qnx-buildfile-lang documentation 1.1.0

Table of Contents

1. Overview . . . 3
  1.1. Feature Matrix . . . 3
  1.2. Project Status & Compatibility . . . 3
  1.3. Resources . . . 3
2. Eclipse Plugin . . . 4
  2.1. Installation . . . 4
  2.2. Usage . . . 4
  2.3. Syntax Highlighting . . . 5
  2.4. Custom Validator . . . 5
3. Visual Studio Code Extension . . . 6
  3.1. Installation . . . 6
  3.2. Usage . . . 6
  3.3. Syntax Highlighting . . . 7
  3.4. Custom Validator . . . 7
4. Command Line Interface . . . 8
  4.1. Installation . . . 8
  4.2. Usage . . . 8
  4.3. Custom Validator . . . 9
5. Java Library . . . 10
  5.1. Installation . . . 10
  5.2. Key Concepts . . . 10
  5.3. Parsing Programmatically . . . 11
  5.4. Custom Validator . . . 12
  5.5. Variable Expansion . . . 14
  5.6. AST Walking . . . 14
6. Changelog . . . 15
  6.1. 1.1.0 . . . 15
    6.1.1. Content Assist for VSCode . . . 15
    6.1.2. Outline / Symbol Navigation . . . 15
    6.1.3. Quick Fixes for VSCode . . . 15
    6.1.4. Standalone Validator Example . . . 15
  6.2. 1.0.8 . . . 15
    6.2.1. Custom Validator JAR Support . . . 15
    6.2.2. Syntax Highlighting . . . 16

Giovanni Vergine, verginegiovanni@gmail.com

Date: Wed Feb 18 19:18:26 EET 2026

1. Overview

QNX buildfiles play a central role in many embedded and automotive systems. As systems scale, buildfiles naturally increase in complexity, which makes early validation, consistency, and policy enforcement increasingly important.

`qnx-buildfile-lang` is a set of Java-based tooling implemented using Xtext for parsing QNX Buildfiles.

This project was built to make QNX development smoother, more reliable, and more automatable for teams working on embedded or automotive platforms.

1.1. Feature Matrix

Feature	Eclipse Plugin	VSCode Extension	CLI	Java Library
Syntax validation	✔	✔	✔	✔
Syntax highlighting	✔	✔		
Content assist / auto-completion	✔	✔		
Outline view	✔	✔		
Custom validator JAR	✔	✔	✔	✔
Variable expansion				✔
Quick fixes	✔	✔		

1.2. Project Status & Compatibility

- Minimum JRE required: 17

1.3. Resources

- Source code: qnx-buildfile-lang
- Issue tracker: qnx-buildfile-lang/issues
- License: Apache License Version 2.0, January 2004

2. Eclipse Plugin

Like most of the Xtext-based projects, an Eclipse Feature is available and is deployed to Maven. Follow the steps below to understand how to get it, install it and use it.

2.1. Installation

Download the Eclipse Repository as a ZIP from Maven Central. Look for `qnx.buildfile.lang.repository-{Version}.zip` in Maven Central

E.g. latest version `1.1.0` can be found at https://repo1.maven.org/maven2/io/github/gvergine/qnx.buildfile.lang.repository/1.1.0/

Once downloded, follow the usual steps to install it in Eclipse:

- Help → Install New Software → Add → Archive…
- Select the ZIP file and proceed with installation
- Restart Eclipse

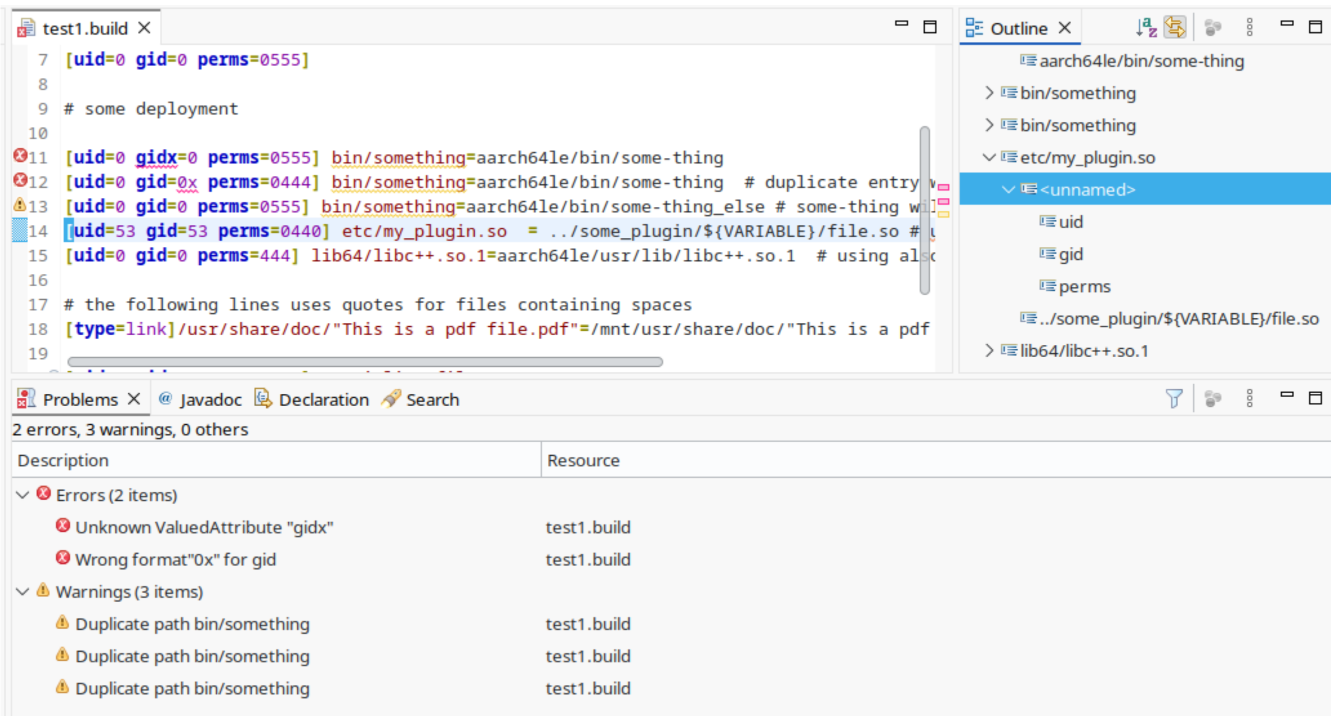
Even if Momentics is Eclipse-based, support for Momentics IDE is still work in progress.

2.2. Usage

Xtext Nature

When opening a QNX .build file for the first time in an Eclipse project, Eclipse will ask if to allow the Xtext Nature for such project. This is mandatory to enable the editor.

Syntax checking and highlighting should be available as in the following picture:



2.3. Syntax Highlighting

The Eclipse plugin provides semantic syntax highlighting with configurable colors. Highlighting styles can be customized under *Preferences → BuildfileDSL → Syntax Coloring*.

2.4. Custom Validator

The Eclipse plugin supports loading an external custom validator JAR at runtime. This allows teams to enforce additional project-specific rules without modifying the plugin itself.

To configure it:

- Go to *Preferences → BuildfileDSL → Custom Validator*
- Browse and select the custom validator JAR file
- Run *Project → Clean* to re-trigger validation with the custom rules

The JAR must contain a `Main-Class` manifest entry pointing to a class that extends `BaseDSLValidator`.

Clearing the path and cleaning the project restores default validation.

An example maven project that builds a custom validator is available here.

3. Visual Studio Code Extension

Xtext supports the Language Server protocol out of the box, that works very well with Visual Studio Code.

3.1. Installation

Download the extension as a `.vsix` from Maven Central. Look for `qnx.buildfile.lang.lsp-{version}.vsix` in Maven Central

E.g. latest version `1.1.0` can be found at https://repo1.maven.org/maven2/io/github/gvergine/qnx.buildfile.lang.lsp/1.1.0/

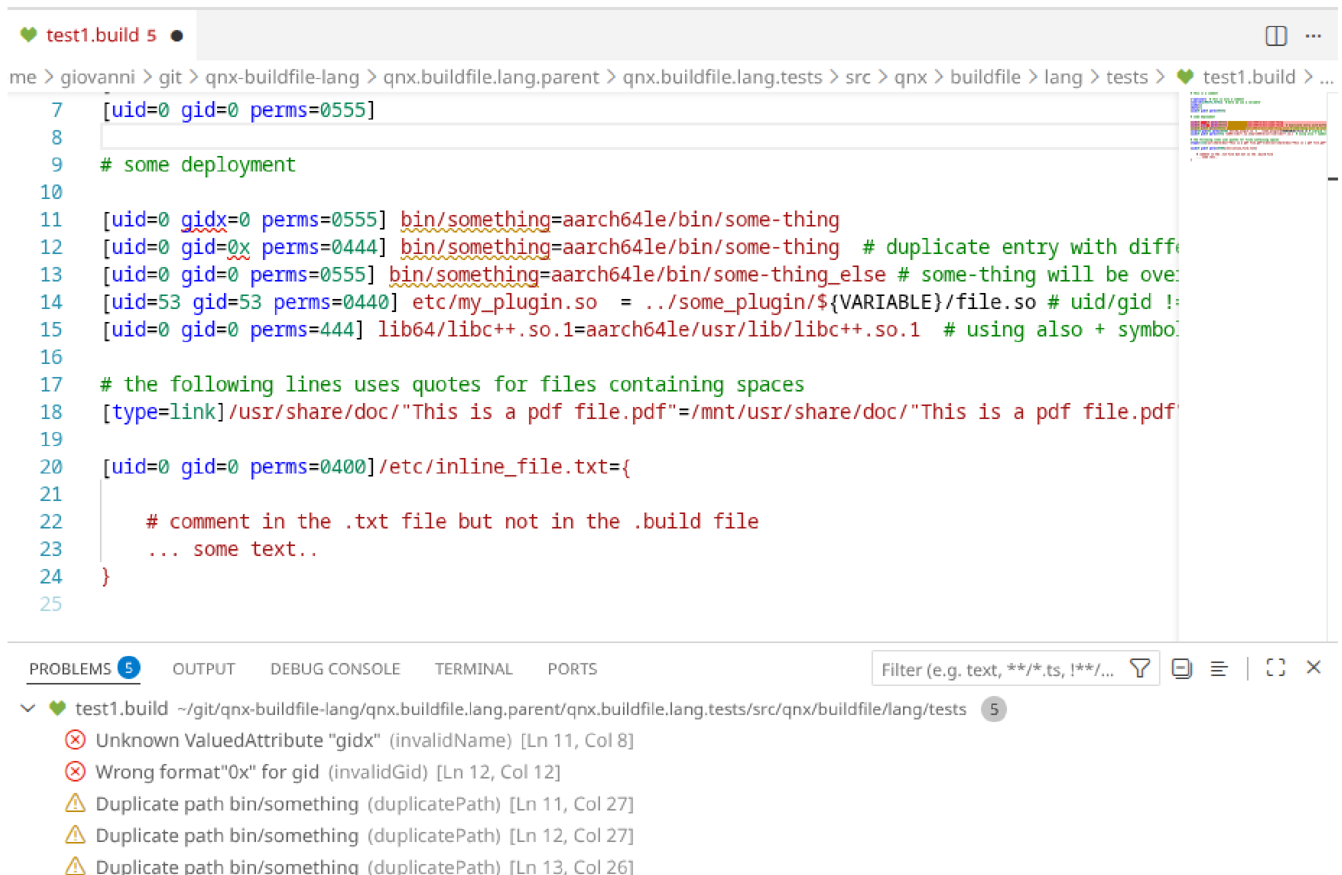
Once downloded, follow these steps for installing in vscode:

```
(optional) $ code --uninstall-extension gvergine.qnx-buildfile-lang
$ code --install-extension path/to/qnx.buildfile.lang.lsp-{version}.vsix
```

Then, restart vscode.

3.2. Usage

Syntax checking and highlighting should be available as in the following picture:



3.3. Syntax Highlighting

The VSCode extension provides syntax highlighting via a TextMate grammar combined with LSP semantic tokens. Colors follow the active VSCode theme.

3.4. Custom Validator

The VSCode extension supports loading an external custom validator JAR at runtime.

To configure it:

- Open VSCode Settings (Ctrl+,)
- Search for `qnx-buildfile-lang`
- Set *Custom Validator Jar Path* to the absolute path of the JAR file

The language server restarts automatically when the setting is changed. A manual restart is also available via the Command Palette (Ctrl+Shift+P → *QNX Buildfile: Restart Language Server*).

Clearing the path triggers an automatic restart and restores default validation.

An example maven project that builds a custom validator is available here.

4. Command Line Interface

A ready-to-use executable jar for command line validation is available as well. This is mostly useful for integration in a script or in a CI/CD pipeline.

It supports the same validation rules of the Eclipse Plugin, but they are printed on the standard error, and the exit code will be 0 only if no errors are found.

4.1. Installation

Download the command line standalone executable jar from Maven Central. Look for `qnx.buildfile.lang.cli-{Version}-shaded.jar` in Maven Central

E.g. latest version `1.1.0` can be found at https://repo1.maven.org/maven2/io/github/gvergine/qnx.buildfile.lang.cli/1.1.0/

4.2. Usage

Java Version

Make sure you use a JRE version 17+

You can specify multiple files on the same command line, e.g:

```
java -jar qnx.buildfile.lang.cli-1.1.0-shaded.jar -i=<inputs>[,<inputs>...] [-i
=<inputs>[,<inputs>...]]...
  -i=<inputs>[,<inputs>...]
         buildfile(s)
```

This example shows a succesful run:

```
$ java -jar qnx.buildfile.lang.cli-1.1.0-shaded.jar -i path/to/first.build -i
path/to/second.build
QNX Buildfile Validator version 1.1.0
Processing path/to/first.build
Processing path/to/second.build
Done - 0 failures
$ echo $?
0
```

Errors and warnings will be reported on standard error, and exit code will be 1 if any error occourred:

```
$ java -jar qnx.buildfile.lang.cli-1.1.0-shaded.jar -i path/to/error.build
QNX Buildfile Validator version 1.1.0
Processing path/to/error.build
```

```
ERROR at path/to/error.build:11: Attribute name "gidh" is not known
Done - 1 failure
$ echo $?
1
```

4.3. Custom Validator

The CLI supports loading a custom validator JAR via the `-c` flag:

```
$ java -jar qnx.buildfile.lang.cli-1.1.0-shaded.jar -c path/to/custom-validator.jar -i
path/to/file.build
```

The JAR must contain a `Main-Class` manifest entry pointing to a class that extends `BaseDSLValidator`.

An example maven project that builds a custom validator is available here.

5. Java Library

The standalone Java library allows you to build your own Java-based tooling. Typical use cases are:

- Extending the validation rules for enforcing additional policies
- Integration with Java-based environments (e.g. Tomcat, Java-based GUIs)
- Code generation (e.g. XML or CSV representation of the buildfile for further processing with non-Java tools)

5.1. Installation

You can get the latest version via Maven Central:

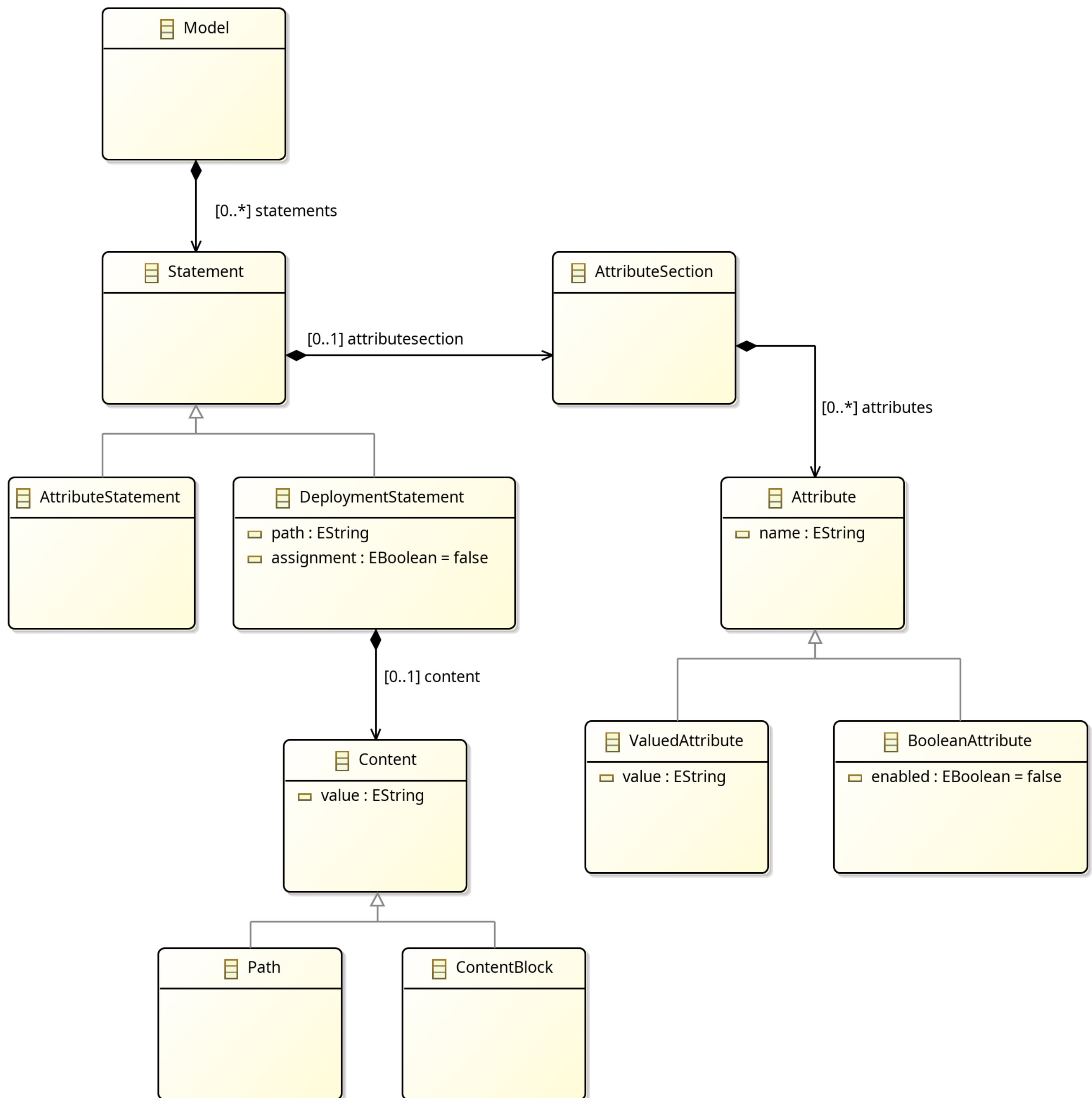
```
<dependency>
  <groupId>io.github.gvergine</groupId>
  <artifactId>qnx.buildfile.lang</artifactId>
  <version>1.1.0</version>
</dependency>
```

5.2. Key Concepts

When using a parser library, it is important first to understand the Abstract Syntax Tree that the parser will produce in memory. When the AST is known, it will be easy to access any part of the model and to validate it or generate output accordingly.

There are multiple ways to see and remember the AST:

- If you are familiar with the Xtext grammar notation, you can understand the AST by looking at it.
- Javadoc is generated and available via Maven as well.
- Referring to the following class diagram:



5.3. Parsing Programmatically

```
import qnx.buildfile.lang.utils.Parser;
import qnx.buildfile.lang.utils.ParsingResult;

// ...

Parser parser = new Parser();
File file = new File(filename);
ParsingResult parseResult = parser.parse(file);

if (parseResult.hasErrors()) {
    // ...
}
```

5.4. Custom Validator

You can write a custom validator by extending `BaseDSLValidator` and annotating check methods with `@Check`:

```
import org.eclipse.xtext.validation.Check;
import qnx.buildfile.lang.attributes.AttributeKeywords;
import qnx.buildfile.lang.buildfileDSL.Attribute;
import qnx.buildfile.lang.buildfileDSL.BuildfileDSLPackage;
import qnx.buildfile.lang.validation.BaseDSLValidator;

public class CustomValidator extends BaseDSLValidator
{
    @Check
    public void checkAttributes(Attribute attribute)
    {
        if (!AttributeKeywords.ALL_ATTRIBUTE_KEYWORDS
                .contains(attribute.getName()))
        {
            error("Attribute name \"" + attribute.getName() + "\" is unknown",
                    BuildfileDSLPackage.Literals.ATTRIBUTE__NAME,
                    "invalidName");
        }
    }
}
```

For example, the following custom validator adds a warning for duplicate paths:

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.eclipse.xtext.validation.Check;

import qnx.buildfile.lang.buildfileDSL.BuildfileDSLPackage;
import qnx.buildfile.lang.buildfileDSL.DeploymentStatement;
import qnx.buildfile.lang.buildfileDSL.Model;
import qnx.buildfile.lang.utils.Walker;
import qnx.buildfile.lang.utils.Walker.IWalker;
import qnx.buildfile.lang.validation.BaseDSLValidator;

public class CustomValidator extends BaseDSLValidator {

    private final static Walker walker = new Walker();

    @Check
    public void checkDuplicates(Model model) {
        Map<String,List<DeploymentStatement>> duplicates = new HashMap<>();
```

```java
        walker.walk(model, new IWalker() {
            @Override
            public void found(DeploymentStatement deploymentStatement)
            {
                String path = deploymentStatement.getPath();

                if (duplicates.containsKey(path))
                {
                    duplicates.get(path).add(deploymentStatement);
                }
                else
                {
                    List<DeploymentStatement> l = new ArrayList<>();
                    l.add(deploymentStatement);
                    duplicates.put(path, l);
                }

            };
        });

        duplicates.forEach((path, deployments) ->{
            if (deployments.size() > 1)
            {
                for (DeploymentStatement deployment : deployments)
                {
                    warning("Duplicate path " + path, deployment,
BuildfileDSLPackage.Literals.DEPLOYMENT_STATEMENT__PATH, "duplicatePath");
                }
            }

        });

    }
}
```

The custom validator JAR must include a `META-INF/MANIFEST.MF` with a `Main-Class` entry pointing to your validator class. It can then be used across all deployment modes: the Eclipse plugin (via preferences), the VSCode extension (via settings), and the CLI (via the `-c` flag).

To use the custom validator programmatically with the Java library, set the system property before creating the parser:

```java
System.setProperty("customValidatorJar", "/path/to/custom-validator.jar");
Parser parser = new Parser();
// ...
```

5.5. Variable Expansion

The library provides a `VariableSubstitutor` utility for resolving `${…}` variable references in the parsed model:

```
import qnx.buildfile.lang.utils.VariableSubstitutor;

// ...
final Map<String, String> envMap = System.getenv();
VariableSubstitutor vs = new VariableSubstitutor();
vs.substituteVariables(model, envMap);
```

5.6. AST Walking

The `Walker` utility provides a visitor-style API for traversing the parsed model. Implement only the `found()` methods you need:

```
import qnx.buildfile.lang.utils.Walker;
import qnx.buildfile.lang.utils.Walker.IWalker;

// ...
Walker walker = new Walker();
walker.walk(model, new IWalker() {
    @Override
    public void found(DeploymentStatement deploymentStatement) {
        System.out.println("Found deployment: " + deploymentStatement.getPath());
    }
});
```

6. Changelog

6.1. 1.1.0

6.1.1. Content Assist for VSCode

- **VSCode**: content assist now proposes boolean attribute names (after +/-), valued attribute names (with trailing =), and known values for attributes like type, autoso, compress, code, and data. Eclipse already had this; VSCode now matches the same proposal set.

6.1.2. Outline / Symbol Navigation

- **VSCode**: new document outline showing deployment statements and attribute statements as top-level entries, with individual attributes as children.
- **Eclipse**: matching outline tree with custom icons for each node type (attribute statements, deployments, boolean attributes, valued attributes).
- Both editors support breadcrumb navigation and Go to Symbol.

6.1.3. Quick Fixes for VSCode

- **VSCode**: quick fixes now suggest closest matching attribute names for typos using Levenshtein distance (e.g. permjs → perms). The first suggestion is marked as preferred for one-click apply. Eclipse already had this; the underlying suggestion logic is now shared between both editors.

6.1.4. Standalone Validator Example

New examples/standalone-validator project demonstrating the full library pipeline: parsing, ${VAR} substitution from environment variables, validation (standard or custom), and directory deployment analysis with file-level replacement suggestions. Supports --strict-vars (fail on unresolved variables), -W (fail on warnings), -e KEY=VALUE (extra variables), and -r (report file output).

6.2. 1.0.8

6.2.1. Custom Validator JAR Support

- **Eclipse**: new preference page under *Preferences → BuildfileDSL → Custom Validator* to configure the path to an external validator JAR. Changes take effect on *Project → Clean*.
- **VSCode**: new setting `qnx-buildfile-lang.customValidatorJarPath` in VSCode Settings. The language server restarts automatically when the setting changes. A manual *QNX Buildfile: Restart Language Server* command is also available in the Command Palette.
- **CLI**: refactored to use the same validation pipeline as Eclipse and VSCode. The `-c` flag continues to work as before.

6.2.2. Syntax Highlighting

- **VSCode**: added TextMate grammar for syntax highlighting — comments, attribute sections, paths, variable references, content blocks, and operators are now colored.
- **Eclipse**: added semantic highlighting with configurable colors. Styles can be customized under *Preferences → BuildfileDSL → Syntax Coloring*.
- Both editors now show consistent highlighting across the same language elements.