Patterns for Distributed Machine Control System Data Sharing

MARKO LEPPÄNEN, Tampere University of Technology
VELI-PEKKA ELORANTA, Tampere University of Technology

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. Data sharing is essential part of achieving distribution as collaboration between different parts of the system requires data exchange.

Categories and Subject Descriptors: **D.3.3 [Software Engineering]**: Software Architectures —*Patterns*

General Terms: Measurement, Design, Security

Additional Key Words and Phrases: Patterns, data exchange, variables

ACM Reference Format:

Leppänen, M. and Eloranta, V-P. 2012. Patterns for Distributed Machine Control System Data Sharing, PLoP'12, 15 pages.

1. INTRODUCTION

Nowadays, it is very common to have a software-based control system in heavy machinery and process automation systems. These machine control systems are often distributed as there are many factors that favor the system to be designed in divide-and-conquer type way. One of the most important factors is that different functional hardware parts of the machine are physically apart from each other and their corresponding control software is usually located in an embedded controller node near the controlled hardware. The nodes must communicate using a come communication media, usually a CAN bus (Controller Area Network), in order to collaborate with each other and perform their own functionalities as a part of a greater whole. It is also common that the nodes on the system bus have very wide variance in their computational capabilities. Usually a machine control system has several simple embedded controllers with relatively limited computational abilities. These nodes are also known as low-end nodes. In addition to these embedded controllers the system may contain at least 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 are 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, which were in the design phase or already on the market, by these companies were inspected during architectural evaluations. Whenever a pattern idea was recognized, the initial pattern drafts were written down. The 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.

.

Author's address: M. Leppänen, PO Box 527, FI-33101 Tampere, FINLAND; email: marko.leppanen@tut.fi; V-P.Eloranta, PO Box 527, FI-33101 Tampere, FINLAND; email: veli-pekka.eloranta@tut.fi

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. A preliminary version of this paper was presented in a writers' workshop at the 18th Conference on Pattern Languages of Programs(PLoP). PLoP'12, October 19-21, Tucson, Arizona, USA. Copyright 2012 is held by the author(s). ACM 978-1-4503-2786-2

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. This 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. So, an arrowhead points to the patterns, which refine the previous patterns 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 of all patterns mentioned in this paper

Pattern name	Solution
CONTROL SYSTEM	Implement control system software that controls the machine and has interfaces to communicate with other machines and systems.
ISOLATED FUNCTIONALITIES	Identify logically connected functionalities and compose these functionalities as manageable sized entities. Implement each of these entities as their own subsystem.
DEVIL MAY CARE	Define a time interval in which the system must reach a steady state. Being in steady state means that the system is yet again ready for normal operations. Before the steady state is reached, the system can generate erroneous alarms that can be neglected.
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.
LOCKER KEY	Store sending node's message data to a slot in a shared memory (locker) and send a message containing only the key to the locker. Receiver then uses the key to retrieve the message data from the locker.
HUMAN-MACHINE INTERFACE	Add a human-machine interface. It consists of ways of presenting information and controls to manipulate the machine. These typically are displays with GUIs, buzzers, joysticks and buttons etc. Separate the way of presenting the information from the controls of the machine with a separate HMI bus.
DIAGNOSTICS	Add a diagnostic component to the system. This component collects data, which enables the system to analyze if some subsystem starts to operate poorly or erroneously. Usually all data values have limits where they should reside and a deviation from this indicates a need for maintenance.
NOTIFICATIONS	Communicate noteworthy or alarming events and state changes in the system using notifications - a dedicated message type for notifications. Provide a way to create, handle and deliver notifications easily and enforce application developers to use these notifications.
REMOTE CONNECTION GATEWAY	Add a remote connection gateway that establishes the connection between the machine and remote party. Remote connection gateway transforms the used messaging scheme to suit the local and remote parties' needs.
VARIABLE MANAGER	For each node add a component, Variable Manager, that stores the system information as variables and provides interfaces for reading and writing them. When a variable is locally updated through the interface, the information is also sent to the bus. Whenever Variable manager notices updated information on the bus, it updates the corresponding variables.

VARIABLE GUARD	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.
VARIABLE VALUE TRANSLATOR	Add a converter service to the Variable Manager. The service includes interface to store and retrieve variables in suitable units regardless of the unit of the variable that the Variable Manager internally uses.
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	Create a mechanism in the system that saves system's state automatically as snapshots either periodically or before system configuration changes. The snapshot of the state creates a checkpoint where the system can return.
PARAMETRIZABLE 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 service that provides counting functionality for different purposes. Service should offer different kind of counters: e.g. non-resetting usage counter, maintenance counter, and resettable counter. The counters can count up or count down.
BLACKBOX	Add a Blackbox component to the system which records selected events occurring in the system for later inspection.
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 resetted 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-specific predetermined period has elapsed, cached value should not be used any more. A new value should be retrieved from the source node.

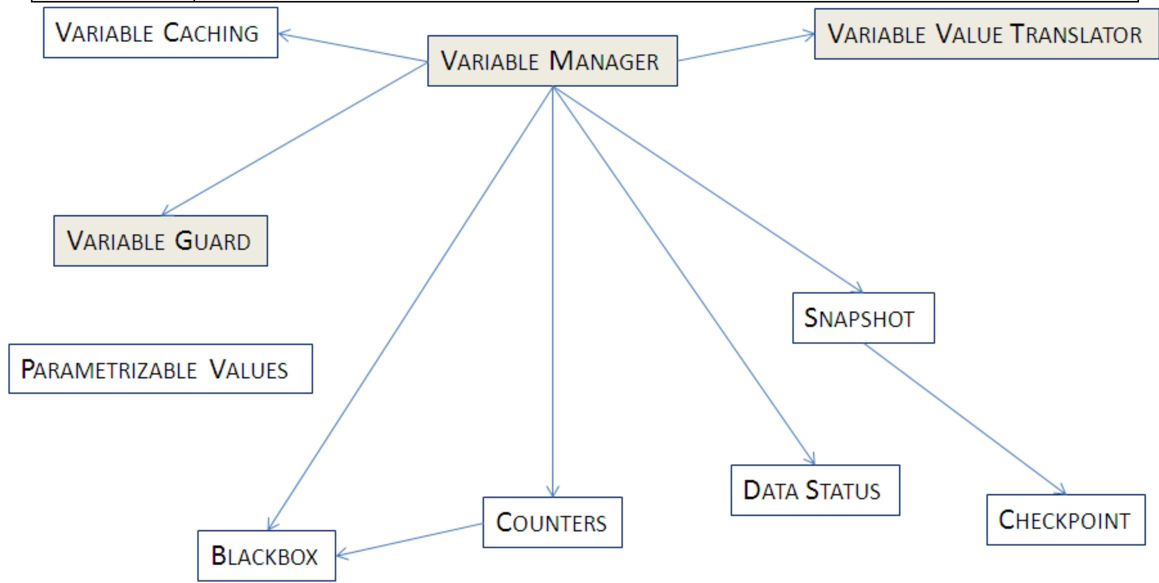


Fig. 1. The data management sublanguage. Patterns that are presented in this paper are highlighted.

Our pattern format closely follows the widely-known Alexandrian format (Alexander 1979). 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 "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 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

...there is a distributed machine CONTROL SYSTEM which consists of several independent nodes. Because ISOLATE FUNCTIONALITIES has been applied, the nodes have their own areas of responsibilities. The nodes connect to a MESSAGE BUS allowing them to collaborate 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 preserved 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 other nodes in the system for co-operation.

.

In order to facilitate efficient collaboration between nodes they need to share information. Usually the purpose of the application on the node is to control a part of the machine and thus, information exchange mechanism is not in the responsibility area of the application.

As the machine responds to the stimulus from outside environment or the operator commands, usually only the information about the current situation is relevant. This means that there is no need for long time storage of the data as history data is only needed for more specific services. But it also means that the information itself may have real-time requirements; a node should have a quick access to all required information. However, as the bus has a limited data transfer speed and communication initialization overhead, all communication between the nodes will have 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. 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. Also, if information sharing is implemented by query-and-answer based scheme, all queries also interrupt the node from its tasks as it has to wait for the answer. The data producing node also has to react to the received query. This tolls the processing capacity of the node and takes time from more urgent activities. The more nodes are present in the system, the more bus capacity they consume and more interruptions will occur as the number of the communicating parties grows.

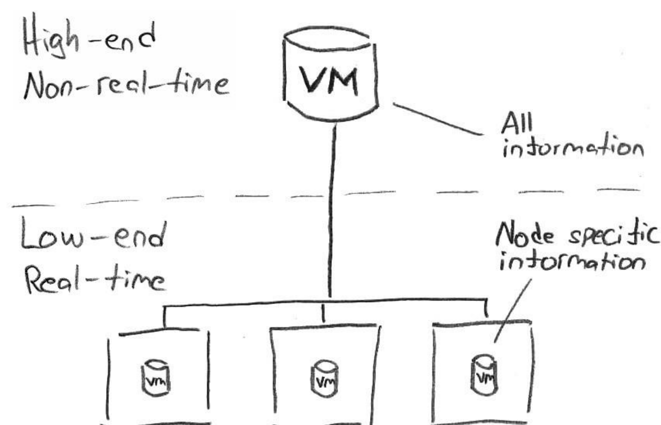
Still, it should be easy to add new consumers of information to the system. The core system may be extended by some optional features that have to co-operate with the rest of the system. Some of these options may be developed long after the core system has been released. This kind of evolution of the

system can cause changes in the needs for processing and sharing the information over time. This means that the developer of an application may not know during the development time that what other applications will use the data this application is producing. Thus, the producers and the consumers must be decoupled from each other. This must be done 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.

All this combined to the fact that there might be lots of information to be managed in a distributed control system mean that the developer of an application may needlessly have to concentrate on the messaging schemes. Different nuggets of information can have different real-time requirements and updating needs. The situation might be made worse by the fact that some nodes have several applications executing on them. These applications can consume and update same data. Poorly designed system may result in same node asking for a certain data several times.

Therefore:
For each node add a component, Variable Manager, that stores the system information as variables and provides interfaces for reading and writing them. When a variable is locally updated through the interface, the information is also sent to the bus. Whenever Variable manager notices updated information on the bus, it updates the corresponding variables.

First, all relevant information that the distributed system uses should be presented as variables. A variable in this context is a uniquely named 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. Some of the data is fundamentally analogous and continuous in nature, usually those variables which are measurements of physical phenomena, such as engine cooling liquid temperature or position of the joystick despite of the fact that they are presented as discrete samples. Some of the data is naturally discrete, like countable items (e.g. amount of felled trees), system state (e.g. language selections) or classifications (e.g. tree species).



Every node has its own set of relevant variables. Relevancy may result from the fact that the node itself produces this data, either by measuring it by sensors or applications residing on the node process it from several data sources. The relevancy might result also from the fact that this node needs the data for further

processing or to control the actuators connected to it. To manage these variables conveniently, a Variable Manager component should be added on every node. It stores all the relevant variables and provides an access to them through interfaces. The interface provides a set of methods to atomically write and read the variables by referencing them with their unique names. The Variable Manager interface guarantees that any data that is currently accessed is mutually excluded and not used by any other party. 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 produced changes in the local variables to the other interested parties, the Variable Manager communicates with other Variable Managers residing on the other nodes via the communication channel. Every Variable Manager listens to the communication channel in order to update all variables which are relevant to it. In order to achieve this, the variables should be mapped to messages on the communication channel, so that every shared variable has a message with which it can be transferred on the bus. The mapped variable on a message is usually referred as a signal. The mapping from variables to messages and vice versa is usually automatized by using an XML file, spreadsheet file or similar configuration tool where the developer can relatively easily design the variable space, so that the unique variable name can be used in the application without caring about the actual messaging scheme.

The communication between Variable Managers 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 one variable. As the these messages are broadcast to the bus, all the other variable managers can listen to the communication medium and receive all sent 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 need to know about its consumers. The data sharing becomes transparent to the application, as the Variable Management just updates the corresponding variable values whenever a node receives a relevant message and if the application updates data that is used somewhere else the Variable Manager sends the corresponding message automatically.

In addition to sharing variables globally, Variable Manager can be used on one node for intra-process communication. These variables are configured to be only local and therefore they have no message mapped to them. This means that they are not sent to the bus at any time, but applications residing on the same node may access them and exchange information with other applications using them. However, if some other application on other node will need these variables later on, the variable manager must be reconfigured to map these variables to messages, so that the Variable Manager on the other node may update its own local variables according to these messages. Thus, developing applications using Variable Manager is alleviated, because the applications can use the same method for storing their local data and there is no fundamental difference in using remote and local values.

Sending the update messages can be triggered on variable change, but also periodic updates can be used. If the measured variable is naturally analogous and continuous, it is advisable to sample the measurement with a fixed rate and send the change messages periodically. Now the recipient can decide if the change in the value is significant for it and noise-induced changes may be ignored by using an application-specific threshold. For example, a boom extension is controlled by joystick Y-axis position, which is sampled with an A/D - converter with five milliseconds sample rate. The position is then stored to the Variable Manager storage as a variable which is mapped to a message which is sent to other nodes cyclically every 5th millisecond. In this way, the boom controller can know if the operator is keeping the control joystick steady or has the position changed and act accordingly. If the information is naturally discrete and presents some sort of state changes, for example language selection, it is not advisable to send the data periodically. Sending messages repeatedly about the already known situation only uses the bus capacity in vain. For further details, see George Washington is Still Dead (Harrison et al. 2000)

It is the responsibility of the designer to decide the strategy 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. Still,

many of the variables need to be to broadcast only once when they are updated. As an example, in the case of CANopen bus (EN 50325-4:2002), the designer must make a selection mapping the variables either to a timer-driven and cyclically sent PDO frame or an event-based SDO frame.

If the node must react almost immediately to some system state change, it may be better not to use the Variable Manager 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. Thus, if the communication need is more event-based and it requires immediate action, NOTIFICATIONS is a more suitable option

When the system starts up, there can be lots of uninitialized variables in the Variable Manager as the system has not yet produced or measured any data. These uninitialized variables must be prevented from being used by adding some metadata; usually one bit will be enough, to the data, which will tell if the data is not yet initialized. This metadata must be accessible through the Variable Manager interface. In addition, the startup phase can cause several variables that are sent on change -basis to trigger their corresponding messages. To prevent the bus being overfilled with these messages, consider using some sort of time interval before triggering the message. The interval may be random or fixed, for further details see DEVIL MAY CARE.

The Variable Manager can also store a time stamp with every variable with other metadata provided by DATA STATUS. This can be 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 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.

All data is not suitable for storing into Variable Manager, as the mutual exclusion and interface overhead causes latency. Further, if the data is updated frequently in the Variable Manager, it may trigger excessive amount of messages. Thus, if the data is such that it needs fast access or its update rate is high, it might be better to store the data only to local memory space of the application. If this kind of data needs to be shared with other processes, LOCKER KEY might be used instead. If the data presents a counter that it is usually just incremented or decremented by a fixed amount, COUNTERS can be used. Using a counter service removes the need to read a variable from the manager, incrementing or decrementing it and writing it back to the Variable Manager by offering a service which does all this with one method call.

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. These services may include collecting data for DIAGNOSTICS in order to monitor the health of the system, a SNAPSHOT of all the data in the system for restoring the state in testing environment, REMOTE CONNECTION GATEWAY for exchanging data with applications residing off-board and VARIABLE VALUE TRANSLATOR to offer data in different unit systems. It also makes system debugging and testing easier as one node has a comprehensive set of all the variables in the system.

Similar patterns have been widely documented. Variable Manager is essentially a special version of CACHING (Kircher and Jain 2004) and can be seen as a version of Data Repository architecture style, as it stores all shared data in a common storage location. Also, CAN Object Dictionary is a simple version of this (EN 50325-4:2002). This pattern is similar to OBSERVER (Gamma et al. 1995), where all registered observers are updated automatically when a single datum entity changes. In Variable Manager, these changes are propagated globally.

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 consumer of the information does not have to know the source of this piece of information. The location-transparency makes it easy to add new sources of information, as the consumers are not interested in the producing party - only the data. However, as the variables are only updated periodically or triggered by changes, the data may be somewhat old when it is needed. The detection of outdated 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. This might lead into problems when the system is scaled up and the amount of stored information grows. Especially, if history data is needed for some service, for example DIAGNOSTICS, the amount of data might grow too large to be handled conveniently anymore. Then dumping of the history data periodically to a database is needed.

With Variable manager, 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 is 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 using it anymore.

It may be hard to design the correct updating and invalidating strategy for a variable. If a variable is updated too often, it causes 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. Yet, all data is not suitable for storing into Variable Manager. The suitability depends on the need of sharing and the requirements for the access rate. Also, Variable Manager data is usually not preserved over power cycles, so it is not suitable for saving data that is needed for long time. If more persistent storage for simple signals is needed, see COUNTERS instead. If the amount of data that should be stored persistently is large, mass storage and some database solution are advisable. These heavy-weight solutions are usually feasible only on high-end nodes of the system.

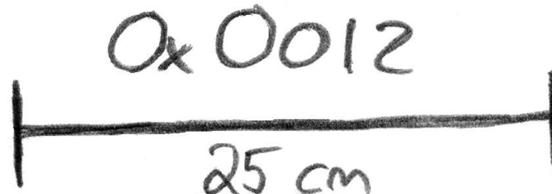
As all the data is shared, any node might update the variables and the other nodes will use them in their operations. Thus, any erroneous or malicious update of the shared data will propagate to whole system. If this is seen as a risk, building a mechanism like VARIABLE GUARD is advisable.

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. 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 long. However, the cabin PC can also use the variable about the feed state to show the operator information how the feed process is going as the user interface refresh rate does not need to be so high.

2.2 Variable Value Translator

…there is a distributed machine CONTROL SYSTEM which consists of several independent nodes which are ISOLATED FUNCTIONALITIES. The nodes gather data from the environment using sensors and perform computations from this data. In order to facilitate collaboration through information sharing with other nodes in the system, the data is presented as variables in VARIABLE MANAGER. Many of the variables represent measurements of physical quantities in various resolutions. However, the units used can vary from one node to other. For example, measured length of a tree may be measured as rotations of a metering wheel by the harvester head node, but an algorithm in the boom control unit that that calculates the needed force to support the tree from the center of mass uses centimeters in its internal algorithms. In order to handle these differences the nodes should be aware in what units a value is measured or calculated.

Different devices may use various measuring units internally, but they should also be able to utilize each other's information on the surrounding environment.



In order to have information about the surrounding physical environment, the control system uses sensors to measure physical quantities. The value of a physical quantity is a product of a numerical value and a unit, e.g. 2.7 meters. Thus, measurement results usually have two parts: magnitude and dimension, which is conveyed as units. However, some special quantities do not have a unit as they are dimensionless such as planar angles or strain, which is defined as ratio of deformed length to initial length; thus the division cancels out the unit of length. In software, the measured values are often compared with other measurements or predefined invariant values. An example of this could be comparing engine coolant temperature to predefined safety limits to ensure that the coolant will not boil. However, comparisons of magnitude can only be done if the units are the same.

In order to measure the physical quantities, the controllers use sensors. The real world causes some change in the sensor that can be interpreted as an analog signal, which is sampled with a known rate and is converted to fixed resolution digital number value. The low-end devices usually use raw values from the analog-to-digital conversion as a basis for their operation. These raw measurement values can be then scaled, for example, to metric units, but some vendors may use different measurement standards (metric system vs. US customary units etc.).

It may be lucrative to develop the machine control system in a distributed manner as ISOLATED FUNCTIONALITIES builds a system architecture where the different parts of the system have clear responsibilities and well-defined interfaces between them. Thus, the nodes and devices of the machine control system may be acquired from multiple vendors; either from subcontractors or as Common-of-the-Shelf (COTS) components. As global software development is now commonplace practice, these vendors may reside in different countries. Due to having different developers, who do not necessarily share the same common vision of the system or even the same culture, the devices may use varying units in their measurements. This happens more likely with COTS components, as the mass-produced device vendors do not usually customize their interfaces to meet single customer's needs.

If measurement values are communicated between the different controllers in the system, the both ends of communication must have the same conception of the values. The variance in units makes adding new devices to the system challenging as the measurements made by the devices and their calculations may be incomprehensible to each other when communicated via message bus. In addition, if a vendor providing a certain component goes out of business or otherwise the component has to be changed during the life

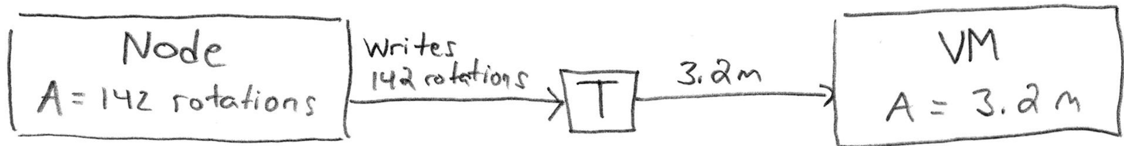
cycle of the software product, it has to be made sure that the replacement devices support the same units as the rest of the system.

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. However, it may not be possible to alter the measurement units from the data producing device as the units are not changeable because of lacking configuration feature or by a vendor-made decision. If different consumers of the produced data require different format for the measured value, it is left to the responsibility of the consumer to use the correct units and resolutions. In some high end products, the measured data can contain additional information, such as same data in other units or a metadata field containing the unit that the data uses. However, it is usual that measurements sent by devices are not configurable and if the device vendor has not implemented the feature of sending the same information in different units or using a metadata field, it is a huge task to implement this afterwards. In addition, these additional fields make the messages longer and the measuring devices may not have the required processing power to calculate the measurements in alternative units.

In addition to the development difficulties with different units, the machine operator must be taken into account if any of the measured values are shown directly in the HUMAN-MACHINE INTERFACE. The machine operator might only be familiar with her own native measuring system and may not be familiar with the units used by the developers of the parts of the system. In order to ensure that the operator can efficiently operate the machine, the user interface should present only units that she is familiar with. For example, if the operator is familiar with measuring speed as kilometers per hour, it may cause dangerous situations if the speedometer readings are shown as miles per hour.

Therefore:
Add a converter service to the Variable Manager. The service includes interface to store and retrieve variables in suitable units regardless of the unit of the variable that the Variable Manager internally uses.

As the low-end nodes have limited resources, their measurements of physical quantities are not usually presented in standard units in their internal operations, but they rather use raw digital 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 are quicker. However, some high end applications may need different units than the core system raw data and thus, a translation service interface is needed. If both SEPARATE REAL-TIME and VARIABLE MANAGER have been applied, there can be one special Variable Manager on the high-end node, which stores all system-wide variable values to one place. The most natural place to implement this translation service interface is the Variable Manager on the high-end node, if such exists. The Variable Manager itself can internally use any suitable units for storing data and the interface can be used to get value of any variable stored in the Variable Manager in any suitable unit for that measured quantity. Also, storing values can be implemented so, that the translation service can get the value and units for a certain and store the variable accordingly. Usually the interface method will have as parameters the name of the variable, which is accessed, the value of that variable and the unit. If the user is writing a variable, the Variable Manager can translate the value to the internal storage unit or store the variable value as it is and change the unit as requested. When reading variables, the user requests the variable name and declares the unit she wants to fetch the value. The interface then returns the value in correct units. If the unit is not suitable for the unit, so that the interface does not know how to translate the value, it will return an error code, exception or informs the user otherwise.



There are multiple strategies for the translation interface to know the conversions between units for a certain variable. One simple strategy to resolve the problem is some kind of configuration file that connects every variable to a certain unit and gives translation multipliers for other suitable units. However, as the system evolves and new units are introduced to the system, someone must maintain the configuration files. This may present a risk of failure, if some variable is not properly configured and uses 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 gargantuan task.

Another way of connecting variables to units to is to store data with a metadata field that contains the unit of the measurement. Then special functions for converting units to others can be used. As an additional benefit, the consumer of the data can safely use the data as it is easy to check in what units the data at hand is. In other words, the metadata can be used to be sure that the nodes do not inadvertently mix different units. If the data is communicated over the bus, the metadata field requires more space from the message and may make the data payload smaller, but the system can handle with less configuration a situation where the producer node is replaced with another that uses different units. Using the metadata field, there is no need for reconfiguration of variables as the variable 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-specific messaging schemes which are not easily changeable.

When designing the conversion interface and translation functions, some issues should be taken into consideration - especially, if the values are compared as ratios or some other higher mathematical operations are performed to them. First, when comparing physical quantities to each other, the rule of thumb is to have them in same units. However, some units are defined as derivations from others. These units can be seen as a specially named mnemonic, for example in International System of Units (SI system), one liter is defined as one cubic decimeter and one Newton is a shorthand for 1 kg * m / s^2. These special units can in most cases be treated and compared as their equivalents. However, this is not true for all cases. For example, fuel consumption may be defined as liters per 100 kilometers. As liters are cubic decimeters, this reduces the unit to square meters, which is unit of area, but it does not make sense to compare these two.

In addition to units, there are many different classes of measurements, where values cannot be easily compared with each other between classes. In simplest cases, the comparisons are done between ordinal measurements, e.g. rankings, which have no units. Although the ordinal measurements have no units, there is naturally no sense of comparing two ordinals of different types, like the amount of read errors in the bus to the amount of drilled holes. However, some types may be composite of each other and thus their units can be converted to be comparable. For example, if three work shifts add always up to one work day, their amounts can be easily compared. See COUNTERS for more information about ordinal measurements
.

Most of the measurements that are of interest in the engineering algorithms are in Stevens' measurement theory (Stevens 1946) classified as interval-based or ratio-based as the measured quantities are mostly continuous in nature. Interval-based measurements, such as time measured from an arbitrary starting epoch and temperature in Celsius define the order of the measured values and the size of interval between two sample measurement points. However, when used for arithmetic operations they do not have any sensible definition for multiplication and division. One cannot define a ratio between two of these measurements as it is not sensible to say "twice as hot" meaning that 20 degrees of Fahrenheit is 10 degrees of Fahrenheit multiplied by two.

However, many measurements can be expressed in a ratio-based way, where the measurement is a ratio between the unit magnitude and the measured quantity. In ratio scale measurement, there must exist a true zero value that is not arbitrary selected. The temperature in Kelvin degrees is an example of ratio-based measurement and has a zero point, where there is no thermal motion, thus the particles have zero kinetic energy. Most measurements in engineering are done in ratio scales. With these measurements ratios between two samples can be defined - it can be said that one measured tree, for example, is twice as long as some other measured tree and 100 Kelvin temperature is twice as hot as 50 K
.
In addition to having these classes, the magnitudes can be linear real numbers or repeating angular values, planar imaginary numbers and voluminous Euclidean vectors. Stevens' classification theory does not take in count for example angular magnitudes, which are confined into a certain range and thus, Nicholas R. Chrisman's article (Chrisman 1998) extends the levels of measurements to have ten classes (Nominal, Graded membership, Ordinal, Interval, Log-Interval, Extensive Ratio, Cyclical Ratio, Derived Ratio, Counts and Absolute) but many of these are not used in the practical applications. As a bottom line, conversions between classes require more attention than just changing the unit of measurement within a class.

All variables in the system can be presented as any suitable units that the user or software modules may require. It is 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 may 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 and the more sophisticated applications may use human-readable units. However, this pattern does not guarantee that the developer remembers to convert the units, so special care must be taken if the system does not use uniformly same units throughout the whole design.

Converting values requires processing power and if done remotely, it slows the system down as remote nodes must send the values to the high-end node for conversion and wait for the result.

An autonomous fork lifter carries loads from a shelf in a warehouse to another shelf or to the loading area and uses several sensors to guide itself to the correct loading spot. These sensors include velocity sensors, limiting switches and so on. In addition, there are many valves and electric motors that are controlled by sending them messages containing the proper amount of movement they have to carry out. All these sensors and actuators have been 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, which are possibly provided by a different vendor. This goal 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 control values in the sent messages are 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 Variable Guard

...there is a distributed machine CONTROL SYSTEM which consists of several independent nodes, which are ISOLATED FUNCTIONALITIES. 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 the VARIABLE MANAGER. However, the system has components that are made by a third party, but their operations rely on the data produced by the system. For example, navigation software is made by a 3rd party vendor due to the map licensing issues, but the GPS location information is produced in the rest of the system. Because this location data is used by the core system and fleet management, the

navigation software should not be able to meddle with the location data or cause any other problems by accidentally altering the any other system state variables. Furthermore, no sensitive data should be accessed by the 3rd party software.

In order to allow third party applications from untrusted parties, the system must have control what information each application can produce and consume.

It is sometimes necessary and beneficial for the machine control system provider 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 (Eklund and Bosch 2012). System openness may promote better software for the platform, as the control system vendor may focus on their core business and let the 3rd party to implement all optional software.

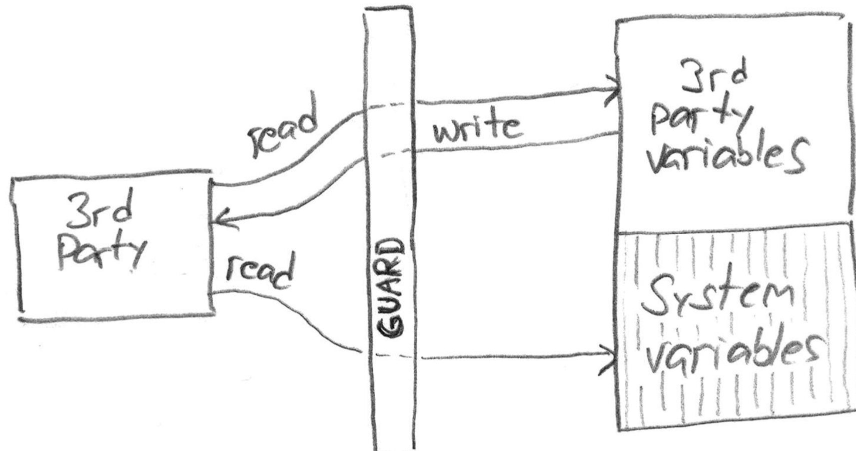
New innovations and ideas are more likely to happen if a larger crowd may develop their own software using the provided platform. As the machine control systems have long life cycles, platform openness may help the system to adapt to the unforeseen requirements and usages of the system. As the VARIABLE 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 related variables in the system. Remote access systems are usually developed as separate projects from the control system, but they need 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 cannot be allowed to access any sensitive information (personal information of the operator, business-critical data and so on) and it may not alter the control information in the core system as this might compromise overall system safety. Even if the 3rd party is trusted, it is easier to prove the core system safety conformance so that it is guaranteed that the 3rd party software cannot even accidentally alter any safety related state variables.

Therefore:
Design a mechanism to guard the variables that 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, which collects and stores global system information to one place. It resides on the high-end node, which usually has more processing power than the low-end nodes. Therefore, it can 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. This interface provides read-only access to the core system variables and may hide some variables completely. The interface can be implemented for example by using a dynamic library file and an interface description, so the inner workings of the variable manager need not to be exposed to the 3rd party developers. In this way, malicious changes to the machine data are not 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 to create their own variables. This can be implemented by using the Variable Guard's interface or configuration file to reserve new named variables. These variables have unlimited reading and writing rights for the 3rd party software, but they are not communicated to the low-end nodes. This makes the application development easier as the developers can use a 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. In more advanced solution, the access to all variables may be configured with a file or some other tool. In this way, the system designer has more control on what variables other parties may read. Usually there is no access by default to any variables and the system designer may customize an interface which has an exclusive access to certain variables.



This pattern is similar to the EXECUTION DOMAIN pattern (Schumacher et al. 2006) and PROTECTION PROXY (Buschmann et al. 1996).

After implementing the Variable Guard, it is possible to allow the 3rd party vendors to develop software on the control system. The 3rd party software can have controlled access to the relevant control system information, which is presented as Variable Manager variables. The variables which the 3rd party software needs can be shared without any fear of malicious or erroneous changes to the system information and all sensitive data is hidden from the 3rd party software developers.

As there is possibility to execute 3rd party software that can be acquired from almost any vendor, the 3rd party software ecosystem may be built in order to generate more revenue. Now the machine control system provider can focus on its core competence, designing machine control systems, and leave all

additional services and features to be provided by other parties. However, this pattern does not provide a way to protect data from 3rd party software that should execute on a low-end node. This kind of control software needs still be developed by the machine control system provider or some other trusted party.

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 all sensitive data is really protected. It is usually better to grant access with variable-by-variable basis, than to have an inclusive access. Feelings of false security may rise when a guard is applied, but badly designed interface may leak information.

A mining drill has to communicate with fleet management software. The fleet management software resides on a server that is located in the mine control room. The communication is carried out using a GPRS modem. The fleet management only needs production and location information from the drill and does not need minute details about the sensor values etc. on the system. Therefore, it has been sensible to add an interface for fleet management software to the machine data that only allows limited access for the fleet management to the control parameters of the machine. The fleet management data on the machine is stored into separate variables that are not communicated via bus. In this way, the mining drill cannot 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 especially wish to 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

Alexander, C. 1979. The Timeless Way of Building. Oxford University Press, New York

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. 1996. Pattern-Oriented Software Architecture, Volume 1: A System of Patterns. Wiley, Chichester, UK

Chrisman, Nicholas R. Rethinking Levels of Measurement for Cartography. 1998. Cartography and Geographic Information Science, Volume 25, Number 4, October 1998 , pp. 231-242(12)

EN 50325-4:2002 Industrial communications subsystem based on ISO 11898 (CAN) for controller-device interfaces - Part 4: CANopen

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Systems. Addison-Wesley. United States.

Harrison, N. B., Foote, B and Rohnert, H. 2000. Pattern Languages of Program Design 4. Addison-Wesley, United States

Kircher, M. and Jain, P. 2004. Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. Wiley, Chichester, UK

ISO 11898: Road vehicles – Controller Area Network (CAN). 2003. ISO, Geneva, Switzerland. http://www.iso.org/.

Eklund, U. and Bosch, J. 2012. Introducing software ecosystems for mass-produced embedded systems. In: Lecture Notes in Business Information Processing, 1, Volume 114, Software Business, Part 2, Part 7. 248–254

Schumacher, M., Fernandez-Buglioni, E., Hybertson, D., Buschmann, F. and Sommerlad, P. 2006. Security Patterns : Integrating Security and Systems Engineering (Wiley Software Patterns Series). John Wiley & Sons

Stevens, S. S. 1946. Science 103(2684), 677–680.