The ASSUME Static Analaysis Exchange Format (preliminary draft - not ready for publication)

Felix Kutzner
Department of Computer Science
Research Group "Verification meets Algorithmics"
Karlsruhe Institute of Technology

February 24, 2017

Chapter 1

Introduction

1.1 Abstract

In this document, we present reference descriptions of tool-agnostic configuration and report formats for static code analysis tools. This effort is a first step towards allowing tighter-coupled usage of static code analysis tools. We initially focus on support for the C programming language, but designed the format with extensibility in mind.

1.2 Purpose

When developing safety-critical software, missing a software bug poses severe risks both for the well-being of the software's users and for the financial and legal safety of its developers. Static code analysis (SCA) is a technique widely used to reduce this kind of risk by checking software for programming errors by analyzing its source code using formal methods. However, each theoretical approach of SCA is limited by drawbacks: for example, the approach of *abstract interpretation* is inherently imprecise, while *static bounded model checking* comes with a tradeoff between computational expensiveness and the meaningfulness of its results. Therefore, the checking process frequently involves expensive manual follow-ups, leading to higher development costs or potentially defective software being deployed in safety-critical systems.

We attack this problem by allowing SCA users to combine the strengths of different tools, even if they pursue different approaches to SCA. In the context of *Work Package 2* of the program *ASSUME: Affordable Safe & Secure Mobility Evolution*, we develop methods for tightly coupling SCA tools via exchanging of intermediate and final analysis results.

To further this effort, we developed means for uniformly configuring SCA tools as well as uniformly presenting their analysis results. Our design is focused on modularity as well as both forward- and backward-compatible extensibility, allowing further aspects of SCA tools (such as support for further programming languages, data range specifications, more elaborate assembly code handling) to be configured in a uniform way. We also took a pragmatic stance when appropriate, for example by allowing the inclusion of tool-specific sections in the configuration files: this way, configuration aspects which we omitted so far in the common configuration can be added to the unified format in a later version, while users retain the ability to fully configure SCA tools as needed.

We chose XML as a metaformat and implemented our design using XML schema definitions (XSD). XML and XSD are widely employed languages, greatly facilitating our extensibility goals via their polymorphic type system, the possibility to declare open content and the reusability of elements and types in new schemata.

This document serves as a reference for the common configuration and report formats.

1.3 Software

We develop the JAVA library `asef4j` for the formats defined in this document. `asef4j` is a portable library for parsing configuration files, writing report files, and emitting SCA tool configurations for tools not directly supporting the configuration/report formats.

1.4 Outline

In Chap. 2, we give definitions of terminology and references to standards used for our own developments. Chap. 3 contains the presentation of the formats introduced in this document. Finally, we provide concrete XML schemata definitions and a hierarchy of check categories in Chap. 4.

Chapter 2

Preliminaries

2.1 Normative references

- XML/XSD standards: `https://www.w3.org/TR/REC-xml/`, `https://www.w3.org/TR/xmlschema11-1/`, `https://www.w3.org/TR/xmlschema11-2/`
- Uniform Resource Identifier (URI) RFC: `https://www.ietf.org/rfc/rfc3986.txt`
- ANSI/ISO C standards: `http://www.open-std.org/JTC1/SC22/WG14/www/standards`, `ISO/IEC 9899:2011`, `ISO/IEC 9899:1999`, `ISO/IEC 9899:1990`, `ISO/IEC 9899:1990 + ISO/IEC 9899 AM1`

2.2 Terminology

- Check
- Check category

2.3 Notes on the XML schema design

- TODO: describe technical goals of our design
- ...then do the usual "problem $\rightsquigarrow$ policies $\rightsquigarrow$ actionable items" thing
- Using venetian blinds design pattern (`http://www.oracle.com/technetwork/java/design-patterns-142138.html`) for reusability, with root elements explicitly marked via annotations
- Using open content in carefully chosen places (`https://www.xml.com/pub/a/2002/07/03/schema_design.html`)
- Approach: for each concern, have a collection of named entities. more collections via open content. concrete configurations via analysis tasks, which contain references to named entities. here, open content too, to allow referencing elements from collections added for new concerns. give examples.
- Within collections of named entities: use XSD subtyping for extensibility.
- Schema modularization: multiple XSD files to facilitate reuse further

2.4 Notes on the specification

- Point out that we're dealing mainly with semantics here, refer to the appendix for the concrete schemata. Use the XSD as a definition of the format's syntax.
- ...therefore, all the XML rules (such as escaping) apply
- $\ll x \gg$ placeholder syntax

Chapter 3

The ASEF configuration format

This chapter contains a description of the ASEF configuration format. The format's syntax is given as an XML schema definition given in Δ^1. Throughout this chapter, the XML namespace `asef` denotes the namespace of the XML schemata definitions given in Δ^2. In Sec. 3.1, we present the configuration format's general structure. In Sec. 3.2, we detail the *local configuration*, i.e. the parts of configuration which may be dependent on the concrete setup of the computer where the static analysis tool is executed. Sec. 3.3 contains the description of the *common global configuration*, which is the tool-independend unified part of the configuration. Finally, we define how tool-specific configuration may be embedded in configurations in Sec. 3.4.

3.1 General structure

Syntactic rule 1. The configuration document's root element shall be `asef:Configuration`.

The structure of the `asef:Configuration` element is given in Fig. 3.1. The elements *«Meta information»*, *«Hardware targets»*, *«Language targets»*, *«Source code modules»*, *«Execution model targets»*, *«Check targets»*, *«Analysis tasks»*, *«Tool-specific configuration»* and *«Local configuration»* are specified in the ensuing sections.

```
<asef:Configuration>
  <asef:GlobalConfiguration>
    <asef:CommonConfiguration>
      \placeholder{Meta information}\\
      \placeholder{Hardware targets}\\
      \placeholder{Language targets}\\
      \placeholder{Source code modules}\\
      \placeholder{Execution model targets}\\
      \placeholder{Check targets}\\
      \placeholder{Analysis tasks}\\
    </asef:CommonConfiguration>
    \placeholder{Tool-specific configuration}
  </asef:GlobalConfiguration>
  \placeholder{Local configuration}\\
</asef:Configuration>
```

Figure 3.1: Structure of the `asef:Configuration` element.

[1] TODO: copy the XSD into this document
[2] TODO: copy the XSD into this document

The configuration is split into two parts: The *global* configuration, contained in the `GlobalConfiguration` element, consists of configuration items which are independent of the concrete environment where the SCA tool is executed. The *local* configuration contains the parts of the configuration which is dependent on the concrete environment where the SCA tool is executed.

3.2 Local configuration

Definition 1. URI substitution rule. An *URI substitution rule* is a pair (x, y) of strings, with x called the *token* and y called the *substitution* of the rule. x must satisfy the regular expression $\_[0-9]+\_$
.

Definition 2. URI substitution element. An *URI substitution rule element* is an XML element named `URISubstitutionRule` of type `asef:URISubstitutionRule`. An URI substitution rule element whose attribute `token` has the value x and whose attribute `substitution` has the value y is said to represent the URI substitution rule (x, y).

Definition 3. Admissible sets of URI substitution rules. A set R of URI substitution rules is called *admissible* iff the following conditions hold:

1. Uniqueness of tokens: $\forall (x_1, y_1), (x_2, y_2) \in R : x_1 = x_2 \rightarrow y_1 = y_2$
2. No cyclic replacement: There are no $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n) \in R$ such that for all $2 \leq i \leq n$, x_i is a substring of x_{i-1} and x_1 is a substring of y_n.

Syntactic rule 2. The *«Local configuration»* element referenced in Fig. 3.1 is an optional XML element named `LocalConfiguration` containing a single element `URISubstitutionRules`, which contains a list of URI substitution rule elements (see Def. 2), representing an admissible set R of URI substitution rules.

Definition 4. Local URI completion. Let R be the set of URI substitution rules represented by URI substitution rule elements within the *«Local configuration»* element of the configuration. The *local completion* of an URI is the result of repeatedly replacing tokens occuring in rules by their respective substitution, until no further replacement is possible.

Semantic rule 1. The location of a file referenced via an URI u given as a value of type `URI` within an ASEF configuration document is given by the local completion of u.

3.3 Common global configuration

3.3.1 The *«Meta information»* element

Definition 5. Configuration meta-information element. A *configuration meta-information element* is an XML element named `Meta`, having the type `asef:Metadata`. The attributes of the configuration meta-information element as well as the values of the elements it contains may have arbitrary string values. The `configurationName` attribute is intended to hold the name of the configuration. The `version` attribute is intended to hold the version of the configuration. The `description` is intended to hold the description of the configuration. The list of `Maintainer` elements contained in a configuration meta-information element is intended to represent the list of persons responsible for maintaining the configuration.

Syntactic rule 3. The *«Meta information»* element referenced in Fig. 3.1 shall be a configuration meta-information element as defined in Def. 5

.

3.3.2 The *«Hardware targets»* element

Definition 6. Hardware target. A *hardware target* is an XML element of type `HardwareTarget`. The *target name* of a hardware target H is the value of the attribute `name` of H.

Syntactic rule 4. The *«Hardware targets»* element referenced in Fig. 3.1 shall be an XML element named `HardwareTargets` containing a sequence S of XML element named `HardwareTarget`, which are hardware targets as defined in Def. 6.

Definition 7. Homogenous hardware targets. Hardware targets of type `HomogenousHardwareTarget` are called *homogenous hardware target*. Let H be a homogenous hardware target.

- The value H.`endianness` determines the byte-level sequential order of integer types composed of multiple bytes. If H.`endianness` = `big`, integers are stored in big-endian order. If H.`endianness` = `little`, integers are stored in little-endian order. If H.`endianness` = `middle`, integers are stored in middle-endian order.
- The value H.`unalignedDereferenceSupported` determines whether unaligned memory accesses shall be treated as errors. Iff H.`unalignedDereferenceSupported` = `false`, the analyzer shall assume that the program is executed on hardware not supporting memory addresses at an address p with an n-byte transfer such that $p \bmod n \neq 0$.
- The value H.`pointerSize` determines the size of data memory addresses in bits. This value must be a multiple of 8.
- The value H.`functionPointerSize` determines the size of function memory addresses in bits. This value must be a multiple of 8.

3.3.3 The *«Language targets»* element

Definition 8. Language target. A *language target* is an XML element of type `LanguageTarget`. The *target name* of a language target L is the value of the attribute `name` of L.

Syntactic rule 5. The *«Language targets»* element referenced in Fig. 3.1 shall be an XML element named `LanguageTargets` containing a sequence S of XML elements named `LanguageTarget` which are language targets as defined in Def. 8. The elements contained in S must be unique in S wrt. their target name. No language target contained in S may have the target name `auto`.

Definition 9. C language targets. Language targets of XML type `CLanguageTarget` are called *C language target*. Language targets of XML type `CLanguageSubtarget` are called *C language subtargets*. The *supertarget name* of a C language subtarget L is the value of the attribute `superTarget` of L. The *supertarget* of L is a C language target or a C language subtarget.

Syntactic rule 6. For each C language subtarget L occuring within the *«Language targets»* element, the supertarget name of L must be the name of a C language target occuring within the *«Language targets»* element of the name of a C language subtarget occuring within the *«Language targets»* element. The supertarget name of L may not be `auto`. Δ^3 Δ^4

Semantic rule 2. For a C language subtarget L with supertarget name S occuring within the *«Language targets»* element, the supertarget of L is the language target with the name S.

Definition 10. C language target semantics. The semantics of a C language target T are defined as given as follows:

1. T.`signedBitfields`: Iff `true`, bitfield variables shall be interpreted as signed variables.

[3] TODO: Exclude cyclic superTarget relations
[4] TODO: Tie supertarget names to supertargets

2. T.`signedEnums`: Iff `true`, enum variables shall be interpreted as signed variables.

3. T.`fpRoundingMode`: TODO: determines FP rounding mode

4. T.`fpConstantRoundingMode`: TODO: determines FP rounding mode

5. T.`enumType`: The C type used for representing **enum** types. The value of T.`enumType` must be the name of a C integer type.

6. T.`inlineAssemblyHandlingMode`: If T.`inlineAssemblyHandlingMode` = `ignore`, inline assembly code occuring in C source files shall be ignored by the analysis tool. If T.`inlineAssemblHandlingMode` = `default`, the analysis tool shall treat inline assembly code using its default settings for the treatment of inline assembly code.

7. T.`standardRevision` : The revision of the C standard using which the C code should be interpreted.

8. T.`initializeStaticVariables`: If `true`, variables with static storage duration shall be assumed to be initialized as defined in the C standard. If if `false`, their values shall be treated as uninitialized.

9. T.`enableVolatile`: If `false`, the **volatile** modifier is ignored for all variables. If true, reads from variables having the **volatile** modifier are interepreted as possibly any value in the range of the corresponding variable.

Δ[5]

Definition 11. C language subtarget semantics. Δ[6]

- Definition: List of preprocessor definitions (directly given)
- Definition: List of include paths (directly given)
- Definition: List of include files (directly given)
- Definition: C language subtarget has a list of preprocessor definitions, include paths, include files
- Definition: Resolved C language subtarget: is a c language subtarget, with its diffs applied to data inherited from super targets

3.3.4 The *«Source code modules»* element

- proper description of stub URIs
- module-level language subtarget
- source files (id, path, subtarget)
- stub requirements (proper description of stub URIs)
- module inclusion specs

[5] TODO: describe meaning of the list of C types
[6] TODO: extension semantics

3.3.5 The *«Execution model targets»* element

Definition 12. Execution model target. An *execution model target* is an XML element of type `ExecutionModelTarget`. The *target name* of an execution model target E is the value of the attribute `name` of E.

Syntactic rule 7. The *«Execution model targets»* element referenced in Fig. 3.1 shall be an XML element named `ExecutionModelTargets` containing a sequence S of XML element named `ExecutionModelTarget`, which are execution model targets as defined in Def. 12.

Definition 13. C synchronous execution model targets. Execution model targets of type `CSynchronousExecutionModelTarget` are called *C synchronous execution model target*. A C synchronous execution model target E signifies that the analysis tool shall check the program under the assumption that the program is executed sequentially. The `EntryPoint` elements contained in E.`EntryPoints` specify the entry point functions of the software under analysis. The static code analysis tool shall consider an execution of the software under analysis reflecting the sequential execution of the entry point functions in the order of their appearance within E.`EntryPoints`.

3.3.6 The *«Check targets»* element

3.3.7 The *«Analysis tasks»* element

- reference items by name, perform analysis according to the semantics of the referenced items

3.4 Tool-specific global configuration

- optimizationLevel, verbosityLevel, timeLimit, customParameters
- forAnalysisTasks structure
- ToolSpecificConfiguration

Chapter 4

Appendices

4.1 Check Categories and Semantics

4.1.1 Notation

PS:X := Polyspace check category X
QPR:X := QPR check category X
AS:X := Astree alarm category X

4.1.2 Assertions

Category	Description	Contains as subset
assert	An assertion failed.	PS:ASRT
assert.user	A user-defined assertion failed.	AS:"Assertion failure", QPR:user.assertion
assert.custom	A custom assertion failed.	QPR:custom.assertion, AS:"User defined alarm", AS:"Check failure"

4.1.3 Numeric checks

Note: "overflow" == overflow or underflow

Category	Description	Contains as subset
numeric		
numeric.overflow	Numeric variable overflow	PS:OVFL, AS:"Overflow in arithmetic", AS:"Initializer range"
numeric.overflow.float	Overflow of floating-point value	
numeric.overflow.int	Overflow of signed integers (result of operation out of range)	QPR:arithmetic.overflow, QPR:shift.overflow
numeric.overflow.conversion	Overflow in integer conversion	AS:"Overflow in conversion"
numeric.overflow.conversion.explicit	Overflow in explicit integer conversion	QPR:explicit.conversion.overflow
numeric.overflow.conversion.implicit	Overflow in implicit integer conversion	QPR:implicit.conversion.overflow
numeric.divbyzero	Division by zero	PS: ZDV
numeric.divbyzero.float	Division by zero (float)	AS:"Float division by zero"
numeric.divbyzero.int	Division by zero (int)	AS:"Integer division by zero", AS:"Integer modulo by zero", AS:"Undefined integer modulo", QPR:divbyzero
numeric.shift	Invalid shift operation	PS: SHF
numeric.shift.rhs	Invalid right-hand side of shift	AS:"Wrong range of second shift argument"
numeric.shift.rhs.amount	Shift amount is too large	QPR:shift.by.amount
numeric.shift.rhs.negative	Shift amount is negative	QPR:shift.by.negative
numeric.shift.lhs	Invalid left-hand side of shift	QPR:shift.of.negative
numeric.floatop.result.nan	Invalid floating-point operation	PS:INVALID_FLOAT_OP
numeric.floatop.result.subnormal	The operation results in a subnormal float	PS:SUBNORMAL_FLOAT
numeric.floatop.arg.nan	Invalid floating-point operation	AS:"Float argument can be NaN or infinity"

4.1.4 Control flow

Category	Description	Contains as subset
controlflow		
controlflow.nonterminating	Nonterminating program execution	
controlflow.nonterminating.loop	Nonterminating loop	PS:NTL, QPR:nonterminating.loop
controlflow.nonterminating.functioncall	Nonterminating function call	PS:NTC
controlflow.recursivecall	Recursive function call	Astree:"Recursive function call"
controlflow.missingfunctiondefinition	Function body missing	Astree:"Stub invocation"
controlflow.unreachablecode	Unreachable code	
controlflow.deadcode	Effectless code	

4.1.5 Data flow

Category	Description	Contains as subset
dataflow dataflow.writeafterwrite	Write to memory location without read since last write	AS:"Write after write"
dataflow.initialization	Use of non-initialized variables	AS:"Use of uninitialized variables", PS:NIRV, PS:NIP
dataflow.initialization.local	Use of non-initialized local variable	PS:NIVL, QPR:initizlized.local
dataflow.initialization.global	Use of non-initialized global variable	PS:NIV, QPR:initialized.global, AS:"Reading global/static variable that was never written to"

4.1.6 C Standard library

Category	Description	Contains as subset
stdlib		PS:STD_LIB
stdlib.memalloc	Invalid arguments for dynamic allocation/deallocation function	AS:"Invalid argument in dynamic memory ..."
stdlib.memaccses	Invalid arguments for memory-accessing function	AS:"Invalid memcpy/bzero/..."
stdlib.numeric	Square root of negative number	AS:"Square root of negative number"

4.1.7 Memory

Category	Description	Contains as subset
mem		PS:COR
mem.ptr	Invalid pointer-related operation	
mem.ptr.deref	Invalid pointer dereference	PS:IDP, QPR:pointer.dereference
mem.ptr.deref.misaligned	Dereference of misaligned pointer	AS:"Dereference of mis-aligned pointer"
mem.ptr.deref.invalid	Dereference of pointer with invalid address	AS:"Dereference of null or invalid pointer"
mem.ptr.deref.field	Incorrect field dereference	AS:"Incorrect field dereference"
mem.ptr.deref.function	Pointer to invalid or null function	AS:"Pointer to invalid or null function"
mem.ptr.op.arrayoutofbounds	Array index out of bounds	PS:OBAI, QPR:arrayindex, AS:"Out-of-bound array index"
mem.ptr.op	Invalid operation on pointer	
mem.ptr.op.arrayoutofbounds	Array index out of bounds	PS:OBAI, QPR:arrayindex, AS:"Out-of-bound array index"
mem.ptr.op.comparison	Invalid pointer comparison	AS:"Invalid pointer comparison"
mem.ptr.op.invalidarithmetic	Arithmetics on invalid pointers	AS:"Arithmetics on invalid pointers" AS:"Subtraction of invalid pointers"
mem.ptr.op.offset	Offset range overflow	AS:"Offset range overflow"
mem.funcall	Invalid function call via pointer	QPR:functionpointer
mem.funcall.args	Invalid function call due to arguments	
mem.funcall.args.num	Wrong number of arguments in function call	AS:"Function call with wrong number of arguments"
mem.funcall.args.unnamed	Call of a function with an unnamed argument	AS:"Function with unnamed arguments"
mem.funcall.args.incompatibletype	Call of a function with incompatible argument type	AS:"Incompatible parameter type in a function call", AS:"Reinterpreting incompatible parameter type in a function call"
mem.funcall.retval	Invalid function call due to return value	
mem.funcall.retval.incompatibletype	Call of a function with incompatible return type	AS:"Incompatible function return type", AS:"Reinterpreting incompatible function return type"
mem.write.constant	Write to constant memory	AS:"Attempt to write to a constant"

4.1.8 Asynchronous execution

Category	Description	Contains as subset
async.racecondition.readwrite	Concurrent read and write to/of memory location	AS:"Read/Write data race"
async.racecondition.writewrite	Concurrent writes to memory location	AS:"Write/Write data race"

4.1.9 Miscellaneous

Category	Description	Contains as subset
misc	Checks not otherwise classified within the ASSUME check category hierarchy	AS:"Invalid usage of concurrency intrinsic", AS:"Invalid usage of OS services", AS:"Run-time error during the evaluation of a constant expression"
misra		
warning		