Examples for the ASSUME Static Code Analysis Tool Exchange Format

Introduction

About this document

This document is a guide to the ASSUME static code analysis tool exchange format examples `ExampleConfiguration.xml` and `ExampleReport.xml`.

Goals and design policies

In the first stage of designing the ASSUME SCA (static code analysis) tool exchange format, we aim to provide a common configuration and report format for SCA tools. To this end, the **configuration format** must allow the specification of analysis tasks as independent from the specifics of individual tools as possible. For example, a common means of specifying the interpretation of programs needs to be established. Also, the SCA tool configuration should facilitate the specification of machine-independent configurations, e.g. hiding differences in the concrete paths of source code files. As a secondary goal, the configuration should be specifiable in a modular way, facilitating the reuse of configurations. With the **report format**, we aim to establish a common hierarchy of check categories with accompanying check semantics, making results obtained from different tools directly comparable. Finally, the formats need to be extensible, allowing the addition of language support and analysis feature configuration options at a later stage.

For the format design, we therefore use a set of guiding policies:

- Common semantics, e.g. for checks, are defined as needed, but should be based on the semantics used by existing SCA tools where no conflicts arise.
- The format may allow tool-dependent configuration, but the volume of necessary tool-dependent configuration should be minimized. Users should be able to specify analysis tasks without any tool-specific configuration at all.
- The format may allow the specification of local configuration, e.g. file paths, but needs to provide means to the user to create configurations where any local settings are as isolated from the rest of the configuration as possible.
- The format should consist of lightweight, reusable structures. Language-dependent structures should be isolated and their integration into the format must allow the addition of new language-dependent structures without breaking backward compatibility.
- Configurations should be composed of small, reusable and extensible parts, which may be aggregated to form a complete SCA tool configuration.

Feedback

The format is work in progress, and feedback (to felix.kutzner@kit.edu) is highly welcome.

Notation

Within XML code, italic serif-typed text <<*X*>> is a placeholder for an element described in section X.

Configuration

In this section, we present most aspects of the configuration format's current state by example (see the file `ExampleConfiguration.xml`).

Top-level structure

A configuration consists of three major parts:

1. the *common configuration* (belonging to the global configuration) describing configuration items for all analysis tools and independent of machines executing the analysis tools,
2. the optional *tool-specific configuration* (also belonging to the global configuration) allowing users to configure analysis tool behaviour which cannot currently be specified in the common configuration,
3. the *local configuration*, which is intended to allow users to specify e.g. concrete directory replacements for the rsp. placeholders used in the other parts of the configuration.

For now, the configuration schema is designed with the usage of XInclude in mind: place the global and local configurations in separate files and create a configuration file by including both. Then, users of the configuration only need to supply a customized local configuration file. The high-level structure of a configuration file is given in the following XML snippet, with `asef` standing for "**A**SSUME **S**CA tool **e**xchange **f**ormat":

```
<?xml version="1.0" encoding="UTF-8"?>
<asef:Configuration xmlns:asef="http://todo.example.com/asef"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://todo.example.com/asef ASC3F.xsd">

<asef:GlobalConfiguration>
  <asef:CommonConfiguration>
       <<Meta information>>
       <<Hardware targets>>
       <<Language targets>>
       <<Source code modules>>
       <<Execution model targets>>
       <<Check targets>>
       <<Analysis tasks>>
  </asef:CommonConfiguration>
  <<Tool-specific configuration>>
<asef:GlobalConfiguration>

<<Local configuration>>

</asef:Configuration>
```

Global configuration

Meta information

Using the optional metadata element, users may store a description of the configuration file.

```
<<Meta information>>=
<Meta configurationName="Example configuration for the ASSUME SCA tool exchange format"
      version="1.0"
      description="This is a description of the example configuration.">
  <Maintainer>Felix Kutzner (KIT)</Maintainer>
</Meta>
```

Hardware targets

To keep configuration files modular, most bits of the configuration are stored in entities identifiable by name, which can then be combined to specify a concrete analysis configuration. We first present *Hardware targets*, which are descriptions of the hardware on which the code to be analyzed is intended to run:

<<Hardware targets>>=

```
<HardwareTargets>
  <HardwareTarget xsi:type="asef:HomogenousHardwareTarget" name="PPC32"
    endianness="big"
    unalignedDereferenceSupported="true"
    pointerSize="32"
    functionPointerSize="32"/>
  <HardwareTarget xsi:type="asef:HomogenousHardwareTarget" name="x86-64"
    endianness="small"
    unalignedDereferenceSupported="true"
    pointerSize="64"
    functionPointerSize="64"/>
</HardwareTargets>
```

Language targets

Language targets are named entities describing how to interpret source code expressed in a given programming language. Currently, the format only supports C language targets. However, the configuration format is designed with extensibility in mind: new language targets, e.g. for C++, may be introduced at a later stage without affecting existing configuration files.

<<Language targets>>=

```
<LanguageTargets>
  <LanguageTarget xsi:type="asef:CLanguageTarget" name="C99"
    signedEnums="true"
    signedBitfields="false"
    fpRoundingMode="downward"
    fpConstantRoundingMode="toNearest"
    enumType="int"
    inlineAssemblyHandlingMode="ignore"
    initializeStaticVariables="true"
    enableVolatile="true"
    standardRevision="C99">
      <Types>
        <Type name="char" size="8" alignment="8" atomic="false"/>
        <Type name="short" size="16" alignment="16" atomic="false"/>
        <Type name="int" size="32" alignment="32" atomic="false"/>
        <Type name="long" size="32" alignment="32" atomic="false"/>
        <Type name="long long" size="64" alignment="64" atomic="false"/>
        <Type name="float" size="32" alignment="32" atomic="false"/>
        <Type name="double" size="64" alignment="64" atomic="false"/>
      </Types>
  </LanguageTarget>
```

A C language *subtarget* extends another C language (sub)target with additional preprocessor definitions, include directory paths, and individual include files needing to be prepended to all C source code files which are analyzed using this subtarget. For example, this way compiler-specific header files may be prepended to the files under analysis. File and directory paths need to be specified as URIs. Within URIs, substrings matching `$_[A-Z]*_$` are placeholders for concrete paths specified in the local configuration. Include directories and files are used in the order

of their appearance within the language targets. Via the `insertionMode` attribute, it may be specified whether the list of include directories rsp. files needs to be prepended or appended to the subtarget parent's list of include directories rsp. files (if applicable).

```
  <LanguageTarget xsi:type="asef:CLanguageSubtarget"
    name="C99 with includes"
    superTarget="C99">
      <PreprocessorDefinitions overrideParentDefinitions="false">
         <Definition identifier="STATIC_ANALYSIS"/>
      </PreprocessorDefinitions>
      <IncludeDirectories insertionMode="append">
         <DirectoryURI>$_LIBINCLUDES_$</DirectoryURI>
      </IncludeDirectories>
      <IncludeFiles insertionMode="append">
         <IncludeFile path="$_SYSINCLUDES_$/compiler_sys.h" local="true"/>
      </IncludeFiles>
  </LanguageTarget>
```

The basic idea is to have a generic C language target and more concrete language targets via subtargets:

```
  <LanguageTarget xsi:type="asef:CLanguageSubtarget"
    name="LT for ExampleModule"
    superTarget="C99 with includes">
      <IncludeDirectories insertionMode="append">
        <DirectoryURI>$_EXAMPLEMODULEPATH_$/include</DirectoryURI>
      </IncludeDirectories>
  </LanguageTarget>

</LanguageTargets>
```

Note: The `LanguageTargets` list is not the only place in which language subtarget elements may occur. For example, within source code modules, language subtargets may be specified for individual files (extending the language target which would otherwise be used to interpret that file).

Source code modules

Source code module elements are named entities describing sets of files needing to be analyzed. Users may specify an optional root URI, relative to which relative `SourceFile` URIs are interpreted. Within source code modules, source files are identifiable by an ID. Source code module elements may also contain information about which parts of the source code needs to be stubbed.

<<Source code modules>>=

```
<SourceModules>
  <SourceModule name="ExampleModule" rootUri="$_EXAMPLEMODULESRC_$">
    <SourceFiles>
      <SourceFile uri="main.c" id="1"/>
```

Needing further preprocessor definitions and header files not visible to other source code files, the file `dodgycode.c` has an individual C language subtarget (extending the language target which would otherwise be used to interpret that file):

```
      <SourceFile uri="dodgycode.c" id="2">
        <LanguageTargetExtension>
          <CLanguageSubtarget superTarget="auto">
            <PreprocessorDefinitions overrideParentDefinitions="false">
              <Definition identifier="ADD_MACRO" expansion="x+y"/>
```

```
            <Definition identifier="KBD_PORT_ADDR" expansion="0x60"/>
            <Definition identifier="KBD_STAT_ADDR" expansion="0x64"/>
            <Definition identifier="KBD_CMD_ADDR" expansion="0x64"/>
          </PreprocessorDefinitions>
          <IncludeFiles insertionMode="append">
            <IncludeFile path="HideProprietaryCExtensions.h" local="true"/>
          </IncludeFiles>
        </CLanguageSubtarget>
      </LanguageTargetExtension>
    </SourceFile>
    <SourceFile uri="shadycode.c" id="3"/>
  </SourceFiles>
```

Suppose the module *ExampleModule* requires function stubs. Since the specification of stubs is language-dependent, the format offers an optional `RequiredCStubs` element within source code modules. There are three ways of requiring a C function to be stubbed: requiring a *visibility controlled* stub means that the stub generator should create a skeleton stub for each visibility-controlling macro identifier specified in the `VisibilityControllingSymbols` list, controlling their visibility via corresponding `#ifdef` directives. Requiring a *universal* stub means requiring a single stub skeleton to be generated, without the stub's visibility being controlled via `#ifdef` directives. Furthermore, *autogen* stubs should not be implemented in source code files, but be generated on-the-fly be the analysis tool. (Note that the analysis tool only needs to interpret *autogen* stubs; the other information may be used for manual stub implementation or by stub skeleton generators.)

Stubbed functions are identified via URIs. For the C programming language, the URI namespace `cstub` is used, whose structure we define as follows:

- Functions *func* having external linkage are described by cstub://globalscope/*func*.
- Functions *func* having internal linkage for the file with ID *fileID* within source code module *module* are described by cstub://filescope/*module*/*fileID*/*func*.

Stubs for functions with external linkage may be assigned to groups. Stub generators should place all stubs of a group into the same file.

```
    <RequiredCStubs>
      <VisibilityControllingSymbols>
        <CVisibilityControllingSymbol name="ENABLE_LLBMC_STUBS"/>
        <CVisibilityControllingSymbol name="ENABLE_ASTREE_STUBS"/>
      </VisibilityControllingSymbols>
      <CUniversalStub uri="cstub://globalscope/open" group="posix_io"/>
      <CUniversalStub uri="cstub://globalscope/read" group="posix_io"/>
      <CUniversalStub uri="cstub://globalscope/write" group="posix_io"/>
      <CVisibilityControlledStub
          uri="cstub://filescope/ExampleModule/2/read_from_kbd"/>
      <CAutogenStub uri="cstub://globalscope/malloc"/>
      <CAutogenStub uri="cstub://globalscope/free"/>
    </RequiredCStubs>
  </SourceModule>
```

For each stub URI, at most one corresponding stub entry may be present in a `RequiredCStubs` element.

A stub module generated using the stub specification given in *ExampleModule* might look like this:

```
  <SourceModule name="ExampleModule_stubs" rootUri="$_EXAMPLEMODULESTUBS_$">
    <SourceFiles>
```

All stubs of the group *posix_io* are implemented in the file `groups/posix_io.c`. The stubs implemented in a given source code file are listed within a `ImplementsStubs` element:

```
      <SourceFile uri="groups/posix_io.c" id="1">
        <ImplementsStubs>
          cstub://globalscope/open
          cstub://globalscope/read
          cstub://globalscope/write
        </ImplementsStubs>
      </SourceFile>
```

The stub *fancy_rng* was not assigned to a group, so it is placed in an individual file:

```
      <SourceFile uri="fancy_rng.c" id="2">
        <ImplementsStubs>cstub://globalscope/fancy_rng</ImplementsStubs>
      </SourceFile>
```

Finally, the static function *read_from_kbd* has a location corresponding to its stub URI:

```
      <SourceFile uri="filescope/ExampleModule/2/read_from_device.c" id="3">
        <ImplementsStubs>
          cstub://filescope/ExampleModule/2/read_from_kbd
        </ImplementsStubs>
      </SourceFile>
    </SourceFiles>
  </SourceModule>
```

Having a module *ExampleModule_stubs* implementing the stubs for *ExampleModule*, we can create a third module composed of the former two. To complicate things, suppose that our third module contains a file implementing the function *fancy_rng*, which is also implemented in the stub module.

```
  <SourceModule name="ExampleModule_joined">
    <SourceFiles>
      <SourceFile uri="someOtherFancyRNG.c" id="1"/>
    </SourceFiles>

    <RequiresModules>
```

We include all files from *ExampleModule*:

```
      <RequiresModule name="ExampleModule"/>
```

We also include all of *ExampleModule_stubs*, however excluding all files implementing the stub cstub://globalscope/fancy_rng:

```
      <RequiresModule name="ExampleModule_stubs">
        <ExcludingFilesProvidingStub uri="cstub://globalscope/fancy_rng"/>
      </RequiresModule>
```

Alternatively, we could have used an `<ExcludingFile module="..." id="..."/>` element to specify a file to be excluded from the inclusion. (Since the required module may require further modules, it is necessary to specify the module name as well as the file's identifier within that module.)

```
    </RequiresModules>
  </SourceModule>
</SourceModules>
```

Note: A language target extension may also be specified at source module level. If so, the language target used for a source file extends the lanugage target of the rsp. source module, which in turn extends the language target used for the analysis task.

Execution model targets

Execution model targets are named entities specifying how software is executed, ie. synchronous/asynchronous execution and entry points:

<<Execution model targets>>=
```
<ExecutionModelTargets>
  <ExecutionModelTarget xsi:type="asef:CSynchronousExecutionModelTarget"
                        name="ExampleExecModel">
    <EntryPoints>
      <EntryPoint>main</EntryPoint>
    </EntryPoints>
  </ExecutionModelTarget>
</ExecutionModelTargets>
```

Check targets

Check targets are named entities in which the user can configure requirements for checks, e.g. which runtime or MISRA checks need to be supported by the static analysis tool. Note that the static code analysis tool should perform all of the checks specified in the check target used for analysis, and that it must note deviations from this configuration in the report (unsupported checks, unsupported failure handling modes).

<<Check targets>>=
```
<CheckTargets>
  <CheckTarget xsi:type="asef:CCheckTarget" name="BasicChecks">
    <CorrectnessCheckCategory name="numeric.divbyzero"
                              failureHandlingMode="wraparound"/>
    <CorrectnessCheckCategory name="mem.ptr.deref" failureHandlingMode="stop"/>
    <!-- ... -->
  </CheckTarget>
</CheckTargets>
```

Analysis tasks

Analysis tasks are named entities representing combinations of hardware targets, source code modules, language targets, check targets and execution model targets, thereby completing a configuration (modulo tool-specific settings). For example, to analyze the source code module *ExampleModule_joined* for the architecture x86-64 as well as for PPC, the user might create the following analysis tasks:

<<Analysis tasks>>=
```
<AnalysisTasks>
  <AnalysisTask name="analyzeExampleModuleOnPPC"
                missingRequiredCapabilityHandlingMode="abort">
    <ReportGeneratorConfiguration documentFormat="assume"/>
    <HardwareTarget>PPC32</HardwareTarget>
    <SourceModule>ExampleModule_joined</SourceModule>
    <LanguageTarget>LT for ExampleModule</LanguageTarget>
    <CheckTarget>BasicChecks</CheckTarget>
    <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
  </AnalysisTask>

  <AnalysisTask name="analyzeExampleModuleOnX86_64"
                missingRequiredCapabilityHandlingMode="abort">
    <ReportGeneratorConfiguration documentFormat="assume"/>
    <HardwareTarget>x86-64</HardwareTarget>
    <SourceModule>ExampleModule_joined</SourceModule>
    <LanguageTarget>LT for ExampleModule</LanguageTarget>
```

```
    <CheckTarget>BasicChecks</CheckTarget>
    <ExecutionModelTarget>ExampleExecModel</ExecutionModelTarget>
  </AnalysisTask>
</AnalysisTasks>
```

Tool-specific configuration

Some aspects of the configuration are too tool-specific to be specified in a common configuration: for example, time and memory limits might be specified using different granularities, and some settings such as loop bounds are dependent on the fundamental approach of the analysis tool. Users may provide *tool configurations* containing basic parameters such as additional command line arguments for the analysis tool and language target extensions containing e.g. further C preprocessor definitions (for a given source code file, the tool-specific language target extension extends the language target which would otherwise be used for the file. If the file has an individual language target, the tool-specific language target extends that target and is used instead).

<<Tool-specific configuration>>=
```
<asef:ToolConfigurations>
  <ToolConfiguration name="QPRVerify" customParameters="--verbose">
    <ForAnalysisTasks>
      <AnalysisTaskName>PPC32</AnalysisTaskName>
      <AnalysisTaskName>x86_64</AnalysisTaskName>
    </ForAnalysisTasks>
    <LanguageTargetExtension>
      <CLanguageSubtarget superTarget="auto">
        <PreprocessorDefinitions overrideParentDefinitions="false">
          <Definition identifier="STATIC_ANALYSIS"/>
          <Definition identifier="ENABLE_LLBMC_STUBS"/>
        </PreprocessorDefinitions>
      </CLanguageSubtarget>
    </LanguageTargetExtension>
```

The content of the `ToolSpecificConfiguration` element is not defined within this configuration format, relying on the vendors of individual analysis tools to do so. For example, in the case of QPR Verify, such a `ToolSpecificConfiguration` might look like this:

```
    <ToolSpecificConfiguration>
      <LoopBound>40</LoopBound>
    </ToolSpecificConfiguration>
  </ToolConfiguration>
</asef:ToolConfigurations>
```

Local configuration

Finally, the *local configuration* element contains replacement rules for URI substrings, which need to be applied to all URIs occuring in the global part of the configuration. For this example, a local configuration might have the following rules:

<<Local configuration>>=
```
<asef:LocalConfiguration>
  <URISubstitutionRules>
    <URISubstitutionRule token="$_LIBINCLUDES_$"
                  substitution="file:///usr/include/libc"/>
    <URISubstitutionRule token="$_SYSINCLUDES_$" substitution="file:///usr/include"/>

    <URISubstitutionRule token="$_EXAMPLEMODULEPATH_$"
                  substitution="file:///users/felix/projects/ExampleProject"/>
```

```
    <URISubstitutionRule token="$_EXAMPLEMODULESRC_$"
                         substitution="$_EXAMPLEMODULEPATH_$/src"/>
    <URISubstitutionRule token="$_EXAMPLEMODULESTUBS_$"
                         substitution="$_EXAMPLEMODULEPATH_$/verification/generatedStubs"/>
    <URISubstitutionRule token="$_EXAMPLEMODULEJOINED_$"
                         substitution="$_EXAMPLEMODULEPATH_$/verification/joined"/>
  </URISubstitutionRules>
</asef:LocalConfiguration>
```

Reports

In this section, we present most aspects of the current state of the report format by example (see `ExampleReport.xml`).

Top-level structure

A report consists of six major parts:

1. a copy of the configuration used to obtain the results (see section „Configuration“),
2. a collection of execution reports,
3. a collection of source code file descriptions,
4. a collection of source code location descriptions,
5. a collection of check results,
6. and a collection of failure traces.

The high-level structure of a report file is given as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<asef:Report xmlns:asef="http://todo.example.com/asef"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" name="Example report"
sourceVersion="String" xsi:schemaLocation="http://todo.example.com/asef ASCCRF.xsd">
  <<Configuration>>
  <<Execution report collection>>
  <<Source file collection>>
  <<Source storage collection>>
  <<Check result collection>>
  <<Failure trace collection>>
</asef:Report>
```

Execution report collection

An *execution report* is a named entity detailing which configuration has been used to configure the analysis tool and contains information about the analysis tool execution, such as warnings. (Note: the concept of source code locations, one of which is referenced in the following example snippet, is explained in a later section.)

```
<<Execution report collection>>=
<asef:ExecutionReports>
  <asef:ExecutionReport
      name="Report for ExampleProject-on-PPC analysis"
      analysisTask="analyzeExampleModuleOnPPC"
      toolParameters="LLBMC"
      analysisBeginDate="2016-12-17T09:30:47Z"
      analysisFinishDate="2016-12-17T09:30:49Z"
      analysisComputerName="i11pc164" toolName="LLBMC">

  <SourceCodeProcessingMessages>
    <SourceCodeProcessingMessage xsi:type="asef:FreeformSourceCodeProcessingMessage"
                locationID="100"
                msg="warning: '&amp;amp;amp;' within '||' [-Wlogical-op-parentheses]"/>
  </SourceCodeProcessingMessages>

  <CheckerMessages> <!-- ... --> </CheckerMessages>
  <ConfigDeviations> <!-- ... --> </ConfigDeviations>
  </asef:ExecutionReport>
</asef:ExecutionReports>
```

Source storage collection

To allow the identification of individual source files as well as e.g. include files not explicitly specified in the configuration's source code modules, reports contain a separate collection of source file descriptions – a list of uniquely identifiable `File` elements. Where possible, files are identified with their corresponding source-module-level descriptions via the module name and their ID within that module. Moreover, a hash sum can be stored for each source file (computed using MD5 unless otherwise specified using the `hashAlg` attribute). The source file descriptions contain language-dependent data: for example, a C source file may be flagged as preprocessed, while a C header file has an include directory attribute. (Again, source file descriptions for other languages can be added without breaking backward compatibility.)

<<Source storage collection>>=

```
<asef:SourceStorages>
  <Storage xsi:type="asef:CSourceFile" id="1"
    path="$_EXAMPLEMODULESRC_$/main.c"
    hashSum="1b826051506f463f07307598fcf12fd6" preprocessed="false"
    originModule="ExampleModule" idInOriginModule="1"/>

  <Storage xsi:type="asef:CSourceFile" id="2"
    path="$_EXAMPLEMODULESRC_$/dodgycode.c"
    hashSum="3b5337aa426bb547efefb97edec54e3e" preprocessed="false"
    originModule="ExampleModule" idInOriginModule="2"/>

  <Storage xsi:type="asef:CSourceFile" id="3"
    path="$_EXAMPLEMODULESRC_$/shadycode.c"
    hashSum="ff702f10bebfa2f1508deb475ded2d65" preprocessed="false"
    originModule="ExampleModule" idInOriginModule="3"/>

  <Storage xsi:type="asef:CHeaderFile" id="4"
    path="$_EXAMPLEMODULEPATH_$/include/ExampleModule.h"
    hashSum="2f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
    includeDirectory=""/>

  <Storage xsi:type="asef:CHeaderFile" id="5"
    path="compiler_sys.h"
    hashSum="3f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
    includeDirectory="$_SYSINCLUDES_$"/>

  <Storage xsi:type="asef:CHeaderFile" id="6"
    path="HideProprietaryCExtensions.h"
    hashSum="4f702f10bebfa2f1508deb475ded2d65" preprocessed="false"
    includeDirectory="$_EXAMPLEMODULESRC_$"/>

  <Storage xsi:type="asef:CSourceFile" id="10"
    path="$_EXAMPLEMODULESTUBS_$/groups/posix_io.c"
    hashSum="8633b81a334995b50b53df83581af093" preprocessed="false"
    originModule="ExampleModule_stubs" idInOriginModule="1"/>

  <Storage xsi:type="asef:CSourceFile" id="11"
    path="$_EXAMPLEMODULESTUBS_$/fancy_rng.c"
    hashSum="b06f74ff6378f4a2629621b3d8aa935f" preprocessed="false"
    originModule="ExampleModule_stubs" idInOriginModule="2"/>

  <Storage xsi:type="asef:CSourceFile" id="12"
    path="$_EXAMPLEMODULESTUBS_$/filescope/ExampleModule/4/read_from_device.c"
    hashSum="c143a9ae806ab2c93ad4f4f593173bf0" preprocessed="false"
    originModule="ExampleModule_stubs" idInOriginModule="3"/>
```

```
 <Storage xsi:type="asef:CSourceFile" id="20"
   path="$_EXAMPLEMODULEJOINED_$/someOtherFancyRNG.c"
   hashSum="591d99a6a84b1e1dbb44395a3fa27d64" preprocessed="false"
   originModule="ExampleModule_joined" idInOriginModule="1"/>
```

Source code stored outside of files, e.g. in preprocessor definitions, can also be identified:

```
  <Storage xsi:type="asef:CFilelessPreprocessorDefinition" id="30">
    <Definition identifier="ADD_MACRO" expansion="x+y"/>
  </Storage>

</asef:SourceStorages>
```

Source location collection

The source location collection contains *locations* identifying places within the files described in the source file collection. The format supports multiple approaches of identifying such places: the C-specific one (including support for macro expansion) is recommended for identifying locations in C source code. Additionally, support is provided for plaintext file/line/column and file/line location specifications. Finally, ranges in source code can be specified using a special location type.

<<Source location collection>>=

```
<asef:Locations>
```

A basic C soure code location (without need for macro expansion) is given by a file/line/column triple:

```
  <Location xsi:type="asef:CsourceRealLocation"
    id="1" storageID="2" lineNo="22" colNo="8"/>
```

If a `CRealLocation` resides within a preprocessor definition, the line number is required to be 1.

```
  <Location xsi:type="asef:CsourceRealLocation"
    id="2" storageID="30" lineNo="1" colNo="2"/>
```

C macro expansions are represented by *macro locations*, consisting of a spelling location ID (the location within the macro definition) and the expansion location ID (the location where the macro is expanded).

```
  <Location xsi:type="asef:CsourceRealLocation"
    id="10" storageID="3" lineNo="1" colNo="1"/>
  <Location xsi:type="asef:CsourceRealLocation"
    id="11" storageID="3" lineNo="1" colNo="1"/>
  <Location xsi:type="asef:CsourceMacroLocation"
    id="12" spellingLocID="10" expansionLocID="11"/>
```

Ranges can be specified using a begin and an end location ID:

```
  <Location xsi:type="asef:CsourceRealLocation"
    id="20" storageID="2" lineNo="22" colNo="12"/>
  <Location xsi:type="asef:CsourceRealLocation"
    id="21" storageID="2" lineNo="22" colNo="20"/>
  <Location xsi:type="asef:RangeLocation"
    id="22" beginLocID="20" endLocID="21"/>
```

Finally, locations can be specified in plaintext files:

```
  <Location xsi:type="asef:PlaintextRealLocation"
    id="30" storageID="20" lineNo="5" colNo="53"/>
```

```
 <Location xsi:type="asef:PlaintextLineLocation"
    id="31" storageID="20" lineNo="5"/>
</asef:Locations>
```

Check result collection

The check result collection contains a `Check` element for each performed check. We plan to establish a common set of check categories (containing e.g. the category *numeric.divbyzero*), onto which the tool-specific check categories (e.g. Polyspace's *ZDV* category) can be mapped. The check result (safe, unsafe or undecided) is given in the check element's *status* attribute. The tool may provide further details, e.g. reasons for an undecided status, in the *statusSupplement* attribute. Furthermore, the tool's internal check category and status are provided using the *internalCategory* rsp. *internalStatus* attributes. Finally, the analysis tool may provide free-form information about the result in the *annotation* field.

<<Check result collection>>=

```
<asef:Checks>
  <ResultsFor executionReport="Report for ExampleProject-on-PPC analysis">

    <Check xsi:type="asef:CCheck" id="1"
      category="numeric.divbyzero"
      status="unsafe" internalCategory="divbyzero"
      internalStatus="unsafe" annotation="">
      <LocID>1</LocID>
    </Check>

    <Check xsi:type="asef:CCheck" id="2"
      category="assertion.user"
      status="safe" statusSupplement="locally" internalCategory="user.assertion"
      internalStatus="safe" annotation="locally">
      <LocID>12</LocID>
    </Check>

    <Check xsi:type="asef:CCheck" id="20"
      category="assertion.user" status="undecided"
      statusSupplement="function-body-missing" internalCategory="user.assertion"
      internalStatus="safe" annotation="function body missing">
      <LocID>22</LocID>
    </Check>

  </ResultsFor>
</asef:Checks>
```

Failure trace collection

Finally, the failure trace collection provides additional information about the failed checks. However, this section remains to be designed.

<<Failure trace collection>>=

```
<asef:FailureTraces>
  <TODO_ConcreteFailureTraces/>
</asef:FailureTraces>
```