ESR Automate

User Manual

Document Version 1.8.1
Document Date August 18, 2021

© 2015-2021 ESR Labs GmbH
All rights reserved.

Contact

ESR Labs GmbH
Balanstr. 73, Haus 11, 4. OG
81541 München
Germany
info@esrlabs.com

Limited Warranty

We do not guarantee for the correctness of the information contained in this document. We appreciate all suggestions and hints to errors to improve the quality of this document. The content of this document may be changed without notice. This document may not be copied or altered, also not partly, without prior notice and permission of E.S.R.Labs.

Contents

1 Introduction

ESR Automate is an AUTOSAR Authoring Tool. It allows creating, maintaining and browsing any AUTOSAR model conforming to an AUTOSAR XML Schema, or in other words: to one of the AUTOSAR metamodels. The main purpose of AUTOSAR models is RTE and Basic Software configuration. Typically, an AUTOSAR model comprises information as defined by the AUTOSAR "Software Component Template" (Software Component Definition) and the AUTOSAR "System Template" (ECUs, Communication Buses, Mappings).

In contrast to most other AUTOSAR tools on the market, Automate features text-based modeling. This approach allows treating models very much like program code and brings a number of benefits over graphical or form/tree-based tools:

- Efficient editing, including search&replace, copy&paste, etc.
- Straightforward use with Version Control Systems
- Supports collaboration in large development teams
- Scales up to very large models
- Integrate seamlessly into existing development environments

The textual representation of an AUTOSAR model is created by Automate Sync Tool which is a core part of Automate. The sync tool is able to turn any set of AUTOSAR XML files into the Automate Text files and vice versa. This conversion is loss-less and doesn't change the assignment of model elements to files, the order of model elements within files and the content of the model. Section Textual Language introduces Automate Text language, section Synchronization Tool describes the usage of the Sync Tool.

Just by transforming AUTOSAR XML into Automate textual syntax the files are much smaller, much more readable and a lot easier to modify. As a second step, Automate provides an easy-to-use macros' system which can further simplify the textual view without loosing information. Macros can be defined

by the user without any programming being involved. Section Macro System explains the Macro System.

Although Automate text files can be viewed and modified with any text editor, it's recommended to use one of the editor plugins which come with Automate. The plugins provide syntax highlighting, auto-completion, reference navigation, error annotations and more, each for a specific host editor. The editor plugins are described in separate manuals.

Please note that this manual is not an introduction into AUTOSAR. It's assumed that the reader is familiar with the AUTOSAR concepts.

2 Installation

Automate runs on Windows and Linux. As a basic requirement, the Ruby runtime system must be installed. This chapter explains the steps to install Ruby and Automate on the different platforms.

For the installation of a specific editor plugin, please refer the separate plugin manual.

2.1 Windows

For Windows, Ruby is readily bundled in a convenient installer which can be downloaded from rubyinstaller.org. Download the latest Ruby 2.7 installer and install it following the installation wizard. During installation check the option "Add Ruby executables to your PATH".

Once Ruby has been installed, execute the "install.bat" batch file included in the Automate distribution package. This will install the required Ruby gems and license files.

```
> install.bat
```

In case you need an interactive diff feature enabled, make sure to install fxruby 1.6.44 gem with:

```
> gem install fxruby -v 1.6.44
```

2.2 Linux

On Linux, install Ruby 2.7 from your Linux distribution package. If your distribution doesn't contain Ruby in that version, another good option is to use the "rvm" tool (rvm). Rvm is a utility for switching between different versions of Ruby which can also be used to install new version from source code bypassing the Linux distribution.

Yet another option is "rbenv", which is very similar to rvm. However, when installing Ruby using rbenv, it doesn't install libruby.so by default, leading to errors when loading C-extensions of Ruby gems. To fix this, set the option below before running rbenv install:

```
> export RUBY_CONFIGURE_OPTS="--enable-shared"
> rbenv install 2.7
```

In case you need an interactive diff feature enabled, make sure to install the development packages:

- libfox-1.6-dev
- libjpeg-dev
- libpng-dev
- libtiff-dev
- libxext-dev
- libxft-dev
- libxrandr-dev
- x11proto-gl-dev
- zlib1g-dev

Depending on your Linux distribution the command for installing these packages could look like (from Ubuntu 20.04):

```
> sudo apt-get install libfox-1.6-dev libjpeg-dev libpng-dev libtiff-dev
   libxext-dev libxft-dev libxrandr-dev x11proto-gl-dev zlib1g-dev
```

You also need to install fxruby 1.6.44 gem with:

```
> gem install fxruby -v 1.6.44
```

Once Ruby has been installed, execute the "install.sh" batch file included in the Automate distribution package. This will install the required Ruby gems and the license files.

```
> ./install.sh
```

If you have trouble installing those gems, please refer to the respective gem documentation:

- Installing Nokogiri
- Installing FXRuby
- Installing Ruby FFI

2.3 License

If you have a license file, please put it into the ".esr" directory inside your home directory (%UserProfile% on Windows and $HOME on Linux) when Automate is installed.

2.4 Updates

In case you already have a previous version of Automate installed, also follow the installation procedure described above as you would on a new installation. This will add the new Ruby gems to your Ruby installation.

When you have several versions of Automate installed, you can use the Ruby "gem" tool included in the Ruby distribution to find out which versions are actually installed:

```
> gem list
```

The output of the "gem" command should contain a line showing the installed versions of Automate:

```
...
esr-automate (1.1.0, 1.0.0)
...
```

By default, the latest installed version of Automate will be used whenever an Automate command is invoked. If you need to run older version, you can do so by specifying the particular version on the command line surrounded by underscores, e.g.:

```
> auto-sync _1.0.0_ ...
```

3 Textual Language

The basic idea behind Automate is to turn AUTOSAR XML into a more legible textual representation. This chapter introduces Automate Textual Language.

3.1 Syntax

Automate uses a *generic* textual syntax. This means that every command expressed in Automate textual language follows the same pattern:

Pattern:

```
<command> (<arg>,)* (<label>: <arg>,)*
```

Example:

```
ArgumentPrototype arg1, direction: in, type: /DT/UInt8
```

The first word in a line is the command. The command keyword is followed by arguments, which can have a label in front of them. All arguments without a label must precede any labeled arguments. Arguments can be of one of the following kinds:

- String: `"x"` or `x`
- Integer: `1`
- Float: `1.0`
- Enumeration Literal: `in` or `"%xx"`
- Reference: `/Interface/ICS1` or `port1`

For string values, the double quotes may be omitted if the string is a valid identifier and doesn't collide with another keyword (it must start with a letter, must not contain any non-word characters and must not be equal to "true" or "false"). Enumeration literals are normally written without quotes but must be surrounded by quotes if they contain non-word characters or digits. References to other model elements normally contain slashes to separate the short names of the target element and its parent elements. However, references may also be shortened under certain preconditions reducing them to just a single word.

Commands may contain other commands. This allows to build up a tree structure of elements very similar to the nesting of XML elements. In order to nest commands, the container command's definition line must end with an

opening curly brace ({). Each of the child commands follows on a separate line. Finally, the nesting container must be ended by a closing curly brace (}) on a separate line:

Pattern:

```
<command> ... {
  <command> ...
  <command> ...
}
```

Example:

```
ClientServerInterface ICS1 {
  OperationPrototype op1 {
    ArgumentPrototype arg1, direction: in, type: /DT/UInt8
  }
}
```

In some special situations it's necessary to specify the role of a child command. This is done by putting a label in front of the nested command. If several child commands are nested using the same role, square brackets ([and]) can be used to group those commands. As with the curly braces, the opening bracket must be on the same line as the label, the closing bracket must be on a separate line:

Pattern:

```
<command> ... {
  <label>:
    <command> ...
  <label>: [
    <command>
    <command>
  ]
}
```

Example:

```
IntegerType UInt8 {
  lowerLimit:
    ARLimit intervalType: closed, value: "0"
  upperLimit:
    ARLimit intervalType: closed, value: "255"
}
```

Comments can be created using the hash sign (#). They extend from the hash sign until the end of the line:

```
# power window control interface
ClientServerInterface IPwControl {
```

```
  ...
}
```

Note that comments can only be used directly in front of a command. The reason is that comments in Automate are transformed into AUTOSAR description model elements and thus it must be possible to associate them with another model element. Note that this also means, that comments can only be put in front of commands representing model elements which can take a description element according to the AUTOSAR metamodel.

Comments in Automate text files will be transformed into AUTOSAR description model elements with the language attribute set to "forAll". The other way around, description elements in ARXML files will only be turned into Automate text file comments if the language is set to "forAll". All other kinds of description model elements will show up as regular child elements nested inside the described element.

Another similar mechanism for attaching information to model elements are annotations. Annotations start with an "at" sign (`@`) and extend to the end of the line like comments. One important application for annotations is macro definition (see Macro System):

```
@macro: AtmDataReceivePoint <name>
DataReceivePoint <name> {
  ...
}
```

When a command has many arguments, lines can become very long. Automate allows lines to be broken apart after a comma (`,`) or after an opening square bracket (`[`). In addition to these "natural" breaking points, new lines can be started anywhere within a command if a backslash (`\`) is inserted before the line break.

Example (break after comma):

```
ArgumentPrototype arg1,
  direction: in,
  type: /DT/UInt8
```

Example (break after comma and square bracket):

```
RunnableEntity run1,
  readVariable: [
    irvar1,
    irvar2
  ]
```

Example (break after backslash):

```
QueuedSenderComSpec \
  dataElement: de1
```

Note however, that the information where a line in an .atm file is broken apart is not stored in .arxml files. This means that line breaks will be removed when a file is transformed into ARXML and back.

In order to overcome this problem, Automate can be instructed to insert line breaks in certain places during the a2t transformation. This is done using the line break extension. See section Line Break Extension for details. Be aware though that even in this case some manually inserted line breaks might be removed if they don't match the configured scheme.

Automate text files should be encoded as UTF-8. Using other encodings for the text files may lead to unintended output in the ARXML files after synchronization. Since the ARXML files created by Automate are UTF-8 encoded, any non-UTF-8 characters originating from the text files will be replaced by the UTF-8 replacement character.

3.2 Semantics

The benefit of generic syntax is that it stays the same even when the semantics of the language change. The semantics of Automate textual language is mainly defined by the selected AUTOSAR metamodel. The current version of Automate comes with the following AUTOSAR metamodels:

- `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.03`, `17.10`, `18.03`, `18.10`, `19.03`, `19.11`, `20.11`

The metamodels included in Automate are fully AUTOSAR compatible, i.e. they allow to model everything allowed by AUTOSAR for a particular metamodel version and nothing more. Or in other words, they enable the user to model any information allowed by the official AUTOSAR XML schema and only that information.

AUTOSAR metamodel classes form the entities of the (abstract) AUTOSAR language. You can view the AUTOSAR metamodel by means of UML class diagrams: Each AUTOSAR release comprises an Enterprise Architect (.EAP) file in the "auxiliary" section which holds the metamodel as a large UML class model. In oder to view the .EAP file you need Enterprise Architect which is also available as a free viewer version (Enterprise Architect Viewer). Make sure to download the "Viewer", not the "Trial" version, as the latter will expire.

There is a one-to-one relationship between AUTOSAR metamodel classes (or UML classes in the .EAP file) and Automate textual language commands: The name of the command is exactly the same as the name of the metamodel class. Further, the arguments allowed by a command correspond to the attributes and non-containment references (UML associations) of the metamodel class. The set of child commands allowed for a particular command is defined by the containment references (UML aggregations) of the metamodel class and its target classes. Each Automate command occurring in an Automate text file creates a model element, i.e. an *instance* of the corresponding metamodel class.

Every argument or child element is associated or aggregated in a specific role, which corresponds to the role name of the metamodel reference (or UML association/aggregation). For command arguments, the role is normally represented by a label in front of the argument. For the sake of brevity, some labels are omitted, making those arguments *unlabeled arguments*. Those arguments are identified by their position in the argument list (unlabeled arguments must be listed before any labeled arguments).

Currently, the only unlabeled argument is "shortName" which is the first argument of most Automate commands.

In the following example, the `ArgumentPrototype` has the "shortName" attribute set to "arg1" (unlabeled argument), the "direction" attribute set to "in", and the "type" reference set to "/DT/UInt8" (both labeled arguments).

```
ArgumentPrototype arg1, direction: in, type: /DT/UInt8
```

Labels for child commands can be omitted as long as the corresponding child class is referenced in only one role. If the role is ambiguous, a label must be put in front of the command. One example is the `IntegerType` command which contains an `ARLimit` in two different roles.

```
IntegerType UInt8 {
  lowerLimit:
    ARLimit intervalType: closed, value: "0"
  upperLimit:
    ARLimit intervalType: closed, value: "255"
}
```

3.3 References

References are pointers from one model element to another model element. There are two kinds of references: containment references and

non-containment references. Nested child commands create model elements which are linked to their container via a containment reference. Model elements which aren't child elements are referenced via non-containment references.

AUTOSAR XML represents non-containment references by means of the *qualified names* of the target elements. A qualified name is the name of a model element preceded by the names of all elements which contain it. In AUTOSAR, the top level containers are packages (`ARPackage`). A reference to the ClientServerInterface "ICS1" in the following example would be represented by the qualified name `/P1/P2/ICS1`.

```
AUTOSAR {
  ARPackage P1 {
    ARPackage P2 {
      ClientServerInterface ICS1 {
        ...
      }
    }
  }
}
```

Automate uses the same mechanism for non-containment references as AUTOSAR XML, i.e. these references are represented by the qualified names of the target elements.

Sometimes, references occur in some specific context which naturally restricts the number of possible target elements. As an example, consider an AUTOSAR receiver port which receives queued data elements. In this case a QueuedReceiverComSpec can be added to the port specifying the queue length for a particular data element.

```
RPortPrototype receiverPort, requiredInterface: /P1/ISR1 {
  QueuedReceiverComSpec queueLength: 1, dataElement: /P1/ISR1/de1
}
```

In this case, the data element referenced by the QueuedReceiverComSpec must be part of the interface which the port implements. Also note that the qualified name of the data element contains the qualified name of the interface because the interface is the container of the data element.

Automate can use the context of a reference in order to shorten the reference string representation. For the example above, it's sufficient to just provide the short name ("de1") of the target data element because the context (the interface) is already clear. References shortened in this way are represented by the short name of the target element without a leading slash:

```
RPortPrototype receiverPort, requiredInterface: /P1/ISR1 {
  QueuedReceiverComSpec queueLength: 1, dataElement: de1
}
```

Short references are especially handy for connectors, mappings and within internal behavior definitions. Automate sync tool (see Synchronization Tool) automatically shortens references when context information is available. See Scope Information for a complete list of references with attached context information.

4 Macro System

The concept of Automate is to represent AUTOSAR models in a concise, textual way. Therefore, it's important to extract the actual information from the AUTOSAR XML files and strip any redundant data. The textual syntax presented in the previous chapter is already a big step towards this goal. As a second step though, Automate macro system can be used to carve out the relevant information even more.

4.1 Macro Concept

The idea behind macros is to "fold" repeating model patterns and hide them behind a meaningful label. During synchronization, Automate sync tool recognizes the macro patterns and replaces them with a macro command. It's important to note that although macros can't cover the full range of possible AUTOSAR models, no information is lost during synchronization (with a possible exception when using calculated attributes). If a macro pattern doesn't match, it's simply not replaced.

Macros are especially useful when they are tailored to project specific modeling patterns. They can even be used to encourage modeling guidelines: When used in an Automate text file, they will produce AUTOSAR XML conforming to the pattern. When AUTOSAR XML is transformed into Automate text files, and the macros don't show up, this indicates a divergence from a modeling pattern.

4.2 Macro Definition

In order to simplify project specific macro definitions, Automate features an easy-to-use macro definition language which doesn't involve any programming. Macros are defined in Automate text files just like the rest of the model. However, those files need to be put aside in a dedicated directory, and the Automate tools need to be informed about the macro directory (see the description of command line options for the different tools below).

A macro consists of a macro pattern (which should be matched on the AUTOSAR source model) and a macro name with arguments (which should be inserted when the pattern is removed). Macro patterns are parts of an AUTOSAR model defined within a file in a macro directory. The actual part is marked by an annotation which also defines the macro name and its arguments. Macro arguments are derived from placeholders used in the pattern. Here is an example:

```
AUTOSAR {
  ARPackage P1 {
    CompositionType Compo1 {
      @macro: AtmAssemblyConnector <name>, <rcp>:, <rport>:, <pcp>:, <
         pport>:
      AssemblyConnectorPrototype <name> {
        AssemblyConnectorPrototype_requester componentPrototype: <rcp>,
           rPortPrototype: <rport>
        AssemblyConnectorPrototype_provider componentPrototype: <pcp>,
           pPortPrototype: <pport>
      }
    }
  }
}
```

The above model defines a macro named "AtmAssemblyConnector". The `@macro:` annotation marks the model subtree which forms the macro pattern: In this case the pattern consists of one AssemblyConnectorPrototype element, a AssemblyConnectorPrototype_requester element and a AssemblyConnectorPrototype_provider element. The macro name is the first word in the macro annotation, and it's followed by the macro's argument definitions.

Macro arguments are retrieved from pattern placeholders. In the example above there are placeholders for the connector name (`<name>`), the requester and provider component prototypes (`<rcp>` and `<pcp>`) and for the requester and provider ports (`<rport>` and `<pport>`). Placeholders are marked by angle brackets and may be given any identifier (starting with a letter and containing

only word characters). The macro argument list consists of the pattern placeholders separated by commas. If the placeholder name is followed by a colon (:), the macro argument will be a labeled argument, otherwise it will be an unlabeled argument.

Some identifiers in Automate text files appear without quotes if they don't contain special characters. In particular, the "shortName" attributes are normally unquoted. Currently, the macro definition can't influence if macro arguments are serialized with or without quotes. However, if a macro argument is named "shortName" it will inherit this property automatically.

With the macro definition above, AssemblyConnectorPrototype elements can be abbreviated by AtmAssemblyConnector elements if the first matches the given pattern (i.e. it doesn't specify any additional attributes like "uuid" which are not part of the pattern):

Model without macro:

```
AUTOSAR {
  ARPackage P1 {
    CompositionType Comp1 {
      AssemblyConnectorPrototype Con1 {
        AssemblyConnectorPrototype_requester componentPrototype: cp1,
           rPortPrototype: port1
        AssemblyConnectorPrototype_provider componentPrototype: cp2,
           pPortPrototype: port2
      }
    }
  }
}
```

Model with macro replaced:

```
AUTOSAR {
  ARPackage P1 {
    CompositionType Comp1 {
      AtmAssemblyConnector Con1, rcp: cp1, rport: port1, pcp: cp2, pport:
         port2
    }
  }
}
```

Note that both, the model without macros and the model with macro use short references. This is possible since the Automate macro system automatically derives scope information from the pattern placeholders and makes it available to the macro. Also note that the pattern placeholder names make up the labels of the macro arguments. The name attribute is unlabeled because no colon is present in the macro definition.

Default values for macro attributes can be specified with the special syntax:

```
@macro: NvBlockAttributes <size = 10000>:, <checksum = crc>:
AdminData {
  @macro: optional
  Sdg gid: "blockAttributes" {
    SdgContents mixed_content_order: <<= nil > {
      @macro: optional
      Sd gid: "size", value: <size>
      @macro: optional
      Sd gid: "checksum", value: <checksum>
    }
  }
}
```

Values of integer, floating point, boolean and string types are supported.

4.3 Calculated Attributes

In case of an assembly connector, the connector name is not really relevant as it doesn't add any useful information. Since AUTOSAR requires a connector name, one popular solution is to derive the connector name from the names of the component prototypes and ports. The Automate macro system supports this by means of calculated attribute values. For the example above, a calculated name attribute could be added in the following way:

```
@macro: AtmAssemblyConnector <rcp>:, <rport>:, <pcp>:, <pport>:
AssemblyConnectorPrototype <= "CON_#{rcp.shortName}_#{rport.shortName}_#{
   pcp.shortName}_#{pport.shortName}"> {
  ...
}
```

In contrast to a regular placeholder, the angle brackets' content starts with an equal sign (=). The equal sign is followed by a string definition in quotes ("). This string in turn may contain placeholders marked by curly braces preceded by a hash sign (#{ }). Within the string placeholders, the macro arguments may be referenced by name, and their attributes can be accessed using a dot (.) followed by the attribute name. The attribute names are defined by the AUTOSAR metamodel for the metamodel class associated with the argument. In the example above, the "shortName" attribute is accessed for component prototypes and ports.

With this modification, the above example changes in the following way:

Model without macro:

```
AUTOSAR {
```

```
  ARPackage P1 {
    CompositionType Comp1 {
      AssemblyConnectorPrototype CON_cp1_port1_cp2_port2 {
        AssemblyConnectorPrototype_requester componentPrototype: cp1,
           rPortPrototype: port1
        AssemblyConnectorPrototype_provider componentPrototype: cp2,
           pPortPrototype: port2
      }
    }
  }
}
```

Model with macro replaced:

```
AUTOSAR {
  ARPackage P1 {
    CompositionType Comp1 {
      AtmAssemblyConnector rcp: cp1, rport: port1, pcp: cp2, pport: port2
    }
  }
}
```

Note the calculated name of the AssemblyConnectorPrototype and the missing name argument of the AtmAssemblyConnector.

By using calculated attribute values like this, the output of the macro transformation is even more concise. The drawback however is, that macros will not match if the value of the source attributes differ from the calculated values. In the example above, the macro would not match if the name of the AssemblyConnectorPrototype would be changed to e.g. “Con1”.

Since names like the connector name aren't really relevant, Automate provides a way to ignore attribute values during the XML to Text transformation but will still insert the calculated value during the Text to XML transformation. In order to activate this feature, the equal sign preceding the calculation rule must be replaced by an opening angle bracket and an equal sign (<=). This syntax should suggest that the rule only applies in the Text to XML direction. With this change, the example macro definition above looks like this (the only difference is the added <):

```
@macro: AtmAssemblyConnector <rcp>:, <rport>:, <pcp>:, <pport>:
AssemblyConnectorPrototype <<= "CON_#{rcp.shortName}_#{rport.shortName}_
   #{pcp.shortName}_#{pport.shortName}"> {
  ...
}
```

The feature of ignoring attribute values in the XML to Test transformation has the drawback that information is actually lost. On the other hand this provides

a way to clean up or normalize model patterns in project specific setups. In the connector example, it would be ensured that every connector name follows the given naming pattern.

4.4 Nested Macros

In some cases, it's necessary to define macros which can be used within other macros. A very gainful application of macros is the abbreviation of CompuMethods to more concise enum definitions. In this case there should be a macro replacing the CompuMethod and another macro representing a single enum literal. The second macro can (only) occur as a child of the first.

The Automate macro language allows to define a nested macro within a container macro. The definition of nested macros is very similar to regular macros. The only difference is that for nested macros, the minimum and maximum occurrence may be defined. This is done using a square bracket notation (`[.,.]`) in front of the nested macros name. The following example defines a macro "EnumCompuMethod" which can contain "EnumLiteral" elements.

```
AUTOSAR {
  ARPackage EnumMacro {
    @macro: EnumCompuMethod <shortName>
    CompuMethod <shortName>, category: "TEXTTABLE" {
        compuInternalToPhys:
          Compu {
            CompuScales {
              @macro: [1,*] EnumLiteral <literal>, <value>
              CompuScale {
                lowerLimit:
                  Limit intervalType: closed, text: <value>
                CompuScaleConstantContents {
                  CompuConst {
                    CompuConstTextContent {
                      Vt text: <literal>
                    }
                  }
                }
                upperLimit:
                  Limit intervalType: closed, text: <value>
              }
            }
          }
    }
  }
}
```

In this example, the EnumLiteral must occur at least once but there is no upper limit (*). If the nested macro would not match at all, the containing macro would also not match since the minimum occurrence requires at least one match. In order to make the container macro match even without any matching child macro, the minimum occurrence should be set to 0.

If an Automate macro doesn't match, it's simply not replaced. This is also the case for nested macros. So in the example above the macro "EnumCompuMethod" would match as long as there is at least one match of the nested macro, but regardless of if there are other kinds of CompuScale contents like rational formulas. In order to forbid non-matching nested content in a macro, the keyword `exclusive` may be applied. With the following change, the above macro would only match if the nested macro matches for *all* CompuScale elements.

```
@macro: exclusive [1,*] EnumLiteral <literal>, <value>
```

4.5 Macro Context

Macro definitions are done within context model elements. In the enum example above, the context for the "EnumCompuMethod" macro is an ARPackage. Macros will only match if the macro pattern occurs in the same context (i.e. within the same container class and the same role) as the macro definition.

This behavior is relevant in cases where the macro pattern's root element can occur in different roles. As an example, an AR 4.0 data receive point is a VariableAccess element. Only the role in which it is connected to its container makes it a data receive point. The following snippet defines a macro "AtmDataRecvPointByArg" with root pattern class VariableAccess. The pattern matching behavior described above ensures that the macro will only match if the VariableAccess occurs in the role "dataReceivePointByArgument" (which actually makes it a data receive point).

```
RunnableEntity {
  dataReceivePointByArgument: [
    @macro: AtmDataRecvPointByArg <port>:, <dataelement>:
    VariableAccess <<= "DRP_#{port.shortName}_#{dataelement.shortName}> {
      AutosarVariableRef {
        VariableInAtomicSWCTypeInstanceRef targetDataPrototype: <
            dataelement>, portPrototype: <port>
      }
    }
  ]
}
```

As another example, consider the following macro for an AdminData within a PPortPrototype:

```
AUTOSAR {
  ARPackage P1 {
    ApplicationSwComponentType SWC1 {
      PPortPrototype port1 {
        @macro: SpecialPort
        AdminData {
          Sdg gid: "specialPort"
        }
      }
    }
  }
}
```

Although AdminData can occur in any Identifiable, the macro "SpecialPort" will only match within instances of PPortPrototype. This behavior can be changed though, by explicitly specifying a superclass in the macro definition using the @<class-name> notation. By changing the macro definition from

```
@macro: SpecialPort
```

to

```
@macro: @PortPrototype SpecialPort
```

the macro will match in all instances of PortPrototype and thus also in RPortPrototype. The topmost class which may be given for the context is the metamodel class actually containing the macro root class. In this case this is Identifiable which contains AdminData.

4.6 Optional Macro Parts

Sometimes, macros don't match because some part of a pattern is missing. Instead of creating a new macro, the missing part may be marked as optional using the @macro: optional syntax. As an example, consider a custom extension to NvBlockNeeds by means of special data. Each additional attribute is expressed as a Sd model element. Now if a particular instance of the NvBlockNeeds is missing one of these attributes, the macro as a whole should still match. In the following macro definition, the line @macro: optional ensures that each of the Sd elements is optional.

```
AUTOSAR {
  ARPackage P1 {
    ApplicationSwComponentType SWC1 {
```

```
      SwcInternalBehavior myInternalBehavior {
        SwcServiceDependency myNvmBlock {
          NvBlockNeeds myNvBlockNeeds {
            @macro: NvBlockAttributes <size>:, <custom1>:
            AdminData {
              @macro: optional
              Sdg gid: "blockAttributes" {
                SdgContents {
                  @macro: optional
                  Sd gid: "size", value: <size>
                  @macro: optional
                  Sd gid: "custom1", value: <custom1>
                }
              }
            }
          }
        }
      }
    }
  }
}
```

If an optional macro part is not present during matching, any contained placeholders will not be set, and the resulting macro model elements will have the corresponding attributes unset. The other way round, if and only if an attribute of the macro model element which is associated with a placeholder in an optional macro part is set, the optional part will be expanded when converting back to ARXML. Since macro attributes are required in order to decide if an optional macro part is present or not, optional macro parts without embedded placeholders are not allowed.

As the above example also shows, optional macro parts may contain optional macro parts. Just as with non-nested optional parts, the part is expanded to ARXML if at least one of the nested placeholder has a value. In this case, this means that the `Sdg` is only generated if it contains at least one `Sd` element.

4.7 Arbitrary Macro Subtrees

Optional macro parts are a way to make macros more universal. There are cases, however, where some contents of a macro pattern should explicitly not be part of the macro. As an example, most model elements in the AR 4.0 metamodel may have a "longName" in form of a `MultilanguageLongName` which in turn may be composed of a number of sub elements. Most of the time, the long name is not relevant for a macro but should rather be ignored

during macro matching and copied to the resulting macro element without changes.

The following example uses the `@macro: any` syntax to mark a subtree as not being part of the macro:

```
AUTOSAR {
  ARPackage P1 {
    @macro: ImplArrayType <shortName>, <type>:, <size>:
    ImplementationDataType <shortName>, category: "ARRAY" {
      ImplementationDataTypeElement <<= shortName+"_Element">,
         arraySizeSemantics: fixedSize, category: "TYPE_REFERENCE" {
        SwDataDefProps {
          SwDataDefPropsConditional implementationDataType: <type>
        }
        PositiveIntegerValueVariationPoint text: <size>
      }
      @macro: any
      MultilanguageLongName
    }
  }
}
```

With this definition, the macro "ImplArrayType" will match no matter if there is a `MultilanguageLongName` contained in the `ImplementationDataType` or not and independent of the elements which might be contained within the `MultilanguageLongName` element. If there is such an element, it will reappear within the macro element after matching without change.

4.8 Referencing Macros

Macros may be used to replace model elements which are referenced by other elements. One example is the EnumCompuMethod introduced above. That EnumCompuMethod is defined as a macro with CompuMethod as the macro pattern root. Thus, it may act as the target of any reference which can point to a CompuMethod.

However, there is one important point to remember when macros should be usable as reference targets: Since references are realized via qualified names, the macro element must have a qualified name. Automates' qualified name calculation works for all elements with a "shortName" attribute. Thus, it's necessary to give the macros' name attribute the name "shortName", which means that the name placeholder must be `<shortName>` as shown in the example above. If the name is set incorrectly, the macro element will have no qualified name. If in this case references pointing to the macro root element

exist, the macro will not match so that the references can be kept in the model.

References pointing to elements which could be collapsed in a macro (child elements of a macro root) will prevent the macro from matching. Otherwise, there would be no way to keep the reference in the text representation.

KNOWN ISSUE: The check if elements collapsed into a macro are referenced is not repeated if the .atm file containing the macro occurrence has already been created. This means that if a macro was successfully applied and later a reference to an element collapsed inside the macro is added in a different file, the macro does not automatically expand. Instead, there will be an unresolved reference on the ATM side. This problem can be resolved by removing and recreating the .atm files containing the respective elements.

4.9 Comments on Macros

Model elements replaced by macros can take comments if the replaced elements can take comments (see section Syntax). As with non-macro elements, comments show up in the model as description elements with the language attribute set to "forAll", and the other way round, only description elements with language "forAll" are turned into comments.

5 Extensions

Automate features an extension mechanism which offers advanced configuration options and can be used to extend Automate with custom functionality.

5.1 Extension Files

Extensions are written in Ruby using concise, extension specific APIs (internal DSLs). The various Automate commands have a command line option `--ext-dir` which allows specifying directories containing extension files. These directories will be searched recursively for files with the extension `.rb`

and those files will be loaded. Note however, that extensions may only use the extension API described below and may not contain arbitrary Ruby code.

In case of an error during extension loading, the loading of the corresponding extension file will be aborted, and the error message will be printed using the logger (log level "error").

5.2 Check Extensions

Check extensions are defined by means of the `check` command. They must have a name and may optionally have an ID (`:id` option). Automate checks that all names and IDs of loaded extensions are unique and throws an error otherwise. The actual check routine is appended to the check command as a Ruby block enclosed by `do` and `end`.

```
check "CheckWithoutId" do
  # check code here
end

check "CheckWithId", :id =>17 do
  # check code here
end
```

Checks may be defined for a specific metamodel class using the `:for` option. In this case the check routine will be run for each instance of the specified class found in the model. The Ruby block therefore needs to take one argument which will be set to the corresponding instance element. If no metamodel class is given, the check will be run once for the whole model.

```
check "PortCheck", :for => PortPrototype do |p|
  # check code here
end
```

Each check definition may be prefixed by an optional description. Descriptions must be marked with the keyword `desc`.

```
desc "this is the description for the following check"
check "CheckWithDescription" do
  # check code here
end
```

In order to report a problem, the check routine may use the keywords `info`, `warn` and `error`. Each of these commands is followed by an error description and an optional context element (option `:context`). The context element is used to assign the error location. If no context is specified and the check is

defined for a particular metamodel class, that class's current instance will implicitly be the context.

The following example checks that all port names start with a lowercase letter:

```
desc "checks that port names start with a lower case letter"
check "PortNaming", :id => 17, :for => PortPrototype do |p|
  if p.shortName !~ /^[a-z]/
    error "port name must start with a lower case letter"
  end
end
```

Check routine implementations make use of Automate AUTOSAR API derived from the AUTOSAR metamodel as described in AUTOSAR Library. Note however, that metamodel class names used with the `:for` option should not be prefixed with package names.

Checks that aren't defined for a particular metamodel class but rather run for the whole model, can use the keyword `model` to access the model. The model object gives access to an "environment" object which in turn allows searching for elements in the whole model.

The following example uses a find operation to check if any EcuInstance is defined in the model and reports a warning otherwise:

```
check "ECUExists" do
  ecus = model.environment.find(:class => EcuInstance)
  if ecus.size == 0
    warn "there are no ECUs defined in the model"
  end
end
```

5.3 Variant Extension

Automate has basic support for model variants. Given a number of model files, the tools can mask out model elements which are unavailable in a certain variant. This avoids broken reference errors when two variants of a reference target exist under the same name.

Since variants are only defined beginning with AUTOSAR 4.0 and various project/company specific variant mechanisms may exist for AUTOSAR 3.x projects, Automate allows customizing variant behavior by means of an extension.

```
variant_element_active do |model, e|
  # variant definition code here
end
```

The `variant_element_active` method should be called with a block taking two arguments: a model object and a model element object. This registers the block for further usage by Automate. Automate will call the block for every model element in order to find out if that element is active in the current variant or not. If the block returns true, the element is active, otherwise it is inactive.

5.4 Macro Method Extension

Macro methods are methods which can be called from within macro definitions or, more precisely, from within the expressions which define calculated attributes. This can be used to access more complex functionality in those expressions. A macro method is defined with the `macro_method` keyword:

```
macro_method "calculated_name" do
  # code here
end
```

A macro method is called by simply using its name in a calculated attribute expression:

```
@macro: AtmAssemblyConnector <rcp>:, <rport>:, <pcp>:, <pport>:
AssemblyConnectorPrototype <= calculated_name> {
  ...
}
```

Macro methods execute in the context of the model element representing the macro and have direct access to methods of this element. So the following definition would allow to calculate the name of the macro `AtmAssemblyConnector` shown above:

```
macro_method "assembly_con_name" do
  "CON_#{rcp.shortName}_#{rport.shortName}_#{pcp.shortName}_#{pport.
     shortName}"
end
```

In addition to instance methods, macro methods can take additional parameters. Those parameters are defined as follows:

```
macro_method "param_example" do |a,b|
  # use parameters a, b, etc in the calculation
end
```

Parameters are passed to macro methods using parenthesis:

```
@macro: AtmAssemblyConnector <rcp>:, <rport>:, <pcp>:, <pport>:
AssemblyConnectorPrototype <= param_example(3,4)> {
```

```
  ...
}
```

5.5 Line Break Extension

Automate is capable of reading files with extra line breaks as described in the Syntax section. When it writes .atm files though, no extra line breaks will be inserted by default. The line break extension allows turning automatic line break insertion on and specifying where exactly line breaks should be inserted.

There are two different kinds of places where line breaks can be inserted: after arguments and within argument value arrays. The line break configuration allows specifying on a per-class and per-attribute/reference level if the corresponding argument should start on a new line and if an array of values for this argument should be broken apart.

The configuration is done using the `line_break` keyword. If no further arguments are provided, line breaks will be turned on for all classes and all attributes/references (and value arrays) apart from the `shortName` attribute.

```
line_break
```

Line breaks for a specific class are turned on by providing the class name as the first argument:

```
line_break PPortPrototype
```

This will turn on line breaks for all attributes/references of the mentioned class, but not for subclasses. Set the `:subtypes` option to `true` to turn it on for all subclasses as well:

```
line_break PortPrototype, :subtypes => true
```

Using the `:*` wild card for the class name is equivalent to `line_break` without any arguments. This is necessary if further options should be provided (see example below).

```
line_break :*
```

If line breaks should be enabled only for some arguments of a particular class, list the names of the corresponding attributes/references within the `:args` option:

```
line_break RunnableEntity, :args => [:canEnterExclusiveArea, :
   sharedCalprmAccess]
```

Configuration of arguments for which value arrays should be broken apart is done similarly using the `:arrays` option.

```
line_break RunnableEntity, :arrays => [:runsInsideExclusiveArea]
```

Multiple `line_break` commands are executed in order with later occurrences overwriting earlier ones. This allows to do a general setup first for all classes using the `:*` wild card and then refine it for specific classes.

```
# for all commands: break for timestamp attribute
line_break :*, :args => [:timestamp]
# for all ports: break for category attribute, not timestamp attribute
line_break PortPrototype, :subtypes => true, :args => [:category]
# for PPortPrototype: break for all attributes/references
line_break PPortPrototype, :args => :*
```

When overwriting, the `:*` wild card is useful to re-enable line breaking for all arguments or value arrays.

Line breaks for macros work just the same way as line breaks for regular model elements. They can also be configured specifically for each macro by using the macro name and the macro argument names with the `line_break` keyword:

```
line_break AtmAssemblyConnector, :args => [:rcp, :pcp]
```

This assumes that the macro "AtmAssemblyConnector" exists with argument names "rcp" and "pcp". It ensures that both the "rcp" and the "pcp" argument each start on a new line.

6 ECUC PARAMETER DEFINITIONS

Automate supports AUTOSAR ECUC parameter definition mechanism described in the following document in AUTOSAR documentation (see AUTOSAR Documentation):

- Methodology and Templates/Templates/AUTOSAR_TPS_ECUConfiguration.pdf

Note that only AUTOSAR versions 4.x are supported.

This mechanism allows to extend the AUTOSAR metamodel with ECUC parameter definition files written in ARXML. Automate finds and reads these files from a defined path (defined by `--def-path` command line option).

Defined ECUC parameters will be available as model elements when syncing, diffing or when using Automate editor. This has the benefit of making Automate model more concise as well as enforcing the restrictions defined for ECUC parameters.

Note that when using `auto-sync`, the ECUC parameter definition files also need to be provided as input files with the appropriate file extension (see Argument Files of the `auto-sync` command).

7 Splittable elements support

Automate supports the AUTOSAR splittable elements, it means that a model could contain elements which may be split up over several files. For this purpose AUTOSAR defines a stereotype `atpSplitable` on a reference or an aggregation.

Please refer to the AUTOSAR specification documents in order to learn more on this feature benefits and limitations.

Splittables support feature could be activated with an `--splittable` option on any command line tool.

8 AUTOSAR Library

Automate comes with an AUTOSAR Library which makes it easy to analyse and manipulate AUTOSAR models represented by ARXML files. The library features a model API derived directly from the official AUTOSAR metamodel and thus allows developers familiar with the AUTOSAR metamodel to get started with very little effort.

8.1 AUTOSAR Documentation

The official AUTOSAR documentation available at http://autosar.org is the recommended source of documentation. The following lists show the most important documents describing the metamodel. You can find these files by visiting the AUTOSAR website, navigating to "Specifications" and then following the subfolder paths given below.

AUTOSAR 3.1, 3.2:

- Methodology and Templates/Templates/AUTOSAR_SoftwareComponentTemplate.pdf
- Methodology and Templates/Templates/AUTOSAR_SystemTemplate.pdf
- Methodology and Templates/Methodology/AUTOSAR_MetaModel.zip

AUTOSAR 4.0:

- Methodology and Templates/Templates/AUTOSAR_TPS_SoftwareComponentTemplate.pdf
- Methodology and Templates/Templates/AUTOSAR_TPS_SystemTemplate.pdf
- Methodology and Templates/Methodology/AUTOSAR_MMOD_MetaModel.zip

The metamodel zip file contains an Enterprise Architect UML class model representing the metamodel. This is the recommended source of information when working the models and metamodels. All metamodel classes and features have a description which can be read within Enterprise Architect. The PDF documents basically contain the same information but add the official AUTOSAR requirements and some additional description. See section Semantics for details about Enterprise Architect.

When navigating the metamodel in Enterprise Architect, make sure to focus on the top level package called "M2" as this is the metamodel description. The package "M1" contains UML object models describing the basic software configuration language.

8.2 Loading Metamodels

Before a model can be created, the corresponding metamodel must be made available. This is done by means of the `MetamodelLoader` which takes the metamodel version as an argument:

```
require 'esr/autosar/metamodel_loader'

MM = ESR::Autosar::MetamodelLoader.load("4.0.3")
```

8.3 Classes and Packages

The model API is directly derived from the metamodel. There is a Ruby class for each metamodel class and there is a Ruby module for each metamodel package. For example in the AR 4.0 metamodel, there is a class `ApplicationSwComponentType` in the nested package `M2/AUTOSARTemplates/SwComponentTempalate/Components`. The Ruby class has the same name and is nested in Ruby modules corresponding the metamodel packages. Thus, the full Ruby name of the class is: `M2::AUTOSARTemplates::SWComponentTemplate::Components::ApplicationSwComponentType`.

In order to instantiate this class, you could use the following code:

```
MM::M2::AUTOSARTemplates::SWComponentTemplate::Components::
   ApplicationSwComponentType.new
```

As this is pretty tedious to write and hard to remember, the AUTOSAR Library includes a shortcut mechanism. The top level Ruby module contains a module called "Flat" which in turn contains links to all metamodel classes. Using the `Flat` module, you don't have to remember the package hierarchy, and your code becomes shorter and independent of the package hierarchy:

```
MM::Flat::ApplicationSwComponentType.new
```

8.4 Attributes and References

Each Ruby metamodel class has methods for each attribute and reference as defined by the metamodel. In the following, attributes and references are also referred to as "features". The name of the read method which is normally used for model navigation is the same as the feature role name in the UML class model.

As an example, the class `SwComponentType` has a reference named "port" pointing to `PortPrototype`. So in order to navigate from a component type to its ports, call the method "port". Since the reference to `PortPrototype` has an unbounded upper multiplicity (*) the method returns an Array. References with an upper multiplicity of 1 return the target model element

instead. The same applies to attributes with upper multiplicity 1, n, or unbounded. The following code prints out the name of each software component in the model along with the number of ports.

```
swcs = model.environment.find(:class => MM::Flat::SwComponentType)
swcs.each do |swc|
  puts swc.shortName
  puts swc.port.size
end
```

Besides the read method, there are also methods for writing the values of an attribute or references. The easiest way of modification is to just assign values to the features. If the multiplicity of an attribute/reference is 1, the value assigned must be a single value, otherwise it must be an Array. If it's an Array, all the existing values will be replaced by the new values. The following example creates a new software component and sets its name and adds two ports.

```
swc = MM::Flat::ApplicationSwComponentType.new
swc.shortName = "Swc1"
swc.port = [
  MM::Flat::PPortPrototype.new(:shortName => "port1"),
  MM::Flat::PPortPrototype.new(:shortName => "port2")
]
```

Note that attributes and references may also be assigned while creating a new element. The Ruby Hash notation is used to provide a mapping of values to features. Assigning a value in this way has the same semantics as assigning a value separately and thus the following code is equivalent to the example above:

```
swc = MM::Flat::ApplicationSwComponentType.new(
  :shortName => "Swc1",
  :port => [
    MM::Flat::PPortPrototype.new(:shortName => "port1"),
    MM::Flat::PPortPrototype.new(:shortName => "port2")
  ])
```

For Array type features, there are two more methods which allow a more fine-grained modification: `add<feature-name>` adds a single value to an array of values and `remove<feature-name>` removes a value. In both cases, the feature name is changed to start with a capital. The adding method can optionally take a zero based index and will insert the value at this position. The following example shows how a port can be added to and removed from a software component:

```
port3 = MM::Flat::PPortPrototype.new(:shortName => "port3")
swc.addPort(port3)
```

```
swc.removePort(port3)
# insert port at the beginning (index = 0)
swc.addPort(port3, 0)
```

Note that the model framework automatically checks all assignments for compliance with the metamodel. For example, a port could not be assigned to "shortName".

8.5 Reverse References

Sometimes, it can be very handy to navigate the references in the reverse direction. Note that references in the official AUTOSAR metamodel are unidirectional in general, but the AUTOSAR Library adds backward references. The name of the backward reference can not be found in the UML class model. Instead, it is derived from class and reference names using the following algorithm:

- the name of the backward reference is the name of the target class (i.e. source class of the original reference) with the first letter changed to lower case
- if this would result in a name clash because the class already contains another reference or attribute with the same name, the name of the forward reference is appended to the reference name, separated by an underscore

For example, in AR 4.0, the reference from `SwComponentType` to PortPrototype is named "port". The reverse reference is named "swComponentType" (name of the class with first letter changed to lowercase).

On the other hand a `RunnableEntity` has a reference called "dataSendPoint" pointing to class `VariableAccess`. The reverse reference is named "runnableEntity_dataSendPoint" (name of the class with first letter lowercase and with the name of the original reference appended). The reason is that `VariableAccess` is referenced from `RunnableEntity` in different roles and thus using only "runnableEntity" as the reverse reference name would not be unique.

Note that the multiplicity of reverse references is usually unbounded. The only exception are reverse references of containment relationships (diamond notation in UML), which have upper bound 1 (an element can only be contained by one other element at a time).

Reverse references are also useful while changing a model. The modeling

framework ensures that bidirectional references stay in sync while the model is being changed. So instead of adding a port to a software component type, the software component may also be assigned to the reverse references:

```
MM::Flat::PPortPrototype.new(:shortName => "port4", :swComponentType =>
   swc)
```

8.6 Model Navigation

Models are typically navigated using the feature read methods as described above. Beyond that, the modeling framework provides a convenient way to concatenate read method calls: Even if a method returns an Array (for multiplicity N), a further method may be chained. In this case, the framework will call the method on every element contained in the Array and build the union of all result sets. The following example retrieves all datatypes used by a `SenderReceiverInterface`:

```
intf.dataElement.type
```

In this example, "dataElement" returns an Array of `VariableDataPrototypes`, each of which has a "type" reference. In the expression above, the "type" reference is evaluated for each `VariableDataPrototype` and the resulting types are unified into a single Array which forms the expression result.

Another handy feature for model navigation are Ruby's built-in Array methods in combination with Ruby blocks: The "select" method allows to filter the contents of an Array for elements satisfying a certain constraint. The constraint itself is expressed as a Ruby block which receives each element as a block argument and includes this element in the result set by returning true while excluding it by returning false. The "collect" method also takes a block with one argument and builds a new Array consisting of the block results. As an example, ports might be filtered for those implementing a `SenderReceiverInterface` before the above expression is applied. As a little extra challenge, `PPortPrototypes` have a reference named "providedInterface" whereas `RPortPrototypes` have a reference named "requiredInterface".

```
swc.port.
  collect{|p| p.is_a?(MM::Flat::PPortPrototype) ? p.providedInterface : p
     .requiredInterface}.
  select{|i| i.is_a?(MM::Flat::SenderReceiverInterface)}.dataElement.type
```

This expression returns the set of all datatypes used by all `SenderReceiverInterfaces` implemented by ports of `swc`. The second line

is a call to "collect" which creates a new Array consisting of the interfaces implemented by either a `PPortPrototype` or a `RPortPrototype`. The third line selects only the `SenderReceiverInterfaces` from that list and navigates to the data elements and datatypes as explained above.

8.7 Loading and Storing Models

Instead of manually creating a model, existing ARXML files may be loaded. The only prerequisite is that the metamodel version of the ARXML files must match the metamodel version loaded using `MetamodelLoader` (see Loading Metamodels. The `ModelLoader` provides a "load" method, which takes a file pattern as an argument. The following example loads all ARXML (.arxml) files contained in the current directory and its subdirectories.

```
require 'esr/autosar/model_loader'

model = ESR::Autosar::ModelLoader.load("**/*.arxml")
```

The loaded model may be analysed or modified using the model API as described above. In case it has been modified, the changes may be written back the original files using the `ModelStorer`:

```
require 'esr/autosar/model_storer'

ESR::Autosar::ModelStorer.store(model)
```

If models are very large, loading may take some time even though the model files are cached. This can be tedious if several scripts need to read the same model (e.g. in a batch file) or if users want a really short response time (e.g. in scripts which query the model). The longer model loading time is mainly due to the fact that the caches must be checked for changes and that cross file references must be resolved.

Model loading can be made a lot faster by using the AUTOSAR library's model dump feature. By calling the `ModelStorer` "store_dump" method, a whole model can be written into one large file. Later on, the model can be reloaded by calling the `ModelLoader` "load_dump" method. The following snippet shows how a model is loaded using the regular load mechanism and then stored using the dump functionality. The dump file is named "test.dump" in this case.

```
require 'esr/autosar/metamodel_loader'
require 'esr/autosar/model_storer'

mm = ESR::Autosar::MetamodelLoader.load("4.0.3")
model = ESR::Autosar::ModelLoader.load(mm, "**/*.arxml")
```

```
ESR::Autosar::ModelStorer.store_dump(model, :filename => "test.dump")
```

Once the dump file exists, it can be reloaded a lot faster using the following code:

```
require 'esr/autosar/metamodel_loader'
require 'esr/autosar/model_loader'

mm = ESR::Autosar::MetamodelLoader.load("4.0.3")
model = ESR::Autosar::ModelLoader.load_dump(:filename => "test.dump")
```

Model load times will be around 5 to 10 times faster using the dump feature compared to regular cached loading. Note that there is no checking if the original model files have changed. The user is responsible for making sure that a new dump file is written every time the original model changes.

9 Synchronization Tool

Automate allows representing AUTOSAR XML in a more concise and readable form. This chapter describes Automate Sync Tool which facilitates the transformation from XML to text and vice versa.

The sync tool is a command line tool named `auto-sync`. If properly installed, the following line typed in a console should print the command line options:

```
> auto-sync -h
```

9.1 Synchronization Directions

`auto-sync` is used to transform AUTOSAR XML files into Automate text files and back. It provides two different subcommands, one for each direction:

- `a2t`: Transform from AUTOSAR XML to Automate text
- `t2a`: Transform from Automate text to AUTOSAR XML

9.2 Synchronization Modes

Synchronization of XML and text files can be done in three different modes:

- File mode: the synchronization result files are put next to the corresponding source files
- Directory mode: the synchronization result files are put in a separate directory next to the source directory
- Direct mapping mode: the synchronization result files can be put into arbitrary locations by specifying an explicit mapping

Consider the following file structure as an example input:

```
+project1/
  +arxml/
    +fileA.arxml
    +fileB.arxml
```

In file mode the source files can be selected for transformation separately. If fileA.arxml and fileB.arxml are chosen, the resulting file structure is:

```
+project1/
  +arxml/
    +fileA.arxml
    +fileA.atm
    +fileB.arxml
    +fileB.atm
```

In directory mode source files can by selected for transformation by directory. If "arxml/" is chosen, the resulting file structure is:

```
+project1/
  +arxml/
    +fileA.arxml
    +fileB.arxml
  +automate/
    +fileA.atm
    +fileB.atm
```

In direct mapping mode, the output depends on the mapping provided. Given the mapping `project1/arxml/fileA.arxml;build/file1.atm`, the input file `fileA.arxml` will be transformed into the output file `build/file1.atm`. See section Direct Mappings for details.

File, directory and direct mapping mode can be used at the same time: All files provided to `auto-sync` via a file pattern will be processed in file mode. All directories provided via a directory pattern will be process in directory mode. If a particular file is selected both via a file pattern and a directory pattern, the directory pattern is preferred, i.e. the target file location is determined according to the directory pattern. All files selected via a direct mapping will be processed according to the mapping. If the direct mapping

selects files which are also selected via file or directory mode, the direct mapping will override the output file.

In directory mode, any excess target files (files with the target file extension) for which no source files exist will be deleted. In direct mapping mode, all mappings for which the source file is not present will delete the target file. By default, the user will be asked to confirm the removal. With the command line option `--force-delete` this confirmation is skipped.

Automate won't delete excess target files outside of directories synchronized in directory mode, and the automatic deletion won't take place if an entire source directory is removed. The reason for that is that in those cases, Automate doesn't know which files are actually excess files. However, this can be changed by using the `--target-pattern` option. The target pattern should be built just like the source pattern, but it should find target files instead of source files. Also in contrast to source patterns, multiple target patterns must be separated by commas instead of whitespace. In a setup where commands for both directions of synchronization exist, the source pattern of one command can be used as the target pattern of the other one and vice versa. Automate will try to delete all files found by the target pattern unless they are recreated by the actual synchronization.

During synchronization, Automate will check if the target files have been modified manually and ask for user confirmation to overwrite changes. With the command line option `--force-overwrite` this confirmation is skipped. The check only works if the files were already synchronized before in either direction, i.e. they have been cached. Without a cache any existing target files will be overwritten.

9.3 File/Directory Patterns

Note that both, file and directory selections can be done by means of so-called glob patterns. Glob patterns can contain single asterisks (*) representing single file/directory wildcards and double asterisks (**) representing directory wildcards which match any directory and all its subdirectories. Here are some examples:

```
project1/arxml/*.arxml

=> project1/arxml/fileA.arxml
   project1/arxml/fileB.arxml

**/*.arxml
```

```
=> project1/arxml/fileA.arxml
   project1/arxml/fileB.arxml

*/arxml

=> project1/arxml
```

It's important to prevent the command shell from expanding the patterns before they get into Automate. In most shells, this can be achieved by surrounding the pattern with quotation marks, e.g. `"**/*.atm"`. For the `--target-pattern` option, this can be achieved by using an equal sign (=) after the option, e.g. `--target-pattern=**/*.atm`.

9.4 Naming Schema

In file mode, source and target files are expected to have special file extensions. By default, AUTOSAR XML files must have the extension `.arxml` and Automate text files must have the extension `.atm`. `auto-sync` will complain if the selected source files do not follow this rule. The target file name will be derived from the source file name by replacing the source file extension by the target file extension. The default extension may be adapted by means of the command line options `--arxml-ext` and `--text-ext`. Note that the file extensions should be specified without a leading dot but may contain dots by themselves.

In directory mode, source and target directories are expected to have special names. By default, directories containing AUTOSAR XML files must be names `arxml` and directories containing Automate text files must be names `automate`. The target directory path will be derived from the source directory path by replacing the last path segment with the target directory name. The default directory names may be adapted by means of the command line options `--arxml-dir` and `--text-dir`.

Since projects may use AUTOSAR XML files with different extensions at the same time, Automate allows specifying several XML file extensions with the `--arxml-ext` option. On the text side however, there is only one file extension allowed. In order to be able to recreate the original XML file extension after a conversion to text and back, `auto-sync` remembers the original file extension as an annotation to the `AUTOSAR` root element. It will do so only for the second and following XML file extensions, not for the first one specified. When no annotation is present in an Automate text file, the first XML file extension specified will be used when converting from text to XML.

In case there are several file extensions specified with the `--arxml-ext` option and there are several files in a certain location which differ only in one these extensions, `a2t` converts only the file with the extension that comes first in the extension list. When there are several files differing only in the file extension selected by the `--target-pattern` option, either none or all of the files will be deleted depending on if the source file is present or not.

9.5 Direct Mappings

In direct mapping mode, the user provides a mapping of source files to target files. In the simplest case this is a mapping from a single source file to a single target file. File mappings are provided on the command line just as file/directory patterns are in the other modes. The mapping is done in the following format

```
<arxml file>;<atm file>
```

This means that independent of the direction of transformation (`a2t` or `t2a`), the ARXML file is always specified first followed by semicolon (;) and the ATM file.

The fixed order in the mapping allows to use the same mapping specification for both directions of transformation. This is particularly useful if the mappings are specified via argument files (see section Argument Files).

9.6 Caching

Automate employs caching in order to speed up model loading and to avoid unnecessary transformations. Cache information is stored within each folder containing either AUTOSAR XML or Automate text files in a sub folder named `.mcache`. The cache is built up during the first model load and transformation. Once the cache is built up, any successive loading or transformation is significantly faster.

The cache is protected with checksums over the cache itself and any dependency files. If the cache is corrupted it will be invalidated and rebuilt.

The cache is also rebuilt when it is accessed by an Automate tool version for which the internal cache format changed. When using different versions of the Automate tool (or other tools based on the AUTOSAR lib) on the same model files, it's possible that the cache is invalidated over and over again. In order to

minimize this effect the user should try to use the same version or at least "similar" versions.

When working with a Version Control System (VCS), it's recommended to not check the cache files into the repository. This avoids differences appearing in both the actual model and the cache. When the model is updated from the VCS, the local cache is updated for the files that have change and is reused for the files which haven't. This way model loading and transformation is fast even after a model update.

9.7 Changed Only Mode

Although Automate introduces minimal differences when converting XML to text and back, sometimes it's desirable to not touch an AUTOSAR XML file at all. This could be in a situation where XML comments or whitespace of a particular file need to be preserved. In oder to still allow users to make changes using Automate, `auto-sync` provides a special mode which writes only those XML files which have been edited as Automate text files. This way only the files actually edited by the user are rewritten whereas the rest of the XML files remain unchanged.

Changed only mode can be selected by means of the command line option `--changed-only`. This option is only meaningful to the text to XML transformation, it doesn't affect the XML to text conversion. In order to check if a file was changed on the text side, Automate relies on its cache which is built during the XML to text transformation. When text files are transformed into XML files, only those XML files are written where the source file has changed according to the checksum stored in the cache or where no cache exists. Writing ARXML files when no cache exists ensures that new or renamed files will be synchronized properly. If the cache is corrupted, the respective XML files will not be written.

This has an important implication: If changed only mode is to be used, the cache files must not be removed or modified. At least not as long as there are changes on the text side which still need to be synchronized back to the XML side. In oder to ensure this, the best way is to do an XML to text synchronization right before the intended change and a text to XML transformation right after the change.

In particular, changes on the text side must be synchronized back to the XML side before an update of Automate is installed. The reason is that Automate updates always invalidate the existing cache. In this case the information

about which file has changed would be lost.

9.8 ARXML DEST Attributes

References in ARXML files contain an XML attribute named "DEST" holding the XML name of the reference's target class. This information is redundant in most of the cases, since the target element itself expresses the target class. In cases where reference targets are not available, the "DEST" attribute could give a hint about the intended target class.

Automate text files don't explicitly hold the information found in the "DEST" attributes, thus making the files easier to read and write. When the text files are converted back to ARXML, the values of the "DEST" attributes are recreated from the reference target elements. When there is no target element because a reference could not be resolved, Automate will try to reconstruct the "DEST" attribute by means of the existing ARXML files: if the target file of a text to ARXML transformation already exists, the "DEST" values found in that file will be used. If no such file exists, Automate will fall back to using the first non-abstract class possible as a reference target.

In order to avoid that the values of the "DEST" attributes are changed by Automate, it's recommended to either keep the original ARXML files in place or to avoid unresolved references. Note that Automate will automatically correct the "DEST" attributes on the next synchronization run where references resolve properly.

9.9 Command Line Options

This section describes the command line options of `auto-sync`. This information is also available in the help message (`-h`).

`-h, --help` Show the help message.

`-m METAMODEL` Set the metamodel version. Supported metamodel versions include: `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.03`, `17.10`, `18.03`, `18.10`, `19.03`, `19.11`, `20.11`

`-l LOGLEVEL` Set the log level. Supported log levels: `debug`, `info`, `warn`, `error`, `fatal`

`--target-pattern PAT1,PAT2` Pattern or pattern list for finding existing target files which are subject to deletion unless they are recreated during synchronization. The format is exactly the same as the one of the source patterns, except that multiple patterns are separated by comma instead

of whitespace. Usage of an equal sign (=) after the option is recommended as it will prevent pattern resolution by most of the command shells.

--force-delete Don't ask for confirmation before deleting excess target files.

--force-overwrite Don't ask for confirmation before overwriting manually modified target files.

--no-progress Don't output progress information while loading, storing and during transformation. This is useful for batch conversions which write into a log file (otherwise the log file is flooded with progress information lines).

--arxml-dir DIR Select the AUTOSAR XML directory name. Default "arxml".

--arxml-ext EXT1,EXT2,EXT3 Select the AUTOSAR XML file extensions. Default "arxml".

--text-dir DIR Select the Automate text directory name. Default "automate".

--text-ext EXT Select the Automate text file extension. Default "atm".

--no-short-refs Switch off short references (references shortened by means of scope information).

--no-checks Switch off model validation before synchronization.

--changed-only Activate changed only mode. Only meaningful for the text to XML transformation.

--write-bom Write UTF-8 BOM into Automate text files. This can be useful for making sure text editors write Automate text files using the UTF-8 encoding.

--macro-path PATH Specify a macro directory or directory pattern. All Automate text files within this directory/directories will be treated as macro definitions.

--def-path PATH Specify a ECUC definition directory or directory pattern. All ARXML files within this directory/directories will be treated as ECUC definitions.

--ext-dir DIR1,DIR2,DIR3 Specify extension directories which will be searched recursively for extension (.rb) files.

--splittable Support AUTOSAR splittable elements.

--use-gem GEM1,GEM2,GEM3 Expert only option allowing to use patch Ruby gems.

--no-ruby-check Expert only option allowing to use different Ruby versions.

--ensure-native Expert only option to verify that native C extensions are loaded.

-v, --version Print version information. Includes versions of all Ruby gems

used.

9.10 Argument Files

All command line options and input/output file names may be specified via one or more argument files. Argument files are plain text files which contain the arguments separated by whitespace (space, tab or newline). They can be provided on the command line by preceding the file name with an "at" (@) character. The content of each argument file will be "expanded" on the command line at the position of the specified argument file name.

Consider the following example invocation of `auto-sync`:

```
auto-sync -m 4.0.3 a2t file1.arxml file2.arxml
```

This is equivalent with the following invocation, given that the two example argument files `options_file` and `arxml_list_file` are present and contain the options and file names as specified below:

```
auto-sync @options_file a2t @arxml_list_file
```

Content of `options_file`:

```
-m 4.0.3
```

Content of `arxml_list_file`

```
file1.arxml
file2.arxml
```

10 Check Tool

The Automate command line tool `auto-check` allows running model checks without starting a synchronization. If properly installed, the following line typed in a console should print the command line options:

```
> auto-check -h
```

10.1 Input Files

The check tool operates solely on ARXML files. Please transform Automate text files to ARXML using `auto-sync` if needed.

The input file pattern format is compatible with `auto-sync`. This means that both, files and directories may be given, either directly or via patterns. In case of directories, the tool will find the ARXML files within the directory recursively by using the file extensions specified with the `--arxml-ext` option.

10.2 Checks

`auto-check` runs all the built-in checks plus any custom checks defined via the check extension language. See Check Extensions for details about how to define custom checks.

10.3 Command Line Options

This section describes the command line options of `auto-check`. This information is also available in the help message (`-h`).

`-h, --help` Show the help message.

`-m METAMODEL` Set the metamodel version. Supported metamodel versions include: `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.03`, `17.10`, `18.03`, `18.11`, `19.03`, `19.11`, `20.11`

`-l LOGLEVEL` Set the log level. Supported log levels: `debug`, `info`, `warn`, `error`, `fatal`

`--no-progress` Don't output progress information while loading. This is useful for batch conversions which write into a log file (otherwise the log file is flooded with progress information lines).

`--ext-dir DIR1,DIR2,DIR3` Specify extension directories which will be searched recursively for extension (.rb) files.

`--arxml-ext EXT1,EXT2,EXT3` Select the AUTOSAR XML file extensions. Default "arxml".

`--splittable` Support AUTOSAR splittable elements.

`--use-gem GEM1,GEM2,GEM3` Expert only option allowing to use patch Ruby gems.

`--no-ruby-check` Expert only option allowing to use different Ruby versions.

`--ensure-native` Expert only option to verify that native C extensions are loaded.

`-v, --version` Print version information. Includes versions of all Ruby gems used.

10.4 Argument Files

See section Argument Files of the `auto-sync` command.

11 Diff Tool

The Automate command line tool `auto-diff` allows comparing ARXML files using a textual view as defined by the Automate textual language. If properly installed, the following line typed in a console should print the command line options:

```
> auto-diff -h
```

`auto-diff` is not a diff tool by itself but rather just converts the ARXML files to text files and delegates all the diffing work to an existing diff tool which the user already has installed.

11.1 Usage

In order to compare two ARXML files, both files should be specified on the command line. In the simplest case this looks like this:

```
> auto-diff fileA.arxml fileB.arxml
```

If the metamodel is not specified, the tool will try to auto-detect the metamodel version. If no diff tool is specified (`--diff-tool`) the standard `diff` command will be used, which is expected to be in the system path (e.g. in a MinGW installation on Windows).

`auto-diff` is especially useful when ARXML is checked into a version control system for viewing the differences in Automate text syntax. For this to work, `auto-diff` should be configured as an external diff tool in the respective version control system client.

```
> auto-diff --diff-tool "/path/to/diff/tool %file1 %file2" fileA.arxml
   fileB.arxml`
```

The full command line of the diff tool to be used should be given to the `--diff-tool` option. Within the string, the special placeholders `%file1` and `%file2` will be replaced by the actual text files which `auto-diff` creates by converting the input files.

By default, `auto-diff` doesn't change the input model during conversion in any way. However, it can be useful to sort model elements in order to get better diff results: If a model element was added to a list of existing elements but also the order of the existing elements was changed, the actual difference, i.e. the addition of the single element, will be hard to see.

There are different sorting options available: * `-o` sorts elements by shortName * `-u` sorts elements by uuid. When both `-o` and `-u` are active, sorting will be done by uuid first and then elements without a uuid will be sorted by shortName. * `-f` flattens the package hierarchy before sorting. This is useful for properly tracking elements which moved from one package to another one.

Instead of running the external diff tool right away, the option `-i` starts `auto-diff` in interactive mode. This opens a little window and allows users to play with the sorting options and even show the diff of the original ARXML. With the `-a` option, the external diff tool isn't started until the user makes a choice in interactive mode by pressing a button.

11.2 Command Line Options

This section describes the command line options of `auto-diff`. This information is also available in the help message (`-h`).

`-h, --help` Show the help message.

`-m METAMODEL` Set the metamodel version, this is optional. If not specified, the tool will try to detect it automatically. Supported metamodel versions include: `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.03`, `17.10`, `18.03`, `18.10`, `19.03`, `19.11`, `20.11`

`-l LOGLEVEL` Set the log level. Supported log levels: `debug`, `info`, `warn`, `error`, `fatal`

`-i, --interactive` Run in interactive mode. There will be a little Window which allows users to change diffing options interactively.

`-a, --ask-first` In interactive mode, wait for the user to start the first diff by clicking a button.

`--diff-tool CMD` The full command line of the diff tool to start. Use `%file1` and `%file2` to refer to the converted input files. Default: `diff %file1 %file2`

-o, --order-by-name Order contained elements by shortName.

-u, --order-by-uuid Order contained elements by AUTOSAR uuid. When combined with –order-by-name, uuid takes precedence. Use with –flatten to order independent of the package structure. This allows to track elements which moved from one package to another.

-f, --flatten Flatten the package hierarchy. This is useful for tracking elements which moved from one package to another one when used in combination with the –order-by-uuid option. The original package name will still be available for reference in an annotation for each element.

-x, --compare-arxml Compare ARXML files. This option disables the ARXML to text conversion and thus effectively deactivates `auto-diff`, which may be useful e.g. for debugging purposes.

--text-ext EXT Select the Automate text file extension, important to load macros. Default "atm".

--no-short-refs Switch off short references (references shortened by means of scope information).

--macro-path PATH Specify a macro directory or directory pattern. All Automate text files within this directory/directories will be treated as macro definitions.

--def-path PATH Specify a ECUC definition directory or directory pattern. All ARXML files within this directory/directories will be treated as ECUC definitions.

--ext-dir DIR1,DIR2,DIR3 Specify extension directories which will be searched recursively for extension (.rb) files.

--splittable Support AUTOSAR splittable elements.

--use-gem GEM1,GEM2,GEM3 Expert only option allowing to use patch Ruby gems.

--no-ruby-check Expert only option allowing to use different Ruby versions.

--ensure-native Expert only option to verify that native C extensions are loaded.

-v, --version Print version information. Includes versions of all Ruby gems used.

11.3 Argument Files

See section Argument Files of the `auto-sync` command.

12 Editor Back-End Service

Automate features editing support for Automate text files via editor plugins. The editor plugins are "thin" front-ends which don't know about the details of the Automate language and the metamodel used. Instead, they communicate with a back-end service via a local socket connection. The back-end service process loads the Automate model and provides the front-ends with any information they require.

The Automate editor back-end service is a command line tool name `automate-rtext-service`. If properly installed, the following line typed in a console should print the command line options:

```
> automate-rtext-service -h
```

12.1 Configuration File

When the user starts to edit an Automate text file in a front-end editor, the editor plugin looks for a file named `.rtext`. It starts searching in the directory containing the file being edited and then walks up the directory hierarchy until the file is found.

The `.rtext` file must contain a configuration for the particular file extension of the file being edited. The configuration content is the command line used for starting up the back-end service. Here is an example:

```
*.atm:
cmd /c automate-rtext-service -m 3.1.4 "."
*.atm40:
cmd /c automate-rtext-service -m 4.0.3 --text-ext atm40 "."
```

This `.rtext` file content specifies configurations for two different file extensions: `*.atm` and `*.atm40`. The file extension patters are provided in separate lines and need to be followed by a colon (`:`). It's also possible to specify several extensions in one line by separating them with commas. The command line which is invoked by the front-end plugin must be given in the next line. These two lines may be repeated in order to include several configurations of file extensions and associated command lines into the same `.rtext` file.

In the above example, the back-end service is started using the Windows console `cmd /c`. This is necessary since `automate-rext-service` is a batch file on Windows. This example also shows how an AUTOSAR 3.1.4 and 4.0.3 model can be used in the same directory. The only prerequisite is that the Automate text files must have different file extensions.

12.2 Model Workspace Selection

The `.rtext` together with the contained command lines defines the model workspace (i.e. which model files belong together): All Automate text files in the directory containing the `.rtext` file and its subdirectory may *potentially* belong to the model. However, only the directories passed to the `automate-rtext-service` command decide which files are actually part of the model.

`automate-rtext-service` can take a directory or directory glob pattern as its command line argument. It will search these directories for Automate text files having the text file extension (default: `.atm`, may be changed with the `--text-ext` options). All files found this way belong to the model. In the above example `.` selects the current directory and therefore all text files contained in this and all subdirectories. In a different situation, this could be restricted to only some sub folders.

12.3 Command Line Options

This section describes the command line options of `automate-rtext-service`. This information is also available in the help message (`-h`).

`-h, --help` Show the help message.

`-m METAMODEL` Set the metamodel version. Supported metamodel versions include: `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.03`, `17.10`, `18.03`, `18.10`, `19.03`, `19.11`, `20.11`

`-l LOGLEVEL` Set the log level. Supported log levels: `debug`, `info`, `warn`, `error`, `fatal`

`--text-ext EXT` Select the Automate text file extension. Default “atm”.

`--no-short-refs` Switch off short references (references shortened by means of scope information).

`--macro-path PATH` Specify a macro directory or directory pattern. All Automate text files within this directory/directories will be treated as macro definitions.

--def-path PATH Specify a ECUC definition directory or directory pattern. All ARXML files within this directory/directories will be treated as ECUC definitions.

--timeout TIMEOUT Specify the idle timeout in seconds after which the service will shut down. Default: 3600.

--ext-dir DIR1,DIR2,DIR3 Specify extension directories which will be searched recursively for extension (.rb) files.

--splittable Support AUTOSAR splittable elements.

--use-gem GEM1,GEM2,GEM3 Expert only option allowing to use patch Ruby gems.

--no-ruby-check Expert only option allowing to use different Ruby versions.

--ensure-native Expert only option to verify that native C extensions are loaded.

-v, --version Print version information. Includes versions of all Ruby gems used.

12.4 Argument Files

See section Argument Files of the `auto-sync` command.

13 Automate web UI

The Automate web UI tool `automate-wui` is a web-service for browsing AUTOSAR components compositions. In order to get a list of arguments and options supported use:

```
automate-wui -h
```

13.1 Input files

The web UI tool operates solely on ARXML files. Please transform Automate text files to ARXML using auto-sync if needed.

13.2 Command Line Options

`-h, --help` Show the help message
`-m METAMODEL` Set the metamodel version. Supported metamodel versions include: `3.1.4`, `3.1.5`, `3.2.1`, `4.0.3`, `4.1.1`, `4.2.1`, `4.2.2`, `4.3.0`, `17.10`, `18.03`, `18.10`, `19.03`, `19.11`, `20.11`
`-p PORT` Set the port for a web-service. Default: 9292
`-t, --splittable` Enable splittables support
`-b, --skip-browser-opening` Don't open browser automatically on startup
`-s, --public-server` Allow web-server accept external connections
`-V, --version` Print version information

13.3 Usage

First, you need to run `automate-wui` with metamodel and input files provided. Wait the tool to start and load the model, when backend is ready to serve you will see log message:

```
INFO : Serving at port 9292...
```

Now you could open `localhost:9292` page using your browser. You will see the main page of the service. On the left side there is a list of components, choose the component you would like to browse, and you will be able to see the given component composition on the right side (figure 1).

The diagram contains the selected component, other components which are connected to the components and the connectors between all these components. You could navigate the diagram with the left mouse button pressed. Also, with the same button you are able to change the components position on the screen. Hover your mouse cursor over the connection to see details. Use your mouse wheel to zoom the diagram in and out.

Left-click on the component selects it on the diagram. Press the "≡" button to get additional component info (figure 2).

Right-click on the component opens context menu (figure 3) which could be used to remove connected components from the diagram and pin components to position on the diagram.

Use the "⟳ Reload model" button in order to reload the model files without `automate-wui` service restart.

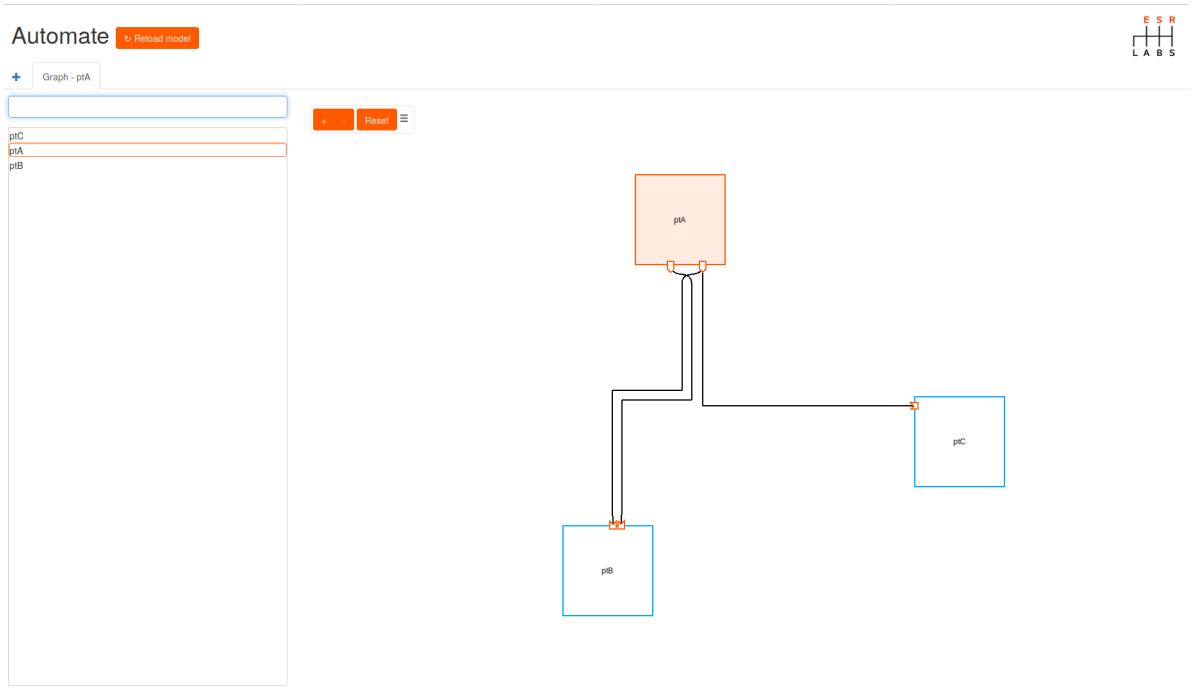

Figure 1: Automate web UI

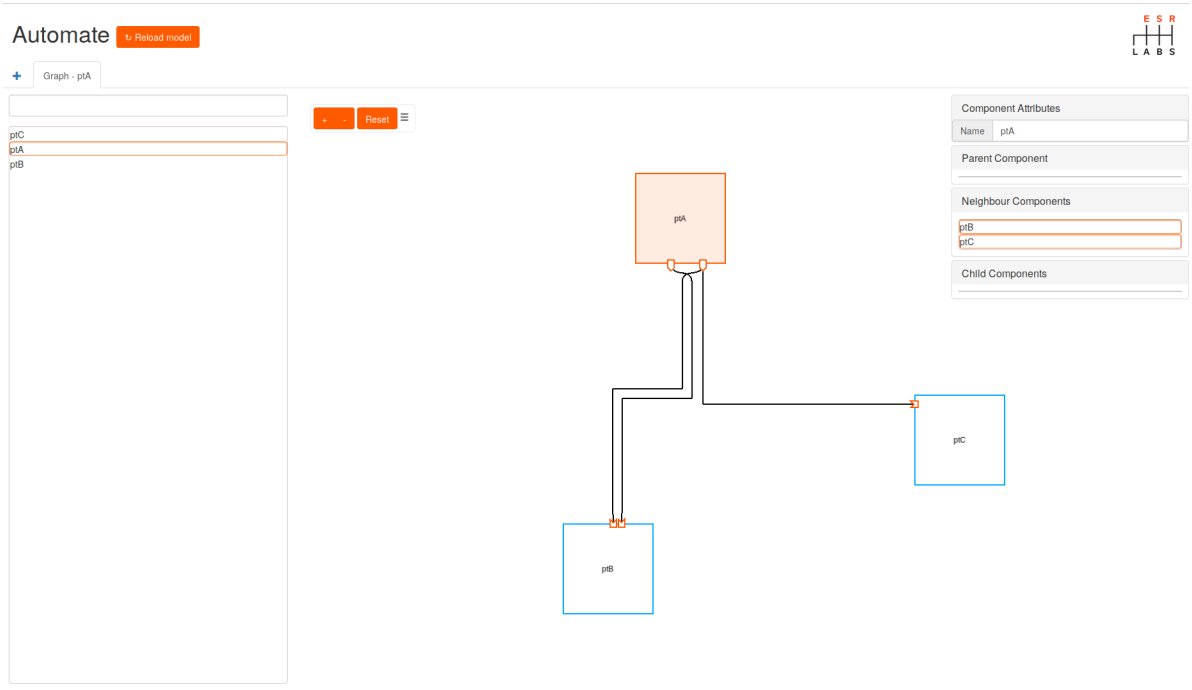

Figure 2: Component additional information

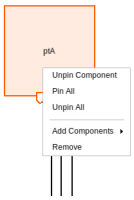

Figure 3: Component context menu

14 Scope Information

This chapter contains a complete list of metamodel references with attached scope information. Scope information is used for additional constraint checks and to shorten references.

14.1 AR 3.1.x / 3.2.x

Metaclass/Command	Reference
OperationPrototype	possibleError
ModeSwitchComSpec	modeGroup
ServerComSpec	operation
ParameterProvideComSpec	parameter
UnqueuedSenderComSpec	dataElement
QueuedSenderComSpec	dataElement
SenderAnnotation	dataElement
ClientComSpec	operation
ParameterRequireComSpec	parameter
QueuedReceiverComSpec	dataElement
UnqueuedReceiverComSpec	dataElement
ReceiverAnnotation	dataElement
CalibrationPortAnnotation	calprmElement
RTEEvent	startOnEvent
RunnableEntity	canEnterExclusiveArea
RunnableEntity	runsInsideExclusiveArea
RunnableEntity	sharedCalprmAccess
RunnableEntity	perInstanceCalprmAccess
RunnableEntity	readVariable
RunnableEntity	writtenVariable
SwcNvBlockNeeds	mirrorBlock
SwcNvBlockNeeds	defaultBlock
WaitPoint	trigger
OperationInvokedEvent_operation	pPortPrototype

Metaclass/Command	Reference
OperationInvokedEvent_operation	operationPrototype
DataReceivedEvent_data	rPortPrototype
DataReceivedEvent_data	dataElementPrototype
DataReceiveErrorEvent_data	rPortPrototype
DataReceiveErrorEvent_data	dataElementPrototype
ModeSwitchEvent_mode	rPortPrototype
ModeSwitchEvent_mode	modeDeclarationGroupPrototype
ModeSwitchEvent_mode	modeDeclaration
CalprmAccess_CalprmElementPrototype	portPrototype
CalprmAccess_CalprmElementPrototype	calprmElementPrototype
DataReceivePoint_dataElement	rPortPrototype
DataReceivePoint_dataElement	dataElementPrototype
ModeSwitchPoint_modeGroup	pPortPrototype
ModeSwitchPoint_modeGroup	modeDeclarationGroupPrototype
ServerCallPoint_operation	rPortPrototype
ServerCallPoint_operation	operationPrototype
DataSendPoint_dataElement	pPortPrototype
DataSendPoint_dataElement	dataElementPrototype
DataWriteAccess_dataElement	pPortPrototype
DataWriteAccess_dataElement	dataElementPrototype
DataReadAccess_dataElement	rPortPrototype
DataReadAccess_dataElement	dataElementPrototype
RoleBasedRPortAssignment	rPortPrototype
RoleBasedPPortAssignment	pPortPrototype
DelegationConnectorPrototype	outerPort
DelegationConnectorPrototype_innerPort	componentPrototype
DelegationConnectorPrototype_innerPort	portPrototype
AssemblyConnectorPrototype_requester	componentPrototype
AssemblyConnectorPrototype_requester	rPortPrototype
AssemblyConnectorPrototype_provider	componentPrototype
AssemblyConnectorPrototype_provider	pPortPrototype
ServiceConnectorPrototype_servicePort	serviceComponentPrototype
ServiceConnectorPrototype_servicePort	portPrototype
ServiceConnectorPrototype_applicationPort	componentPrototype
ServiceConnectorPrototype_applicationPort	portPrototype
SenderReceiverToSignalMapping_dataElement	softwareComposition
SenderReceiverToSignalMapping_dataElement	componentPrototype
SenderReceiverToSignalMapping_dataElement	portPrototype
SenderReceiverToSignalMapping_dataElement	dataElement
SenderReceiverToSignalGroupMapping_dataElement	softwareComposition
SenderReceiverToSignalGroupMapping_dataElement	componentPrototype
SenderReceiverToSignalGroupMapping_dataElement	portPrototype
SenderReceiverToSignalGroupMapping_dataElement	dataElement
SwCompToEcuMapping_component	softwareComposition
SwCompToEcuMapping_component	componentPrototype
SwCompToEcuMapping_component	targetComponentPrototype
SwCompToImplMapping_component	softwareComposition
SwCompToImplMapping_component	componentPrototype

Metaclass/Command	Reference
SwCompToImplMapping_component	targetComponentPrototype
SenderRecRecordElementMapping	recordElement
ArrayElement	arrayElement

14.2 AR 4.0.x / 4.1.x / 4.2.x / 4.3.x / 17.x / 18.x / 19.x / 20.x

Metaclass/Command	Reference
ClientServerOperation	possibleError
ModeSwitchSenderComSpec	modeGroup
ServerComSpec	operation
ParameterProvideComSpec	parameter
NonqueuedSenderComSpec	dataElement
QueuedSenderComSpec	dataElement
NvProvideComSpec	variable
SenderAnnotation	dataElement
ClientComSpec	operation
NvRequireComSpec	variable
ParameterRequireComSpec	parameter
QueuedReceiverComSpec	dataElement
NonqueuedReceiverComSpec	dataElement
ReceiverAnnotation	dataElement
ClientServerAnnotation	operation
NvDataPortAnnotation	variable
ParameterPortAnnotation	parameter
ModePortAnnotation	modeGroup
TriggerPortAnnotation	trigger
RunnableEntity	canEnterExclusiveArea
RunnableEntity	runsInsideExclusiveArea
WaitPoint	trigger
RTEEvent	startOnEvent
RVariableInAtomicSwcInstanceRef	contextRPort
RVariableInAtomicSwcInstanceRef	targetDataElement
POperationInAtomicSwcInstanceRef	contextPPort
POperationInAtomicSwcInstanceRef	targetProvidedOperation
RModeInAtomicSwcInstanceRef	contextPort
RModeInAtomicSwcInstanceRef	contextModeDeclarationGroupPrototype
RModeInAtomicSwcInstanceRef	targetModeDeclaration
RTriggerInAtomicSwcInstanceRef	contextRPort
RTriggerInAtomicSwcInstanceRef	targetTrigger
PModeGroupInAtomicSwcInstanceRef	contextPPort
PModeGroupInAtomicSwcInstanceRef	targetModeGroup
RModeGroupInAtomicSWCInstanceRef	contextRPort
RModeGroupInAtomicSWCInstanceRef	targetModeGroup
ROperationInAtomicSwcInstanceRef	contextRPort
ROperationInAtomicSwcInstanceRef	targetRequiredOperation
AsynchronousServerCallResultPoint	asynchronousServerCallPoint

Metaclass/Command	Reference
PTriggerInAtomicSwcTypeInstanceRef	contextPPort
PTriggerInAtomicSwcTypeInstanceRef	targetTrigger
AutosarParameterRef	localParameter
ParameterInAtomicSWCTypeInstanceRef	portPrototype
ParameterInAtomicSWCTypeInstanceRef	rootParameterDataPrototype
ParameterInAtomicSWCTypeInstanceRef	contextDataPrototype
ParameterInAtomicSWCTypeInstanceRef	targetDataPrototype
AutosarVariableRef	localVariable
VariableInAtomicSWCTypeInstanceRef	portPrototype
VariableInAtomicSWCTypeInstanceRef	rootVariableDataPrototype
VariableInAtomicSWCTypeInstanceRef	contextDataPrototype
VariableInAtomicSWCTypeInstanceRef	targetDataPrototype
ArVariableInImplementationDataInstanceRef	portPrototype
ArVariableInImplementationDataInstanceRef	rootVariableDataPrototype
ArVariableInImplementationDataInstanceRef	contextDataPrototype
ArVariableInImplementationDataInstanceRef	targetDataPrototype
PortAPIOption	port
RoleBasedPortAssignment	portPrototype
RoleBasedDataAssignment	usedPim
DelegationSwConnector	outerPort
RPortInCompositionInstanceRef	contextComponent
RPortInCompositionInstanceRef	targetRPort
PPortInCompositionInstanceRef	contextComponent
PPortInCompositionInstanceRef	targetPPort
VariableDataPrototypeInSystemInstanceRef	contextComposition
VariableDataPrototypeInSystemInstanceRef	contextComponent
VariableDataPrototypeInSystemInstanceRef	contextPort
VariableDataPrototypeInSystemInstanceRef	targetDataPrototype
SenderRecRecordElementMapping	applicationRecordElement [partial]
SenderRecRecordElementMapping	implementationRecordElement [partial]
IndexedArrayElement	applicationArrayElement [partial]
IndexedArrayElement	implementationArrayElement [partial]
ComponentInSystemInstanceRef	contextComposition
ComponentInSystemInstanceRef	contextComponent
ComponentInSystemInstanceRef	targetComponent

15 Changelog

1.0.0

- Initial release

1.0.1

- Added support for UTF-8 BOM in Automate text files
- Added –write-bom option to auto-sync
- Fixed ARXML instantiator exception when characters are found outside of the root XML element
- Fixed ARXML serializer to not write non-UTF-8 characters into XML files
- Fixed ARXML error message line numbers to be more precise and sort errors by line number

1.1.0

- Added AUTOSAR 4.1.1 metamodel
- Added –target-pattern option to check for excess target files [MATE-27]
- Added extraction of ARXML DEST attributes from target files [MATE-34]
- Added check to detect target file name clashes [MATE-16]
- Added –no-checks option to switch off checks during synchronization [MATE-32]
- Fixed –change-only mode to also synchronize new files [MATE-18]
- Fixed ARXML serializer to output self closing tags instead of empty tags [MATE-41]
- Fixed unresolved reference error messages to be more precise [MATE-20]
- Fixed duplicate name check to ignore elements without a shortname [MATE-19]
- Fixed file selection in case file and directory patterns are mixed [MATE-22]
- Fixed installer to not show warning on internet connection problems [MATE-15]
- Fixed installer to prevent installation of newer gems from the internet [MATE-14]

1.1.1

- Fixed output of user confirmation request before deleting a file [MATE-44]

1.2.0

- Added auto-check command line tool [MATE-33]
- Added support for check extensions [MATE-10]
- Added extension for controlling role label generation in Automate files [MATE-50]
- Added support for optional macro parts [MATE-53]
- Added support for arbitrary subtrees below a macro [MATE-48]

- Made macro context more specific, added context configuration option [MATE-17]
- Fixed exception when macro contains named element within an element with name placeholder [MATE-46]
- Fixed exception when scope reference points to wrong type [MATE-45]
- Fixed sporadic exception when scopes depend on one of their own features [MATE-60]
- Fixed exception on macro expansion on multiple elements in a one-role [MATE-55]
- Fixed exception when reducing a macro in a file with parse errors [MATE-54]
- Fixed exception on wrong macro names [MATE-57]
- Fixed order of macro arguments to obey order of definition [MATE-31]
- Fixed back-end crashes when files are deleted during loading [MATE-58]
- Fixed wrong sort order of prioritized arguments [MATE-61]
- Don't write mixed_content_order attribute if all child elements have the same type [MATE-52]
- Improved tool performance [MATE-62]

1.2.1

- Fixed exception when a macro matches a pattern without a subtree in an 'any' placeholder [MATE-69]

1.2.2

- Fixed incorrect warning message "any outside of macro definition" on MIData2 [MATE-73]
- Fixed reference completion not using scopes when extensions are loaded [MATE-77]

1.3.0

- Added AR4.0 and AR4.1 scope information [MATE-85]
- Added load/store support for models in a single dump file [MATE-84]
- Fixed stack level too deep exception in auto-sync [MATE-82]
- Fixed problems when a macro is used in another maro's 'any' slot [MATE-83]
- Fixed macro matching ignoring incoming references [MATE-78]
- Fixed exception when local relative reference uses an external scope [MATE-80]
- Fixed relative references not updated when scopes change [MATE-79]
- Improved performance of auto completion [MATE-28]
- Improved performance of text parser and serializer

- Improved performance of model file reload in editor backend

1.3.1

- Added Ruby 2.0 support [MATE-94]
- Fixed big integers being truncated during sync [MATE-90]
- Fixed exception when ARXML -REF element has no content [MATE-91]

1.3.2

- Fixed some references getting lost on a2t (e.g. SdgContents/sdx) [MATE-95]
- Fixed some unresolved refs not being reported (e.g. SdgContents/sdx) [MATE-96]
- Fixed AutosarSerializer crashing on excess mixed_content_order values [MATE-97]

1.3.3

- Added AR.3.2 scope support [MATE-99]
- Fixed macros not being reduced when root is referenced [MATE-98]
- Fixed macro scopes and short refs with AR4.x [MATE-100]

1.4.0

- Added auto-diff command [MATE-105]
- Added basic variant support [MATE-104]
- Support multiple input files differing only in the file extension [MATE-102]
- Support file extensions containing dots [MATE-106]
- Fixed t2a not updating references in ARXML files when only context elements of short references have changed [MATE-101]
- Fixed exception when auto-check is called with an ARXML directory [MATE-103]

1.4.1

- Fixed duplicate name errors being reported for inactive elements [MATE-109]
- Fixed inactive elements not being marked in element search results [MATE-111]
- Fixed scope check not ignoring inactive elements [MATE-112]
- Fixed missing backward references for elements duplicated in a variant [MATE-113]

1.4.2

- Added macro method support [MATE-120]
- Added missing scope information for SwcToImplMapping [MATE-115]
- Fixed exception when following backward reference on not-yet-saved elements when variants are present [MATE-121]
- Fixed complete and follow references not working for scoped references when multiple variants are present [MATE-119]
- Fixed wrong error message "missing target" if several inactive targets exist [MATE-117]

1.4.3

- Added support for direct file mappings to sync command [MATE-150]
- Added argument file support [MATE-151]
- Added AR 4.2.1 metamodel [MATE-152]
- Improved cache invalidation strategy to avoid unnecessary flushes [MATE-153]
- Fixed resolution of nested short refs [MATE-140]
- Fixed wrong scope information [MATE-134]

1.4.4

- Fixed file name case issue related to checking for duplicate filenames and deleting unmatched files [MATE-154]
- Fixed exception when trying to delete an unmatched file twice [MATE-155]

1.4.5

- Fixed references to elements getting lost if target is reduced to a macro without a name [MATE-156]
- Fixed exception when short refs are active in above case [MATE-156]

1.5.0

- Added support for line breaks in ATM files [MATE-124]
- Added support for comments on macros [MATE-74]
- Added overwrite check for modified files to auto-sync [MATE-123]
- Added option to ensure that native extension are loaded [MATE-132]
- Added AR 4.2.1 scope information
- Added AR 4.2.2 metamodel
- Added Ruby 2.1 support
- Fixed exception with t2a when some macros could not be expanded [MATE-137]
- Fixed exception when macro uses 'name' argument name instead of 'shortName' [MATE-135]

- Fixed unexpected overwriting of modified target files (–changed-only option and no source file change) [MATE-128]
- Fixed extension file processing order on Linux
- Fixed auto-diff stdout/stderr propagation on Linux
- Improved performance of ARXML writing
- Improved tool startup times

1.5.1

- Fixed ARXML output having wrong schema location/xmlns for AR4.1+ [MATE-162]

1.5.2

- Fixed exception on variant-inactive macro replaced elements [MATE-165]

1.5.3

- Switched Numerical type representation from Integer to String [MATE-188]

1.6.0

- Switched to Ruby 2.3 and 64-bit only
- Switched licensing
- Added automatic conversion of integer values in place of expected float values
- Added automatic conversion of integer and float values in place of expected string values
- Added scope information for ClientServerToSignalMapping and TriggerToSignalMapping

1.6.2

- Fixed licensing issues

1.6.3

- Fixed inner macro broken references issue

1.7.0

- Added automate-wui component
- Added AUTOSAR adaptive metamodels 17.03, 17.10
- Added support for Ruby 2.0-2.3 and 32-bit
- Implemented fallback to RubyEncoder licensing
- Added splittables support

1.7.1

- Improved handling of non-existing macro path
- Fixed FlexNet issue with Trusted Storage misconfiguration
- Implemented splittables support for automate-wui
- Added AUTOSAR adaptive 18.03 metamodel
- Added AUTOSAR 4.3.1 metamodel

1.7.2

- Fixed automate-rtext-service crash on macro change [MATE-226]
- Implemented macro argument default value support [MATE-207]
- Added AUTOSAR adaptive 18.10 metamodel
- Added AUTOSAR 4.4.0 metamodel

1.7.3

- Added AUTOSAR adaptive 19.03 metamodel
- Fixed licensing issues

1.7.4

- Fixed scope for RoleBasedPortAssignment [MATE-242]

1.8.0

- Switched to Ruby 2.7
- Interactive diff feature is now optional, diff works without fxruby gem

1.8.1

- Fixed adaptive metamodels (float attributes) [MATE-253]
- Added support for autosar_lib license feature
- Fixed model store issue
- Fixed frozen string issue [MATE-249]