

Check – Validation Language

Reference Documentation

Sven Efftinge, sven@efftinge.de, www.efftinge.de

PRIMARY SPONSORS



Informatik AG



Enterprise
Software Solutions



SECONDARY SPONSOR



Table of Contents

INTRODUCTION.....	3
CHECK FILES.....	3
IMPORT STATEMENTS.....	4
EXTENSION IMPORT STATEMENT.....	4
COMMENTS.....	4
CHECKS.....	4
WORKFLOW COMPONENT.....	5

Introduction

An important concept in model-driven software development is the concept of models. Another important concept is the idea of having domain-specific languages (DSLs) that are used to describe domain-specific stuff in a short and concise manner. *Check* is a domain-specific language that is specialized on model validation. Model validation, like any other validation, should happen as early as possible in a development process. It's best to directly integrate it in the model editor. If you have a long chain of components, for instance, containing model transformation steps, you may want to validate the state of your models several times. *Check* can be used to specify all of those constraints. It is based on the *oAW expressions framework*, so it can be used with all kinds of model representations as long as a suitable meta model implementation is available (or you implement it yourself).

Check files

A check file must be located in the Java class path of the used execution context. Its file extension must be *chk*. Let's have a look at a check file.

```
import my::metamodel;
extension other::ExtensionFile;

/**
 * Entities without a name are useless
 */
context my::Entity ERROR 'an entity must have a name!':
    this.name != null
;

/**
 *
 */
context my::Entity WARNING 'the name '+name+' is rather short!':
    this.name.length() > 3
;
```

This example shows all of the available statements. The structure of a check file is:

- import statements
- extension import statements
- checks

We will talk about each kind of statement one by one.

Import Statements

By using an import statement one can import name spaces of different types. (see expression reference documentation for a description of type names)

The syntax is:

```
import my::imported::namespace;
```

There are no static imports or any similar concept in *Check*. Therefore, you cannot write:

```
import my::imported::namespace::*; // WRONG!  
import my::Type; // WRONG!
```

Extension Import Statement

You can import extension files (see *Extend Reference*) using an extension statement. The syntax is:

```
extension fully::qualified::ExtensionFileName;
```

Comments

A check file can have single- and multiline comments.

The syntax for single line comments is:

```
// my comment
```

Multiline comments are written like this:

```
/* My  
multi line  
comment */
```

Checks

A *check* is a constraint for elements of a specific type. The syntax is:

```
context TypeName (ERROR|WARNING) msg-expression-using-this :  
constraint-expression-using-this  
;
```

If a given model element is of the specified Type, the constraint expression will be evaluated. If the constraint does not hold (the result of the evaluation is 'false'), the msg-expression is evaluated and the resulted String is stored in a so called *Issue*, together with the element on which the constraint was evaluated and the severity. The severity is specified using the keywords *ERROR* or *WARNING*.

The list of issues is collected and returned.

That's all!

Workflow component

If you want to invoke such a validation from out of a workflow, you should add a declaration of the *Check* component to your workflow description (see *Workflow Engine Reference*).

A typical configuration of the check component might look like this:

```
<component
  class="org.openarchitectureware.check.CheckComponent">
  <metaModel class="org.open...emf.EmfMetaModel">
    <metaModelPackage value="org.component.ComponentPackage"/>
  </metaModel>
  <checkFile value="example::Checks"/>
  <expression value="myModel.get(0).eAllContents"/>
</component>
```

The qualified name of the component is `org.openarchitectureware.check.Component`.

The check *Component* needs to know what kind of meta models your checks are based on. Therefore at least one dependency `metaModel` must be configured. In the example we configured an *EMF* meta model called `component`. If you have multiple interconnected models made up of different meta models (and meta meta models) you can specify more than one meta model.

The `checkFile` property expects a fully qualified name pointing to the check file containing all the checks.

The content of the `expression` property is evaluated (using the expressions engine, that is configured with all the meta models specified before). The result is converted to a list. The check engine checks each element contained in the list against the constraints found in the check file.

There are three additional configurable properties:

- `abortOnError`: If set to `true` the *Component* will throw an exception (to stop the workflow) if the issues list contains any errors. Default is `true`.
- `logIssues`: If set to `true` the component will log all issues contained in the resulted list (using `org.apache.commons.logging`). Default is `true`.
- `useWorkflowIssues`: If this is set to `true` the *Component* will store all found issues in the `Issues` object, handed in by the *workflow engine*. Default is `false`

About our Sponsors

itemis GmbH & Co. KG is an independent IT service company with an emphasis on consulting, coaching, and software development. Every single itemis expert provides many years of project experience and widespread knowledge about all object oriented and component based software development issues - especially in the field of model driven software development.

b+m is the founder of the openArchitectureWare project. The software was originally developed within the scope of many successful projects. b+m opened the software to the community in late 2003. All of the paradigms of Model-Driven Software Development including Product Line Engineering and not only the generator framework have become a key concept for product and customer specific development at b+m. b+m customers can make use of long time experience and substantial know-how in that field. Located at the company headquarters in Melsdorf/Kiel and at its subsidiaries in Berlin, Cottbus, Hamburg, Hanover and Kiel the b+m staff of 205 provides practical solutions for customized business applications, business process optimization and comprehensive architecture, project and quality management.

oose Innovative Informatik GmbH offers coaching, consulting and training in all themes about software engineering. The main focus of their activities are software architecture, requirements engineering and project-management. oose have first-hand information and experience, because our staff take actively part with others in actual trends, standards and innovations. Our staff support this and pass their know-how regularly on by writing and publishing books or being speaker at conferences, etc. Within the OMG oose collaborate actively on the specifications of the UML and also the SysML.

MID Enterprise Software Solutions GmbH is a leading supplier of optimized tool environments for standardsbased and model-centric software development as well as business process modeling. This includes professional tool consulting and tool components to build a complete tool environment using the best techniques and tool modules available - Architectural and Operational Excellence. With innovatorAOX, MID provides a holistic standard tool environment for object- and function-oriented software development as well as business process and data modeling to help its customers establish highly efficient processes and tool environments for software production, The unique and seamless integration of business process modeling into the development process ensures an unprecedented level of convergence of business requirements and implemented IT systems. Project members from all departments speak the same language and all requirements are clearly described.