SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.2: Modeling of information-flow restrictions

Paul Muntean, TUM

SIBASE
Technische Universität München
Chair for IT Security
Workgroup TP 5.1

SIBASE Report

TP 5.1 – AP 5.1.2: Modeling of information-flow restrictions

Editor: Paul Muntean (paul@sec.in.tum.de)

Authors:	Paul Muntean, TUM
TP Responsible:	Paul Muntean
Version:	03. September 2014
Submission date:	29. August 2014

Funded by BMBF under grant number 01IS13020

Version	Date	Author	Comment
0.1	10.07.2014	P.Muntean	Initial version
0.2	14.08.2014	P.Muntean	Correction of misspelled English words
0.3	18.08.2014	P.Muntean	Formatting improvements
0.4	22.08.2014	P.Muntean	Correction of unclear English statements
0.5	25.08.2014	P.Muntean	Formatting the comment mapping algorithm
0.6	27.08.2014	P.Muntean	General formatting improvements
0.7	29.08.2014	P.Muntean	General formatting improvements
0.8	01.09.2014	P.Muntean	Added general xText grammar explanations
0.9	02.09.2014	P.Muntean	Added general formatting improvements
1.0	03.09.2014	P.Muntean	Correction of method header grammar

Contents

List of Figures

Listings

List of Tables

List of Acronyms

ANTLR	ANother Tool for Language Recognition
API	Application Programming Interface
AST	Abstract Syntax Tree
ECORE	EMF File Extension
EMF	Eclipse Modeling Framework
ESC	Extended Static Checking
IDE	Integrated Development Environment
IE	Information Exposure
IF	Information Flow
IFC	Information Flow Control
IFW	Information Flow Weakness
LOC	Lines of Code
OCL	Object Constraint Language
OS	Operating System
SAE	Static Analysis Engine
SW	Software
UML	Unified Modeling Language
xTend	EMF File Extension
xText	EMF File Extension

Chapter 1: Introduction

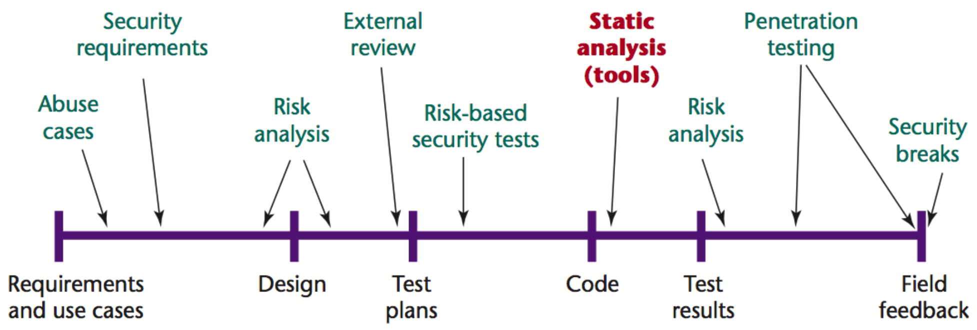


Figure 1.1.: The software development life cycle with focus on security

Figure 1.1 [1] presents the standard software life-cycle with focus on security. It can be observed that static analysis tools are usually used only after the code was written. We argue for annotating other software artifacts (UML models) even in early stages of software development in order to incorporate security concerns into the software design phase. We present in this chapter the capabilities, limitations and possible improvements of our annotation language which is used for annotating UML state charts and source code with focus on information flow restriction.

1.1 State of The Art

The detection of information exposure weaknesses [2] (a subcategory of Information Flow Weaknesses (IFW)) uses dynamic analysis techniques [3–5], static analysis techniques [6–10], and hybrid techniques which combine static and dynamic approaches [11]. The static techniques need to know which parts of the code are *sinks* (functions which use information provided as parameters) or *sources* (functions which provide information in form of return value) and which variables are tagged and need to be propagated based on the Control Flow Graph (CFG). A solution for tagging *sinks*, *sources* and *variables* in source code is based on libraries which contain all needed annotations attached to function declarations. This approach plays an important role mainly for static analysis bug detection techniques where the information available during program run-time is not available nor the interaction with the environment can be fully simulated. Extended Static Checking (ESC) [12] is a promising research area which tries to cope with the shortage of not having the program run-time information. During extended static analysis additional information is provided to the static analysis process. This information can be used to define trust boundaries and tag variables. Textual annotations are usually manually added by the

user in source code. At the same time annotations can be automatically generated and inserted into source code. ESC can be used to eliminate bugs in a late stage of the software project when code development is finished. Tagging and checking for information exposure bugs during the design phase would eliminate the potential of implementing software bugs which can only be removed very costly afterwards. Thus security concerns should be enforced into source code right after the conceptual phase of the project.
The paper [13] presents five challenges concerning ESC. The last challenge reports of the annotation as being a very time consuming burden and is therefore disliked by some programming teams. The authors argue about the fact that annotations can cover design decisions and enhance the quality of source code. We argue that annotations are necessary in order to do ESC and the user needs a kind of assistance tool that helps selecting the suited annotation based on the current context. Thus the annotation burden needed for learning and applying the language should be reduced. At the same time adding annotations to reusable code libraries reduces even more the annotation burden since libraries can be reused, shared and changed by software development teams.
The main limitations of an annotation language used for tagging trust boundaries and variables in order to detect information flow bugs are in our opinion:
Main Limitation:

1. Annotation languages can not be applied out of the box on both modeling and source code level for introducing information flow restrictions

Other Limitations:

1. Annotation perceived to be a heavy burden
   - No language assistant/editor
   - High burden to learn and use the language
2. Language characteristics
   - Annotation languages are not easily extensible
   - Annotation languages tags are hard-coded for a limited set of tags
   - Annotation languages are not expressive enough
   - Some annotation languages don't have library support, thus low reuse

To address the *Main Limitation* we need to think about how textual annotations can be abstracted so that these can be used to annotate UML models as well. One possibility is to use graphical post-its (text boxes in which textual annotations can be directly typed in) or to have a kind of graphical pallet from which symbols having a one to one mapping to textual annotations can be selected and attached to different parts of the UML model. This offers the possibility to analyze the models by generating code and attaching the annotations directly to the code. UML state charts can be also analyzed by simulating execution and using the annotations as intermediary states. Also OCL annotations, which were inserted to be interpreted in a user defined way, can be used for information propagation analysis.
To address point 1 of *Other Limitations* a language assistant editor is needed on the model and code level that is context sensitive and that can help to suggest suited annotations. To address the other limitations from point 2 the language should be built around a code infrastructure that

makes the supporting code easily adaptable to changes. The set of tags should be suited and sufficient for the intended task. The usage of annotations should be possible for any file type in order to support library reuse.
Annotation languages have made a significant impact on static analysis. For example, Microsoft's SAL annotations [14] helped to detect more then 1000 potential security vulnerabilities in Windows code [15]. In addition, several other annotation languages including FlowCaml[16], Jif [17], Fable [18], AURA[19] and FINE [20] express IFC related concerns.
However, none of the annotation languages has support for both the modeling and source code level in order to introduce information-flow restrictions. Most of the effort concerned with annotating is focused on textual files. We argue for annotating other software artifacts even in early stages of software development. In our opinion this has the following advantages:

1. Security concerns can be integrated into software design artifacts in an early phase of development
2. Bridging the communication gap between software security designers and software developers
3. Increased level of software security and quality

1.2 Why Do We Need a New Policy Language?

The goal of our research was to design an annotation language that can be used for annotating text files and UML state charts by tagging sinks, sources, trust boundaries, sanitization and declassification functions. We are aware that there are a lot of other policy languages that could be used for annotation of text files and UML models but none of them is directly usable right out of the box for our purpose.
Our policy language design requirements are:

1. The policy language should be used for annotating UML state charts and source code files in order to restrict certain information flows
2. The policy language should be be integrated as annotation language in the Eclipse IDE with focus on customizability and extensibility
3. The set of policy language annotation tags should not be fixed
4. The parser for the language should be automatically generated.You do not want to write a full-blown C/C++ language parser!
5. The parsed annotations should be converted to objects and no string tokenizers should be used

After taking into consideration all these language requirements we came to the conclusion that there is no open source policy language that could be used to detect information exposure weaknesses and API misusage (library functions and system calls) by annotating models and source code.

1.3 Idea and Contribution

Starting from our previous mentioned policy language requirements we designed an xText [21] based grammar that is used to parse the whole C/C++ 11 language. The C/C++ source code file extensions (.h, .hh, .hhh, .hxx, .c, .cpp) and UML state chart annotation post-its (graphical boxes which can be attached to different parts of a UML state chart diagram) can be annotated with policy language restrictions. From the point of view of the format of the used textual annotation tags our annotation language is similar with [22, 23]. `//@` is used as starting tag for single-line comments. A multi-line comment is starting with the tag `/*@` and ending with the tag `@*/`.
We obtained an ECORE model (a one to one mapping from our xText grammar to the ECORE grammar representation) that can be reused for integrating the policy language into an UML state chart editor. Treating the annotation tags as EObjects opens new possibilities for annotating UML models.
Our lightweight annotation language (reduced set of annotation tags, extensible and reduced grammar size) uses recursion for dealing with different expression types and has only 4 top level entity objects. Multiple nested recursions are used in our grammar recognizing complex C/C++ expressions. The policy language grammar has about 400 LOC with code comments included and in our opinion it is very small. Source code generation is also supported by using xTend, ANTLR and .mwe2 files. With few adaptations we could use our grammar to parse other programming languages as well.
The result is an extensible policy language and a highly reusable source code implementation that can easily be used for annotating models and source files.

Chapter 2: Language Design

This section discusses the five major challenges having to deal with when detecting information exposure bugs:

1. converting comments and code into annotations
2. introducing correct annotations into source files
3. annotating UML state charts and source code files
4. dealing with scattered annotations
5. attaching annotations to the appropriate function declarations with the goal of detecting IE bugs

Section 2.1 presents the five major challenges of our policy language definition and usage and gives an overview of the solution. Section 2.2 describes the textual tags and their usage. Section 2.3 describes the challenges related to comment extraction and gives a brief overview of the implementation. Section 3.4 gives a brief overview of possibilities to use our policy language together with static analysis.

2.1 Challenges and Overview of our Solution

2.1.0.1 Converting Comments and Code into Annotations

The main advantages of using annotations extracted from source code comments are:

1. direct mapping between annotation and source code element (function, variable, etc.)
2. annotations have the same textual representation as source code
3. annotations are human and machine readable

Extracting annotations from comments and code is promising but at the same time it can be quite challenging. First, comments are ambiguous and written in free form; developers can express the same meaning using different words, phrases, sentence structure, etc. It is difficult to automatically and precisely analyze comments in order to extract the correct annotations from them. Furthermore we want that the annotations added by the user to be accurate and syntactically correct, as these annotations are intended to be used in the source code in order to help developers to better understand the analyzed program and to detect information exposure bugs.

To address the above challenges we rely on a fully automated work-flow for generating the policy language parser. The design work-flow is based on xText grammar. The xText grammar is written in text format and at the same time has an ECORE representation where every rule from the grammar is represented as an entity object in a UML class diagram. The language parser can be used to convert textual comments into an ECORE representation. An ECORE representation of a comment is a higher level object that can be instantiated and that maps the whole comment hierarchy into an object oriented hierarchy. An ECORE comment object has fields which can contain other ECORE objects used to represent parts of the original comment. It can be checked if the objects are `null` or if they are instantiated. If they are not `null` then all the parts of the top level object can be retrieved and used for example during extended static analysis.

2.1.0.2 Adding Correct Annotations into Source Files

Writing annotations into a source code file seems to be not so complicated at first but from the beginning the user needs to know the language features and how to use them. At the same time it is important to add syntactically correct annotations into the source file since these annotations are used later on. This challenge was of main importance since the textual annotations were converted afterwards into ECORE objects. The annotations were extracted afterwards from these complex objects. A syntactically incorrect annotation would not permit the parser to create a valid object representation.
In order to prevent the insertion of syntactically incorrect comments a text editor is used to add the comments into a source code file. During typing, the language editor is parsing the whole file in a closed loop. Based on the current context (current line and column number where the user is typing) the editor offers the most appropriate annotation proposals. The advantage is that the user is not overwhelmed by all the possible input elements of the grammar at the same time. Only elements related to the annotation language are suggested. If the user was typing or adding something which was not syntactically correct then this added element was highlighted with a red zig-zag line underneath. At the same time the added language tags were highlighted in the text file in dark red color and in bold font after typing. This creates a clear visual separation between annotated code and comments and helps the user to insert syntactically correct annotations.

2.1.0.3 Annotating UML Models and Source Code Files

One of the main challenges which were posed by our language design requirements was the possibility to annotate both UML models and source code files using the same annotation language. Thus, avoiding the need of learning a new type of language abstraction for annotating UML models. This language abstraction could be represented by a graphical pallet where annotations are represented with symbols which the user can pick and attach to the UML model. We addressed this challenge by using the same textual policy language on both annotation levels. As in the source code file the user has a small typing box similar to a post-it where he can type his annotations and then attach the box to the desired model part. The same error correction and syntax highlighting mechanisms are available to the user as mentioned in section 2.1.0.2.

2.1.0.4 Dealing with Scattered Annotations

The problem of dealing with scattered annotations can be formulated as the challenge of designing an efficient parsing algorithm that that can be used to parse whole text files by mapping annotations

to source code and skips source code elements that are not annotated.
In the following we present a scattered annotations scenario. This scenario is presented in detail in section 4.1.2. The five C header files used in this scenario have in total 3886 LOC, including annotations (`stdio.h` 950 lines, `stdlib.h` 969 lines, `string.h` 648 lines, `time.h` 421 lines, `wchar.h` 898 lines). The five header files contain only six annotations attached to six function declarations. In this situation we say that the annotations are scattered over less then 1 percent of the total number of function declarations. In this scenario an efficient technique should be employed that skips not annotated function declarations and uses a structured file representation format (AST, ECORE, etc.).
We addressed this challenge by providing the possibility to separately parse header files or other types of files. We use the term *translation unit* to denote any type of parsable file (source, header, etc.). Translation units that are source code files were not analyzed since we decided to not insert annotations in this type of files at this stage of development. Each analyzed translation unit was converted to an AST representation. In this way it was possible to get all the contained annotations separated from the source code. Annotations were directly recognized with no need of implementation adaptation since they were fully compatible with the C comments syntax.

2.1.0.5 Attaching Annotations to The Right Function Declarations

Listing 2.1: Example function declarations from stdio.h. The "..." indicate 0 or N blank lines

```
//@ @function sink
...
int    putc(int, FILE *);
...
//@ @function source
int    fgetc(FILE *);
...
```

Between the inserted annotation comment and the function declaration normally there should be no space or new line so that the function declaration follows directly after the annotation text. If spaces or new lines are mistakenly inserted between the annotation and the function declaration or parts of an annotated library don't conform to annotation best-practice rules then the analysis should be capable to deal with this as well. The solution is a robust algorithm used for reading annotations from files. The challenge is to attach the correct annotation to the function declaration. Normally the annotation should be attached to the following (the first function declaration which is present on the next text line(s)) function declaration. A correct mapping for the comments and function declarations presented in listing 2.1 would be that the comment `//@ @function sink` should be mapped to `int putc(int, FILE *);` and that `//@ @function source` should be mapped to `int fgetc(FILE *);`.
This challenge was addressed by developing a parsing algorithm which maps annotations to function declarations. First, the algorithm has a input list of all the annotations obtained from each translation unit represented in AST form. Second, the files are rescanned in order to detect the next function declaration which follows (the first function declaration which is present on the next text line(s)) directly after the file location (line number) from where the annotation was extracted. The line number and the file name from where the comment is extracted are known

from the previous translation unit parsing process. This avoids confusion when for example the same comment is introduced twice but attached to different function declarations.

2.2 Annotation Language Design

As we are concerned with tagging trust boundaries and variables for detecting information exposure weaknesses and API usage errors, we designed annotations in single and multi-line format. Single-line comments contain less information and are written on one text line. Multi-line comments are written on many lines and contain usually more information about the tagged source code. Figure 2.1 presents the single-line annotation format used to annotate function declarations. Figure 2.2 presents an example of a single-line annotation format. A single-line annotation can not replace a multi-line annotation since it can address only one function declaration name or one parameter at a time. The `@ TargetType` is explained in Table 2.1. The `AnnotationTag` is presented in Table 2.2 and Table 2.4. The `ParameterName` and `Comment` are explained in Table 2.5.

```
@ TargetType AnnotationTag Comment
```

Figure 2.1.: Single-line annotation format used for annotating function declarations

```
//@ @function sink myComment
```

Figure 2.2.: Example of a single-line annotation used for annotating a function declaration

Figure 2.3 presents the single-line annotation format used to annotate parameter declarations. Figure 2.4 presents an example of a single-line annotation format.

```
@ TargetType ParameterName AnnotationTag Comment
```

Figure 2.3.: Single-line annotation format used for annotating parameter declarations

```
//@ @parameter parameterName confidential myComment
```

Figure 2.4.: Example of a single-line annotation used for annotating a parameter declaration

Figure 2.5 presents the multi-line annotation format used to annotate function declarations. Figure 2.6 presents an example of a multi-line annotation format.

```
/*@ @TargetType AnnotationTag myComment1
 * @TargetType ParamterName AnnotationTag myComment2 @*/
```

Figure 2.5.: Multi-line annotation format used for annotating function declarations

```
/*@ @function source myComment1
 * @pre functionName1 myComment2
 * @post functionName2 myComment3
 * @parameter password confidential myComment4
 * @parameter type sensitive myComment5
 * @parameter key confidential myComment6 @*/
```

Figure 2.6.: Example of a multi-line annotation: *function declaration* tagged as source, parameters *password*, *key* and *type* tagged as confidential and sensitive.

TargetType	Description
`@function`	annotation tag used for function declaration
`@pre`	annotate previous function call name
`@post`	annotate next function call name
`@parameter`	annotation tag used for function parameter name

Table 2.1.: Policy language target types. The target types are used as first element on each new line of a comment

TargetType	AnnotationTag	Description
`@function`	`sink`	the function uses the information
	`source`	the function provides the information
	`declassification`	the function declassifies information
	`sanitization`	the function sanitizes information
	`trust_boundary`	the function is a trust boundary

Table 2.2.: The annotation tags presented in the table can be used in combination with the target tag `@function`

TargetType	Description
`@pre`	previous function call name
`@post`	next function call name

Table 2.3.: The target tags are used to annotate previous or next functions calls in a *chain* of function calls

TargetType	AnnotationTag	Description
`@parameter`	`confidential`	confidential information tag
	`sensitive`	sensitive information tag

Table 2.4.: The annotation tags can be used in combination with the target tag `@parameter`

String	Description
`ParameterName`	the tagged parameter name
`Comment`	optional textual comment

Table 2.5.: `ParameterName` is used to specify the annotated parameter name and `Comment` is used to specify an optional text comment

Table 2.1 presents the language annotation tags available in our policy language. The target tag `@function` is used to attach a comment to a function declaration. The target tag `@parameter` is used to attach an annotation to a parameter name. Table 2.2 and Table 2.4 present the flags which can be attached to the tag `@function` and `@parameter`, respectively. Table 2.5 contains the strings `ParameterName` which is the name of the tagged parameter and `Comment` represents an optional string comment.
Most of the single-line annotation formats presented in related work [22, 23] begin with the tag `//@` or `//*@` and multi-line annotation formats start with the tag `/*@` and end with the tag `@*/`. This type of comment like annotation formats are used because the user normally wants to introduce annotations that are at the same time valid language comments. On the other hand it is quite difficult and potentially not necessary to extend a programming language in order to insert new syntax elements that need to be compiled too. For supporting ESC it is sufficient to add annotations as comments into the source code. The annotations need to be parsed afterwards and interpreted by the static code analysis.
Typically, there are two ways to integrate annotations in software: the first approach is based on adding annotations into comments so that they are backward compatible. The second approach is based on adding new language keywords, which can ensure that annotations evolve with the source code but is not backward compatible (old compilers would not support the new syntax elements). Either would work for our approach. We choose the first approach for backward compatibility.

2.3 Annotation Extraction and Mapping

This section describes how the annotations are extracted from the comments. Based on the parser and the ECORE technology we will describe how annotations will be mapped to function declarations.

2.3.0.6 Extracting Annotation from Comments

The annotation extraction process from comments consists of two steps: (1) *Comment Extraction*: extracting annotations containing comments from the source code files. (2) *Annotation Object Generation*: after the textual comments are extracted from source code they have to be converted to a meaningful ECORE representation.

1. *Comment Extraction:* Each translation unit (source file) is converted into an AST representation. The AST representation is an instantiated object containing all the structure of the file in a well structured manner. From this structured representation we can get all comments. The AST object can retrieve all the comments in string format. All the comments will be put into a map where the key is the function declaration (string format)

and the value is the string representation of the comment. The mapping from comments to function declarations is done by the algorithm 1 .

2. *Annotation Object Generation:* We use an object representation of each comment attached to each function declaration. This offers a few advantages in comparison to tokenizing the string and trying to extract the annotations from the comments. First, the grammar entities correspond to a one-to-one mapping to an ECORE representation. The ECORE representation is similar to UML class diagram abstraction. The parser used for parsing the language is registered with the ECORE model representing the grammar. In this context registered, means that the parser *knows* how to convert text comments into the appropriate ECORE object representation. An ECORE object is a hierarchical instantiation of a textual comment. It has getter and setter methods for all the attributes contained in the object's internal structure. Another characteristic of an ECORE object is that it can be deeply nested. This means that grammar syntax contains complex entities composed of many other sub-components (sub-entities) that can be easily represented and managed by an ECORE object.

2.3.0.7 Comments Mapping to Function Declarations

The original algorithm presented in Appendix A for mapping comments to function declarations has 135 lines of code. The algorithm 1 presents the basic structure of the mapping algorithm. The algorithm gets as input a list of translation units (header files, .h file extension) and returns a map having as key the function declaration in string format (as it was inserted in the C library file) and as value the string representing the attached textual comment. In a latter step the value field (currently a string) from the map will be converted to an internal object representation.
The algorithm 1 is performing the mapping in five steps:

1. Step 1: a translation unit file is extracted from the translation unit list.
2. Step 2: all the comments are extracted from the translation unit.
3. Step 3: the comments which are not annotation comments are filtered out.
4. Step 4.a and 4.b: single-line and multi-line comments are handled differently in order to not mix them up.
5. Step 5: at the end of the main loop, the single-line and multi-line hash maps are merged into one map.

Algorithm 1 Comments to function declaration mapping algorithm

```
function MAP MAPCOMMENTSTOFUNCTIONDECLARATIONS($comments$)
    initialize $CL \leftarrow 30$;                                  ▷ maximum multi-line comment length
    initialize $MLS \leftarrow /*@$;                                ▷ multi-line comment beginning string
    initialize $SLS \leftarrow //@$;                                ▷ single-line comment beginning string
    initialize $headerfunctionterminator \leftarrow$ ;
    initialize $commentsattributemap$;
    for all $comment$ from $comments$ do                            ▷ Step 1
        $file \leftarrow comment$;
        initialize $line$, $headerfunctionterminator \leftarrow$ ;
        initialize $bufferInput \leftarrow file$;
        initialize $commentsmap$, $methodattributemap$;
        initialize $buffer$, $bufferOut$;
        while $line$ not empty do                                   ▷ Step 2
            $line \leftarrow bufferInput$;                          ▷ get new line
            $buffer \leftarrow line$;                               ▷ add $line$ to $buffer$
            $buffer \leftarrow newline$;                            ▷ add $newline$ to $buffer$
            $pattern \leftarrow ("/*@|//@")$
            $matcher \leftarrow pattern$;
            if $matcher$ has found then                             ▷ Step 3
                $bufferOut \leftarrow line$;
                $bufferOut \leftarrow newline$;
            end if
        end while
        initialize $singleline \leftarrow null$;
        initialize $nextline1 \leftarrow null$;
        initialize $nextline2 \leftarrow null$;
        initialize $nextline3 \leftarrow null$;
        initialize $nextline4 \leftarrow null$;
        initialize $concat \leftarrow emptystring$;
        initialize $linenumber \leftarrow 1$;
        initialize $temp \leftarrow 0$;
        initialize $commentnumberoflines \leftarrow 0$;
        while $singleline \leftarrow bufferOut$ and not empty do
            if $singleline$ starts $MLS$ then                       ▷ Step 4.a
                $commentnumberoflines \leftarrow 0$;
                $concat \leftarrow singleline$;
                $nextline1 \leftarrow bufferOut$;
                while $nextline1$ contains $MLS$ and $commentnumberoflines < CL$ do
                    $temp \leftarrow 1$;
                    $linenumber \leftarrow linenumber + 1$;
                    $concat \leftarrow newline + nextline1$;
                    $nextline1 \leftarrow bufferOut$;
                    $commentnumberoflines \leftarrow commentnumberoflines + 1$;
                end while
                if $temp$ equal 1 then
                    $commentsmap \leftarrow linenumber$ and $concat$;
                    $temp \leftarrow 0$;
```

Algorithm 1 Comments to function declaration mapping algorithm (continued)

```
48:                     $concat \leftarrow emptystring$;
49:                 end if
50:                 while $nextline1$ remove empty spaces equals $emptystring$ do
51:                     $nextline1 \leftarrow bufferOut$;
52:                 end while
53:                 if $nextLine1$ contains $headerfunctionterminator$ then
54:                     $methodattributemap \leftarrow linenumber$ and $nextline1$
55:                 else if $nextline1$ doesn't contain $headerfunctionterminator$ then
56:                     while $nextline1$ doesn't contain $headerfunctionterminator$ do
57:                         $nextline3 \leftarrow bufferOut$;
58:                         $nextline1 \leftarrow nextline1 + space + nextline3$;
59:                     end while
60:                     $methodattributemap \leftarrow linenumber$ and $nextline1$
61:                 end if
62:             end if
63:             if $singleline$ contains $SLS$ then                                ▷ Step 4.b
64:                 $commentsmap \leftarrow linenumber$ and $singleline$;
65:                 $nextline2 \leftarrow bufferOut$;
66:                 if $nextline2$ not equals $emptystring$ and $nextline2$ contains ; then
67:                     $methodattributemap \leftarrow linenumber$ and $nextline2$;
68:                 else
69:                     while $nextline2$ equals $emptystring$ do
70:                         $nextline2 \leftarrow bufferOut$;
71:                     end while
72:                     if $nextline2$ contains $headerfunctionterminator$ then
73:                         $methodattributemap \leftarrow linenumber$ and $nextline2$;
74:                     else if $nextline2$ doesn't contain $headerfunctionterminator$ then
75:                         while $nextline2$ doesn't contain $headerfunctionterminator$ do
76:                             $nextline4 \leftarrow bufferOut$;
77:                             $nextline2 \leftarrow nextline2 + space + nextline4$;
78:                         end while
79:                         $methodeattributemap \leftarrow linenumber$ and $nextline2$;
80:                     end if
81:                 end if
82:             end if
83:             $linenumber \leftarrow linenumber + 1$;
84:         end while
85:         for all x from commentsmap do                                          ▷ Step 5
86:             $p1 \leftarrow commentsmap$ get the x'th element;
87:             $p2 \leftarrow methodeattributemap$ get the x'th element;
88:             if $p2$ doesn't contain the SLS then ▷ avoid putting single-line comments twice
89:                 $commentsattributemap \leftarrow p1$ and $p2$; ▷ add single, multi-line func. decl.
90:             end if
91:         end for
92:     end for
93:     return $commentsattributemap$;
94: end function
```

2.3.0.8 Extracting Annotation from UML State Charts

Starting from the requirement that the annotations should be textual and should be attachable to UML state charts as we generate source code from annotated UML state charts. After we have generated source code the method of extracting annotations is similar to the one presented in the previous section.

2.3.0.9 Annotation Usage

After running the algorithm presented in listing 1 we get a map containing headers and comments mapped together. After this step we convert each comment string value from the previous map into an ECORE object representation. This object contains all the information previously defined in the string comment in a structured object oriented way. The resulting hash map containing as key the function declaration string and as value the ECORE object representation will be used to tag internal variables and function calls during static analysis, for example.

2.4 Static Analysis

ESC will be used to perform extended static source code analysis. The annotations should be extracted in advance and attached to variables and functions as the analysis is visiting each node on a potentially buggy path.

2.4.0.10 Annotation Propagation

Annotation propagation will be based on explicit information flow. Implicit information propagation is also possible. The tags will be attached initially to function parameters and, based on explicit interference rules, the parameter tags and function tags will be propagated to other variables and function call parameters as the analysis proceeds sequentially on each path.

2.4.0.11 Annotation Checking and Bug Detection

As previously described the attached tags will be propagated based on explicit information flow or implicit information flow. At each step of the static analysis the attached annotation is checked and propagated accordingly to the information inference rules. A bug is detected for example if a confidential tagged variable is transmitting its tag along an execution path and a trust boundary is reached by this tagged variable.

Chapter 3: Implementation

3.1 xText Grammar Implementation

The implementation of our policy annotation language is based on the code infrastructure offered by a standard xText Eclipse project. The standard xText Eclipse project is composed of four sub-projects. The first project outlined in figure 3.1A, contains a `Hello World` xText file used for grammar definition and a .mwe2 file used for generating source code for the language parser, UI text editor and the jUnit test cases. The second project, figure 3.1B, is an SDK project containing the files: `build.properties` and `feature.xml`. The third project shown in figure 3.1C, contains some jUnit test cases for testing the whole code infrastructure. The fourth project shown in figure 3.1D, is a stand-alone UI text editor containing the generated language parser and multiple customisation classes for defining the behaviour of the textual language editor.

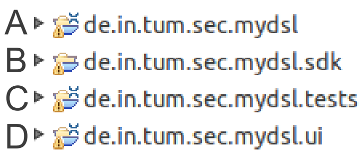

A ▸ de.in.tum.sec.mydsl
B ▸ de.in.tum.sec.mydsl.sdk
C ▸ de.in.tum.sec.mydsl.tests
D ▸ de.in.tum.sec.mydsl.ui

Figure 3.1.: The xText de.in.tum.sec project

3.1.1 xText Grammar Structure

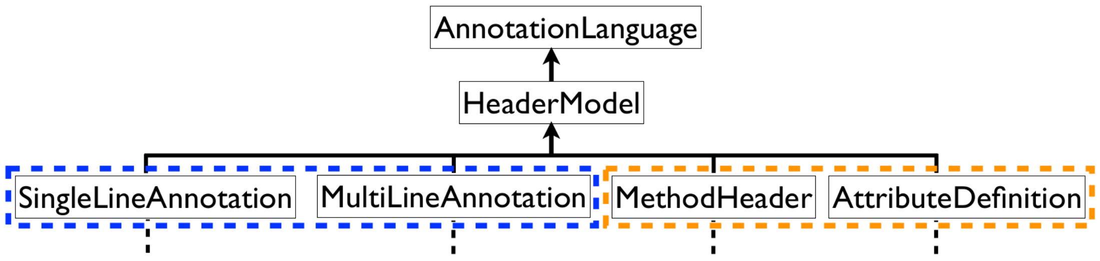


Figure 3.2.: Annotation language grammar hierarchy. The blue dotted line encloses annotation entities and the orange dotted line encloses syntax recognition entities

Figure 3.2 presents the top level structure of our annotation language. Figure 3.2 highlights a clear

separation between annotation and syntax recognition entities. Our annotation language follows a top down definition structure. At the top root node we have the `AnnotationLanguage` entity object which is composed of a `HeaderModel` entity. The `HeaderModel` entity is composed of a list of four entity objects. The `SingleLineAnnotation` and `MultiLineAnnotation` entities are responsible for defining the language annotations entities. The `MethodHeader` and `AttributeDefinition` entity objects are needed for C/C++ syntax recognition. During syntax analysis, all C/C++ language constructs need to be recognisable by the parser since we want to offer annotation proposals to the user. These proposals are context sensitive regarding the position of the currently edited syntax line and column. If a C/C++ expression is not properly recognized by the parser then the proposal mechanism doesn't work from that location until the end of the file. This means that proposals are no more suggested to the user and that the proposals input window doesn't appear.
We think that our xText based grammar has high potential for extensibility. First, we use a clear separation between language annotation entities and syntax recognition entities. Second, our hierarchical language structure offers the possibility to easily add other language elements. Third, the recursivity rules used in the syntax recognition entities can be used to recognize other programming languages.

3.1.2 xText Syntax Graphs

The grammar syntax graphs are graphical representations of the grammar entities. This section presents the main grammar entities from an dynamic perspective whereas in section 3.1.3 the same grammar entities will be presented from a static perspective. All syntax graphs are integrated in a parallel execution from left to right. This means that when the language is parsed the components searched for in the language are read from left to right and in a parallel. Syntax graphs are used to indicate possible paths for syntax recognition. As previously mentioned the syntax graphs are useful to get a better understanding about grammar entities and recursions. At the same time the syntax graphs present the required annotation tags used for trust boundaries (sink, sources, etc) annotation. Annotation entities are used to define annotation language entities (single and multi-line annotations) and syntax recognition entities (method header entity and attribute definition entity) are used to recognize syntax elements. Boxes represent keywords and lines represent possible execution paths. SYMBOLS represents a list of all supported grammar symbols for example: `%, &, #, $` etc. The characters `\n, \s, \r` are used to denote line breack, a space and respectivelly a carriage return. Each syntax graph indicates from left to right the dynamic recognition process of C/C++ syntax elements.

3.1.2.1 Annotation Type Entities

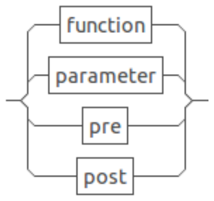


Figure 3.3.: Annotation type syntax graph

Figure 3.3 presents the xText syntax graph for 4 annotation tag categories. It is possible to add other parameter tags by simply adding them in the appropriate grammar list. For example the language grammar could be extended to support the annotation of not only function parameters and function declarations but also interfaces for pre and post function execution restriction.

3.1.2.2 Parameter Annotation Tags

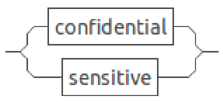


Figure 3.4.: Parameters annotation syntax graph

Figure 3.4 presents the xText syntax graph of two annotation tags used for tagging function parameters. It is possible to add other parameter tags by simply adding them in the appropriate grammar list. The resulting syntax graph would have additional boxes represented in parallel with the already existing two.

3.1.2.3 Function Annotation Tags

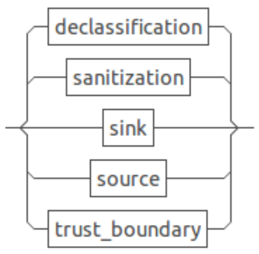


Figure 3.5.: Function annotation syntax graph

Figure 3.5 presents the xText syntax graph for 5 annotation tags used for tagging function declarations. It is possible to add additional function tags by simply adding them in the appropriate grammar list. As previously mentioned the syntax graph would contain additional boxes represented in parallel with the existing ones.

3.1.2.4 Single-line Annotation Entity

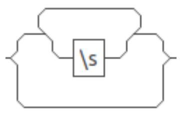


Figure 3.6.: Syntax graph with two branches

Figure 3.6 presents a recursion with two `if` branches. The interior if branch contains on one branch a block and on the other branch no block. The block represents the empty space. The interior and exterior branches can be traversed by the parsing algorithm recursively.

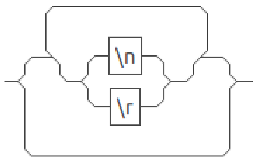


Figure 3.7.: Syntax graph with three branches

Figure 3.7 presents a recursion with three `if` branches. This syntax graph appears also as part of other syntax graphs presented in this section. In all other syntax graphs it has the same meaning as the one explained in this section. The interior `if` branch contains on the upper branch no block and on the lower branch another `if` branch. The most interior `if` branch contains two identical branches. The first upper interior `if` branch contains the `\n` symbol and the second branch contains the `\r` symbol. Both symbols are not represented in the boxes. The upper branch is used by the most exterior upper branch during the recursion.

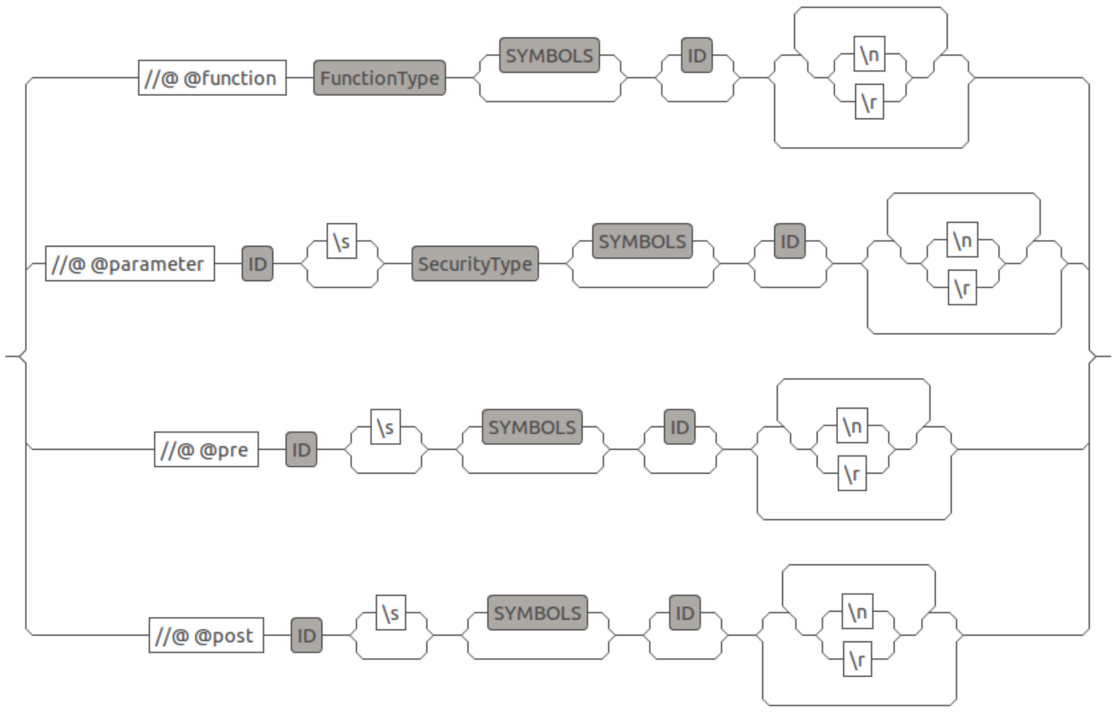


Figure 3.8.: Single line annotation syntax graph

Figure 3.8 presents the xText syntax graph for the single-line annotation entity. The singe-line annotation entity is used to define single-line comments. The syntax graph has two possible execution paths on which single-line function or parameter tags can be specified. It is possible to add other tags by simply adding a new branch in the syntax graph. For example the language grammar could be extended to support the annotation of not only function parameters `//@ @parameter`, function declarations `//@ @function`, previous function declarations `//@ @pre` and post function declarations `//@ @post` but also interfaces (//@ @interface), etc. The resulting syntax graph would have additional branches represented in parallel with the already existing ones. The terminal elements represented on each `//@ @function`, `//@ @parameter`,

`//@ @pre` and `//@ @post` branch represent graphical representations of recursions. `ID` is a grammar placeholder for every type of string which is not a terminal symbol. `SYMBOLS` is a list of language symbols. `SecurityType` is a list containing the flags `confidential` and `sensitive`.

3.1.2.5 Multi-line Annotation Entity

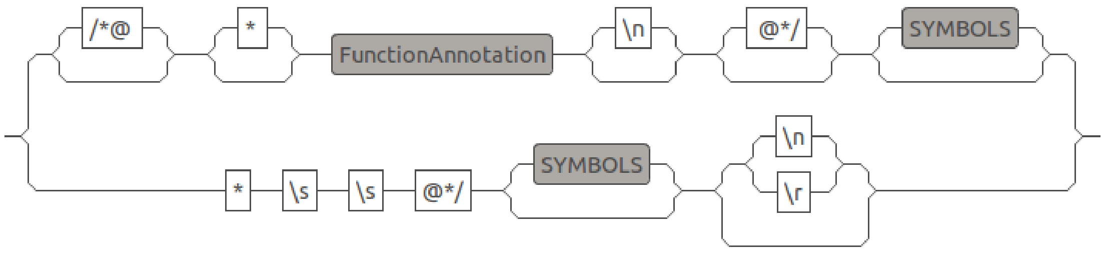


Figure 3.9.: Multi-line annotation syntax graph

Figure 3.9 presents the xText syntax graph for the multi-line annotation entity. The multi-line annotation entity is used to define multi-line annotation comments. The multi-line annotations are composed of function, pre, post and parameter tags and can be defined on more than one text line. The multi-line annotation entity can be extended to support different types of annotations by just adding another parallel branch in the syntax graph.

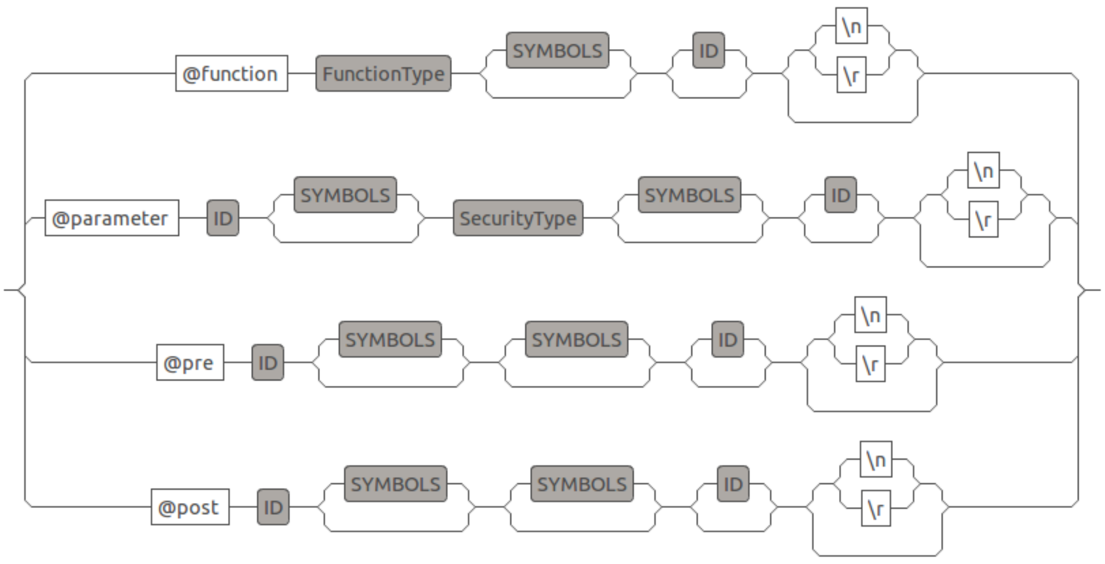


Figure 3.10.: Multi-line annotation tags defined in four syntax graph branches

The implementation details of the `FunctionAnnotation` entity are highlighted in Figure 3.10. Each of the four branches begins with one of the tags: `@function`, `@parameter`, `@pre` and `@post`. The object `FunctionType` present on the upper branch contains a static defined list containing the flags: `trust boundary`, `declassification`, `sanitization`, `sink` and `source`. The object `SecurityType` is composed of a static defined list containing the flags: `sensitive` and confidential. The objects SYMBOLS and ID were already described in a previous section. The recursion element present at the end of each branch is described in section 3.7.

3.1.2.6 Method Header Entity

Figure 3.11 presents the syntax graph of the method header entity. The method header entity is used to recognize syntax elements representing function declarations. The method header entity is composed of a series of SYMBOLS, recursions and expressions. The * symbol is used to denote the fact that a parameter in the list of function declaration parameters could be a pointer. The Expression entity object is presented in detail in listing 3.5 and is used for recognizing C expressions. Multiplicity is achieved by traversing the loops (parallel lines) n times. Since we want to use our annotation language in an object oriented context, too, we decided to name this entity method header entity and not function declaration entity. This is just a convention that can easily be changed if needed.

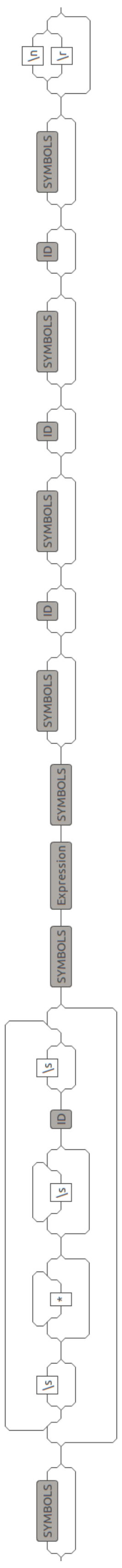


Figure 3.11.: Method header graph. Parallel lines are used for modeling symbol multiplicity and recursions.

3.1.2.7 Attribute Definition Entity

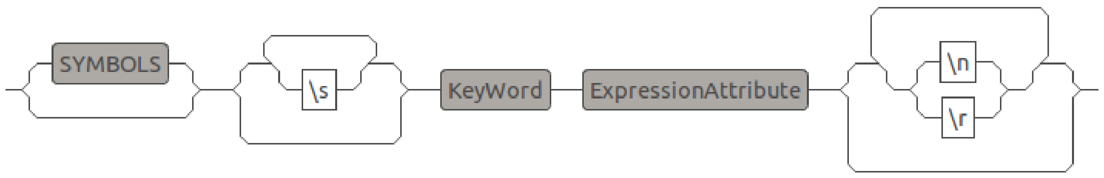


Figure 3.12.: Atribute definition syntax graph

Figure 3.12 presents the syntax graph of the attribute definition entity. This entity is used for recognizing any type of C/C++ expression which is not a function declaration or an annotation. The syntax graph contains one recursion at the beginning after the `SYMBOLS` object and one at the and of the branch. Both are presented in figures 3.6 and 3.7. `ExpressionAttribute` is used for recognizing the attributes of C expressions. The syntax graph presents the linear structure of the recognition process used for identifying attributes of C/C++ expressions.

3.1.3 xText Language Grammar

The language grammar of our policy language is defined using the xText grammar. This section presents the main grammar elements from a static perspective whereas in section 3.1.2 the same grammar entities were presented from a dynamic perspective. The advantage of a language grammar written in xText in comparison to an language grammar defined in ANTLR is that we get a powerful code infrastructure that can be reused right out of the box. First, a language parser is obtained that is capable of interpreting the language annotations as EObjects (ECORE representations of grammar entities). Second, a UI text editor is generated that already contains all the customization points needed for changing its editing behavior. It is not within the scope of this report to present all elements of our annotation language. We will highlight only the four entity objects present at the top of our grammar hierarchy. Also, the entity objects used to construct these top level objects will be briefly presented. In this section we highlight how to extend our language in order to add new grammar elements. The symbol `?` means an optional syntax element and the symbol `*` means multiplicity which can be between `0..N`.

Listing 3.1: Usage of a entity rule inside another entity rule

```
MyEntityRule:
...
myDelimiterVariable = SYMBOLS;
...
);
```

Listing 3.1 presents the usage of the entity rule `SYMBOLS` inside the entity rule `MyEntityRule` through the usage of an assignment. If the user tryes to use an entity rule inside another rule without the usage of an asignment in order to compose complex syntactical expressions then the `xText` autovalidation would complain.

Listing 3.2: Single-line annotation grammar

```
SingleLineAnnotation returns SingleLineAnnotation:
 1:  {SingleLineAnnotation}(
 2:     result +='//@ @function ' functionType = FunctionType
 3:     ((firstDelimiter = SYMBOLS))? ((nameofTheComment = ID))?
 4:                               ('\n' | '\r')*
 5:            | '//@ @parameter ' parameter = ID (' ')?
 6:              securityType = SecurityType
 7:     ((firstDelimiter = SYMBOLS))?((nameofTheComment = ID))?
 8:                                ('\n'|'\r')*
 9:            | '//@ @pre '   parameter=ID (' ')?
10:     ((firstDelimiter = SYMBOLS))?
11:                                ((nameComment=ID))?
12:                               ('\n' | '\r')*
13:          | '//@ @post '   parameter=ID (' ')?
14:   ((firstDelimiter = SYMBOLS))?
15:                              ((nameComment=ID))?
16:                            ('\n' | '\r')*
17: );
```

In listing 3.2 the entity object `SingleLineAnnotation` is used for adding the single-line annotation elements to our annotation language. It returns an object of the same type. The return object type is used during the creation of EObjects from textual annotations. `SingleLineAnnotation` is composed out of the list `result`. The list `result` is composed of four branches which are separated by the or symbol `|` present at line 5. The first branch in the list `result`, present in listing 3.2 at line 2, contains the string `'//@ @function '` which together with the `FunctionType` entity object composes one possible user typed single-line annotation statement. The second branch in the list `result` contains the string `'//@ @parameter '` followed by any type of `ID` and then an optional free space `(' ')?`. After `(' ')?` the entity object `SecurityType` defines a set of two possible language annotations, `confidential` or `sensitive`. All four branches terminate with an optional `((firstDelimiter = SYMBOLS))?` containing the set of all symbols available in the language. `((nameofTheComment = ID))?` is a placeholder for every type of string. At the end of each branch, lines 4, 9, 12 and 16, the new line or the return xText grammar symbols are present, `('\n' | '\r')*`, with multiplicity `0..N`. This offers the possibility to have a new line or a return symbol defined at the end of a single-line annotation.

Listing 3.3: Multi-line annotation entity grammar

```
0:  MultilineAnnotation returns MultilineAnnotation:
1:   {MultilineAnnotation}(
2:      rule += ('/*@ ')?  ('* ')?
3:      functionAnnotation = FunctionAnnotation
4:    ('\n')? (' @*/')? (firstDelimiter = SYMBOLS)?
5:              |
6: ('*') ' ' ' ' ('@*/')(firstDelimiter = SYMBOLS)?('\n'|'\r')?
7:   );
```

The entity object `MultilineAnnotation` presented in listing 3.3 is used for adding the multi-line annotation elements to our annotation language. At the beginning of a multi-line annotation the characters `/*@` are needed and at the end of a multi-line annotation the characters `@*/` are needed. It returns an object of the same type. This is used during the creation of EObjects from textual annotations. `MultilineAnnotation` contains the list `rule`. The list `rule` is composed of two branches which are separated by the or symbol `|` at line 5. The first branch in the list `rule` contains the strings `'/*@ '`, `'* '` and the `functionAnnotation` entity object, on line 3. The first two components are optional because of the `?` symbol. The `functionAnnotation` entity object presented in the syntax graph 3.10 defines all annotation tags which can be added to multi-line comments. The first `MultilineAnnotation` entity branch has three optional elements at the end of the statement: `'\n'` new line, the string `' @*/'` and the `((firstDelimiter = SYMBOLS))?` symbol entity meaning that at the end of this statement we can optionally have any kind of symbol. The second branch in the list `rule` contains the string `('*') ' ' ' ' ('@*/')`. This describes the interior of a multi-line annotation. This is followed by `((firstDelimiter = SYMBOLS))?` and `('\n' | '\r')` which are described above.

Listing 3.4: Method header entity grammar

```
0:  MethodHeader returns MethodHeader : {MethodHeader}(
1:  ((firstDelimiter = SYMBOLS)? (' '?'*'* ' '* ID ' '?)*
2:    secondDelimiter = SYMBOLS
3:  exp += Expression
4:      thirdDelimiter = SYMBOLS)
5: ((fourthDelimiter = SYMBOLS)?(ID?)
6:    (fifthDelimiter = SYMBOLS)?(ID?)
7:    (sixthDelimiter = SYMBOLS)?(ID?))
8:   seventhDelimiter = SYMBOLS?
9:  )('\n' | '\r')?;
```

The entity object `MethodHeader`, shown in listing 3.4, is used for recognizing any kind of C/C++ statement representing a function declaration. It returns an object of the same type. This is used during the creation of EObjects from textual annotations. The list `exp` is composed of *before* and *after* components.
As allready mentioned in listing 3.1 the usage of entity rules inside other rules is not possible in xText without the assignment of an variable to the used rule. In listing 3.4 seven delimiter variables are used in order to model all possible apperarances of C/C++ function declaration elements. Function declaration elements are modeled using seven delimiter variables. The values which can go inside them are at the right of the assignments.
The *before* components are represented by the elements from line number 1 to 2. We have an optional `SYMBOL`, optional space `' '`, optional `'*'` with multiplicity `0..N`, an optional space `' '` with multiplicity followed by an `ID` and optional space. All these elements except `firstDelimiter` have multiplicity `0..N`. The inside `?` symbol on the first line makes sense since the ending multiplicity, `0..N`, is addressing the whole terminal part of the `firstDelimiter` and not individual elements. `secondDelimiter = SYMBOLS`, line 2, is used to make the language capable of recognizing any symbol which could follow afterwards.
The list `exp` contains the entity object `Expression` and also the `thirdDelimiter` up to the `seventhDelimiter`. The five delimiters are used for handling throw statements declared at the end of function definitions. Standard C function declarations have throw statements that are not used. These throw statements are actually only in C++ used. The compiler replaces them with the appropriate code during code compilation. If a function has more throw statements then these five delimiters need to be replaced by a more general recursion. This remains at this

stage future work. The last part of each delimiter is a series of optional names (`ID`) and symbols (`SYMBOLS`) used to recognize any type of expression which could fit here. The `MethodHeader` entity object rule is terminated with an optional new line or return `('\n' | '\r')`.

Listing 3.5: Expression and EntityRef grammar entities

```
Expression returns Ref:
    EntityRef(
    ({Expression.ref = current}
    (firstDelimiter += SYMBOLS) tail = EntityRef)*
   );
EntityRef returns Ref:
    {EntityRef}(
    secondDelimiter += SpecialExpression
   )*;
```

`Expression` presented in listing 3.5 is used inside the `MethodHeader` entity for recognizing series of the `*` symbol. The `*` symbol is used in method headers for declaring pointer parameters. `EntityRef` is used inside the `Expression` entity object in order to define a recursion. `SpecialExpression` is presented in detail in Appendix C.

Listing 3.6: Atribute definition grammar entity

```
0: AttributeDefinition:
1:  {AttributeDefinition}(
2:      attribute_def += (SYMBOLS)?
3:    (' '?)*extension += KeyWord
4:         (extension_2  += ExpressionAttribute)
5:                          )('\n'|'\r')*;
```

The entity object `AttributeDefinition`, listing 3.6, is used for recognizing any kind of C/C++ attributes and definitions. It returns an object of the same type. This entity is used during the creation of EObjects from textual annotations. `AttributeDefinition` is the most compact entity rule used in our grammar but at the same time the most complex one. It is composed of three lists presented in listing 3.6 at lines 2, 3 and 4: `attribute_def`, `extension` and `extension_2`. The list `attribute_def` is composed of the lists: `extension` and `extension_2`. The first component of `attribute_def` is an optional `SYMBOL` and space symbol and then the entity object `KeyWord`. `KeyWord` contains all the language keywords like: `define`, `undef`, `ifdef`, `if`, `endif`, `typedef`, etc. The second component of `attribute_def` is the list `extension_2` containing the entity object `ExpressionAttribute` which contains a recursion used for recognizing C/C++ expressions containing the entity object `SYMBOLS` and the entity `EntityRef` which was previously presented. At the end of `AttributeDefinition` we have an optional new line or return `('\n' | '\r')` with multiplicity `0..N`.

3.2 xText Language Artifacts

3.2.1 Reusable Language Artifacts

Figure 3.1 available on page 24 presents the basic structure of the default xText project. The standard xText project is designed for code reuse, rapid grammar expandability and it is composed

of 4 sub-projects. The grammar can be extended by editing the `.xtext` file and code can be generated by editing the `.mwe2` file.
Both files are contained in the .mydsl subproject. The user can generate new code by first editing the .xtext file and then recompiling it by pressing the sub-menu `Run As->Generate xText Artifacts` and then optionally he can edit the `.mwe2` file. Code can be generated by pressing `Run As->MWE2 Workflow`. The last step generates code in the sub-projects `.mydsl`, `mydsl.tests and mydsl.ui`. The code generated in these sub-projects can be internally or externally used in other Eclipse projects. We will describe the code generated in each sub-project.

3.2.1.1 Parser Infrastructure

The code generated in `de.in.tum.sec.mydsl` can be reused as stand-alone code or can be integrated into other Eclipse plug-in infrastructures and used during run-time. We used the second approach to integrate the generated code into our existing SAE [24] engine. The generated code is composed of 13 packages containing a fixed number of classes depending mostly on the number of entity objects contained in the xText language grammar. The packages contain classes for stand-alone (only one run after the `Run` button is pressed) and run-time (as you type the annotations) running. For running the code as stand-alone the class `MyDslStandaloneSetup` should be used. For running the code during run-time (in closed loop) the class `MyDslRuntimeModule` should be used. Also, entity classes representing each entity object from the xText language grammar are contained in the 13 packages and interfaces used to map the entity objects hierarchy defined by the grammar into code. The packages contain the classes `MyDslAntlrTokenFileProvidertoken` and `MyDslParser` which extends the class `AbstractAntlrParser`. The `InternalMyDslLexer` and `InternalMyDslParser` have all the lexing and parsing rules hard-wired inside and two files. The file `InternalMyDsl.g` contains grammar rules and the file `InternalMyDsl.tokens` contain the language tokens.
The class `MyDslSemanticSequencer` is used for creating semantic sequences from the context and a semantic object. The `MyDslSyntacticSequencer` is used for instantiating matching rules for all the rules defined in the grammar. `MyDslGrammarAccess` is used for accessing grammar rules and a `AbstractMyDslValidator` which can be extended for adding language validation rules. `MyDslFormatter` is used for configuring the language formatting. `MyDslGenerator` is used for code generation. `MyDslScopeProvider` is used for doing some actions based on the current language scope. `MyDslValidator` is used for adding validation functionalities to the language. A model folder contains the generated ECORE model of the whole xText language grammar. We reused all the 13 packages in our SAE engine code infrastructure. The class `MyDslRuntimeModule` was used to integrate the language parser into our existing infrastructure.

3.2.1.2 jUnit Tests

The package `de.in.tum.sec.mydsl.tests` contains two generated classes. The first class `MyDslInjectorProvider` implements the interface `IInjectorProvider` and the interface `IRegistryConfigurator`. The interfaces are used to obtain an injector instance from the `GlobalRegistries` class contained in `org.eclipse.xtext.junit4.GlobalRegistries`.
The second class `MyDslUiInjectorProvider` implements the interface `IInjectorProvider` used for obtaining an injector instance from the `MyDslActivator` class. The injector instance is used together with `com.google.inject.Injector` to write test cases for unit testing the

generated code infrastructure. The classes are used in order to get an injector instance.

3.2.1.3 xText UI Editor

The code generated in `de.in.tum.sec.mydsl.ui` package is a stand-alone Eclipse plug-in used for annotating text files. It contains 14 packages from which 4 packages contain only `.xtend` files used for code generation customization. `MyDslUiModule` extends the abstract class `AbstractMyDslUiModule` and contains only the constructor method. The constructor method calls `super(plug-in)` in the superclass. `AbstractMyDslUiModule` extends `DefaultUiModule` and contains some binding methods. `MyDslExecutableExtensionFactory` is used to get the plug-in bundle and the plug-in injector singleton instance. The assist package contains `AbstractMyDslProposalProvider` which extends `TerminalsProposalProvider`. It is used for the definition of rule completion.
The `MyDslParser` class presented in section 3.2.1.1 extends `AbstractContentAssistParser` and represents the main parser class used by the editor. `PartialMyDslContentAssistParser` extends `MyDslParser` and provides an `AbstractInternalContentAssistParser` object in order to get language elements. Copies of the `InternalMyDslLexer`, `InternalMyDslParser`, `InternalMyDsl.g` and `InternalMyDsl.tokens` are presented in the section 3.2.1.1.
`MyDslActivator` is the activator class for this plug-in project. The `MyDslProposalProvider` class is used for filtering proposals which the user gets to see when the user presses `CTRL+Space`. `MyDslDescriptionLabelProvider` is used for customizing the language labels.
The `MyDslOutlineTreeProvider` is used for getting the outline of the current position of the mouse cursor when typing the annotations and based on this information it is possible to do some operations on the language. `MyDslQuickfixProvider` is used for providing quick-fixes. The `MyDslProposalProvider` class is used for filtering the proposed language elements so that we get only the comment entities which are relevant for annotation definition.

3.2.1.4 Parser Code Integration

In the class `ProgramStructureFacade` a new object of type `AnnotationExecution` is instantiated having an array list as parameter containing `IASTTranslationUnit` objects. The instance object `IASTTranslationUnit` is the AST representation of a source or header file. This offers the possibility to analyze annotations contained in C source or header files. Currently we annotate only header files. In the future we plan to be capable to add annotations also inside method bodies. As a consequence source files need to be handled as well. The `AnnotationExecution` constructor instantiates a `TranslationunitMapper` object which maps the annotation to the corresponding C header definitions. In the same constructor an `AnnotationParser` instance is created and used to map the previously obtained annotation map to objects. This means that the raw string comment is converted to an EObject representation containing the whole language hierarchy as this was defined in the language grammar. Afterwards each object representing a comment is inserted into a map having the annotation header raw string as the key and the EObject comment as the value. An EObject comment object can be of type `FunctionComment` or `ParameterComment`.

3.2.2 Two Click Language Extensibility Work-flow

Figure 3.13 presents the steps needed to extend the language grammar and how to generate the parser code infrastructure and the the UI text editor presented in the previous section.

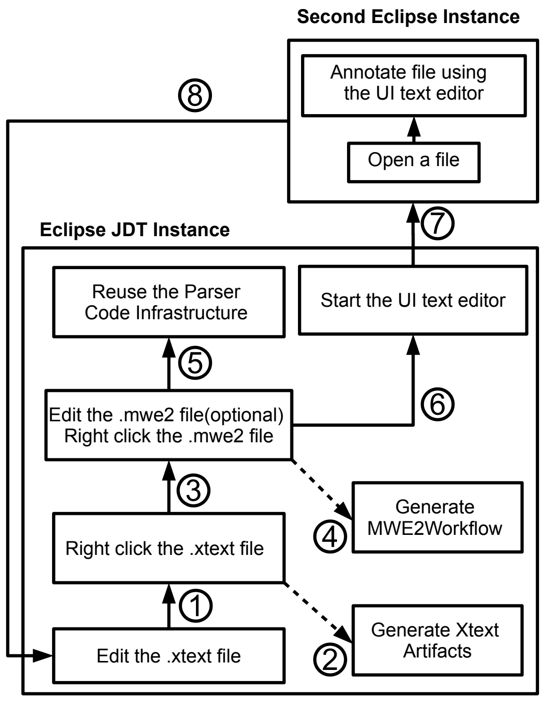

Second Eclipse Instance
Annotate file using the UI text editor
Open a file
⑧
⑦
Eclipse JDT Instance
Reuse the Parser Code Infrastructure
Start the UI text editor
⑤
Edit the .mwe2 file(optional) Right click the .mwe2 file
⑥
③
Generate MWE2Workflow
④
Right click the .xtext file
①
Generate Xtext Artifacts
Edit the .xtext file
②

Figure 3.13.: Work-flow used for extending the language grammar, generating a new code infrastructure and annotating textual files

The user has the possibility to easily extend the language by first editing the .xtext grammar file and then right clicking on the same file indicated in figure 3.13 with ① and generating the xText artifacts indicated with number ②. Afterwards the user can edit the .mwe2 file indicated with number ③ and generate code indicated with number ④. Number ⑤ indicates the parser code infrastructure generation and the code updating for the UI text editor. Number ⑥ indicates the starting of the UI text editor. Number ⑦ indicates the launch of a second Eclipse instance. The user can edit the header files contained for example in a library and reuse the files during static analysis. Number ⑧ indicates this process which can be repeated iteratively. The process is repeated in order to extend the language grammar and generate new code infrastructures.

Chapter 4: Applying the Policy Language

4.1 Scenarios

This section presents two scenarios where our policy language was used. We mainly present the possibilities to annotate text files contained in an arbitrary library and UML state charts. The goal of presenting these two scenarios is to give a sense in which domains and how our annotation language can be used. We emphasize the types of user input possibilities and briefly explain how the annotations are extracted and used during static analysis.

4.1.1 Annotation of C/C++ Libraries

Figure 4.1 presents the work-flow for annotating C header files contained in an arbitrary library. As indicated in step ① the user annotates the `getenv()` function declaration contained in `stdio.h` file and the `printf()` function declaration contained in the `stdlib.h` file. After annotating the files, the comments are extracted from the header files indicated in step ②. In step ③ comments are mapped to function declarations. Here the comments are associated to function declarations. After mapping the comments to function declarations the annotations are converted into EObjects corresponding to the entity objects specified by the grammar. Number ④ indicates the static analysis step in which the annotation objects are used in order to tag parameters or trust boundaries. Number ⑤ indicates a bug report if a bug was detected.

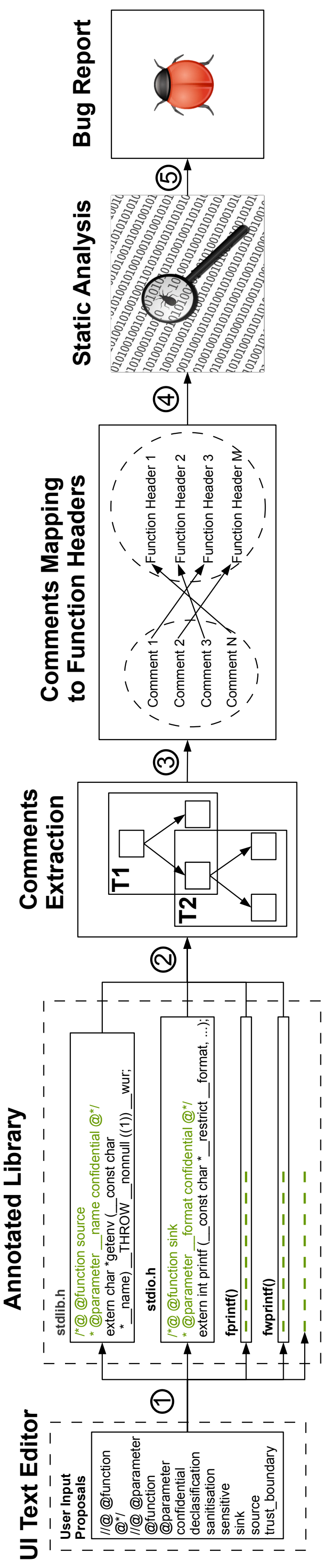


Figure 4.1.: Library annotation work-flow

4.1.2 Annotating Trust-Boundaries in Header Files

For annotating the trust boundaries in header files we created an Eclipse workspace where we imported the UI language text editor and our IF checker, both as Eclipse plug-ins. We started one of the plug-ins. We imported the test programs contained in the test cases CWE-526 [25] (Information Exposure Through Environmental Variables), CWE-534 [26] (Information Exposure Through Debug Log Files) and CWE-535 [27] (Information Exposure Through Shell Error Message) as separate C/C++ projects into the Eclipse CDT workspace. As a second Eclipse instance was started containing the previously imported test programs.
We created a copy of the C standard library containing some of the header files used by the selected test programs. The folder was imported as a virtual folder into the workspace. This offers the advantage that it is not necessary to physically copy the folder into each test case.
In the background the UI text editor plug-in was running in the second Eclipse instance. By double clicking on one of the files (only files with a supported file extension e.g. ".h", ".hpp", etc. can be annotated) contained in one of the projects the UI language text editor pops up a message window offering the possibility to parse the context of the selected file.
By positioning the mouse cursor above the function declaration and pressing `CTRL+Space` a pop-up menu which is shown in figure 4.2 offers the annotations available in that context.
The user doesn't need to know the exact syntax of the policy annotation language. The user only has to select the desired annotation tags from the list.

```
/*@
//@ @function
@*/
//@ @parameter
@function
@parameter
@pre
@post
confidential
declassification
sanitization
sensitive
sink
source
trust_boundary
```

Figure 4.2.: UI text editor proposals list

The function declarations `getenv()` and `printf()` available in the files `stdio.h` and `stdlib.h`, respectively, were annotated. These function declarations represent the trust boundaries of the test programs contained in the test case CWE-526. The added annotations are highlighted in listing 4.1 and listing 4.2 with green color.

Listing 4.1: Annotated getenv() function declaration

```
/*@ @function source
 * @parameter __name confidential @*/
 extern char *getenv (__const char
                 *__name) __THROW __nonnull ((1)) __wur;
```

Listing 4.2: Annotated printf() function declaration

```
/*@ @function sink
 * @parameter __format confidential @*/
extern int printf (__const char *__restrict __format, ...);
```

The test cases CWE-534 and CWE-535 contain the trust boundaries: `LogonUserA()`, `LogonUserW()`, `fprintf()` and `fwprintf()`. The trust boundaries (sources) `LogonUserA()` and `LogonUserW()` contained in the Windows OS header file `windows.h` were annotated. Since this header file is not available under Ubuntu OS we created the header file `dummyHeader.h` containing the annotated function declarations. Also, the trust boundaries (sinks) `fprintf()` contained in `stdio.h` and `fwprintf()` contained in the `wchar.h` were annotated. The annotations are highlighted in listings 4.3, 4.4, 4.5 and 4.6 with green color.

Listing 4.3: Annotated LogonUserA() function declaration

```
/*@ @function source
 * @parameter password confidential @*/
int LogonUserA(
            char *username,
            char *domain,
            char *password,
            int LOGON32_LOGON_NETWORK,
            int LOGON32_PROVIDER_DEFAULT,
           HANDLE pHandle);
```

Listing 4.4: Annotated fprintf() function declaration

```
//@ @function sink
extern int fprintf(
FILE *__restrict __stream,
__const char *__restrict __format, ...);
```

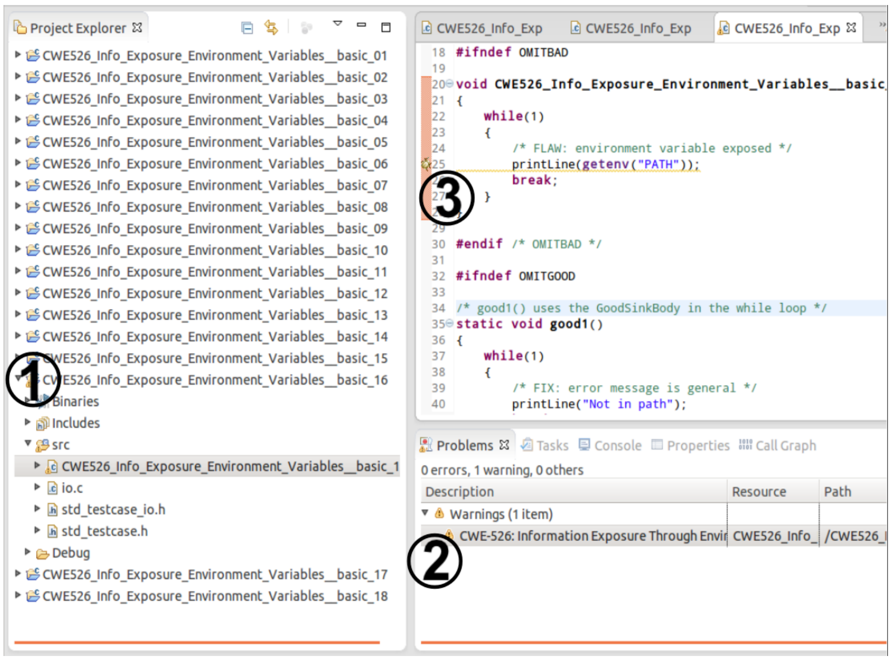
Listing 4.5: Annotated LogonUserW() function declaration

```
/*@ @function source
 * @parameter password confidential @*/
int LogonUserW(
            char *username,
            char *domain,
            char *password,
            int LOGON32_LOGON_NETWORK,
            int LOGON32_PROVIDER_DEFAULT,
           HANDLE pHandle);
```

Listing 4.6: Annotated fwprintf() function declaration

```
//@ @function sink
extern int fwprintf(
__FILE *__restrict __stream,
__const wchar_t *__restrict __format,
...);
```

After the annotation process we run the previously mentioned IF checker on each of the test cases separately. In the first step all the files available in one project are parsed. In the second step an algorithm creates the mappings between annotation and function declarations. Our mapping algorithm is capable of associating annotations and function declaration even when they are separated by multiple empty lines. The annotations and function declarations are put into a map. The IF checker is based on header function models. Each function model has a static field containing the function declaration from the header files. The function model execute method checks, if the function annotation map contains the statically defined function as the key. If this is the case then an `EObject` is retrieved from the map and all annotations are used in the `execute()` method. The annotations extracted from the `EObject` are used to tag a function model as a trust-boundary or to tag confidential variables. Annotation extraction from `EObjects` is done by querying, `annotationObject.getTrustBoundary()`, the object about the values stored in his attribute fields. In this way we avoided hard coding the function declaration models into the `execute()` method but rather having the initialization done in the header files as function annotations.



```
CWE526_Info_Exposure_Environment_Variables__basic_01
CWE526_Info_Exposure_Environment_Variables__basic_02
CWE526_Info_Exposure_Environment_Variables__basic_03
CWE526_Info_Exposure_Environment_Variables__basic_04
CWE526_Info_Exposure_Environment_Variables__basic_05
CWE526_Info_Exposure_Environment_Variables__basic_06
CWE526_Info_Exposure_Environment_Variables__basic_07
CWE526_Info_Exposure_Environment_Variables__basic_08
CWE526_Info_Exposure_Environment_Variables__basic_09
CWE526_Info_Exposure_Environment_Variables__basic_10
CWE526_Info_Exposure_Environment_Variables__basic_11
CWE526_Info_Exposure_Environment_Variables__basic_12
CWE526_Info_Exposure_Environment_Variables__basic_13
CWE526_Info_Exposure_Environment_Variables__basic_14
CWE526_Info_Exposure_Environment_Variables__basic_15
CWE526_Info_Exposure_Environment_Variables__basic_16
  Binaries
  Includes
  src
    CWE526_Info_Exposure_Environment_Variables__basic_1
    io.c
    std_testcase_io.h
    std_testcase.h
  Debug
CWE526_Info_Exposure_Environment_Variables__basic_17
CWE526_Info_Exposure_Environment_Variables__basic_18
```

```
18 #ifndef OMITBAD
19
20 void CWE526_Info_Exposure_Environment_Variables__basic
21 {
22     while(1)
23     {
24         /* FLAW: environment variable exposed */
25         printLine(getenv("PATH"));
26         break;
27     }
28
29
30 #endif /* OMITBAD */
31
32 #ifndef OMITGOOD
33
34 /* good1() uses the GoodSinkBody in the while loop */
35 static void good1()
36 {
37     while(1)
38     {
39         /* FIX: error message is general */
40         printLine("Not in path");
```

```
0 errors, 1 warning, 0 others
Description                                       Resource       Path
▼ Warnings (1 item)
   CWE-526: Information Exposure Through Envir   CWE526_Info_   /CWE526_
```

Figure 4.3.: Bug report presentation in Eclipse

Bug reports are issued when a previously annotated variable as confidential or sensitive is sent to a function by calling it. The function call represents the passing of a confidential/sensitive variable over a trust-boundary. The numbers ①, ② and ③ in figure 4.3 indicate the analyzed test program, the bug report and the bug location (line number) in a file.

4.1.3 Annotation of UML State Charts

Figure 4.4 presents the steps needed to annotate an UML state chart and and detect bugs during static analysis.
Steps $\textcircled{1a}$ and $\textcircled{1b}$ represent the input methods that can be used to annotate UML state charts. In $\textcircled{1a}$, the user has a pallet with symbols which have the same significance as the language annotation tags, shown in step $\textcircled{1b}$. The only difference between the two methods is that the pallet represents a higher level of abstraction regarding the input technique.
With the input technique shown in step $\textcircled{1b}$, the user user can edit the UML state chart by providing grammar annotation elements proposed from the pop-up menu. This input method is similar to the one presented in the first scenario. Numbers $\textcircled{2a}$, $\textcircled{2b}$, $\textcircled{2c}$, $\textcircled{2d}$ and $\textcircled{2e}$ represent attached annotations to functions. After the annotation process is done there are two possible ways to proceed. First, in the upper branch in step $\textcircled{3a}$ the header source files are generated from the UML state charts and are then analyzed in scenario 1. Numbers ④, ⑤, ⑥ and ⑦ represent comment extraction, comment mapping, static analysis and bug report generation, respectively. Second, in the lower branch in step $\textcircled{3b}$, static code analysis is done by simulating the execution of a state chart. Comments don't need to be extracted or mapped since they are already attached and mapped to the state chart. During the state chart simulation process tags attached to symbolic variables and are propagated. When a tainted function call is triggered it is checked if the previous function has the same name as current function tag. The previous function name is stored in the current target tag "`@pre previousFunctionName`". If the check is not true then a bug report is issued. This is indicated with number $\textcircled{4b}$.

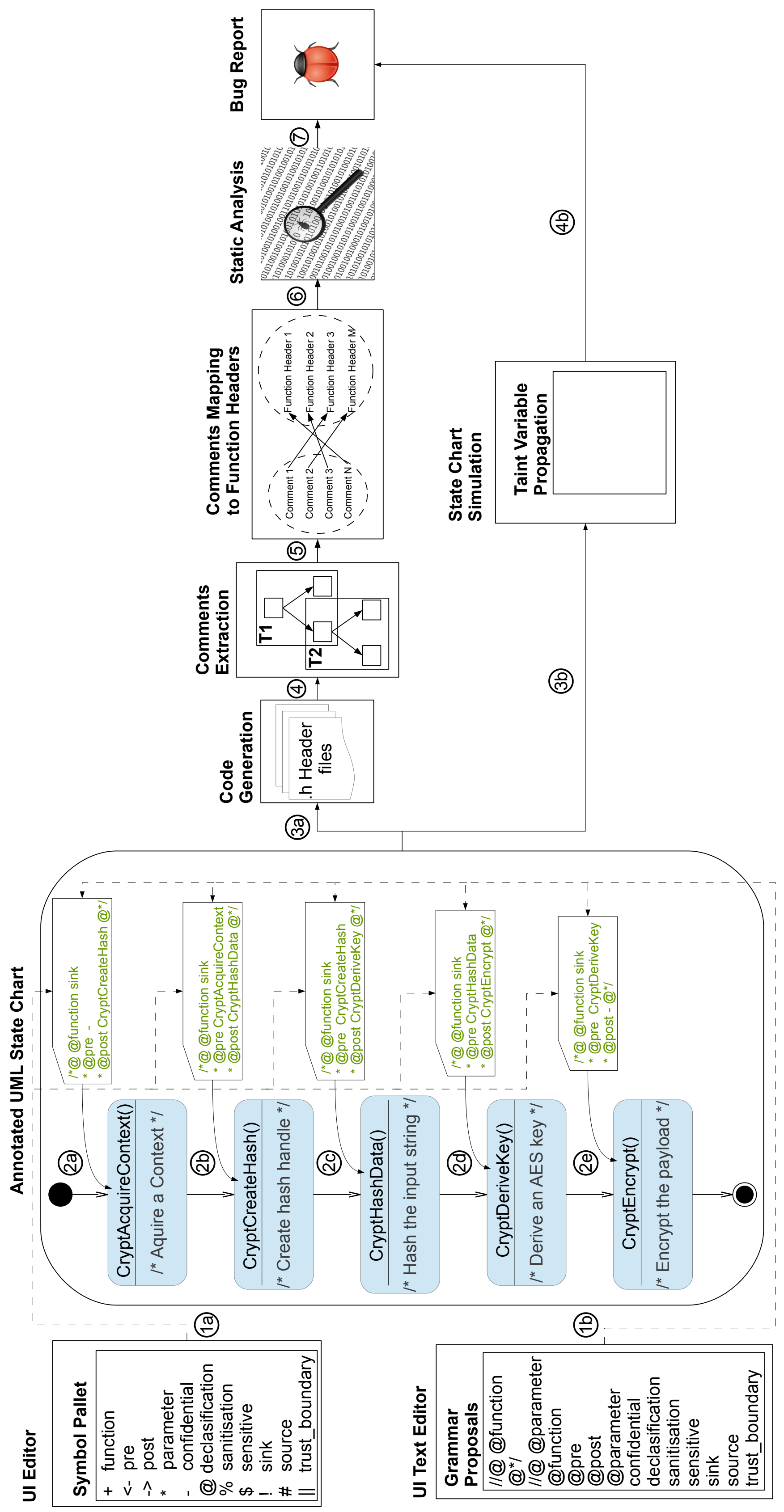


Figure 4.4.: UML state chart annotation work-flow

4.1.4 Annotating a Cryptographic Algorithm using UML State Charts

We selected CWE-325 [28] (Missing Required Cryptographic Step) because we wanted to detect API misusage bugs by remodeling a cryptographic algorithm using UML state charts. When software does not implement a required step (a required step means in this context a step that an algorithm has to perform so that it conforms to its predefined order of steps) in a cryptographic algorithm it results in a weaker encryption than advertised by that algorithm. Listing 4.7 presents a code snippet contained in a text program extracted from CWE-325. Our goal is to detect misusage of APIs by checking if the neighbouring function calls in a chain of function calls match the annotation tags previously attached to the function declarations.

Listing 4.7: Five steps cryptographic algorithm code snippet extracted from CWE-325. Green text represents C code comments and no annotations

```c
/* Copy plaintext into payload buffer */
memcpy(payload, PAYLOAD, payloadLen);
/* Aquire a Context */
if(!CryptAcquireContext(&hCryptProv, NULL,
    MS_ENH_RSA_AES_PROV, PROV_RSA_AES, 0)){
                        break;
}
/* FIX: All required steps are present */
/* Create hash handle */
if(!CryptCreateHash(hCryptProv, CALG_SHA_256,
      0, 0, &hHash)){
                        break;
}
/* Hash the input string */
if(!CryptHashData(hHash, (BYTE*)hashData,
                       strlen(hashData), 0)){
                        break;
}
/* Derive an AES key from the hash */
if(!CryptDeriveKey(hCryptProv, CALG_AES_256,
                        hHash, 0, &hKey)){
                        break;
}
/* Encrypt the payload */
if(!CryptEncrypt(hKey, 0, 1, 0, payload,
      &payloadLen, sizeof(payload))){
                        break;
}
```

After attaching annotations to the UML state chart code can be generated or the execution of the state chart can be simulated as described in section 4.1.3. The annotations can be used in the same manner as is done for header files. This can be achieved by generating source code from the annotated UML state charts. A state can be converted to a function declaration and the annotation post-it can be converted to annotation text attached to the function declaration. The

next steps are similar to the ones presented in Section 4.1.2.
We remodeled the code snippet presented in listing 4.7 with UML state charts and annotated each function call. We have annotated each function contained in figure 4.4 with the tag `sink` and we attached the tags `@pre` and `@post` containing the previous function call name and the next function call name. In order to detect API misusage bugs it is needed to check from the first function call in the chain of function calls if the previous function name is the same with the one stored in the `@pre` tag. Additionally the `@post` tag can be checked if the next function call name is the same as the one stored in the `@post` tag. If there is a mismatch regarding the `@pre` or `@post` tag then the checking algorithm is stopped and a bug report is issued. It is possible that there are other function calls or code between the algorithm function calls. Thus, it is not sufficient to look only at the previous node or the next node when checking the `@pre` and `@post` tags on an execution path. It is necessary to look back and forward in the whole current context. Thus, this is a necessary and sufficient condition since the function calls of the cryptographic algorithm manipulate only local variables.
If the annotations are used as presented in figure 4.4 on the lower branch indicated with ⓷b and ⓸b then the post-it boxes containing the annotations can be interpreted as final states. These final states can be executed just before one of the annotated states is executed. In this way the annotation is used before a state is about to be executed. Another possibility is to integrate the post-it annotation inside or after the annotated state in order to specify internal UML state execution restrictions or post UML state execution restrictions.

Chapter 5: Conclusions

The policy language presented in this report is intended to be used for annotating UML state charts and source code files with the goal of annotating sinks, sources, trust boundaries, sanitization and declassification functions. The language should allow or disallow certain information flows or information flow volumes.
Briefly the main challenges faced during the design and implementation phase of our language are presented in the report. We successfully implemented a policy language based on xText that can be used on the modeling level and also on the source code level of an application. Language editors are presented and we give a brief overview of two scenarios for language usage.
The language implementation is based on generality and can be easily integrated into other IDEs. We would like to emphasize the fact that our language is lightweight and is suited for large projects as well as for scenarios where an extended set of annotation tags is needed.
We conclude that a successful annotation language in the general sense should only pose a minimal learning and usage burden on the user. The annotation tags should rather be chosen based on the current context. The user should not have to search through a long list of possible annotation tags. The language should be flexible in the sense that the set of annotation tags should be easily extendable and the code supporting infrastructure (parser, lexer, etc.) should be automatically generated rather than written by hand. In this sense we present a simple work-flow that guarantees that the language can easily be extended regarding the previous two mentioned perspectives. In our opinion the annotation process itself should be automated by propagating annotations without the need of human intervention. At this stage this still remains future work.
Concluding, we'd like to emphasize that we developed a policy language that can easily be used right out of the box for annotating different types of artifacts in two design phases of a software project (design and coding) by providing a suitable policy language and tools to introduce security concerns in order to detect information exposure bugs.

Bibliography

[1] B. Chess and G. McGraw, “Static Analysis for Security ,” *IEEE Security & Privacy*, November/December 2004.

[2] Mitre, “CWE-200: Information Exposure ,” tech. rep., Mitre, http://cwe.mitre.org/data/definitions/200.html, accessed on July 2014.

[3] J. S. Fenton, “Memoryless subsystems,” *Computer Journal*, vol. 17, pp. 143–147, May 1974.

[4] A. Sabelfeld and A. Russo, “From dynamic to static and back: Riding the roller coaster of information-flow control research ,” *Proceedings of Andrei Ershov International Conference on Perspectives of System Informatics*, pp. 352– 365, 2009.

[5] T. Avgerinos, S. Cha, B. L. T. Hao, and D. Brumley, “AEG: Automatic Exploit Generation,” *Proceedings of the Network and Distributed System Security Symposium (NDSS 11)*, February 2011.

[6] M. Guarnieri, P. E. Khoury, and G. Serme, “Security vulnerabilities detection and protection using Eclipse ,” *ECLIPSE-IT 2011, 6th Workshop of the Italian Eclipse Community*, September 2011.

[7] D. Volpano, G. Smith, and C. Irvine, “A sound type system for secure flow analysis,” *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, 1996.

[8] V. Simonet, “The Flow Caml System: documentation and user’s manual ,” tech. rep., INRIA, July 2003.

[9] X. Xiao, N. Tillmann, M. Fahndrich, J. de Halleux, and M. Moskal, “Transparent Privacy Control via Static Information Flow Analysis ,” tech. rep., Microsoft, August 2011.

[10] A. C. Myers, “JFlow: Practical Mostly-Static Information Flow Control,” *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL ’99)*, January 1999.

[11] S. Moore and S. Chong, “Static analysis for efficient hybrid information-flow control ,” *CSF ’11 Proceedings of the IEEE 24th Computer Security Foundations Symposium*, pp. 146–160, 2011.

[12] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe, “Extended Static Checking ,” *Compaq SRC Research Report 159*, 1998.

[13] K. R. M. Leino, “Extended Static Checking: a Ten-Year Pesrsective,” *Proceeding Informatics - 10 Years Back. 10 Years Ahead*, pp. 157–175, January 2001.

[14] Microsoft, “MSDN run-time library reference - SAL annotations.” http://msdn.microsoft.com/en-us/library/ms235402.aspx. Microsoft.

[15] T. Ball, B. Hackett, S. Lahiri, and S. Qadeer, “Annotation-based property checking for systems software,” tech. rep., Microsoft, May 2008.

[16] V. Simonet, *FlowCaml in a nutshell.* In G. Hutton, ed. APPSEM-II, 2003.

[17] S. Chong, A. C. Myers, N. Nystrom, L. Zheng, and S. Zdancewic, “Jif: Java + information flow,” July 2006. Software release.

[18] N. Swamy, B. J. Corcoran, and M. Hicks, “Fable: A language for enforcing user-defined security policies ,” *In S&P*, 2008.

[19] L. Jia, J. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic, “Aura: A programming language for authorization and audit ,” *ICFP*, 2008.

[20] N. Swamy, J. Chen, and R. Chugh, “Enforcing Stateful Authorization and Information Flow Policies in FINE ,” *In proceedings of ESOP 2010: 19th European Symposium on Programming*, March 2010.

[21] Eclipse, “xText Documentation,” tech. rep., Eclipse, iTemis, http://www.eclipse.org/Xtext/documentation.html, accessed on July 2014.

[22] D. S. Rosenblum, “A practical approach to programming with assertions,” *IEEE Transactions on software engineering*, vol. 21, January 1995.

[23] D. S. Rosenblum, “Towards a method of programming with assertions ,” *ACM*, 1992.

[24] A. Ibing, “SMT-Constrained Symbolic Execution for Eclipse CDT/Codan ,” *Workshop on Formal Methods in the Development of Software*, 2013.

[25] Mitre, “CWE-526: Information Exposure Through Environmental Variables ,” http://cwe.mitre.org/data/definitions/526.html.

[26] Mitre, “CWE-534: Information Exposure Through Debug Log Files ,” http://cwe.mitre.org/data/definitions/534.html.

[27] Mitre, “CWE-535: Information Exposure Through Shell Error Message ,” http://cwe.mitre.org/data/definitions/535.html.

[28] Mitre, “CWE-325: Missing Required Cryptographic Step,” http://cwe.mitre.org/data/definitions/325.html.

Appendix A: Comments to function declarations mapping code

```java
public class TranslationunitMapper extends ASTCommenter{
    /** The map. */
    public HashMap<Integer, String> map = new HashMap<Integer, String>();
    /** The list. */
    public StringBuffer theList = null;
    /** The allfilescontent. */
    public String allfilescontent =null;
    /** The comments method attribute map. */
    HashMap<String, String> commentsMethodAttributeMap = null;
    /** The maximum multiline comment length. */
    private static final int maximumMultilineCommentLength = 30;
    /** The Constant multiLineStartingString. */
    private static final String multiLineStartingString = "/*@";
    /** The Constant multiLineBeginningString. */
    private static final String multiLineBeginningString = "* @";
    /** The Constant singleLineStartingString. */
    private static final String singleLineStartingString = "//@";
    /** The Constant newLine. */
    private static final String newLine = "\n";
    /** The Constant emptyString. */
    private static final String emptyString = "";
    //this is needed in order to have after each parameter comma an empty space
    /** The Constant space. */
    private static final String space = " ";
    /** The Constant headerFunctionTerminator. */
    private static final String headerFunctionTerminator = ";";
    /**
     * Instantiates a new parser commenter.
     */
    public TranslationunitMapper(){
```

```
    commentsMethodAttributeMap = new HashMap<String, String>();
  }
  /**
   * Gets the comments.
   * @param commentsList the comments list
   * @return the comments
   * @throws IOException Signals that an I/O exception has occurred.
   */
  public synchronized HashMap<String, String> getComments(ArrayList<IASTTranslationUnit> commentsList) throws IOException{
    Pattern pattern;
    Matcher matcher;
    for (IASTTranslationUnit ast : commentsList){
      MyLogger.log_parser("Analyzing translation unit: "+ast.getContainingFilename());
      //commentList = ast.getComments();
      String allHeaderFileContents = new String(ast.getOriginatingTranslationUnit().getContents());
      StringBuffer      stringBuffer = new StringBuffer();
      StringBuffer stringBufferAnnotation = new StringBuffer();
      String line;
      HashMap<Integer, String> commentsMap = new HashMap<Integer, String>();
      HashMap<Integer, String> methodAttributeMap = new HashMap<Integer, String>();

      //Variable bufferedReader parses the header file in string format
      BufferedReader bufferedReader = new BufferedReader(new StringReader(allHeaderFileContents));
      while ((line = bufferedReader.readLine()) != null) {
        stringBuffer.append(line);
        stringBuffer.append(newLine);
        /* This part parses the code which is annotated with our annotation
        */
        pattern = Pattern.compile("/*@ | //@");
        matcher = pattern.matcher(stringBuffer);
        if (matcher.find()){
          stringBufferAnnotation.append(line);
          stringBufferAnnotation.append(newLine);
        }
      }
      String allContentOfFileWithAnnotation = stringBufferAnnotation.toString();
      //variable bufreader read the string line by line
      BufferedReader bufReader = new BufferedReader(new StringReader(allContentOfFileWithAnnotation));
      String singleLine = null;
      String nextLine1 = null;
      String nextLine2 = null;
      String nextLine3 = null;
      String nextLine4 = null;
      String concat = emptyString;
      int lineNo = 1;
      int temp = 0;
      int commentNumberOfLines = 0;
      /** This portion create to HashMaps.One for comments and other for methods and attribute
      */
      while( (singleLine=bufReader.readLine()) != null ){
        //adding multiline comments
        if(singleLine.startsWith(multiLineStartingString)){
          commentNumberOfLines = 0;
          concat += singleLine;
          nextLine1 = bufReader.readLine();
          //avoid reading comments longer then a given size
          while(nextLine1.contains(multiLineBeginningString) && commentNumberOfLines < maximumMultilineCommentLength){
            temp = 1;
            lineNo++;
            concat += newLine+nextLine1;
            nextLine1 = bufReader.readLine();
            commentNumberOfLines++;
          }
          if((temp == 1)){
            commentsMap.put(lineNo, concat);
            temp = 0;
            concat = emptyString;
          }
          while(nextLine1.trim().equals(emptyString)){
            nextLine1=bufReader.readLine();
          }
          //put single line function declarations
          if(nextLine1.contains(headerFunctionTerminator)){
            methodAttributeMap.put(lineNo, nextLine1);
            //put multi-line function declarations
          }else if(!nextLine1.contains(
    headerFunctionTerminator))
            while(!nextLine1.contains(headerFunctionTerminator)){
              nextLine3 = bufReader.readLine();
              nextLine1 = nextLine1.trim() + space + nextLine3.trim();
            }
          //put the function declaration part in the map
          methodAttributeMap.put(lineNo, nextLine1.trim());
        }
        //adding single line comments
        //there will be only one single line comment added
        //the one that is just before the function declaration
        //the other ones will be omitted
        if(singleLine.contains(singleLineStartingString)){
          commentsMap.put(lineNo, singleLine);
          nextLine2 = bufReader.readLine();
          //avoid putting blank spaces, search for the ; symbol
          if(!nextLine2.trim().equals(emptyString) && nextLine2.contains(";");
            methodAttributeMap.put(lineNo, nextLine2);
          }
          else{
            //avoid putting a blank line into the map
            while(nextLine2.trim().equals(emptyString))
              nextLine2 = bufReader.readLine();
          }
          //put single-line function declarations
          if(nextLine2.contains(headerFunctionTerminator)){
            methodAttributeMap.put(lineNo, nextLine2);
            //put multi-line function declarations
          }else if(!nextLine2.contains(headerFunctionTerminator))
            while(!nextLine2.contains(headerFunctionTerminator)){
              nextLine4 = bufReader.readLine();
              nextLine2 = nextLine2.trim() + space + nextLine4.trim();
            }
          //put the function declaration part in the map
          methodAttributeMap.put(lineNo, nextLine2.trim());
        }
      }
      lineNo++;
    }
```

```
      //System.out.println(commentsMap+"\n");
      //System.out.println(methodAttributeMap+"\n");
      /** Combine two HashMaps into one map with comments and
    method_attribute pairs
      */
      for (Integer x : commentsMap.keySet()) {
        String y1 = commentsMap.get(x);
        String x1 = methodAttributeMap.get(x);
        //removing non-valid function declarations
        //both single and multi-line comments are added to the
        //commentsMethodAttributeMap
        if(!x1.contains(singleLineStartingString))
          commentsMethodAttributeMap.put(x1,y1);
      }
    }
    return commentsMethodAttributeMap;
  }
}
```

Appendix B: The mwe2 Configuration File

This source code is used for generating the reusable code infrastructure. The green highlighted text in this section represents usual Java code comments and it doesn't represent annotation language comments.

```
module de.in.tum.sec.mydsl.GenerateMyDsl
import org.eclipse.emf.mwe.utils.*
import org.eclipse.xtext.generator.*
import org.eclipse.xtext.ui.generator.*
var grammarURI = "classpath:/org/xtext/example/mydsl/MyDsl.
    xtext"
//list of supported file extensions by the editor
var fileExtensions = "h, hh, hhh, hxx, c, cpp, C, Cpp"
var projectName = "de.in.tum.sec.mydsl"
var runtimeProject = "../${projectName}"
var generateXtendStub = true
var encoding = "UTF-8"

Workflow {
bean = StandaloneSetup {
scanClassPath = true
platformUri = "${runtimeProject}/.."
//The following two lines can be removed, if Xbase is not used
    .
registerGeneratedEPackage = "org.eclipse.xtext.xbase.
    XbasePackage"
registerGenModelFile = "platform:/resource/org.eclipse.xtext.
    xbase/model/Xbase.genmodel"
}

component = DirectoryCleaner {
directory = "${runtimeProject}/src-gen"
}

component = DirectoryCleaner {
directory = "${runtimeProject}/model"
}

```

```
component = DirectoryCleaner {
directory = "${runtimeProject}.ui/src-gen"
}

component = DirectoryCleaner {
directory = "${runtimeProject}.tests/src-gen"
}

component = Generator {
pathRtProject = runtimeProject
pathUiProject = "${runtimeProject}.ui"
pathTestProject = "${runtimeProject}.tests"
projectNameRt = projectName
projectNameUi = "${projectName}.ui"
encoding = encoding
language = auto-inject {
uri = grammarURI

//Java API to access grammar elements(required by several other fragments)
fragment = grammarAccess.GrammarAccessFragment auto-inject {}

//generates Java API for the generated EPackages
fragment = ecore.EMFGeneratorFragment auto-inject {}

//the old serialization component
//fragment = parseTreeConstructor.ParseTreeConstructorFragment auto-inject {}

//serializer 2.0
fragment = serializer.SerializerFragment auto-inject {
generateStub = false
}

//a custom ResourceFactory for use with EMF
fragment = resourceFactory.ResourceFactoryFragment auto-inject {}

// The antlr parser generator fragment.
fragment = parser.antlr.XtextAntlrGeneratorFragment auto-inject {
options = {
backtrack = true
}
//execution timeout
antlrParam = "-Xconversiontimeout" antlrParam = "10000"
```

```
}

//Xtend-based API for validation
fragment = validation.ValidatorFragment auto-inject {
//composedCheck = "org.eclipse.xtext.validation.
    ImportUriValidator"
//composedCheck = "org.eclipse.xtext.validation.
    NamesAreUniqueValidator"
}

//old scoping and exporting API
//fragment = scoping.ImportURIScopingFragment auto-inject {}
//fragment = exporting.SimpleNamesFragment auto-inject {}

//scoping and exporting API
fragment = scoping.ImportNamespacesScopingFragment auto-inject
     {}
fragment = exporting.QualifiedNamesFragment auto-inject {}
fragment = builder.BuilderIntegrationFragment auto-inject {}

//generator API
fragment = generator.GeneratorFragment auto-inject {}

//formatter API
fragment = formatting.FormatterFragment auto-inject {}

//labeling API
fragment = labeling.LabelProviderFragment auto-inject {}

//outline API
fragment = outline.OutlineTreeProviderFragment auto-inject {}
fragment = outline.QuickOutlineFragment auto-inject {}

//quickfix API
fragment = quickfix.QuickfixProviderFragment auto-inject {}

//content assist API
fragment = contentAssist.ContentAssistFragment auto-inject {}

//generates a more lightweight Antlr parser and lexer tailored
     for content assist
fragment = parser.antlr.XtextAntlrUiGeneratorFragment auto-
    inject {
options = {
backtrack=true
}
```

```
//execution timeout
antlrParam = "-Xconversiontimeout" antlrParam = "10000"
}

//generates junit test support classes into Generator#
    pathTestProject
fragment = junit.Junit4Fragment auto-inject {}

//rename refactoring
fragment = refactoring.RefactorElementNameFragment auto-inject
     {}

//provides the necessary bindings for java types integration
fragment = types.TypesGeneratorFragment auto-inject {}

//generates the required bindings only if the grammar inherits
     from Xbase
fragment = xbase.XbaseGeneratorFragment auto-inject {}

//provides a preference page for template proposals
fragment = templates.CodetemplatesGeneratorFragment auto-
    inject {}

//provides a compare view
fragment = compare.CompareFragment auto-inject {}
   }
  }
}
```

Appendix C: Policy Language Grammar

This .xtext file contains the whole grammar of our policy language. The green highlighted text in this section represents usual Java code comments and it doesn’t represent annotation language comments.

```
grammar de.in.tum.sec.mydsl.MyDsl with
org.eclipse.xtext.common.Terminals

/** [xText Grammar description.  Logger]
    [Other notes: used for adding annotations to
    .h, .hh, .hhh, .hxx, .c, .cpp, .C and .Cpp files"]
    This grammar is intended to be used for annotating the
    above mentioned header files.
    @author Paul Muntean
    @version  Revision : 0.3
                  Date : 14.05.2014
                  Hour : 18:10:15 PM
**/
generate myDsl "http://www.xtext.org/example/mydsl/MyDsl"

/**
 * @AnnotationLanguage: top root node of the annotation
    language
 */
AnnotationLanguage:
     element += HeaderModel*
;

/**
 * @SingleLineAnnotation: entity used for single-line
    annotations
 * @MultilineAnnotation: entity used for multi-line
    annotations
 * @MethodHeader: entity used for recognizing any kind of C/C
   ++ headers
 * @AttributeDefinition: entity used for recognizing any kind
    of variable definition
 */
```

```
HeaderModel:
    headers += SingleLineAnnotation
             | MultilineAnnotation
             | MethodHeader
             | AttributeDefinition
;

/**
 * @AttributeDefinition : entity used for recognizing any kind of statement which begins with the symbol #
 */
AttributeDefinition:
  {AttributeDefinition}(attribute_def += (SYMBOLS)?(' '?)* extension += KeyWord (extension_2 += ExpressionAttribute))
  ('\n'|'\r')*
;

/**
 * @ExpressionAttribute : atribute of the AttributeDefinition
 */
ExpressionAttribute returns Ref:
     EntityRef(({Expression.ref = current}
     (symbols_attr += SYMBOLS) tail = EntityRef)*)
;

/**
 * @MethodHeader :this recognizes any kind of method headers
 */
MethodHeader returns MethodHeader :
    {MethodHeader}(
      ((firstDelimiter = SYMBOLS)?(' '?'*'* ' '* ID ' '?)*
       secondDelimiter = SYMBOLS exp += Expression
        thirdDelimiter = SYMBOLS)
     ((fourthDelimiter = SYMBOLS)?(ID?)
       (fifthDelimiter = SYMBOLS)?(ID?)
       (sixthDelimiter = SYMBOLS)?(ID?))
      seventhDelimiter = SYMBOLS?
     )  ('\n' | '\r')?
;

/**
 * @Expression :used for recognizing expressions inside a MethodHeader
 *              :it contains one recursion defined on the current entity object Expression.ref = current
 */
```

```
Expression returns Ref:
    EntityRef(({Expression.ref = current} (symbols += SYMBOLS)
    tail = EntityRef)*)
;

/**
 * @EntityRef :@Expression contains @EntityRef, this is a list
    of entitys
 */
EntityRef returns Ref:
    {EntityRef} (entity += SpecialExpression)*
;
/**
 * @IDSpace :contains a left recursion on the currrent
 *          :used for identityfing expressions with a space in
    front
 */
IDSpace:
  EntityRef ({IDSpace.left=current} (' ')*
  right=SpecialExpression)*
;

/**
 * @SpecialExpression :expressions containing stars
 */
SpecialExpression:
  {Entity}(rules += ID
             | '**'((name0 = SYMBOLS)?)(ID)?
             |       name1 = SYMBOLS(
                     name2 = SYMBOLS)?(
                     name3 = SYMBOLS)?(
                     name4 = SYMBOLS)?(ID)?
             | INT
           )
;

/**
 * @SpaceID :used for recognizing spaces followed be ID
 */
SpaceID:
  {SpaceID}(expr+= (' ')* ID?)*
;

/**
 * @MultilineAnnotation :used for adding multiline annotations
 */
```

```
MultilineAnnotation returns MultilineAnnotation:
   {MultilineAnnotation}(
                                       rule += ('/*@ ')?  ('* ')?
   functionAnnotation = FunctionAnnotation
          ('\n')? (' @*/')? (name0 = SYMBOLS)?
                                                |
    ('*') ' ' ' ' ('@*/') (name1 = SYMBOLS)? ('\n' | '\r')?
   )
;

/**
 * @FunctionAnnotation :used for function annotations
 */
FunctionAnnotation returns FunctionAnnotation:
   {FunctionAnnotation}(
   result += '@function '  functionType = FunctionType
                                               ((name0 = SYMBOLS))?
                                      ((nameComment = ID))?
                                                  ('\n' | '\r')?
   //supported without space before confidential and sensitive
              | '@parameter ' parameter = ID
                                             (name0 = SYMBOLS)?
                                 securityType = SecurityType
                                            ((name1 = SYMBOLS))?
                                  ((nameComment = ID))?
                                              ('\n' | '\r')?
   //for annotating pre and post functions
              | '@pre '  parameter=ID (name0 = SYMBOLS)?
                                                  ((name2 = SYMBOLS))?
                                         ((nameComment = ID))?
                                                     ('\n' | '\r')?
              | '@post ' parameter=ID (name0 = SYMBOLS)?
                                                  ((name3 = SYMBOLS))?
                                         ((nameComment = ID))?
                                                     ('\n' | '\r')?

   )
;

/**
 * @SingleLineAnnotation :used for adding single line
   annotations
 */
SingleLineAnnotation returns SingleLineAnnotation:
   {SingleLineAnnotation}(
    result +=  '//@ @function ' functionType = FunctionType
```

```
                                            ((name0 = SYMBOLS))?
                                      ((nameComment = ID))?
                                              ('\n' | '\r')*
//supported without space before confidential and sensitive
            | '//@ @parameter ' parameter = ID (' ')?
                                    securityType = SecurityType
                                         ((name1 = SYMBOLS))?
                                   ((nameComment = ID))?
                                           ('\n' | '\r')*
//for annotating pre and post functions
            | '//@ @pre '   parameter = ID (' ')?
                                    ((name2 = SYMBOLS))?
                              ((nameComment = ID))?
                                      ('\n' | '\r')*
            | '//@ @post '  parameter = ID (' ')?
                                    ((name3 = SYMBOLS))?
                              ((nameComment = ID))?
                                      ('\n' | '\r')*
  )
;

/**
 * @AnnotationType :annotation types
 * :annotations can addres whole functions or parameters of a
   function
 */
enum AnnotationType:
                     function
                    |parameter
;

/**
 * @FunctionType :annotaions types for functions
 */
enum FunctionType:
                    declasification
                  | sanitisation
                  | sink
                  | source
                  | trust_boundary
;

/**
 * @SecurityType :annotations types for parameters
 */
enum SecurityType:
```

```
                    confidential
                  | sensitive
;

/**
 * @KeyWord : list of C/C++ keywords
 */
KeyWord returns KeyWord:
        {KeyWord}(
                  rule = '__BEGIN_DECLS'
                       | '__BEGIN_NAMESPACE_STD'
                       | '__BEGIN_NAMESPACE_C99'
                       | '__END_DECLS'
                       | '__END_NAMESPACE_STD'
                       | '__END_NAMESPACE_C99'
                       | '__USING_NAMESPACE_STD'
                       | 'define'
                       | 'ifndef'
                       | 'undef'
                       | 'ifdef'
                       | 'if'
                       | 'include'
                       | 'include_next'
                       | 'pragma'
                       | 'else'
                       | 'elif'
                       | 'error'
                       | 'typedef'
                       | 'class'
                       | 'endif'
                       | 'source'
  )
;

/**
 * @SYMBOLS : all available C/C++ symbols
 */
SYMBOLS: {SYMBOLS}
          (symbols += ','
                    | '.'
                    | '..'
                    | '...'
                    | 'x' //don't know what this
                          //symbol is, it is in math.h
                    | ';'
                    | ' '
```

```
                              | '*'
                              | '* '
                              | '['
                              | ']'
                              | '\n'
                              | '('
                              | ')'
                              | '>>'
                              | '<<'
                              | '>'
                              | '<'
                              | '^'
                              | '+'
                              | '-'
                              | '/'
                              | BackSlash
                              | '%'
                              | '|'
                              | '->'
                              | '<-'
                              | '='
                              | '?'
                              | '!'
                              | DoubleQuote
                              | SingleQuote
                              | ':'
                              | '&'
                              | '~'
                              | '#'
                              | CURLY_OPEN
                              | CURLY_CLOSE
                              | INT
                              | name0=KeyWord
                         //used to bypass the reserved xText keyword
                         //source can be used as a function call
                         //in C/C++ headers or source files
                )
  ;

/**
 * @StructDefinition
 * :used for identifying structs
 * :this is future work.
 * :This can optionally be used in the future
 * :when {} is removed as multi-line
 * :comment and the body of the struct will be available
 */
StructDefinition:
  'typedef ' ID (name0=SYMBOLS) name1=ID CURLY_OPEN
    attr += ID*
  CURLY_CLOSE (name2=SYMBOLS) name3=ID (name4=SYMBOLS)?
;

/**
 * @SingleQuote
 * :declaration of ' and avoiding overriding the terminal STRING
 */
SingleQuote:
     MY_STRING
 ;

/**
 * @DoubleQuote
 * :declaration of " and avoiding overriding the terminal STRING
 */

DoubleQuote :
              STRING
            | DOUBLE_DQ_STRING
 ;

/**
 * @BackSlash
 * :declaration of \ and avoiding overriding the terminal STRING
 */
BackSlash :
            STRING
          | MY_BACKSLASH
 ;

/**
 * @MY_BACKSLASH :double backslash
 */
 terminal MY_BACKSLASH: '\\' ;

 /**
 * @DQ_STRING :double quote declaration
 */
 terminal DOUBLE_DQ_STRING :   "'\"' ~('\"')* '\"'";

 /**
 * @SQ_STRING :single quote declaration
 */
 terminal DOUBLE_SQ_STRING :    "'\''  ~('\'')*  '\''";

 /**
 * @CURLY_OPEN :open curly bracket declaration
 */
 terminal CURLY_OPEN:    '{';

 /**
 * @CURLY_CLOSE :close curly bracket declaration
 */
 terminal CURLY_CLOSE:   '}';

 /**
 * @MY_STRING :modified string terminals, it allows any kind
 * of symbols inside the single and double quotation marks \
 */
terminal MY_STRING : '"'  (  '\\'  |  !('\\'|'"')  )*  '"'
                   | "'"  (  '\\'  |  !('\\'|"'")  )*  "'"
;

 /**
 * @SL_COMMENT : all strings which follow // | || | }
 * will be a single line comment
 */
 terminal SL_COMMENT : '//'!('@')  !('\n'|'\r')* ('\n'|'\r')*
                       //'}' can optionally be used to disable the
                       //method bodies together with multi-line
                       //line {} comment
                       //| '}' !('\n'|'\r')* ('\n'|'\r')*
 ;

/**
 * @ML_COMMENT :@/* multi-line comment excluding @ from inside
  *            :{} multi-line comment
 */
 terminal ML_COMMENT : '/*' !('@') -> !('@')'*/'
                        !('\n'|'\r')* ('\n'|'\r')*
                       //'{' -> '}' can optionally be used optional
                        //to disable the method bodies together
                        //with single line { comment
```

```
                                  //|  '{'  ->  '}'  ('\n'|'\r')?
  ;
```