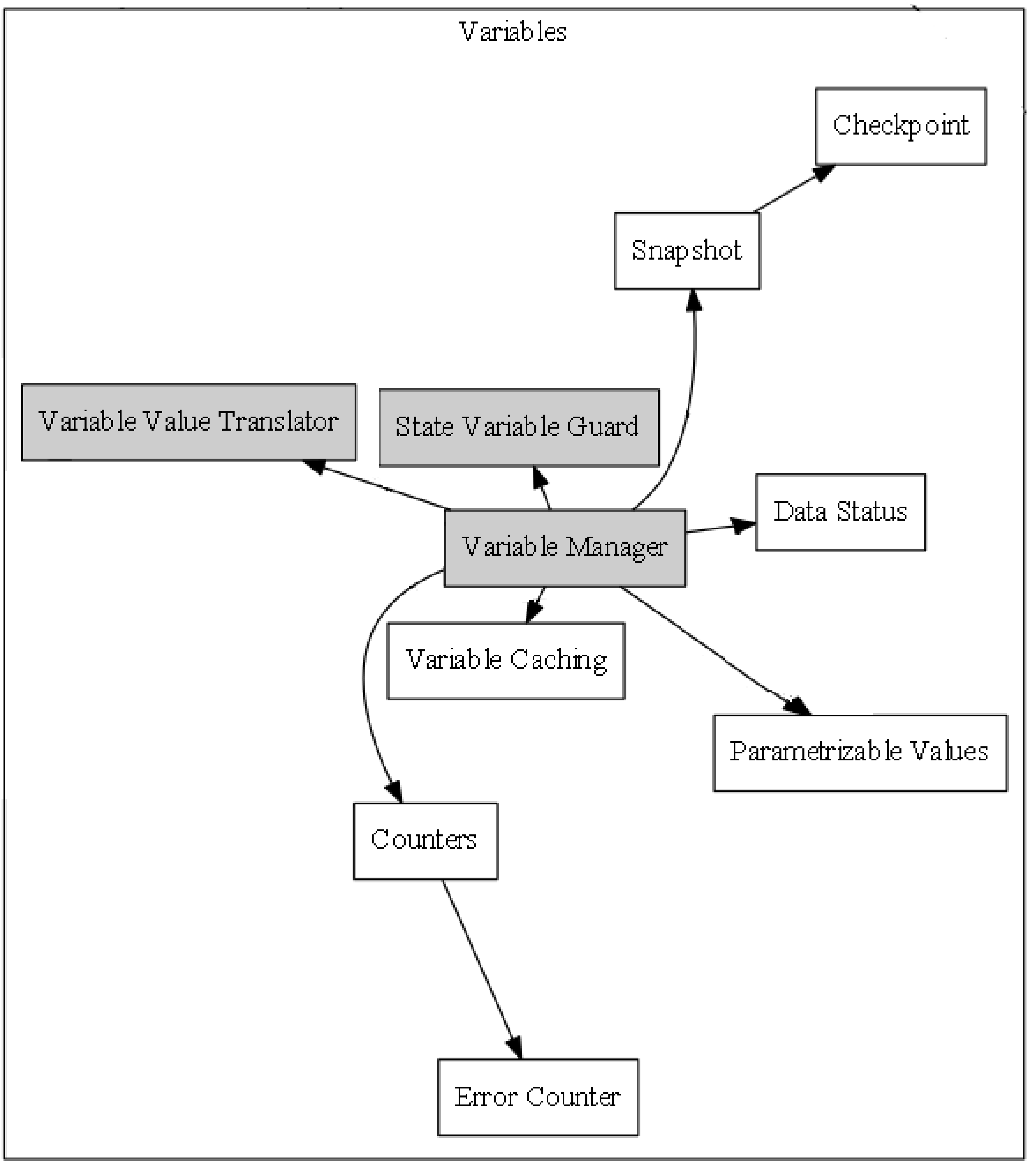


Fig. 1. The variable sublanguage. Patterns that are presented in this paper are highlighted.

Our pattern format closely follows the widely-known Alexandrian format [1]. First we present the context for the problem. Then, the problem is concentrated in a couple of sentences that are printed with a bold font face. After that, a short discussion about all forces that are affecting the problem is given. In a way, it is a list of things to consider when solving this problem. Then, after word "Therefore:" the quick summarization of the solution is given. Then, after a three star transition line, the solution is discussed in a detail. This section should answer all the forces that were left open in the previous section. Then an another star transition marks the end of the section. This section describes briefly the consequences of applying this pattern. After the last star transition a real life example of the usage of this pattern is given.

2.1 Variable Manager

...you have a distributed machine CONTROL SYSTEM which consists of several independent nodes. Because ISOLATE FUNCTIONALITIES has been applied, the nodes have their own responsibilities to attend to. The nodes have a communication bus between them allowing them to co-operate in order to perform the tasks initiated by the machine operator. In order to carry out these operations, the nodes gather data from their environment using sensors and perform computations using this sensory data as inputs. In addition to this information, the performed actions alter the system state and this state information should be stored as well. From all this information, some is purely for local use in the node itself, but some of the information is needed also in some other nodes in the system for co-operation.

System wide information should be shared efficiently in the distributed embedded system.

In order to co-operate successfully with other nodes, a node must have access to all the required information to carry out its own duty. However, only some of this necessary information is produced locally, so the node needs to get complementary data from the other nodes. In addition to consuming information from the other nodes, the node may produce information that is needed elsewhere in the system. So, some sort of communication between nodes is necessary. As the system is distributed, the communication can be carried out using a shared channel, a bus, allowing the nodes to share information. However, as the bus has a limited data transfer speed and communication initialization overhead, all communication between the nodes has some inherent latency. Because of this latency, it is not sufficient to use simple query-and-answer based communication scheme to share the information as it is usually too slow for real-time environment.

As the system has strict real-time requirements, a node should have a quick access to the required information. If the information sharing is implemented by query-and-answer based scheme, all queries also interrupt the source node from its tasks as it has to react to the received query. This taxes the processing capacity of the node and takes time from more urgent activities.

In addition to having latency, the communication channel also has limited bandwidth, so the remote information cannot be accessed remotely whenever demanded as constant queries of updated information from other nodes tax the bus capacity. The more nodes are present in the system, the more bus capacity they consume as the number of the communicating parties grows.

It should be easy to add new nodes to the system as the core system may be extended by some optional features that have to co-operate with the rest of the system. Some of these options may be developed long after the core system has been released. This means that the core system has to be prepared to new consumers of the produced information. The producer and the consumer must be decoupled from each other in such manner, that the producer does not need to know which other nodes use the information it provides and the consumer does not need to know where the information it acquired originates from.

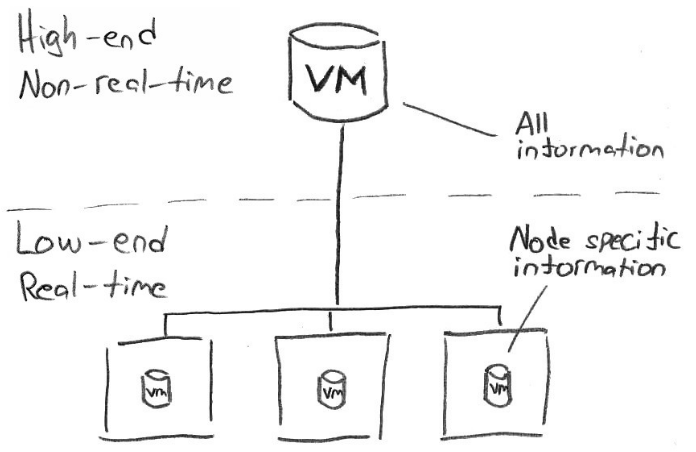
In addition to vendor-made new features, some new features could be made by a third party. These third-party components should have a way to access some of the information that is produced by the system if they need it in order to carry out their function.

Therefore:

For each node add a component, VARIABLE MANAGER, that stores the system information as

variables and provides an interface for reading and writing them. Whenever this component notices new information on the bus, it updates all the corresponding variables. When a variable is locally updated through the interface, the information is also sent to the bus.

*   *   *



All relevant information should be presented as variables. A variable in this context is a piece of information that is necessary for the operation of the machine, such as engine temperature or hydraulic pressure. A variable can also present some production data, such as the amount of drilled holes or the species of a tree in the harvester head. The variable manager component provides an access through an interface to these variables for the node. Whenever a node processes data, receives a new sensor reading or otherwise produces new data that is relevant to the system operation it must update its own variable values which correspond to this new information.

In order to propagate the local variable changes, the variable manager communicates with other nodes using the shared communication medium. This communication is carried out by broadcasting messages which carry the values of the local variables of the source node. Depending on the message size and allowed space for data, these messages can include several variable values or just those which have changed since the last message. As the messages are broadcast to the bus, all the other variable managers can listen to the communication medium and receive all the messages. In this way, the recipient node may decide if the information in a message is relevant to its function and the producer of the data does not know about its users. When a node receives a relevant message, the variable management component updates the corresponding variable values. Because the nodes can use the same method for storing their local data, there is no fundamental difference in using remote and local values.

The variable manager also stores a time stamp with every variable. This is done to prevent the usage of outdated values. So, whenever the node needs to use a variable value, the variable manager checks the time stamp for how old the data is. The time stamp is compared to a predefined time limit and if the data has gotten old, the variable manager may request a new value using a broadcast message that triggers the source node to send a new message containing the current value of the requested variable. However, these kinds of requests take time from all participating nodes and the real-time performance of the system may be threatened. Therefore, updating the values using a polling mechanism should be considered only as an error handling mechanism and the polling should be an abnormal situation which should be avoided. To prevent it from happening, all crucial data must be updated in a cyclic manner using a high enough refresh rate. It is the responsibility of the designer to decide if the variable is updated in a cyclic way after fixed intervals or whenever the value is changed. This configuration requires detailed knowledge of the nature of the variables and their assumed refresh rates. For example, a measurement with 50 ms sample rate might be needed on the bus with a similar rate in order to ensure smooth operation of the control system. On the other hand, if the measured variable is slowly changing one, like outside temperature, it might be reasonable to update this variable only when it changes more than a predefined threshold. For example, in the case of CANopen bus[2], the selection can be between a timer-driven and cyclically sent PDO frame and an event-based SDO frame.

If the node must react almost immediately to a some system state change, it may be better not to use the variable management component to store the data at all. If an urgent message is received, the node just interrupts the current task and carries out the actions mandated by the message. In this way, the data that is used as the basis for the action is always fresh and the node needs not to allocate storage space for the variables.

If SEPARATE REAL-TIME is used, it may be advisable to implement a special variable manager on the high-end node which stores all the information in the system. This enables new services to the system, like DIAGNOSTICS data gathering, a SNAPSHOT of all the data in the system, REMOTE ACCESS and VARIABLE TRANSLATOR. It also makes system debugging and testing easier as one node has a comprehensive set of all the variables in the system.

This pattern is a special version of Caching [3] and can be seen as a version of Data Repository too. CAN Object Dictionary is a simple version of this [4].

* * *

Using the variable manager component, the system can share all the information in a uniform way throughout the system. All shared information can be rendered location-transparent, as the receiver does not have to know the sender of the message. The location-transparency makes it easy to add new sources of information, as the receivers are not interested in the producing party - only the data.

However, as the variables are only updated periodically or by request, the data may be old when it is needed. The detection of out-dated data and requesting new values takes time and thus the reaction times may be longer. The prolonged reaction times may make this solution not suitable for event-based systems.

As the variables are stored locally to the node, they consume storage space from the node. Usually nodes are quite low-end devices so these resources may already be scarce.

It is easier for a developer to use a consistent interface and variable naming scheme to access data regardless of if the original data is produced locally or remotely. It it also possible to implement tools which can aid in data visualization and debugging when all important data is presented using variables. However, variable namespace may become cluttered during the long life cycle of a control system, especially if the system has many shared variables. When a new variable is introduced, the designer must make sure that the name assigned to it is not already used. Moreover, whenever a variable is removed, it must be made sure that no one is anymore using it.

It may be hard to design the correct updating and invalidating strategy for a variable. If a variable is updated too often, it may cause excessive bus load. On the other hand, if the remote nodes get an update too seldom, their calculations may be imprecise and some important events may pass unnoticed.

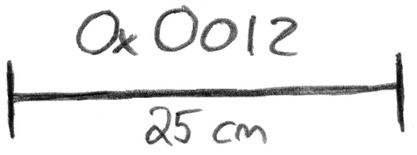
* * *

A forest harvester control system consists of multiple nodes, some of which are low-end control nodes and one is a high-end control PC. These nodes are connected with a CAN bus. When the operator wants to fell a tree, she uses the controls in the cabin to command the control PC. The operator has to feed in some attributes of the current tree such as its specimen type and the desired length of the logs. These attributes are shared as variables in the system, so they propagate through the connecting bus to the nodes as periodic messages. The harvester head node uses these variables in its own operations, such as calculating the correct feed speed of the tree in order to saw logs of the desired length. As the harvester head node feeds the tree, it measures the length of the log and shares this as a variable. The harvester head node uses this information in its own operation as the feeding must be stopped at the right moment. This is fully local operation as the round trip latency for communicating this information to another node and getting a stop command as a reply would be too great. However, the cabin PC also uses this variable to show the operator information about the feeding process.

2.2 Variable Value Translator

...you have a distributed machine CONTROL SYSTEM which consists of several independent nodes. These nodes must co-operate in order to perform the tasks the machine operator wants to. The nodes gather data from the environment using sensors, perform computations from this data and the actions of the nodes usually produce state information, which are presented as variables. Usually the measured variables in the nodes represent different physical quantities in different resolutions. For example, a measured length of a tree may be measured as rotations of a metering wheel in one node and an algorithm in other node that calculates the center of mass uses centimeters.

Varying and different measuring units that the components in the system use should be supported seamlessly.



The nodes and devices of the machine control system may be acquired from multiple vendors which may originate from different countries. The nodes may use different units in their measurements. Usually the nodes use unscaled raw values or standard units in their measurements. Different vendors may use different measurement systems (metric system vs. standard units etc.) This makes adding new devices to the system challenging as the measurements and calculation parameters may be incompatible with each other. In addition, if a component providing vendor goes out of business or otherwise has to be changed during the life cycle of the software product, it has to be made sure that the replacement devices support the same units as the rest of the system.

Mixing up units, such as adding inches and centimeters together while calculating distances, may cause extremely dangerous situations and even loss of life, so the system should be designed so that possibility of this kind human error is minimized.

It may not be possible to change the measurement units from the producing device as the units are not configurable by vendor-made decision. However, different consumers of the produced data may require different format for the measured value and it is the responsibility of the consumer to use the correct units and resolutions.

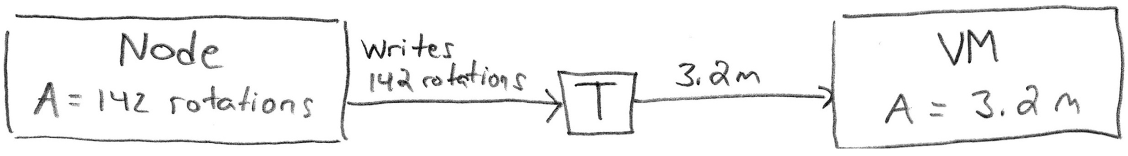
The messages that contain the measured data may contain additional information, such as same data in other units or a metadata field containing the unit that the data uses. However, messages sent by devices are usually not configurable. This means that if the device vendor has not enabled sending the same information in different units or using a metadata field, it is a huge task to implement this afterwards. The reasons why the vendors may omit this data may include the additional length to the message caused by this information and the low processing power in the measuring devices.

The machine operator might only be familiar with her own native measuring system. She may not be familiar with the units used by some suppliers in some parts of the systems. However, in order to ensure that the operator can efficiently operate the machine, the user interface should present only units that she is familiar with.

Therefore:

Add a converter service to the system. This service converts data to correct unit, e.g. from mph to km/h or from voltage in a sensor to temperature in Fahrenheit, when the data is requested.

* * *



If both SEPARATE REAL-TIME and VARIABLE MANAGER have been applied, there can be one special Variable Manager on the system, that collects all variable values to one place. It usually resides on the high-end node. As the low-end nodes have limited resources, they usually don't use physical quantities, but rather some raw sensor data. In this way, they do not need any special converting services for data that is for local use only. Thus the sending, receiving and processing the data is quicker. However, some high end applications may need different units than the core system raw data and thus, a translation service interface is needed. The most natural place to implement this translation interface is the variable manager on the high end node, if such exists. This interface can output any variable it knows in other units.

There are multiple strategies to resolve how the translation interface can know the correct units for a certain variable. The converter layer can know the correct unit for a variable either by a separate configuration or from a metadata field in the messages which contain the data. The metadata field requires more space from the message and may make the data payload smaller, but it can handle easily a situation where the producer node is replaced with another that uses different units. Using the metadata field, there is no need for reconfiguration as the message itself carries all the necessary information. It also encapsulates all variable-specific information in one node and no other node need to know how a variable is measured or how it is used in other nodes. However, usually this is not feasible as the low-end devices have vendor-spesific messaging schemes which are not easily changeable. The metadata can be used to be sure that the nodes do not inadvertently mix different units.

The other strategy to resolve the problem is some kind of configuration file that connects every variable to a certain unit. It is easier to add new devices that send only predefined messages where there are no room for metadata, but now someone must maintain the configuration files. There is a great risk of failure, if some variable is configured to use wrong units. This kind of misconfiguration may have catastrophic consequences. In a large system, there might be thousands of variables, so maintaining all required configuration data is a gargatuan task.

The converter interface can offer services for changing a unit of some variable to a other unit. This allows different algorithms in a certain node to use different resolutions in the calculations. In a more extreme case, an algorithm can use different measurement system than the other parts of the system if it only uses variables as its interface to the outside world.

* * *

All variables in the system can be in any units that the user or software modules may require.

It it easier to develop software as the developer does not need to know which units are in use in other parts of the system as she choose freely from any convenient unit system. It is also easier to test the system as the hardware-dependent raw values are used only locally in the nodes.

However, converting values requires some processing power and slows the system down as remote nodes must send the values to the high-end node for conversion and wait for the result. Therefore, all conversions take at least twice the time what sending one message does.

If the system is not properly designed, this pattern can not guarantee that all nodes always use the same units.

* * *

An autonomous fork lifter uses several different sensors to carry loads from a shelf in a warehouse to another shelf or to the loading area. These sensors include velocity sensors, limiting switches and so on. In addition, there are many valves and electric motors that must be controlled by sending them messages containing the proper amount of movement they have to carry out. All these sensors and actuators are acquired from different sub-contractors and may use different units and the scaling of the raw values vary from a device to device. The fork lifter has a long lifespan and it must be made sure that broken sensors and actuators are easily replaceable with new devices, that are possibly provided by a different vendor. This is achieved by using a converter service in the Variable Manager of the main node. It converts the received messages which contain the measured values into SI units. The main node software uses consistently SI units in all calculations. All sent control messages are then translated to correct units for the receiving actuator. In this way, the sensors and the actuators may use freely any units they want to as long as the converter is properly configured.

2.3 State Variable Guard

...you have a distributed machine CONTROL SYSTEM which consists of several independent nodes. The nodes gather data from the environment using sensors, perform computations from this data and the actions of the nodes usually produce state information. This has been remedied using VARIABLE MANAGER. However, the system may have components that are made by a third party, but their operations rely on the data produced by the the system. For example, a navigation software is made by a 3rd party, but it needs the GPS location information from the rest of the system. This 3rd party software should not be able to alter the location data or cause any other problems by accidentally altering the system state variables. Furthermore, no sensitive data should be accessed by the 3rd party software.

Access or modifications to the state variables should be limited to only trusted parties

It is sometimes necessary and beneficial to allow third party vendors to be able to implement their own software upon the machine control system. There can be a plethora of reasons why the machine control system provider should open up their platform to other companies too [5]. System openness may promote better software for the platform, as the control system vendor may focus on their core business. New innovations and ideas are more likely to happen if others may develop their own software using the provided platform. As the systems have long life cycles, some degree of openness may help the system to adapt to the unforeseen requirements and usages of the system. As the VARIABLE MANAGER acts as the normal interface to the system's data, the variables the 3rd party software needs should be readable for them.

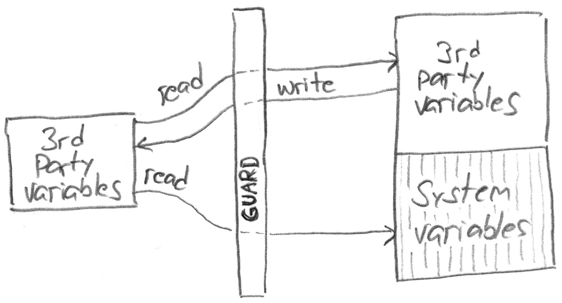
In addition to developing new features, even in-house development may benefit from some degree of openness. Testing the control system is easier when the testing platform can access the variables in the system. Remote access systems are usually developed as separate projects from the control system, but it needs a well-defined access to the control system variables. Thus, it resembles a trusted 3rd party application.

However, openness must not override safety and security. The 3rd party software can not be allowed to access sensitive information (personal information of the operator, business-critical data and such) and it may not alter the basic data in the machine control system as this might compromise overall system safety. Even if the 3rd party is trusted, it is easier to test the system safety conformance so that the 3rd party software can not even accidentally alter any safety related state variables.

Therefore:

Design a mechanism to guard the state variables, which checks if an application is allowed to read variable values or submit their own changes to system state information. The mechanism is the only component that can directly access Variable Manager.

* * *



When both SEPARATE REAL-TIME and VARIABLE MANAGER have been applied, the system can support one special Variable Manager, that collects all variable values to one place. It resides on the high end node, which also has more processing power than the low-end nodes. Therefore it is used to provide a platform for 3rd party software. If a 3rd party software wishes to read variables from the Variable Manager, its developers are given a special interface and documentation how to use it. The interface can be implemented for example by using a dynamic library file and a interface description, so the inner workings of the variable manager need not to be exposed to the 3rd party developers. An other way to implement this pattern is to add an additional node to the system which communicates only with the high-end node using a dedicated bus and has only a read-only access to all the data. In this way no malicious changes to the machine data are possible.

The Variable Guard interface is one-way access to the all core information and keeps track that no sensitive (personal or businesswise) information is allowed to be read. If the 3rd party developers wish to store their own data in the Variable Manager, they should be able have some own variables. This is by using the Variable Guard's interface to reserve new named variables. These variables have unlimited reading and writing rights for the 3rd party software. This helps the developers as they can use an uniform way to access data regardless of its origin. However, if the 3rd party software developers may freely allocate new variables, they may excessively tax the storage space on the high-end node and the access times may become longer as the Variable Manager must handle a large amount of variables. Therefore, some kind of limit to amount of 3rd party variables should be enforced in the Variable Guard interface.

This pattern is similar to the EXECUTION DOMAIN pattern [6] and PROTECTION PROXY [7].

* * *

The system can have 3rd party software without any fear of malicious or erroneous changes to the system variables. Also all sensitive data is hidden from the 3rd party software developers. The 3rd party software ecosystem may be built in order to generate more revenue. However, this pattern does not provide a way to protect data from a 3rd party software that should execute on a low-end node.

The 3rd party developers may use in an uniform way both their own data and the system core data.

Testing and remote access systems can have an access to control system data.

However, special care should be taken when designing the interface, so that the guard really protects the variables. Feelings of false security may rise when a guard is applied, but badly designed interface may leak information due to a buffer overflow.

* * *

A mining drill has to communicate with a fleet management software. The fleet management software resides on a server that is physically far apart from the drill machine. The communication is carried out using a GPRS modem. The fleet management only needs some production and location information from the drill and does not need minute details about the sensor values etc. on the system. Therefore, the it has been sensible to add an additional node with a GPRS modem to the mining drill bus. This node communicates with the high-end node using a variable guard system to access a limited amount of data in the system. This node then transmits this data with an integrated modem to the fleet management server. In this way, the mining drill cannot not send any sensitive data to the fleet management and the machine operator does not have access to the management data in the fleet management server.

3 Acknowledgements

We wish to especially thank our colleagues Dr. Johannes Koskinen and Ville Reijonen for their valuable help and input during the gathering process of these patterns. We also wish to thank all industrial partners for their willingness to provide the opportunities for the pattern mining. These companies include Areva T&D, John Deere Forestry, Kone, Metso Automation, Sandvik Mining and Construction, Creanex, Rocla, SKS Control and Tana. In addition, we would also wish to thank Nokia Foundation and Pirkanmaan rahasto for their scholarships and grants which have aided us in writing these patterns. Finally, huge thanks to our shepherd, Robert S. Hanmer, for his valuable suggestions and help in improving this paper.

References

1. Alexander, C.: The Timeless Way of Building. Oxford University Press, New York (1979)
2. CiA: CANopen Specification. CiA, NÃ$\frac{1}{4}$rnberg, Germany. http://www.can-cia.org/.
3. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. Wiley, Chichester, UK (2004)
4. ISO 11898: Road vehicles – Controller Area Network (CAN). ISO, Geneva, Switzerland. (2003) http://www.iso.org/.
5. Eklund, U., Bosch, J.: Introducing software ecosystems for mass-produced embedded systems. In: Lecture Notes in Business Information Processing, 1, Volume 114, Software Business, Part 2, Part 7. (2012) 248–254
6. Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns : Integrating Security and Systems Engineering (Wiley Software Patterns Series). John Wiley & Sons (mar 2006)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK (1996)