JUHA-MATTI VANHATUPA

VARMA: PATTERN-DRIVEN TOOL SUPPORT FOR XML-BASED VARIATION MANAGEMENT

Master of Science Thesis

Subject approved by Department Council
8th May, 2006
Examiner: Prof. Kai Koskimies (TUT)
Dr. Tech. Imed Hammouda (TUT)
Preface

The work for this thesis was done in 2006 in the Institute of Software systems in the INARI project, which is funded by TEKES (Finnish Funding Agency for Technology and Innovation) and several companies: Nokia Research Center, Nokia Networks, John Deere, TietoEnator and Plenware.

The author would like to thank his examiners Professor Kai Koskimies and Imed Hammouda. I also like to thank the INARI team for their comments and advices.

I also wish to thank my family, especially my brother Timo for their support.

Tampere March 3, 2007

___________________________
Juha-Matti Vanhatupa
Tullikamarinaukio 1 A 104
33100 TAMPERE
tel. 0400-565696
Abstract

Controlling a huge XML configuration is difficult and time-consuming. It is hard for the user to find important XML elements and XML elements of configuration may also have hidden dependencies, which are important on the target domain, but cannot be seen from file itself. One solution is to offer tool support.

The result of this work is VARMA, a prototype tool for a variation management. VARMA can be used to read XML schema and to create a new configuration based on the schema. It creates a graphical representation of a configuration file and guides the user in creation and controlling the XML configurations. VARMA editor was evaluated by creating a configuration file for a condition monitoring application of John Deere harvester.

XML configuration files of the condition monitoring program are huge and complicated. Their XML elements also have dependencies, which cannot be known without being familiar to the application domain. These dependencies are configured to the XML schema, which is given to VARMA. By using VARMA it is possible to control condition monitoring program configurations and to ensure their consistency and validness.
Tiivistelmä

Tiivistelmä

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan osasto
Ohjelmistotekniikka

VANHATUPA, JUHA-MATTI: VARMA: Pattern-driven tool support for XML-based variation management
Diplomityö: 53 sivua
Tarkastajat: Prof. Kai Koskimies ja TkT Imed Hammouda
Rahoittaja: INARI projekti
Maaliskuu 2007

Avainsanat: variaationhallinta, malli, konfiguraatio

Suuren XML konfiguraation hallinta on hankalaa ja aikaa vievää. Käyttäjän on vaikea löytää tärkeitä XML elementtejä ja XML elementeillä voi olla yhteyksiä, jotka ovat tärkeitä konfiguraation käytölle, mutta niitä ei voi nähdä XML tiedostosta itsestään. Yksi ratkaisu ongelmaan on käyttää apuna tähän suunniteltua ohjelmistoa.

Tämän työn tulos on VARMA, variaationhallintatyökalun prototyyppi. VARMAa voidaan käyttää konfiguraation XML scheman lukemiseen ja luomaan uusia schemaan perustuvia konfiguraatioita. VARMA luo graafisen esityksen konfiguraatiosta ja ohjaa käyttäjää konfiguraation rakentamisessa. VARMA editoria arvioitiin luomalla konfiguraatio John Deeren harvesterin kunnonvalvontaohjelmistolle.

Table of Contents

Preface .................................................................................................................................. ii
Abstract .................................................................................................................................. iii
Tiivistelmä ............................................................................................................................ iv
Table of Contents .................................................................................................................. v
1 Introduction ........................................................................................................................ 1
2 Variation management ....................................................................................................... 3
  2.1 Product Platform ......................................................................................................... 3
  2.2 Variation Points .......................................................................................................... 5
  2.3 Configuration Management ........................................................................................ 5
  2.4 XML in Program Configurations ................................................................................ 6
3 Patterns as a Specialization Mechanism ............................................................................ 8
  3.1 JavaFrames Patterns .................................................................................................... 9
  3.2 Tools: JavaFrames and its Extensions ...................................................................... 10
4 Harald Platform ............................................................................................................... 14
  4.1 Architecture .............................................................................................................. 15
  4.2 Generic Measurement Configuration ........................................................................ 15
    4.2.1 Message Conversions Block .............................................................................. 16
    4.2.2 State Conversions Block .................................................................................... 17
    4.2.3 Groups Block ..................................................................................................... 19
5 VARMA Editor ............................................................................................................... 23
  5.1 Overview ................................................................................................................... 23
  5.2 VARMA Role Types ................................................................................................ 24
    5.2.1 Component Role ................................................................................................ 25
    5.2.2 Attribute Role .................................................................................................... 25
  5.3 XML Roles ............................................................................................................... 25
  5.4 XML Schema ............................................................................................................ 25
    5.4.1 Setting parent components ................................................................................. 26
    5.4.2 Configuring Dependencies between Components ............................................. 26
    5.4.4 Description Attribute ......................................................................................... 28
    5.4.5 Example Schema ............................................................................................... 28
1 Introduction

The size and complexity of software systems is constantly increasing. The increasing size and complexity of developed software systems is a problem because companies do not have infinite number of workers or endless time to use in software development processes. This situation creates more need for reusing program code. A major addition to reuse approaches since 1990s is software product families. The idea of a product family is to use a product platform, which implements common structure and behavior of a product family. Product family members are created by configuring the product platform either by using configuration files or by modifying component collection of the product. Creating a valid configuration file is not always easy, especially if the configuration is large or the system is unfamiliar to the configuration creator. One solution is to offer tool support for configuration creation.

Extensible Markup Language (XML) [XML] can be a useful tool in program configuration, because it is programming language independent and human and machine-readable. When using XML configuration, configuration file structure is defined in XML schema file.

Patterns are a recent software engineering problem-solving technique [Ga et al. 05]. Patterns are a way to document software architecture and programming solutions. Those are used largely for describing software architecture and design. Patterns are a useful specialization mechanism [HaKo06] and they can be used to guide the developer through a configuration creation process. Specialization patterns are system-specific and describe a known solution to specialize a particular framework or other system, like a configuration.

This thesis aims at offering tool support for controlling XML configuration files and especially for creating general solution that can be used for different configuration domains. The created editor VARMA is a prototype pattern-based variation management tool. It can be used to read XML schema and using patterns as a mechanism to specialize configuration, to create new configuration files based on a schema. VARMA can be used to create a graphical presentation of created configuration. This presentation helps the user
to understand the structure of a configuration. VARMA also offers real-time support for testing the validity of the configuration.

The purpose of INARI project is to study the requirements of an architecture centric software development environment, and to develop prototype architecting tools as extensions of existing CASE tools and programming environments to satisfy those requirements. This thesis is part of the John Deere case study carried out by INARI project [Ha et al. 05]. VARMA editor was evaluated by creating a configuration file for a condition monitoring application of a John Deere harvester. The configuration file of the condition monitoring application is complex and its components have hidden dependencies. It was an excellent test subject for VARMA.

This thesis is structured as follows. Chapter two consists of theory of variation management. Chapter three describes using patterns as specialization mechanism and presents tools existing when the work began. Chapter four explains the structure of Harald harvester platform. Chapter five concentrates on VARMA schema configuration. Chapter six is implementation documentation. Chapter seven presents what the VARMA editor can do. Chapter eight is final words and notices.
2 Variation management

Software reuse can drastically decrease cost of software development and maintenance [Bo]. Software reuse also causes variability needs. Same software components can be part of various software systems if their behavior can be modified easily. Variability moves from mechanics and hardware to software. Software is an immaterial artifact and easier to modify than physical parts. Design decisions are delayed as long as economically feasible.

Size and complexity, small number of software, delayed designs decisions and variability demands are all leading towards a world, where software systems are large reusable units. Product families are results of this approach.

2.1 Product Platform

A product family is a group of programs sharing the same architecture and is designed for the same purpose in a certain domain. Product platform is a software system, which implements the common structure and behavior of a product family. Using product platform enables reusing the same code in all members of a product family. Koskimies and Mikkonen [KoMi05] enumerate benefits using product platforms:

- Better quality, because code has already been tested in many products.
- Faster program development, because most of the code is already done when a new product is developed.
- Easier product management, because the same kind of working methods can be used in all product family projects.
- Easier proceeding to new projects, because developing environments and tools are already familiar.
- Product standardizing, because many components are common to all products, e.g. user interface.
- Effective work, because once the expensive architecture design is done, it does not need to be repeated.
Product platform is also a way to create variation. It contains different types of components:

- Mandatory, component is included to every product.
- Optional, component can be included, but is not necessary.
- Alternative, one component of a certain collection has to be chosen.

This is shown in figure 2.1.

Also some components are only present in certain products.

![Figure 2.1. Product platform components.](image)

Product platforms also create problems. Architecture design is rarely perfect with the first product, because only the using of a concrete product reveals certain problems. Good design and testing in early stages can of course reduce these problems. Changing the architecture is difficult because modifications to the platform affect all products. Especially large changes in architecture can be very expensive. Using product platform generates some application performance loss to the products, because of high number of interfaces, inheritance, use of dynamic binding and call-back function calls.

Variation creation in product family has problems. [Ho et al 06] lists two core issues. The first is complexity of product family, high number of variation points, variants and
dependencies. This implies that the domain and application engineers have no overview of the variability. Maintaining the product family during domain engineering and selecting variants during application engineering becomes almost unmanageable by individuals. The second core issue is the large number of implicit properties (e.g. dependencies) of variation points and variants. These properties are undocumented and either unknown or only known to experts.

### 2.2 Variation Points

The variability in software product lines is made explicit through variation points. A variation point represents delayed design decision. Designing of a variation point requires several steps: separation of a stable and variant behavior, definition of an interface between these types of behavior, the design of a variant management mechanism and the implementation of one or more variants [Bo et al. 02]. Completing (binding) a variation point enables some functionality or variation of it.

Implementation of variation points can also be problematic because variability requirements do not map nicely onto programming languages. Code, which creates some type of variation can be scattered over multiple parts of the system and multiple features can be tangled within one piece of the system. Addition, it is impossible to know all the consequences of binding of a certain variation point. This operation is prone to human errors.

### 2.3 Configuration Management

Careful procedures are needed to manage the vast number of elements (source code modules, documentation, change request, etc.) that are being created over the lifetime of large software. This is called configuration management [VI93]. Besides the case of different program or component versions, different configurations come about if the same components can be assembled into a system in more than one way.
Configuration management is concerned with the management of all artifacts produced in the course of a software development project. It entails the following major activities:

- Configuration items must be identified and defined. A configuration item is a collection of elements that is treated as one unit for the purpose of configuration management. Examples of configuration items are: a source module, a requirement specification document, a user guide document.
- The release and change of these items must be controlled throughout the software life cycle. This means that orderly procedures must be established as to whom is authorized to change or release configuration items.
- The status of configuration items and change requests must be recorded and reported. For example, the status of a change request could be: proposed, approved, rejected or incorporated.

2.4 XML in Program Configurations

Extensible Markup Language (XML) is a simple, very flexible text format derived from Standard Generalized Markup Language (SGML, ISO 8879:1986) [SGML]. Originally designed to meet the challenges of large-scale electronic publishing, it supports Unicode, allowing almost any information in any written human language to be communicated. XML is also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere [XML]. One main advantage of XML is that it is simultaneously a human- and machine-readable form.

XML Schema is an XML-based alternative to Document Type Definition (DTD). The purpose of DTD is to define the legal building blocks of an XML document. [XML2]. XML schema is successor of DTD because of following reasons:

- XML Schemas are extensible to future additions
- XML Schemas are richer and more powerful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types
- XML Schemas support namespaces
Using XML schema also creates problems. The main problems in the use of XML schema stem from its complexity. The XML Schema language is also referred to as XML Schema Definition (XSD). The main problems are:

- XSD specification is very large, which makes it difficult to understand and implement.
- The XML-based syntax leads to verbosity in schema description, which makes XML schemas harder to read and write.
- Schema validation also makes extra work for regular XML parsing. This might be important in real-time systems.
- Some of its features are very limited or rarely useful.

XML can be a useful tool in program configuration, because of its language independency. Many web programs are configured by using XML. It is also used in program data transmission as message format.
3 Patterns as a Specialization Mechanism

In software engineering the concept of patterns is old and its instantiations are many, e.g. architectural patterns [Bu et al. 96], analysis patterns [Fo97] and performance patterns [SmWi97].

A well-known and popular type of patterns is design patterns [Ga et al. 05]. A design pattern is a solution for a particular object-oriented design problem in a particular context. A design pattern has four essential elements, unambiguous name, the problem definition, solution explaining needed elements and relationships between them, and consequences describing the results and trade-offs for applying the pattern. For example, composite design pattern tells how to treat primitive and composite objects exactly the same. The composite design pattern is shown in Figure 3.1.

![Figure 3.1. The Composite design pattern.](image)

Usually design patterns are represented as class diagrams, even Unified Modeling Language (UML) does not have own syntax for design patterns. This implies that the information about design pattern use in UML must be described in some informal way. For example, by using collaboration symbol or by drawing a circle around the classes related to the design pattern.
Usually there are also some disadvantages for using design pattern. In this case, the usage of Composite design pattern creates more complicated class hierarchy and increases the number of used classes. It requires implementing container related methods, such as addChild() and getChild(), in a leaf class, which is otherwise unnecessary. The idea of a design pattern is to solve a problem, not to cause them. Typical bad solutions are sometimes called anti-patterns [Ak96].

As opposite to design patterns, specialization patterns [HaKo06] [Ha et al. 01a] are system specific. They describe a known solution for specialization of a particular framework. Some of the classes appearing in a pattern description are actual framework classes. Others represent product specific classes that vary in different instances of specialization pattern.

3.1 JavaFrames Patterns

JavaFrames patterns are specialization patterns [Ha02]. JavaFrames is a prototype environment for task-oriented development. It allows precise specification of various kinds of reuse needs, like (design) patterns, coding conventions, and framework extension points (hot spots). These are modeled as role-based patterns forming the specialization model of a reusable system. Based on such a specification JavaFrames is able to provide task-based programming assistance to the specializer. For example, it is possible to specify the specialization interface of a framework. In this case the user is required to have detailed knowledge about the internal structure of the base classes, which means the framework is a white box framework. By using this specialization interface to override right methods in the framework it is possible to get framework-specific programming wizard for it [Ja06]. Even though JavaFrames is a prototype, it has long evolution and it has proven to be a useful tool in several case studies, which included real-world problems.

A role represents JavaFrames patterns basic building block. JavaFrames pattern consist of a set of roles. A role describes a concrete implementation element or other aspect of the solution. Each role may have a number of child roles. The structure of roles corresponds to the structure of elements in the solution. Instantiation of a pattern means that the pattern user creates, modifies or locates elements so that roles became fulfilled [Ha05a]. For
example, in case of UML, a class role is bound to a concrete class and cannot be bound to other UML elements like packages or operations. This type checking is done by user interface wizards.

Roles also have multiplicity. This way one role can represent a set of elements. A certain role may be bound more than once. The multiplicity of a role specifies the minimum and maximum number of concrete elements that can be bound to the role when the pattern is applied. Role cardinality can be [0..1], [0..n], [1..1], [1..n]. When a role is created its cardinality is [1..1] and can be changed later.

Patterns can extend other patterns. The extended pattern is more specified than the original one. The extending pattern inherits roles from the parent pattern. Extending a pattern is a common way to create binding tasks. When a second pattern inherits roles from parent pattern and the corresponding counterparts do not exist, tasks for generating the missing components are created.

A role is bound to a certain object when it is fulfilled. Roles with links that end to a missing object are said to be unresolved. The user can repair unresolved links if he wants to by creating new objects for missing link ends or by locating existing ones. For example, a class role is bound to a certain class and that class is deleted. The class role becomes unresolved. The user can correct this situation if he creates a new class for the class role or locates an existing class.

3.2 Tools: JavaFrames and its Extensions

JavaFrames is an environment providing tools to create and use precise pattern specifications. Currently JavaFrames is integrated into the Eclipse environment [HaKo06]. It is implemented as a set of Eclipse plugins. The original JavaFrames is for Java programming. The JavaFrames environment has been extended with support for UML, XML, and general text files. The core of the tool, which consists of tasking engine and support for Java files is called JavaFrames, and those extended support tools are called INARI.

First the UML part of INARI was integrated with external case-tool Rational Rose. The tool was using Eclipse and Rational Rose, connected with socket connection. However
combination of INARI and Rose was difficult for a person with little experience in modeling tools [La et al. 05]. Afterwards the INARI project has integrated UML plugin for IBM Rational Software Architect (RSA). RSA is an integrated design and development tool that leverages model driven development with UML for creating well-architected applications and services [Sw et al. 05]. RSA is basically Eclipse with a large set of plugins. Thus, there is no longer need for an external UML-tool. Figure 3.1 is shows a class diagram in RSA and a JavaFrames view.

XML plugin, the XML part of INARI, is extending JavaFrames for XML. The roles of XML plugin are XML related, linking to XML elements, XML attributes, XML nodes, etc.

![Figure 3.1. Rational Software Development Platform with INARI UMLPlugin.](image)

The user interface of JavaFrames is divided into two main views: **Architecture View** and **Pattern View**. The Architecture View shows patterns that can be organized hierarchically by using architecture nodes. In Figure 3.1 there are no architectural nodes present. The Pattern View shows the contents of a single pattern. It is divided into three panes: tasks,
properties and hierarchy. Each of them has a slightly different meaning depending on the working context. The same view can be used both for modifying patterns and for deploying them. In Figure 3.1 the pane tasks, which contains task list is brought to front.

An important part of JavaFrames tools is an automatically updated task list. The task list is guiding the developer to follow pattern specification. Each unbound role that can be bound in the current situation, taking into account dependencies to the other roles and their cardinalities, becomes a task [La et al. 05]. In the example, shown in figure 3.1, the task list has a task for the developer to provide second operation for the Interface1. The task is generated based on an extended pattern. Interface role in the extended pattern has two operations and the concrete element, which is bound into that interface, has only one. So the task for providing the other operation is generated.

There are several types of tasks:

- **Binding task.** Binding tasks are used to bind an element to a particular role. To perform this task the user must create or select a corresponding element. Typical binding task could be, for example, “Provide class C”.
- **Notification task.** Notification tasks are used to notify or advise the pattern user.
- **Grouping task.** Grouping tasks are used to organize other tasks and so they guide the pattern instantiation process.
- **Repair task.** Repairing tasks are used to repair violations in the pattern specification.

Tasks also have states. A task can be in one of the following states: done, pending, optional, or mandatory. A task is done if it has been performed. Performing a binding task means binding a particular element to a role, either by locating an existing element or by generating a new one. Performing a notification task means that the pattern user acknowledges the task. Performing a grouping task means the pattern user wants to start performing the subtasks. Performing a repair task means that the pattern user repairs the violation that triggered the task. These violations are against pattern specification, for example, in a certain framework class B must inherit class A, but it does not. A repairing task is created and when it is performed it corrects the relationship and after it class B is inheriting class A.
Pending task is waiting for its pre-conditions to fulfill and it is not shown on the task list. It will appear there when these conditions are filled. Role cannot be bound if it depends on another role, which has not yet been bound. Tasks extracted from dependent roles cannot be performed before the tasks to fulfill the other role have been fulfilled.

Optional and mandatory states depend on multiplicity settings set to a particular role. Mandatory tasks must be performed. Optional tasks are not necessary to perform. For example, if the cardinality of a certain role is 1..n, the first binding task is mandatory. The tasks for binding more elements to the role are optional. Mandatory tasks in the JavaFrames task list are marked using a red circular dot attached to the task title and optional tasks are marked with white hollow circular dot.
4 Harald Platform

Machines have replaced human labor in forestry. Harvesters are a type of heavy vehicles designed for cut-to-length logging operations for felling, delimming and bucking trees. A single harvester can do the work of several men. Harvesters are one of the most complicated moving vehicles in the world. Inside of a harvester there are several communicating electronic components, which consist of hundreds of thousands lines of embedded programs. These machines are very expensive and their productivity is straightly related to their operation time. Every breakdown decreases operation time of the harvester and decreases its productivity. Continuous condition monitoring of the harvester is the best way to prevent unexpected breakdowns and so to reduce the time harvester spends in repairs [Ko01]. As an example, Timberjack 770 harvester is shown in Figure 4.1.

![Timberjack 770 harvester](image)

**Figure 4.1. 770 Timberjack harvester.**

The Harald platform is a condition monitoring program framework. The monitoring program gets its information from the Controller Area Network (CAN) bus. There are multiple sensors and other devices connected to CAN bus. The framework also enables communication between other programs in the Harald platform [Ko05].
4.1 Architecture

Harald platform uses XML-based messages for internal communication. The main component in Harald platform is the core component. All other components see only this core component when communicating.

All components are configured using XML-based configuration files. A component reads certain values specified in the XML file. This work concentrates on specifying the Generic measurement configurations because it is the most difficult component to configure in the Harald platform.

4.2 Generic Measurement Configuration

Each component in Harald platform has its own configuration file. Generic measurement configuration is a configuration file for generic measurement service component. Generic measurement service is the component of Harald platform that can be used to define measurements. Measurements record data of internal messages of the Harald platform. The generic measurement service can, for example, draw a diagram about the use of a certain component. Sample XML blocks and tables in this section are from Generic Measurement Service [Ko06].

Generic measurement configuration can be written into a single file or it can consist of several files using include-keywords. The structure of the configuration remains unchanged. Using different files for each part of the configuration can make it more understandable. This is especially important if any tool support like VARMA is not used.

The top level of configuration structure is shown in listing 4.1. The most important part of the generic measurement configuration is inside the <parameters>-block. It is the changing part, which contains information about the measurements. Configuration inside <parameters>-block has three main blocks. These blocks are message conversions, state conversions and measurement groups.
Listing 4.1. Top level of configuration.

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<Settings xmlns:hs="http://harald.cs.tut.fi/schema">
  <Subsystem name="GenericMeasurementService"
    priority="LOW"
    hide="TRUE"
    sync="TRUE"
    visibility="PRIVATE"
    guid="{0B33764F-6215-469D-92B3-AF75ACEC6E56}" >
    <Parameters>
      <!--Message conversions, state conversions and measurement groups -- >
    </Parameters>
  </Subsystem>
</Settings>
```

4.2.1 Message Conversions Block

Message conversions are used to transform a CAN message into an internal message that can be processed by various components of a harvester.

4.2.1.1 Message Conversion

Normal message conversion is the simplest block. It is an XML element, which has only four mandatory elements inside it.

4.2.1.2 Binary Message Conversion

Binary message conversion is a collection of eight required and two optional attribute roles. The most important element in binary message conversion is `<TargetDataName>`-block, which has an explicit dependency to attribute `SourceDataName` of `StateConfiguration`. This dependency can not be seen from the XML. An example of a message conversion configuration is shown in Listing 4.2. Descriptions for elements of binary message conversion are shown in Table 4.1.

Listing 4.2.BinaryMessageConversion.

```xml
<BinaryMessageConversion>
  <CanBusID>TMC</CanBusID>
  <CanMsgID>136</CanMsgID>
  <TargetDataName>Engine.RPM</TargetDataName>
  <Signed>FALSE</Signed>
  <ByteCount>2</ByteCount>
  <StartByte>0</StartByte>
  <BitOrder>RL</BitOrder>
  <ByteOrder>LR</ByteOrder>
  <Multiplier>1</Multiplier>
  <Addition>0</Addition>
</BinaryMessageConversion>
```
Table 4.1. Descriptions for BinaryMessageConversion fields.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>CanBusID</td>
<td><em>Incoming data.</em> Identifying the used CAN-channel.</td>
<td>Yes</td>
</tr>
<tr>
<td>CanMsgID</td>
<td><em>Incoming data.</em> Identifying the message on the chosen channel that this conversion is going to handle.</td>
<td>Yes</td>
</tr>
<tr>
<td>TargetDataName</td>
<td><em>Outgoing data.</em> This is the name of the message action produced by the message conversion.</td>
<td>Yes</td>
</tr>
<tr>
<td>Signed</td>
<td>Whether the CAN-message is a signed byte or not.</td>
<td>Yes</td>
</tr>
<tr>
<td>ByteCount</td>
<td>The amount of bytes to be read from the CAN message in this conversion.</td>
<td>Yes</td>
</tr>
<tr>
<td>StartByte</td>
<td>The number of the byte in the CAN message this conversion will start from.</td>
<td>Yes</td>
</tr>
<tr>
<td>BitOrder</td>
<td>Whether the bits in one byte of the CAN-message are in a right-to-left (RL) or a left-to-right (LR) order.</td>
<td>Yes</td>
</tr>
<tr>
<td>ByteOrder</td>
<td>Whether the bytes in the CAN-message are in a left-to-right (LR) or a right-to-left (RL) order.</td>
<td>Yes</td>
</tr>
<tr>
<td>Multiplier</td>
<td>A scaling number that can be used to multiply the converted value.</td>
<td>No</td>
</tr>
<tr>
<td>Addition</td>
<td>A scaling number that can be used to add to the converted value.</td>
<td>No</td>
</tr>
</tbody>
</table>

4.2.2 State Conversions Block

State configurations and state combinations are more advanced tools to construct measurements. Those can be configured with rules and logical operations.

4.2.2.1 State Configuration

A state configuration tracks the value of an event and produces another event when the rule given in the configuration is crossed. Likewise every time the rule is crossed over to the deactive state, the configuration produces another event. These events can be used for constructing measurements or state combinations, which again can be used to build measurements. An example of state configuration is shown in Listing 4.3.
Listing 4.3. StateConfiguration.

```xml
<StateConfiguration>
  <SourceDataName>Engine.RPM</SourceDataName>
  <ActivationDataName>Engine.On</ActivationDataName>
  <DeActivationDataName>Engine.Off</DeActivationDataName>
  <Continous>TRUE</Continous>
  <Debug>FALSE</Debug>
  <Rule>
    <Type>Numeric</Type>
    <Operator>MoreThan</Operator>
    <Value>1</Value>
  </Rule>
</StateConfiguration>
```

Rule-block contains information about type of the value of tracked event. In this example, the type is numeric, and the rule is *MoreThan* and 1 is the threshold value, which is compared against *SourceData* to determine possible state changes. Descriptions for elements of state configuration are shown in Table 4.2.

Table 4.2. Descriptions for StateConfiguration fields.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>SourceDataName</td>
<td><em>Incoming data.</em> Identifies the event, which value this state configuration is tracking.</td>
<td>Yes</td>
</tr>
<tr>
<td>ActivationDataName</td>
<td><em>Outgoing data.</em> The name of the message action produced when the state configuration enters active-state.</td>
<td>Yes</td>
</tr>
<tr>
<td>DeActivationDataName</td>
<td><em>Outgoing data.</em> The name of the message action produced when the state configuration enters deactive-state.</td>
<td>Yes</td>
</tr>
<tr>
<td>Continuous</td>
<td>Determines how actively the configuration produces messages. If set to TRUE, every time the source action is received the configuration produces the action corresponding to its state. If set to FALSE, the configuration produces messages only when its state changes. Default is FALSE.</td>
<td>No</td>
</tr>
<tr>
<td>Debug</td>
<td>A debug setting. If set to TRUE various debug information is displayed in the console.</td>
<td>No</td>
</tr>
<tr>
<td>Rule</td>
<td>The rules that define the behavior of this configuration.</td>
<td>Yes</td>
</tr>
<tr>
<td>Rule – Type</td>
<td>Type of the value of the tracked event. Choices are “Numeric” and “String”.</td>
<td>Yes</td>
</tr>
<tr>
<td>Rule – Operator</td>
<td>The mathematical operation used when the value of the source data is evaluated</td>
<td>Yes</td>
</tr>
</tbody>
</table>
for determining the state of this configuration. Choices are “LessThan”, “Equal”, “NotEqual” or “MoreThan”.

| Rule – Value                              | The threshold value that the value of the source event is compared against, using the chosen operator. | Yes |

4.2.2.2 State Combination

State combinations are the most advanced tool in constructing measurements. They can track any number of events and apply a number of logical operations to these events. The state combination produces an activation and deactivation event much like a state configuration, but condition of triggering is defined by state configurations instead of the numeric value of message. For example, a state combination may produce a new message when both Engine.On and Engine.Off messages have been received.

4.2.3 Groups Block

All measurements must be part of a measurement group. The purpose of a measurement group is to collect the same kind measurements into one entity.

4.2.3.1 Groups

A group has five required attributes, which, for example, define the database structure. Each group can hold any number of measurements. An example of configuring a measurement group is shown in Listing 4.4. Descriptions for elements of group are shown in Table 4.3.


```xml
<Group comment="Engine related measurements">
  <ResetOnRetrieve>TRUE</ResetOnRetrieve>
  <GroupName>RPMTimes</GroupName>
  <GroupPath>Engine</GroupPath>
  <ProducedDataEvent>DB.INSERT</ProducedDataEvent>
  <TriggeringDataEvent>TIMER.TEST</TriggeringDataEvent>
  ** The Measurements **
  <Measurement type="...
```

(next section)...

```xml
</Group>```
Table 4.3. Descriptions for group fields.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResetOnRetrieve</td>
<td>Determines whether measurements of this group reset their value when the group receives the triggering data event.</td>
<td>Yes</td>
</tr>
<tr>
<td>GroupName</td>
<td>The name of the measurement group.</td>
<td>Yes</td>
</tr>
<tr>
<td>GroupPath</td>
<td>The name of the XML block that this group will create to the database. Each block is stored in a folder of the same name under the database root. Multiple groups can share a GroupPath, in which case both the groups' contents are written into the same block.</td>
<td>Yes</td>
</tr>
<tr>
<td>ProducedDataEvent</td>
<td>The name of the event produced when the group triggers. DB.INSERT is the event used to write the data into the database.</td>
<td>Yes</td>
</tr>
<tr>
<td>TriggeringDataEvent</td>
<td>The name of the event that triggers the group. This can be any message, but mostly a timer event is used to trigger the groups.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

4.2.3.2 Measurements

Configuration of a measurement is divided into two blocks. The common-block includes all the settings, which are used in every measurement. Specific-block includes settings specific to particular measurement. The type of the measurement is defined in the opening <Measurement>-tag using the type-attribute.

Common block has one required attribute, the name of the measurement, and four optional attributes. There is no limit on the number of activation/deactivation events. However, if an ActivationDataName element is not found, the measurement is active by default. On the other hand, if an ActivationDataName element is found, but it is empty, the measurement will never start (assuming there are no other activation elements). Contents of the specific block vary depending on what the purpose of measurement is. Descriptions for elements of measurement common block are shown in Table 4.4.
Table 4.4. Descriptions for common block of measurement fields.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>The name of the measurement.</td>
<td>Yes</td>
</tr>
<tr>
<td>Unit</td>
<td>The measurement unit.</td>
<td>No</td>
</tr>
<tr>
<td>Debug</td>
<td>The status of debug mode. In debug mode a measurement prints various information in the console.</td>
<td>No</td>
</tr>
<tr>
<td>ActivationDataName</td>
<td>An event that activates the measurement.</td>
<td>No*</td>
</tr>
<tr>
<td>DeActivationDataName</td>
<td>An event that deactivates the measurement.</td>
<td>No*</td>
</tr>
</tbody>
</table>

*: There is no limit to the number of activation/deactivation events. However, if an ActivationDataName element is not found, the measurement is active by default. On the other hand, if an ActivationDataName element is found, but it is empty, the measurement will never start (assuming there are no other activation elements).

Matrix.CumulativeTime measurement builds a matrix where the X-axis represents the values of one event and the Y-axis represents the values of another event. The split limits for the axes can be defined by the user. The information stored in the matrix is the cumulative time spent at a certain x-value / y-value pair. Listing 4.4 is shows an example of configuring a measurement type Matrix.CumulativeTime and after it in table 4.5 descriptions for specific part of measurement Matrix.CumulativeTime.


```xml
<Measurement type="Matrix.CumulativeTime">
  <Common>
    <Name>RPMAndLoadTimes</Name>
    <ActivationDataName></ActivationDataName>
    <DeActivationDataName></DeActivationDataName>
    <Debug>FALSE</Debug>
  </Common>
  <Specific>
    <YAxisDataName>Engine.RPM</YAxisDataName>
    <XAxisDataName>Engine.Load</XAxisDataName>
    <YSplitLimits>800:850:900:950:1000:1050</YSplitLimits>
    <XSplitLimits>0:5:10:15:20:25</XSplitLimits>
  </Specific>
</Measurement>
```
Table 4.5. Descriptions for specific block of measurement Matrix.Cumulative.Time fields.

<table>
<thead>
<tr>
<th>Element name</th>
<th>Element description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>XAxisDataName</td>
<td>The name of the event to be traced on the X-axis.</td>
<td>Yes</td>
</tr>
<tr>
<td>YAxisDataName</td>
<td>The name of the event to be traced on the Y-axis.</td>
<td>Yes</td>
</tr>
<tr>
<td>xSplitLimits</td>
<td>A list of split limits for the X-axis.</td>
<td>Yes</td>
</tr>
<tr>
<td>ySplitLimits</td>
<td>A list of split limits for the Y-axis.</td>
<td>Yes</td>
</tr>
</tbody>
</table>

NOTE: The split limits are given separated by a colon. For example 0:5:10.

In Harald platform there are dozens of different measurements, whose XML elements inside the specific block differ. For the purposes of this work the placement of different XML elements is important, however the meaning of XML elements is not.
5 VARMA Editor

VARMA editor is prototype pattern-based variation management tool. The editor has two kinds of users. First, an expert in the configuration domain writes the schema used in VARMA. Then anyone can use VARMA to read the schema and to create XML configurations using it without knowledge of configuration domain.

5.1 Overview

VARMA can be used to read XML schema and to create JavaFrames patterns based on the read schema. The schema defines what kind of components can be used in configuration files and how components depend on each other. When a schema is read the general configuration pattern is created. This pattern is created from the read schema, which defines what kind of elements the configuration can contain. The process is shown in Figure 5.1.

![Figure 5.1. Process to XML.](image)

This tool helps the user to understand and create configuration files. It can be used to read XML schemas of different kind of configurations. Currently the prototype does not support XML attributes. Support for XML attributes will be added in further development of the editor. VARMA is built on top of JavaFrames.

When the user starts creating a configuration, two more patterns are created. The second pattern contains pre bound XML elements, which can not be removed by the user. These elements are defined in the configuration file of VARMA: output.txt. The third pattern is extending the other two patterns and it is representing a configuration instance. Configuration instance pattern is empty when it is created. The user can use this pattern instance to create his configuration. The VARMA overview is shown in the Figure 5.2.
5.2 VARMA Role Types

Two new role types were created for VARMA: a component role and an attribute role. These role types are very abstract and can be used in different configuration domains. A full description of implementing new role types in JavaFrames is discussed in [Ha05a].
5.2.1 Component Role

Components may contain other components or attributes, or both. For example, a group always contains five attributes and/or any number of measurements component. In theory, there is no limit how deep this hierarchy can be, but because the editor is representing bound elements in a diagram, needed space can become a problem.

In the schema, setting the value of an XML attribute `base` to true tells to the schema-parser that this element is a highest level component. Highest level components are created to the pattern root. XML elements of type string are modeled with an attribute role and complex-types are modeled with component roles.

5.2.2 Attribute Role

Attribute role is modeling a single element, which has a name and a value. Attribute role is always a child of a component role. Attribute name and value are both string types without length limitation.

5.3 XML Roles

XML roles are role types, which are bound to artifacts in XML. Those artifacts are concrete elements in the current domain, XML, XML elements, XML attributes, etc [Ha05c]. VARMA is using XML roles for XML creation and XML plugin for XML parsing.

Component roles and attribute roles of VARMA are linked to the XML element roles using JavaFrames dependencies. Those XML elements contain XPath expressions to the real XML in the Document Object Model (DOM) tree and using those XPath expressions the corresponding XML blocks can be found and edited.

5.4 XML Schema

Possible elements of the XML configuration file are defined in the particular XML schema. The user uses tasks to generate new elements. New elements are always bound to certain roles, depending on the element type defined in the XML schema, to a component role or an attribute role. Element, whose type is a complex type and defined later in the schema is a component and element with type “`xsd:string`” is an attribute.
In addition to the normal schema definition, the XML parser of VARMA also understands few other attributes. Base attribute defines component to be a parent component. Attributes delement and dattribute are used to configure dependencies between components. The user adds these attributes to the schema before using VARMA to read the schema. The user must set some components to parent components when using VARMA, because when a new configuration is created, VARMA creates generation tasks only for parent components. The user does not need to configure any dependencies to the schema if there are no relationships between components.

5.4.1 Setting parent components

Attribute base is a Boolean value, which defines that this component is a parent component. Listing 5.1 shows a line of XML schema defining the base component BinaryMessageConversion.


```xml
<xsd:element name="BinaryMessageConversion" type="BinaryMessageConversionType" minOccurs="0" maxOccurs="unbound" base="true"/>
```

5.4.2 Configuring Dependencies between Components

Attributes delement and dattribute are used in dependency configuring. Those are always used together. The dependencies are always from attribute to attribute. Attribute delement specifies the type of the parent component that is needed to make this component useful and dattribute specifies the attribute.

Both delement and dattribute attributes are numbered because one attribute can depend on several targets. Even if there is only one demanded element and dependent attribute, configured XML attributes are delement1 and dattribute1. The number of delement and dattributes must be equal and if several attributes are needed then same element must be in several delements. Because there is no way to know in which order the user is generating new components, it is possible to configure dependencies to both ends. For example, in a situation where attribute at component A needs attribute from component B (and vice versa), in both A’s and B’s the needed components and attributes can be read from A’s and B’s schema. This kind of configuring ensures tasks for missing
components and attributes are created correctly. An example of an attribute TargetDataName, which needs other attribute, is shown in Listing 5.2.

**Listing 5.2. attribute TargetDataName needing attribute SourceDataName.**

```xml
<xsd:element name="TargetDataName" type="xsd:string"
delement1="StateConfiguration" dattribute1="SourceDataName"/>
```

Sometimes an attribute is needed from several other attributes. At this situation when second attribute is created also an instance of note role is created. Task description of note role describes what element and attribute are needed several times.

If the user wants to get rid of a note role, e.g. he does not need the information anymore, he can delete the note role from the extended pattern and the note role task is destroyed.

### 5.4.3 Component Cardinalities

XML schema attributes modify patterns created to the editor. If minOccurs attribute is zero, the element appears optional in patterns, in other words it gets cardinality 0... If maxOccurs is specified to unbound, the element gets cardinality ..n. For example in Harald specification all these five base elements have minOccurs 0 and maxOccurs unbound, so in patterns they appear optional and have cardinality 0..n. If nothing is specified, element has no attributes which affect to cardinality. Component appears as a mandatory task and has cardinality 1..1.

Sometimes only one of certain elements can appear in the XML file. XML schema is using `<xs:choice>` definition. This kind of situation is handled with specific JavaFrames role, XOR-role and only one of created tasks can be performed [Ha05b]. Use of XOR-role does not appear in produced configuration XML or configuration diagram in any way.

An example of using XOR-role is shown in Figures 5.4 and 5.5. One logical operation must be between any number of term components. The first figure shows patterns before the user has selected the logical operation and the second figure shows those after he has selected this logical operation and created three term components.
5.4.4 Description Attribute

Attribute description can be used to document component specific information. It is a string, which is used as a task description in VARMA. The task description is shown in Tasks pane of Pattern view. An example of description attribute is shown in Listing 5.3.

Listing 5.3. Description attribute.

```xml
<xsd:element name="Group" type="GroupType" minOccurs="0" maxOccurs="unbound" base="true" description="Group holding measurements"/>
```

5.4.5 Example Schema

In Listing 5.4 there is a small example schema containing producer and consumer components. If there is not a configuration to the producer then the consumer is useless, a dependency is configured between them. The consumer component must know the address of producing component. VARMA can read the schema and create configuration consisting producers and consumers.
In listing 5.5 there is an example of XML configuration created with the example schema using VARMA. Every time a consumer is created, a mandatory task for creating a corresponding producer, which has the address, is created. In this created configuration there are two Producer components with addresses port 1 and port 5 and one consumer component. In this example the Consumer component is receiving power from first ProducerComponent in port 1.
Listing 5.5. The produced XML.

```xml
<ConsumersAndProducers>
  <Producer>
    <Quality>
      A </Quality>
    <ProducingPower>
      Huge </ProducingPower>
    <Comment>
      Producer component number one </Comment>
    <Address>
      1 </Address>
  </Producer>
  <Producer>
    <Quality>
      B </Quality>
    <ProducingPower>
      Medium </ProducingPower>
    <Comment>
      Producer component number two </Comment>
    <Address>
      5 </Address>
  </Producer>
  <Consumer>
    <ProducingPower>
      light </ProducingPower>
    <SubProductA>
      <PowerOne>
        Light power type one </PowerOne>
    </SubProductA>
    <Comment>
      Configuration for greedy consumer component </Comment>
    <ProducerAddress>
      1 </ProducerAddress>
  </Consumer>
</ConsumersAndProducers>
```

The configuration file output.txt used with this schema is shown in listing 5.6. The output.txt file structure is described later in chapter 6.

Listing 5.6. output.txt.

```
#
<ConsumersAndProducers
Con
Pro
-"
6 VARMA Architecture Design

VARMA is implemented as Eclipse plugins. It is also using several other plugins, JavaFrames for task creation, XML plugin to create XML roles, GEF and draw2d for the graphical side.

6.1 Platform

Eclipse has plugin architecture. The basic mechanism of extensibility in Eclipse allows new plugins to add new processing elements to existing plugins. Extension points are places where other tools (plugins) can contribute functionality. Everything is written as a plugin, except the platform base. Eclipse provides the run time engine that starts the platform base and dynamically discovers plugins. Each plugin is a component, which describes itself to the system using manifest (plugin.xml) file. Eclipse maintains registry of installed plugins and this registry is updated at each start-up [Ec06a].

JavaFrames itself has been integrated into Eclipse as a set of Eclipse plugins. Also the editor is introduced to the system as a new plugin. It is extending Eclipses editor extension point (org.eclipse.ui.editor). To work this editor needs also role plugin, with initializes role types of VARMA into patterns. This role plugin provides its wizard-based user interface to the system. If new role types are needed in future, for example, to support XML-attributes, classes for new role types and user interface classes (wizards) are added to this role plugin. Specializing own role types for Java Frames is discussed in [Ha05a] chapter 7.

The editor itself uses GEF (The Graphical Editor Framework) and Draw2d (painting and layout plugin) plugins. Using an editing framework like GEF gives ready functionality like undo, redo, deleting, moving graphical elements and printing diagrams. It also makes software more complex because model-side must follow GEFs architecture. Each model element must have corresponding editpart (GEF-frameworks controller-part) and responding figure (GEF-framework view-part). This figure object may be its own class or created by using figure base class. If a new role type also needs a graphical representation then a new model, editpart and possible figure classes have to be implemented. This work is not comprehensive description about GEF and more information about it can be found from other sources, for example, in [Ec06b].
6.2 Editor Structure

VARMA consists of three Eclipse plugins. The two main plugins are: the editor side plugin and the roleplugin with is pattern side of the tool. Plugininitializer of role plugin initializes new role types and role type icons to the rootplugin. It is extending JavaFrames plugins Plugin initializer extension point. The third plugin is VARMA XML Editor, which is extended from regular Eclipse text editor. It implements folding in text editor. Editor architecture of the two main plugins (editor plugin and role plugin) is shown in Figure 6.1. The figure concentrates on the process of XML producing from schema to configuration file and all dependencies between components are not shown.
Figure 6.1. VARMA Editor Architecture.

6.2.1 Role plugin

The role plugin is JavaFrames side of the editor software. It contains the role types of VARMA, user interfaces for elements of those role types and code related to the binding of roles. It also contains important classes related to reading and producing of XML and task creation. Those classes need information of existing roles.

6.2.1.1 Role Plugin Core Classes

The PatternDeleteListener is a listener class. It listens for changes in roles. When role is deleted the PatternDeleteListener deletes the corresponding model element and if the deleted element was a component, it deletes all its attributes.
The AssemblyApplausePluginListener checks, if the PatternDeleteListener is active every time when roles are modified. If not, AssemblyApplausePluginListener starts it. The AssemblyApplausePluginListener itself is run when plugins are started.

The PluginInitializer initializes role types and icons for role types to Root of JavaFrames, so they can be used in the program. Use of the PluginInitializer is specified in plugin.XML file. It is extending Initializer-extension point in JavaFrames. PluginInitializers are typical for each JavaFrames extension plugins.

The CoreTaskFactory is the component behind updating the tasks created from configured dependencies. Every time an element is generated into the diagram, task factory is called. CoreTaskFactory creates more tasks if new elements are needed. It can also delete tasks if the generated element performed several tasks.

Inside the CoreTaskFactory there are two lists, which contain information about the dependencies between elements and attributes. Taskmodels list holds information about those attribute types, which need other attributes. Currenttasks list keeps track about tasks, which are created because of taskmodels and their states. Is the task performed or not and what component and attribute is causing it.

The XMLUtils class contains methods for running XPath expressions to find XML nodes and other helper methods related to XML roles.

6.2.1.2 Schema Parser
The Schema parser is an important part of the editor. It is a class responsible for reading the XML schema and building patterns from it. The schema parser also calls the CoreTaskFactory methods to create contains of taskmodels list.

The Schema parser is also responsible for updating patterns when VARMA GEF model is loaded from file. Without this functionality all pattern binding would be broken when a model is loaded.
6.2.1.3 Pattern Package

There is a Role-class and a BindingUtils-class for both of the two role types. Role-class defines the semantics for a certain role. Each role is associated with a semantics-object that defines the behavior of the role.

The BindingUtils-class is responsible for binding a role to a certain element. Its functionality is called when the user has given the needed information through user interface wizards.

6.2.1.4 Pattern.ui Package

Pattern.ui.package contains a generation wizard class for both role types. The wizards are responsible for the generation of elements. Also the base classes for the wizards belong to this package.

6.2.2 Editor Plugin

The editor plugin is GEF side of the editor software. Parts implementing GEFs MVC architecture are located inside this plugin.

6.2.2.1 Model

Model package holds everything in editors GEF model. It is the only package saved when saving file. Editparts are built based on these model elements. Model consist classes for element hierarchy: abstract base classes and classes for concrete elements. It also contains several helper classes to support model modifications.

6.2.2.2 EditParts

Editparts are GEFs MVC architectures controllers. They act as a link between model and figures. When using GEF, editparts are deleted and created continuously. Figures, GEFs MVCs view-parts, are created in editparts createFigure-method. There are no own classes for figures, because the graphical figures used in the diagram are so simple that they can be created inside the createFigure method with a few lines of code.


6.2.2.3 XML Writer

The XML writer is the class, which writes XML. It uses Output.txt file to find right places for each model element. There is no guarantee in which order the components in the same level are written.

When VARMA is creating XML in real-time using XML roles, it does not use XML writer. The XML writer is a class for creating a whole XML file from the GEF model at once.

VARMA can not read XML files; it only reads GEF model-files of XML configuration. Those files are GEF-model of configuration and they consist of all information about elements of diagram. After this model-file is read, it is possible to export the configuration to XML again.

6.2.2.4 Output.txt-file

File named output.txt in Eclipse’s base directory defines which elements are written into what base blocks. Inside each block the names of the elements are given. It is sufficient to give a prefix of the name, which uniquely determinates the element. Higher level elements are marked with < codemarks, # is the codemark for a new block and – is the codemark for a file end. There is no limitation how many higher level elements a block can contain. It depends from the configuration file structure.

An example of output.txt is listed here:

```
#  
<MsgConversionConfiguration
<MessageConversions
  Binary
  Message
  
#  
<StateConversionConfiguration
<StateConversions
  StateConf
  StateComb
  
#  
<GenericMeasurement
<Groups
  Group
  
```
6.2.2.5 XML File

Higher level blocks are read from output file. For example, in this case MsgConversionConfiguration is BLOCK1A and MessageConversions is BLOCK1B. Elements, which names start with strings written after <MessageConversions are written inside BLOCK1B in the XML configuration file, in this case, binarymessageconversions and messageconversions.

Generated XML-file structure is listed below:

BLOCK1A
  BLOCK1B
    XML from program
  BLOCK1B
BLOCK1A
BLOCK2A
  BLOCK2B
    XML from program
  BLOCK2B
BLOCK2A
BLOCK3A
  BLOCK3B
    XML from program
  BLOCK3B
BLOCK3A

From the sample configuration before, the resulting XML is created as follows:

```xml
<MsgConversionConfiguration>
  <MessageConversions>
    elements starting with word message and binary are placed here.
  </MessageConversions>
</MsgConversionConfiguration>
<StateConversionConfiguration>
  <StateConversions>
    elements starting with word StateConf and StateComb are placed here.
  </StateConversions>
</StateConversionConfiguration>
<GenericMeasurement>
  <Groups>
    elements starting with word group is placed here.
  </Groups>
</GenericMeasurement>
```
6.2.2 VARMA XML Editor

VARMA XML Editor plugin is Eclipse text editor implementing folding of XML blocks. This feature is used in “show source” action. If VARMA XML editor is not open when show source action is performed it is opened automatically. VARMA XML Editor also colours XML blocks.

6.3 Example Sequence of Using VARMA

Figure 6.2 shows an example of creating empty configuration from XML schema and adding a first element into the configuration. The Schema reader reads XML schema and creates configuration pattern. The configuration pattern holds the structure information of all roles. User’s own configuration pattern is extending this pattern and creates the tasks based on this base pattern.

When the user executes the “new configuration” command from the user interface, the rebound elements pattern, the user configuration pattern and the XML file are created. When binding rebound elements, VARMA is using the content of output.txt as described in previous chapter.

Figure 6.2. A sequence diagram of creating new configuration and creating the first element to it.
7 Using the Editor

This chapter introduces the usage of VARMA. How it can be used to create a configuration file. The chapter concentrates on the user interface. The first step in using VARMA is to create a new XML schema for the configuration. If the schema exists it is possible to use it by adding few special attributes to it. The person who writes the schema should be an expert in the configuration domain. After the schema is created anyone can use VARMA to create configurations. The schema creation and configuration for VARMA is discussed in chapter 6.

7.1 Starting of Use

The starting point for new configuration (the schema is read and new configuration is created) is shown in Figure 7.1. The diagram is empty because there are no elements. Rebound XML elements are created in the XML file at right.

The editor is started by creating a file with the extension “conf” to Eclipse. After the file is created VARMA opens automatically. The schema is read by choosing “read schema” from VARMA actions menu and specifying the location of the schema file in the disk. Then the editor reads the XML schema. Based on this schema it creates patterns. After reading the schema file, the patterns are ready, but the graphical presentation is still empty. The user can now create a new configuration file. This is done by choosing “create new configuration” from the same menu. The read XML schema declares what kind of elements can be bound to roles and thus can be generated to the diagram. When a new configuration is created, VARMA creates a new XML file. The configuration XML is generated into this file.
Figure 7.1. The starting point for new configuration.
7.2 User Interface

Like any Eclipse editor, the user interface layout of VARMA can be modified in several ways. Views places, sizes and order may be changed.

The editor represents the elements of the XML configuration as graphical shapes. The elements are those graphical shapes, components and attributes. Different elements are shown in Figure 7.2. Highest level elements are represented as attribute roles; these are not shown in a diagram as independent units, but inside a parent element. Second level elements, the parents of attributes are represented as component roles and shown in the diagram as rectangles. Second level elements can be nested. Those second level elements appear as large rectangles with attribute names and values written inside of them. Lowest level element is the root element, the diagram.

Figure 7.2. Group holding one Measurement and BinaryMessageConversion holding two attributes.
The editor uses automatic layout for elements. The automatic layout in work is shown in Figure 7.3. New element is generated to the right from previous one and each line contains max ten elements. After the tenth elements the next ones are created under existing ones. Their y-coordinates are calculated from the height of the highest element in the upper line. Second level elements can be nested and always when generating an element inside other element, the sizes of the parent element and its parents are calculated again.

Elements are displayed with colours. Elements with the same name have the same colour. This should help the user to find certain elements and see placements of same kind of elements by just one look. Colours of elements are saved to the model side, so they remain same when the diagram is loaded again. There are ten built-in colours in the CoreTaskFactory-class and if more element names are present, the rest is represented as black. If in the future large schemas are handled and more colours are needed, adding them is easy. The programmer just has to specify RGB values of new colours in the CoreTaskFactory-class.
Dependencies between elements are represented with arrows. These arrows are drawn between component elements even though the dependency is actually from attribute to attribute. The dependencies do not affect the result XML. These are shown in the diagram to help the user to understand the configuration structure.

7.3 Handling Elements

Graphical elements of VARMA can be handled in many ways. Those can be created, deleted, compressed, resized, moved, etc. These modifications reflect to the result XML. Everything in VARMA is handled through patterns and elements, direct editing of XML is not needed. Direct editing of XML is not recommended, because the user can easily make errors. However the user can do certain actions without harm, for example, remove unbound XML blocks.

7.3.1 Creating New Elements

The task engine creates generating tasks for each base component type introduced in the XML schema. The user uses these tasks to generate XML elements into the diagram. When the user performs a binding task by generating a component, the following happens:

- The generated component appears into diagram.
- Component role is bound to the generated component.
- Tasks for creating its attributes and child components are created.
- XML role corresponding to this component role is created. The reference of XML role is the XPath expression, which is used to locate the actual XML block.
- XML block corresponding to this XML role is inserted to the configuration XML.
- XML role is bound to the XML block.

The XML side is functioning the same way when the user performs a binding task by generating an attribute. Otherwise functionality is slightly different:

- The generated attribute appears in parent component.
- Attribute role is bound to the generated attribute.
- New tasks appear if generated attribute needs other attributes or existing tasks are deleted if the generated attribute filled several tasks at once.
Tasks mentioned in the last bullet are created based on the dependency information that the user has specified in XML schema.

Figure 7.4 represents the results of binding a component role to a generated binary message conversion. Binary message conversion has appeared into the diagram and the corresponding XML role has been created and the XML block has been inserted into the configuration file. In Figure 7.4 arrows are pointing the component and the corresponding XML block.

Figure 7.4. The user has created BinaryMessageConversion.

There cannot be a situation where an element exists, but is not bound to a certain role. Therefore no locating of elements is possible and there is no user interface for that. This lack of locating elements also makes sure that single element cannot be bound to more than one role. However, one role can be instated several times. Locating existing elements for binding them to roles is normal feature of other JavaFrames-based programs.

If the schema has no mandatory elements, at first all creating tasks are optional. For example, if the user wants to create a new binary message conversion into the XML file,
she performs a binary message conversion binding task. After this new tasks for creating attributes inside this binary message conversion appear. If some of these binary message conversions attributes refer to non-existing element attributes, a task for creating a particular element will appear and also tasks for creating its attributes will appear after the element is created. If the generated element is an attribute, which an other attribute needs, the generation wizard proposes a right value for it. If there are several attributes needing the generated one, the wizard lists all needed values in its note field. It also shows values already filled with same named attributes. In Figure 7.5 the user has created target data name attribute and therefore task engine has created mandatory task for creating state configuration. The target data name attribute has dependency to source data name attribute in state configuration.

Figure 7.5. BinaryMessageConversion TargetDataName attribute needs StateConfiguration with attribute SourceDataName.
7.3.2 Deleting Elements

An element can be deleted from pattern view by deleting the role bound to it. The editor event listeners listen the pattern view and delete the corresponding element from the editor model when a role is deleted.

7.3.3 Compressing Components

Components have two different states: They can be compressed or expanded. After creation the component is expanded. When user double-clicks a component it is compressed. Compressing a component hides its attributes and resizes it to the default size. New double-click expands the component again. The user may compress components if the diagram space is ending or overlapping components are hiding attribute names and values. Components can also be resized by dragging its borders.

7.3.4 Notification Tasks

When more than one instance of a certain attribute-component combination is needed a notification task is created. Task description of Notification task is the information about needed component and attribute combination.

7.3.5 Locating XML and Graphical Shape of Component

Locate component function is activated by choosing “show source” action from user interface. The action opens XML file if it is not open in VARMA XML editor and compress all other XML blocks than the chosen one. It also focuses the corresponding component shape from diagram. Attributes cannot be located, but it is possible to locate the parent component of attribute. The user has located component binary message conversion in the Figure 7.6.
7.4 Two Ways to Produce XML Configuration

VARMA can create XML in two ways: step by step or the whole XML configuration from model at once. XML is always created step by step into XML configuration file, but the user can ignore this file and XML roles and export VARMA GEF model to XML configuration when he thinks the configuration is ready.

7.4.1 Real-time XML Production

When an element is created to the diagram and a role is bound to it, the corresponding XML role is also created and is bound to the created XML block in the result file. This allows the user to see results of his actions instantly in the XML all the time. Step by step process is easier to follow to the user [Ha et. al 01b].

Contents of the output.txt-file are bound when a new configuration is created. Thus highest level XML elements are written into the XML at the start. Those highest level elements cannot be deleted during the user session.
7.4.2 Exporting GEF model to XML at Once

Before exporting, the user should ensure that there are not any mandatory tasks present. If there is, then the result the XML file might be invalid. After performing all generation tasks, appeared related tasks, created all components and attributes he wants to include in his configuration, the user can export editor model into XML configuration file.

The user chooses “Write XML” from VARMA actions menu. The editor opens a dialog and asks filename and location for exported XML. Then the editor exports its GEF model into XML. This kind of creation of XML is one-step process.
8 Conclusions

Schema reading is working concept. By using VARMA it was possible to read Harald configuration schema and create configuration file for Generic measurement service. Experiences showed that there are still some usability issues regarding the integration to the environment.

8.1 Benefits

The usage of VARMA creates many benefits. Two main issues are related to the guidance of the user.

- The user of VARMA does not need to be an expert in the application domain.
- The user of VARMA does not need to know anything about XML schema. Once the schema of configuration is written, it can be used to create configurations.
- The editor is good at handling large entities. Contents of a huge single file can be shown in a diagram and the consequences of modifications are instantly reflected to the task list as new tasks.
- Validity of the XML can be shown in the task list. If there are mandatory tasks present, then the XML file is invalid. The XML file is valid if there are no tasks or only optional tasks remain.

8.2 Problems

There are also a few limitations to the usage of VARMA:

- VARMA does not support XML attributes.
- VARMA can only read GEF model files of XML, not XML file directly. So direct editing of XML files is not possible if user wants modified file to be edited in the editor later.
- Writing correct XML schema for VARMA is not a trivial task and is vulnerable to human errors. Schema writing should be a task for an expert in the application domain.
8.3 Generalization & Further Development Ideas

VARMA is not developed only for Harald configurations; the implementation can be used to read different XML schemas for any configuration description. Our aim was to create a general solution to control XML configurations; this is achieved by reading the incoming XML schema and output file structure from files. This enables the usage of VARMA for different configuration domains. Small experiments for reading and using different XML schemas were run. These schemas consisted XML elements and attributes, which VARMA supports. Notice that dependencies and base-attributes have to be configured to the schema before use. Editor handled all of these small test schemas but XML schema language is a complex tool and this tool cannot handle all special cases.

One feature for future implementation is also some way to control the order in which elements at the same level in XML schema are written. This feature is not needed for Harald configuration, but for other configurations this might be important. JavaFrames pattern engine always arranges roles into alphabetical order. Because XML Elements are written into file in same order as roles, some extra data, where this wanted order is stored, would be needed.

Using different icons for different component types could be a possible further development idea. These icons could also be read dynamically from schema. At the moment all component roles and all attribute roles use same icons.
Bibliography


