SrcML: A language-neutral source code representation as a basis for extending languages in Intentional Programming

Diplomarbeit an der Universität Ulm
Fakultät für Informatik

vorgelegt von

Frank Raiser

Erstgutachter:	Prof. Dr. H. Partsch
Zweitgutachter:	Prof. Dr. F. Schweiggert

Juli 2006

Abstract

This thesis presents an XML-based representation of source code, called SrcML, which is used as a basis for Intentional Programming. The combination of SrcML and Intentional Programming is investigated with a focus placed on extensibility. Three exemplary extensions are provided: the addition of a new statement to the Java programming language, a Lisp-like syntax for Java source code, and the creation of control flow graphs in extensible environments.

Contents

1 Introduction

The *Extensible Markup Language* (XML) [Con96] is gaining a lot of popularity as a format for storing data in a machine processable way. XML has found its way into many areas of application development and numerous tools have been created for processing XML data. Recently we see a lot of momentum [MCM02, Bad00, MK00, ST03] for storing source code in XML, as this yields huge benefits by being able to apply existing XML tools to source code.

In a lab course at the University of Ulm a parser for the Java programming language was extended to output the source code in the *Source code Markup Language* (SrcML). SrcML is a subset of XML to store the syntactic elements found in source code. This approach offers many advantages, for example for a tool which needs to know classes and methods declared in a source code: Currently this is difficult to implement when taking more than one programming language into account. In an XML-based format, however, the solution is a simple XPath [Con99] expression which even works for all languages storeable as SrcML documents sharing a similar concept of classes.

```
Integer a = Integer.parseInt (args [ 1 ]);
```

Listing 1: example of a variable declaration in Java

These benefits can be seen in the example given in Listing 1, which is Java source code declaring a variable and initializing it. Although being easily readable, it takes more effort to develop a tool which recognizes this variable declaration. Listing 2 contains this variable declaration stored as SrcML, thus making its syntactic structure explicit. Although both versions contain the same information, the latter is better suited for processing by a computer program. For example it is easy to find variable declarations: The structure of the source code is read by existing XML parsers and testing for the `variable` node is sufficient.

```
<variables>
 <variable name="a">
  <type name="Integer"/>
  <init>
   <expr>
    <call name="parseInt">
     <callee>
      <expr>
       <identifier name="Integer"/>
      </expr>
     </callee>
     <arguments>
      <expr>
       <array_access>
        <array>
         <expr>
          <identifier name="args"/>
         </expr>
        </array>
        <index>
         <expr>
          <constant value="1"/>
         </expr>
        </index>
       </array_access>
      </expr>
     </arguments>
    </call>
```

```
      </expr>
    </init>
  </variable>
</variables>
```

Listing 2: example of a variable declaration in SrcML

Another important advantage is the possibility to formulate queries in a more precise way than traditional string-based searches, or even tools like `grep`. Consider we want to find the declaration of the variable `a`. Simply executing `grep a` accepts the letter `a` occurring in `parseInt` and `args`, resulting in false positives. Even an advanced search for the word `a` returns too many hits, as the word `a` occurs frequently in comments. Working with an XML format the query can explicitly restrict the result to variable declarations.

Because the initial SrcML format is biased towards the Java programming language, we decided to recreate it to ensure its applicability to most mainstream object-oriented languages. The constant evolvement of programming languages requires the XML storage format to be extensible while existing tools based on it should continue working despite of any changes.

While the creation of the custom parser during the lab course was an invaluable learning experience, its maintenance cost is very high. This cost can be reduced by reusing the parser provided by the Eclipse platform. Therefore we combine the development of the SrcML format with the necessary implementations to use it within the Eclipse environment.

We further conceived several shared properties between SrcML and Intentional Programming (IP), which [CE00] describes as “a new, groundbreaking extendible programming and metaprogramming environment”. Both place an emphasis on extensibility with IP providing three types of extensibility. We show that the extensibility of the SrcML format corresponds to the addition of new intentions to an IP environment. The data structure used to represent source code in an IP environment further bares a close resemblance to SrcML documents and similar to the language neutrality in the SrcML format IP is abstracting from concrete programming languages through the use of intentions which can again be modelled in SrcML. This work therefore provides a prototypical implementation for several properties of IP based on SrcML.

To demonstrate that SrcML is suited for all types of extensibility available for IP we present corresponding exemplary extensions. The similarity between extending the SrcML format and providing a new intention for the IP environment is shown by an extension to the Java programming language. This further emphasises, that IP encourages customization of programming languages through the addition of domain specific intentions.

A major principle of IP are *active source operations*, which we define in due time. One of these operations displays source code to the developer. Considering the underlying XML format it is evident, that the syntactic structure of source code is already available with the help of a standard XML parser. Therefore we can arbitrarily display source code without having to parse it again. As an example for this concept we present an alternative implementation of an active source operation which displays Java source code in a Lisp-like syntax.

For the last type of extensibility we provide a new active source operation which creates control flow graphs for a SrcML document. In an IP environment classical algorithms like this become non-trivial. Compiler literature [WM97, Muc97] ignores the actual construction of control flow graphs, as it is straightforward with the underlying programming language being fully determined. Using Intentional Programming, however, the algorithm has to create control flow graphs for source code consisting of types of statements which will be added in the future. This entails a separation of the algorithm into a core and additional parts which provide the core with the necessary language specific informations.

1.1 Goals

As XML formats are defined by XML schemas [Con01] the definition of SrcML requires the creation of a corresponding schema. In order to develop this schema a set of design criteria has to be found which is based on the problems we want to solve with SrcML. After the SrcML format is finalized we want to provide a corresponding implementation for the Eclipse platform, [GE03] which should be able to transform classical Java source code into SrcML and back again. It should further be integrated into Eclipse such that SrcML Java projects can automatically be parsed and SrcML files can be viewed in a readable presentation in an editor window.

After having a working implementation of SrcML available for the Eclipse platform the next goal is to combine it with concepts from Intentional Programming. To this end, we need to take a closer look at these concepts and investigate how to provide implementations for them based on SrcML, which we the want to integrate into Eclipse as well. To demonstrate the different types of extensibility provided by IP we want to implement the above-mentioned extensions. Furthermore we want to develop a special IP editor in Eclipse which allows using Intentional Programming to its fullest.

A high priority is assigned to providing a comprehensive application of SrcML and IP. We chose the creation of control flow graphs for this application, as it highlights many of the properties of SrcML and IP. Hence we want to reuse the above-mentioned implementations to be able to generate control flow graphs for SrcML documents directly from the Eclipse environment.

1.2 Outline

The main sections of this work are split into a theoretical part and a part describing the implementation for the concepts discussed in the theoretical part. The reader may therefore choose to ignore the implementational aspects, although a large amount of this work consists of the accompanying implementations.

After this section we discuss the development of the SrcML format. To this end, Section 2.1 first examines the problems found in the old SrcML format developed ad-hoc during a lab course. We then present an exemplary Java source code for a simple arithmetic example in Section 2.2 which is used throughout the whole work. Section 2.3 then presents the criteria used for the development of the SrcML schema and discusses the problems which occurred and how we solved them. We further present an introduction to the Eclipse platform in Section 2.4 including how to develop plug-ins for this platform. Section 2.5 concludes the work on SrcML by presenting an overview of all SrcML related implementations.

After discussing SrcML Section 3 presents Intentional Programming and provides an overview of its properties. We begin in Section 3.1 by trying to define Intentional Programming. Unfortunately as the word *intention* in IP reveals this definition has to remain slightly informal. We then take a look at the different properties of Intentional Programming in Section 3.2, before discussing the special IP editor in more detail in Section 3.3. Before discussing the detailed implementations created for IP based on SrcML in Section 3.5, we present how to use Intentional Programming to extend the Java programming language in Section 3.4. In Section 3.6 the previously mentioned extension which displays Java source code in a Lisp-like syntax concludes the discussion of Intentional Programming.

The next section discusses the creation of control flow graphs using SrcML and IP. As mentioned earlier the construction is non-trivial for reasons which are explained in more detail in Section 4.1. Section 4.2 provides a formal definition of control flow graphs and presents our construction algorithm along with a proof of its correctness. The implementation for creating and displaying control flow graphs using SrcML and IP in the Eclipse platform is introduced in Section 4.3, before it is applied to the arithmetic example in Section 4.4.

Due to the large amount of implementations created as part of this work, Section 5 provides a high-level overview. Details on implementations related to the specific topics are discussed in the corresponding sections while Section 5.1 focuses on the installation and usage of these implementations. Section 5.2 then details how to reuse our implementation from a programming point of view. Section 5.3 further explains how test driven development (TDD) was applied to our implementations to improve its quality.

Finally Section 6 provides a summary evaluating to what extent our goals were met. It further presents an overview over the related work in the area of XML source code representations and Intentional Programming and discusses future work.

1.3 Conventions

The ideas in this work and the accompanying implementations are mainly directed at software developers. For the implementation created for this work the developers are therefore considered "users". As typical end-users are unaffected the terms "developer" and "user" are used interchangeably in this work.

As customary in an English-language thesis, we use the first person plural form "we" to refer either to the reader and the author or to the author only, depending on the context.

2 Source code Markup Language (SrcML)

This chapter explains the concept behind the *Source code Markup Language* (SrcML) which is used as a foundation for the remaining parts of this work. It is assumed that the reader is familiar with the *Extensible Markup Language* (*XML*) as specified in [Con96] and the *XML Schema* as specified in [Con01]. Section 2.1 takes a quick glance at the SrcML project as it existed prior to this work and the lessons learned from that, before Section 2.3 goes into details about the development of the new SrcML project and Section 2.2 introduces the arithmetic example which is being reused throughout the remainder of this work.

SrcML is an XML representation of source code which makes its syntactic structure explicit. There are many libraries available to process XML data and SrcML wants to take advantage of that by allowing developers to work on source code stored in SrcML using these existing libraries. The motivating idea is to stop storing source code as plain text files, which are hard to evaluate with a computer program, in order to store the syntactic structure of the source code in such a way that it is easy to handle this data with existing XML tools.

Developers are used to working on plain-text files when dealing with source code and we try to examine the advantages and disadvantages of changing how source code is stored. For example, it is hard to determine if a given plain-text file actually contains source code, as only due to the addition of a parser – or more generally a compiler – the text is interpreted as source code. This is not bad per se, however, we believe that this dependency on custom parsers is hindering the development of tools which can work on source code.

Currently a parser still needs to be written before one can even perform very simple tasks on existing source code. Developers therefore often create programs which work on source code by for example evaluating simple regular expressions , although it is very hard to get these regular expressions correct, as they only see the text on a line-by-line basis as opposed to the syntactic structure of the source code and therefore remain context free. We argue that an explicit way to access source code on its structural level leads to faster and easier development of more reliable tools.

SrcML is a proposal for filling this gap. As SrcML is an XML-based format, the syntactic structure of source code can be directly represented. In the most simple case one could directly transform the *Abstract Syntax Tree* (AST) [WM97] into an XML document. Parsers for XML documents are available for almost all programming languages currently in use and thus such a document could easily be accessed even by novice programmers.

For SrcML we decided, that the format should not be a direct representation of the abstract syntax tree. If plain-text files are going to be replaced by XML files one might as well try to get as many advantages out of this process as possible. One major problem the computer industry is facing today is the huge amount of programming languages available which makes it extremely hard to develop tools supporting several programming languages. This is inherently visible in the problem mentioned above: a parser is needed for every new language and even if such a parser is available the resulting abstract syntax trees of programs in various languages may look very different.

SrcML therefore tries to be a common basis in which most of the standard syntactic elements found in programming languages can be stored. Nevertheless it should be pointed out that the SrcML schema developed as part of this work is emphasized on object-oriented languages, but extensions for functional or logical elements are possible (see Section 2.3.4). This approach offers many advantages: Imagine a tool which needs to know classes and methods declared in a source code. Currently this is rather difficult to implement as soon as one takes two programming languages into account. In an XML-based format, however, the solution to this problem would be a simple XPath [Con99] expression. And the very same XPath expression works for all languages which share a similar concept of classes and can be stored as SrcML documents. This idealistic approach to the problem cannot hold up to reality and in fact only very few or limited tools can be developed for one programming language and automatically work for other languages. SrcML tries to simplify adding support for another language by taking advantage of as many commonalities as possible.

2.1 Previous SrcML project

Another format, which we call SrcML$_{\text{Old}}$, has been developed earlier in lab courses at the University of Ulm and is presented in [Rai04]. This section details why the work at hand is considered a successor and explains some of the lessons learned from that project.

The format originally developed in the previous project was highly dependant on the Java programming language, whereas this work emphasizes the language neutrality of such a format. One reason for the dependence on Java was that the project originated from a custom Java parser. Therefore the schema is not sufficient for storing arbitrary source code and needs to be improved. During the transition from Java 1.4 to version 1.5 it became clear that for a project like SrcML$_{\text{Old}}$it is inadvisable to use a custom parser as the maintenance is too expensive. Section 2.3.1 discusses the development of the new schema which was created in order to alleviate this problem.

The existing SrcML$_{\text{Old}}$project also provided several so-called *platforms* , each of which is a collection of specific functionality configurable through the use of plug-ins. After reconsidering the above points we decided to rewrite the project on the basis of the Eclipse architecture outlined in Section 2.4. Eclipse provides two very important features which have been implemented similarly in the project: a Java parser and a plug-in architecture. Using the Java parser provided by Eclipse dramatically reduces the amount of maintenance needed and using the supplied plug-in architecture obsoletes the various platforms found in the project and in fact provides a more generic way of adding functionality – again at a reduced amount of maintenance.

As the plug-in architecture of Eclipse is very different from the platform architecture used previously those parts of the code are rendered useless. Additionally, the Java grammar used in the previous project is not reusable, as Eclipse already performs the complete parsing process. Furthermore the API available in the previous project is not reusable either due to the major changes in the SrcML format. This means that the implementation created for this work is independent from the previous project except for being influenced by the ideas and experiences gained from it. The idea of storing source code in XML remains the same, but we reconsidered the platforms used for extensions and combined this extensibility with Intentional Programming.

Finally the ideas taken from the SrcML$_{\text{Old}}$project have been merged with ideas from Intentional Programming which is described in Section 3. As we found out, SrcML is a very suitable format for the data structure used to represent source code in Intentional Programming.

2.2 Arithmetic example

Before the detailed discussions of the design criteria an example source code is introduced at this point which is going to be used throughout the remainder of this work. The arithmetic example in Listing 3 is a simple program which reads the three arguments given to it and if the first argument equals the string for one of the four elementary arithmetic operations it performs the corresponding arithmetic operation. This functionality is realized with a simple chain of `if` statements. For simplicity error checking is neglected, so the program crashes if for example not enough arguments are provided. Furthermore the `Example` class is derived from `Object`, which is the default in Java, and is implementing `Cloneable`, in order to demonstrate inheritance in SrcML.

```java
public class Example extends Object implements Cloneable
{
    public static void main(String ... args) throws Exception {
        String op = args[0];
        Integer a = Integer.parseInt(args[1]);
        Integer b = Integer.parseInt(args[2]);
        if ("plus".equals(op)) System.out.println(a+b);
        else if ("minus".equals(op)) System.out.println(a-b);
        else if ("mul".equals(op)) System.out.println(a*b);
        else if ("div".equals(op)) System.out.println(a/b);
        else System.err.println("unknown operation");
    }
}
```

Listing 3: arithmetic example in Java

The complete SrcML representation of this program can be found in Appendix D. Listing 4 is an excerpt of the SrcML document representing the class declaration with omissions indicated by XML comments. The original program can be found in the SrcML document again: Sometimes a literal token is included, as in the case of the `public` modifier, and at other times the SrcML document is abstracting from the original source code, as in the case of the inheritance where special tags are used. The semantics of the XML tags found in Listing 4 are not relevant at this point and the listing only serves as an early introduction to how source code stored in the SrcML format looks like.

```
<!-- ... -->
  <type_decl name="Example" type="class">
    <modifiers>
      <modifier name="public"/>
    </modifiers>
    <inheritance>
      <inherits type="implementation">
        <type name="Object"/>
      </inherits>
      <inherits type="type">
        <type name="Cloneable"/>
      </inherits>
    </inheritance>
    <method name="main">
      <!-- ... -->
    </method>
  </type_decl>
<!-- ... -->
```

Listing 4: class declaration for arithmetic example

Listing 4 already shows that different concepts of a programming language can be represented similarly in SrcML. Although the example contains two different types of inheritance , clearly separated through the `extends` and `implements` keywords in the Java source code, both of these are represented with `inherits` in SrcML. The reason for this is seen in the additional specification of the type of inheritance. There are two well-known types of inheritance in object-oriented programming: type inheritance and implementation inheritance. The `Example` Java class contains both kinds of inheritance. Note that an interface in Java which inherits methods from other interfaces is originally written with an `extends` keyword in Java. This is a discrepancy, as a class uses the `extends` keyword for implementation inheritance. In the case of an interface, however, it is a type inheritance. When transformed to the SrcML format, we can remain consistent such that the two different usages of the `extends` keyword lead to two different types of inheritance being stored reflecting the exact type of inheritance. Problems like this often influenced the design of the SrcML format and are discussed in more detail in Section 2.3.

2.3 Design criteria

After analyzing the SrcML$_{\text{Old}}$ project we agreed on a set of design criteria for the new project. This section presents these criteria with a short description of each, before the following sections go into details about how these criteria can be implemented:

Definition 2.1. Design criteria for the SrcML project:

- usage of XML and XML schemas

- language neutrality
- querying of source code
- extensibility

Language neutrality was already mentioned and it should be pointed out again that for the purpose of this work a restriction to object-oriented languages was made. More precisely three major object-oriented programming languages – C++, C#, and Java – were closely examined for common syntactic structures as detailed in section 2.3.2.

By the choice of using XML and the abundance of available parsers for it, it is guaranteed that there are many existing tools compatible with SrcML. Any other easily parseable format could have been used as well, but XML has proven to be a reliable standard in the past years and there are parsers available for a huge share of programming languages. Using XML also allows developers to make use of existing XML tools and apply them to source code. XML is also a format which is easily readable by humans as well as machines. Despite of its readability it is noteworthy to point out that SrcML does not imply developers edit source code directly in its XML-based form. See Section 3.3 for more details of how we believe advanced source code editing might look like with the help of SrcML and Intentional Programming. Furthermore the usage of XML schemas [Con01] allows automatic verification of SrcML documents. The previous SrcML project used Document Type Definitions for this purpose, but over the course of the last years XML schemas have established themselves as a successor.

One very important design emphasis was placed on querying of source code. Queries are used in almost all tools working on source code. Analyses usually perform many queries, whereas tools which modify the source code tend to need fewer queries to find the positions in which to make changes. In either case the SrcML format makes it simple to create a query. Using a format like SrcML which gives access to the syntactic structure of the source code allows developers to formulate precise queries as discussed in Section 2.3.3.

Another very important aspect was to create a format which is highly extensible. Programming languages are most probably going to change in the future, but for the SrcML format to remain usable it has to provide a way to adapt to necessary changes. The SrcML schema is therefore left open for extensions as described in Section 2.3.4. When adding support for a new language to the SrcML format, as many syntactic structures as possible should be shared and new structures should only be added if unavoidable. This is a very important aspect for developing tools with the existing SrcML format in mind, which are able to handle the new language as best as possible. Furthermore this is also an important point when combining SrcML with Intentional Programming as is explained in Section 3.

2.3.1 Extensible Markup Language (XML) and XML schemas

This section covers details of the development of the SrcML schema. As mentioned earlier the three major programming languages C++, C#, and Java were compared for common syntactic structures in order to create a schema suitable for the presentation of object-oriented source code.

The following paragraphs provides a short introduction to XML schemas and argue about the need for developing such a schema for SrcML. A few notes will be added with respect to the practices used when developing the schema. The problems which appeared during the development of this schema are discussed in Sections 2.3.2 and 2.3.3. Section 2.3.4 discusses how the extensibility design criteria influenced the schema creation.

Introduction to XML schemas

XML schemas as specified in [Con01] are used to describe the syntactic structure of XML documents. In our case the SrcML schema is used to specify how source code stored in this format should look like. It is important to realize that XML schemas are XML documents themselves and thus can be processed easily. This is useful when verifying SrcML documents against the schema, i.e. to test if a given document is syntactically correct according to the chosen schema.

This automatic verifiability is one important aspect of why a SrcML schema is needed. Another aspect is that a schema gives developers a precise resource on the structure of SrcML documents which is useful when developing tools to work with this format. As such the schema itself serves as documentation. Before discussing the SrcML schema in detail it should be ensured that the vocabulary used for this subject is properly introduced:

Definition 2.2.

- XML documents are made up of *tags*. Every start-tag is followed by a corresponding end-tag except for empty-element tags. XML tags can include *attributes* which are key/value pairs, other tags, and simple text. This work adheres to the extensible markup language as defined in [Con96].
- The *Document Object Model* (DOM) "provides a standard set of objects for representing [...] XML documents [...] and a standard interface for accessing and manipulating them" [Con98]. A DOM is therefore a means of how XML documents are kept in a program's memory space.
- When an XML document is represented as a DOM the objects representing tags are called *elements*. Due to the hierarchical nature of XML documents the resulting DOM is a tree structure on which the standard vocabulary for graph theory can be applied. Most notably, this work talks about *child* and *parent* elements in such a tree structure.

Remark 2.3. Due to the close correspondence between tags, elements, and DOM nodes we use these terms interchangeably even when the context is different from the one given in the definition. For example an XML tag could have a parent element, although tags are defined for XML documents and elements for their representations in the memory space.

In early stages of the schema development, documentation for each tag was provided directly within the schema itself. This increases the file size of the schema which is undesirable. When performing verifications against the schema, it is often necessary to download a copy of it from the internet in which case including the complete documentation results in a major slowdown. For example the unit tests we use for schema verification (see Section 5.3) perform much better with a tailored schema, because each file results in the complete schema being read again. Therefore the individual tags are now commented on the webpage of the SrcML project [Rai04] and this documentation can also be found in Appendix B.

During the development of the SrcML schema, *Best Practices* as found in [Cos05] have been honored. This mainly influenced the namespace exposure and elements:

Every XML schema is also associated with a namespace which allows XML tag names to be reused. The SrcML schema is using the *http://srcml.de* namespace and various other namespaces like *http://srcml.de/ext/java* are used for extensions to the original schema. The namespace is exposed such that all instance documents will have to specify namespaces explicitly. This was influenced by the idea of using SrcML for domain specific languages and Intentional Programming as discussed in Section 3 which can result in instance documents using several small languages at once and therefore several namespaces. Having to specify all namespaces explicitly avoids conflicts caused by eventually occurring tags with equal names. Usually the main SrcML namespace is used as the default namespace such that the namespace prefix can be omitted for the respective tags.

Every SrcML tag is declared as a type as well as an element. For the SrcML schema, types are used to declare the syntactical structures. However when extending the schema and adding new schemas it is more straightforward to work with elements. This improves the consistency of tag names, as they are already included when working with elements, whereas the use of a type requires the tag name to be specified as well. While it is feasible to use the same tag names for a type throughout the SrcML schema itself, it is harder to enforce these tag names in third party schemas using only types.

In the following, we discuss the schema declarations for the `inheritance` tag from Listing 4 in an exemplary way to demonstrate how these declarations are created. The final declaration of this tag can be seen in Listing 5 which is an excerpt of the SrcML schema found in Listing 34 in Appendix A.

```
<xs:complexType name="Tinheritance">
  <xs:sequence>
    <xs:element name="inherits" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="modifiers" type="SrcML:Tmodifiers"
          minOccurs="0" />

          <xs:element name="type" type="SrcML:Ttype" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:attribute name="type" />

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:any namespace="##other" minOccurs="0"
                                maxOccurs="unbounded" />
  </xs:sequence>

  <xs:anyAttribute processContents="skip" />
</xs:complexType>
```

Listing 5: SrcML schema for inheritance

The `complexType` element is used to declare a new type which can then be used to declare elements which make up the structure of a document. From Listing 4, it can be deduced that an `inheritance` element is used in the `type_decl`'s declaration. The "T" prefix found in names has been used throughout the schema for names referring to type declarations. The `sequence` then describes the structure of an `inheritance` element which is declared to consist of an unlimited number of `inherits` elements. Each of which is consisting most notably of a `type` which represents the inherited type. Furthermore the optional `modifiers` element can be used for programming languages, which allow to influence the inheritance process through the use of keywords. An example for the usage of this element is the C++ language which allows inheritance to be modified with the `public`, `protected`, `private`, or `virtual` keywords. Although the number of `inherits` elements is unlimited the above declaration implicitly contains a `minOccurs="1"` which means if an `inheritance` element is used at least one `inherits` child element has to be present. The remaining lines in Listing 4 are used for the extension mechanism which is discussed in Section 2.3.4.

Problems

After identifying similar syntactic constructs in C++, C#, and Java it is not always clear how to translate these into the SrcML schema. For example it is possible to use individual `class` and `interface` tags for classes and interface, or one `type_decl` tag used for both type declarations. As a case example we show two such controversial tags: the `type_decl` and `expr` tags. In all problematic cases a closer look was taken to the advantages and disadvantages of possible solutions according to the criteria set up for the schema creation.

One such problem was how to represent typical object-oriented type declaration in SrcML. These include classes, interfaces, enumerations, structs, and annotations amongst others. There are two possible solutions to this problem: Either each of these declarations is represented using an individual tag – `class`, `interface`, `enumeration`, and so on – or a common declaration tag – for example `type_decl` – is introduced which can handle all of them. One could also try to combine only some of these declarations and treat the remaining ones independently, but this seemed to be a rather inconsistent and counter-intuitive way: When considering additional languages it would be hard to define which declaration types should be combined and which not, as the available types are not even known at this time.

A closer examination of the two possible solutions with respect to the design criteria did not reveal a preferable method either: If we use a tag handling all type declarations – respectively called `type_decl` – we will need an attribute to distinguish the exact kinds of type declarations. With the attribute value being a string, this approach is very easy to extend without even requiring an additional schema. Individual tags are also easy to extend by simply adding new tags, although this would increase the number of available tags. So both approaches are generally providing a sufficient means of extensibility.

Apart from the design criteria, we argue that individual tags more closely resemble an abstract syntax tree whereas a common tag for type declarations provides a better abstraction – pun intended. Using many individual tags also increases the size of the schema file. It therefore appeared that there is no clear approach which should be taken because of its advantages and it is a matter of taste which approach gets used. For this work the choice was made in favor of a common `type_decl` tag unifying all kinds of type declarations.

A similar problem which occurred for the `expr` tag is discussed in Section 2.3.3, as the solutions make a significant difference for queries. In general we always try to look at the different possible solutions and determine which one is preferable in terms of the above criteria. A complete list of the tags which have been created for the SrcML schema including informal descriptions of what we intend to use them for is given in Appendix B.

2.3.2 Language neutrality

A design criteria for the new SrcML project was achieving language neutrality as far as possible. To this end, the general purpose programming languages C++, C#, and Java have been taken into account. As mentioned earlier, the emphasis is placed on object-oriented (OO) languages and we consider these languages to be representatives for most features found in OO languages.

In order to abstract from the specifics of a language, we tried to identify common syntactic structures shared by those languages on the grammar level based on [Str98, csh, G$^+$05]. However this does not guarantee completeness such that every valid – in the sense of compilable – source code can be equivalently represented in SrcML. The problem when trying to prove completeness is hidden in the technicalities of the associated grammars. Each language comes with a grammar which has different properties and grammars from official documentations differ significantly from the grammars apparently used in parsers. So as an additional help the informal language specifications were taken into consideration as well which describe features of the language independently from its grammar. We then combined these features with the grammar and thereby tried to represent similar features with the same SrcML constructs.

As a proof for the completeness of this approach is very hard and even a successful proof would not offer any additional insights, we decided to only perform an empirical verification based on the implementation described in Section 2.4. To this end, unit tests have been created which perform conversions from Java source code to SrcML testing various properties. Additionally transformations are made back to Java source code and once more into SrcML after which the two SrcML representations are compared. As the SrcML documents are an abstract view of the syntax of the language in a standardized XML format, this comparison is easier to make, as differences cannot be created by simple character artifacts like whitespace or newlines. With the help of this unit test, the complete code base of the Eclipse project was used to justify the confidence in the developed SrcML format. Additionally selected examples have been verified manually.

The SrcML schema also allows instance documents, which are not equivalent to a source code file in any given programming language. For example C++ does not directly support interfaces and Java does not support multiple inheritance, but a SrcML document is allowed to contain both. More specifically, a document can be constructed including all features from all supported programming languages. So one should bear in mind that performing a validation against the SrcML schema does not guarantee a syntactically correct source code document for any programming language. This is a rather theoretical problem though, as practically there will be very few occasions in which a source code is constructed by randomly inserting new parts. Existing source code can be transformed into its SrcML representation in which case the document is consistent in that it represents a source code in the given programming language. Modifications to existing SrcML documents should be aware of the underlying programming language to ensure the creation of documents which represent source code in that language. It should be pointed out that while it is possible to for example add interfaces to a SrcML document, this should only be done after checking the provided metainformation on whether the target programming language supports interfaces. As a summary, validation of instance documents is generally an approximation for a syntactically correct source code document but does not verify semantics.

The previous problem goes hand in hand with the lack of semantic information in this format. Language specific semantics may be used in constructing a SrcML document, because often a parser is unable to determine the correct syntactical structure without the knowledge of the programming language's semantics. For example linking the usage of variables to their declaration already requires knowledge about the variable binding in the given language. The semantics used for the creation of a SrcML document are not included in the resulting document. There are several reasons for neglecting semantics in this format:

number of tags would increase If the semantics were directly represented in the SrcML files this would require new tags to be used. Considering that two programming languages are generally much more different in their semantics than their syntactic structures this would lead to an enormous increase of the number of required tags.

semantics are programming language specific Due to the programming language specific nature of semantics this would also be a violation of the language neutrality criteria.

semantics could be added as an extension The final reason not to include semantics is that there is always the option of adding them as an extension as described in Section 2.3.4. After all the SrcML format is a container for the syntactic structure of a program, not its semantics.

2.3.3 Querying source code

With queries being one of the main design criteria of the SrcML schema, this section examines the advantages and disadvantages when working with source code in an XML format. Some examples for queries include searching for declared classes, methods defined in a class, or finding the declaration of a variable. Queries are formulated as XPath expressions which comes naturally, regarding that [Con99] states: "XPath is a language for addressing parts of an XML document".

A disadvantage of XPath queries is their length and design decisions can directly translate into longer queries. Additionally the various namespaces probably present in a SrcML document require elements in the query to be explicitly specified by their namespace which again adds to the overall length of a query. However the majority of queries are contained in tools or dynamically created depending on user inputs which makes this a bearable disadvantage.

For the user of the SrcML format the structure contained in it allows for very precise queries. A main advantage of these queries is that they can contain structure: Therefore a query would not try to find "MyClass", but instead a type declaration for a type called "MyClass". This automatically reduces false positives compared to using standard tools like `grep` . There are many possible occurrences for the literal string "MyClass", which are totally unrelated to the type declaration being searched for. The string could appear in a comment or string constant. It could also be used as a variable or function name and so on. A `grep`-based search would return a positive hit in all these cases. Naturally this also makes XPath queries harder to write due to the inherent structure which has to be known to the developer of the query.

Additionally a distinction was made between simple and complex queries. Simple queries are queries which can easily be created without investing too many thoughts, usually searching for simple tokens similar to `grep`. Complex queries instead tend to involve more complex structures and can end up being several lines long. Those queries are considered to be created by tools and thus disadvantages like the query length are not as important as for simple queries. Examples for simple queries include finding the declaration of a certain type, getting all methods declared in an interface, or the number of variables declared in a class. An example for a more complex query would be searching for the declaration of a type which inherits a certain interface and implements a certain method of it with the help of an if statement.

Finding the declaration of a class called "MyClass" requires a simple XPath query like the one shown in Listing 6 line 1. Developing such queries requires understanding of the SrcML schema for the required tag names and structures. Complex queries require a bit more effort to create. Nevertheless these queries are often suitable for dynamic creation by tools and the more complex a query gets the more likely it is that it's used only by the tool and not directly exposed to the user.

```
1  //type_decl[@name=”MyClass” and @type=”class”]
2  //type_decl[@name=”Example” and @type=”class”]/method
3  count(//variables/variable)
4  //type_decl[@name=”Example” and @type=”class” and
5      inheritance/inherits[type[@name=”Cloneable” and @type=”type]] and
6      method[@name=”main” and block//if]]
```

Listing 6: example queries (namespaces neglected)

Listing 6 lines 2-6 also show some example queries on the SrcML document for the arithmetic example which can be found in Appendix D. Line 2 is a simple query which, when evaluated, results in all methods declared in the `Example` class. The simple query in line 3 is used to count the number of variables declared in the document. The query in lines 4-6 is a complex query which searches the document for a type declaration of a class called `Example` which implements the `Cloneable` interface and has a `main` method including an if-statement in its body.

expr tag

The `expr` tag mentioned above poses the following problem: Should expressions be represented in a SrcML document directly, for example as assignments, method calls, etc.? Or is it better to add an additional `expr` tag as a container for all kinds of expressions? This problem is similar to the `type_decl` problem mentioned earlier. This time the individual tags required for assignments, method calls, and so on, are necessary and the question is whether to add a generic `expr` tag for the purpose of abstraction.

Both ways can be compared in terms of formulating queries. While it seemed appropriate to abstract from type declarations it is rather hard and unrewarding to abstract at the expression level. When performing queries which reach down to the expression level it is most likely that these queries are programming language specific. At this point it appears to be preferable to have a closer resemblance to the abstract syntax tree. So we did not further evaluate other options like combining certain expressions into more abstracted tags.

The queries with an included `expr` tag obviously tend to get longer, as they contain the common `expr` tag as well as the individual tag for the actual expression. Due to the fact that very often abstract syntax trees contain expressions nested inside expressions this increase of the query length can be very significant. The same argument also applies to the file size of a SrcML document, as expressions make up a major part of every source code document. Experimental estimations have shown that transforming Java source code from normal `.java` files to their SrcML representation leads to an average increase of the file's size by 5 times. When additionally using the `expr` tag the size is increased by about 10 times instead. This argument was not considered very important,

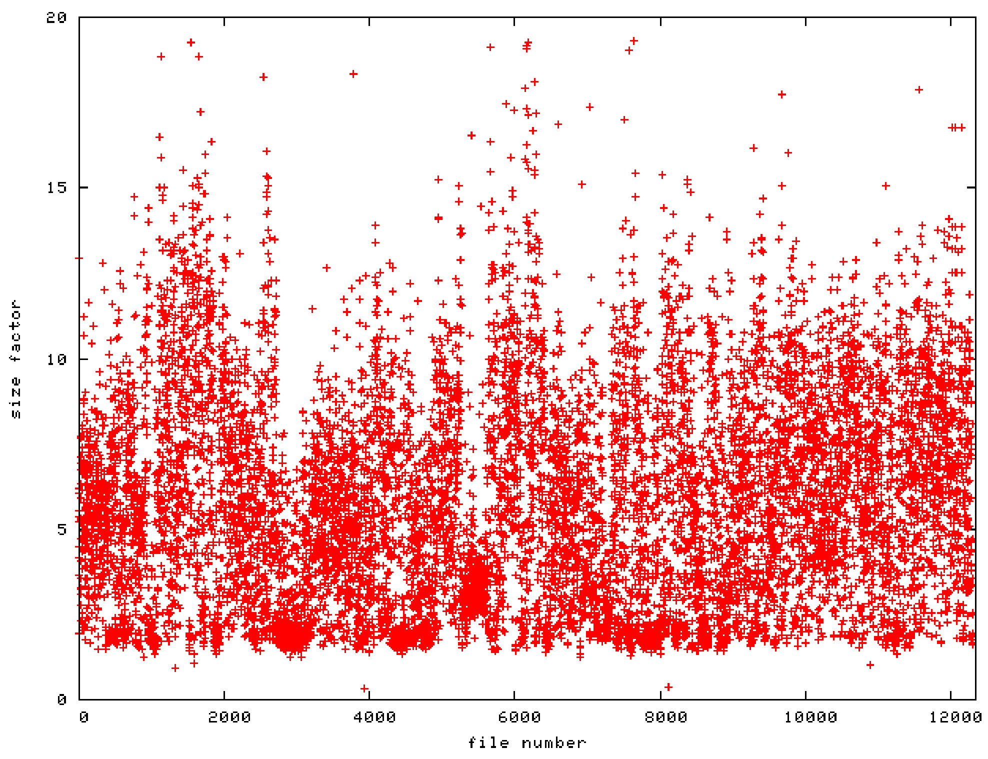


Figure 1: Size factors for Eclipse 3.1.2 source code files

as source code sizes are very small compared to current hard disk sizes. In the example case of transforming the complete source code of Eclipse 3.1.2, which is about 100MB in size, the resulting documents take up about 700MB. But for more moderate amounts of source code a size increase of factor 10 or even 20 is bearable and will be ever less significant with the ongoing developments in the hard disk sector. As a last resort it is also possible to store source code in a compressed format which, due to the repetitive nature of XML documents, significantly decreases the required storage size.

As can be seen in Figure 1 the average size factor when converting `.java` files to SrcML is around 6-7. Nevertheless larger factors can be seen for files which include many nested expressions. The data from Figure 1 was gained by converting the complete source code of Eclipse 3.1.2 consisting of roughly 12000 files into their SrcML representations and comparing the file sizes afterwards. The files which show a factor close to 1 are a result of the current implementation not converting non-javadoc comments into SrcML as they cannot be safely associated with the element they are commenting. Overall the missing comments are not affecting the size factor, as their addition to the SrcML document only has a constant-sized overhead. Considering the addition of metainformations to SrcML documents we estimate an average size factor of 20 or more is possible including data traditionally contained in several other files.

The `expr` tag offers a way to query for expressions as such, without needing to know what specific kind of expression is used. As discussed earlier, abstractions on the expression level seem to not be very helpful which makes the `expr` tag a separator between an abstracted view of an object-oriented source code and the detailed programming language specific expressions. Additionally when the extensibility design criteria is considered it becomes clear, that handling an arbitrary amount of expressions without a common `expr` container is resulting in significantly longer queries. Especially the simple query for an expression as such ends up being very complicated and lengthy, as all possible available expressions would have to be included. Due to these reasons the final decision was to use the `expr` tag and accept the storage size overhead it creates.

In summary the SrcML format offers tool developers a means of powerful queries on source code. These queries could also be exposed to experienced users, for example to provide a more sophisticated search functionality in editors comparable to the use of regular expressions found in many current editors. It is also noteworthy that the criteria for language neutrality can be combined with querying, as many simple queries can be used unchanged for different programming languages. This in turn reduces the development time of multi-language tools using such queries.

2.3.4 Extensibility

The extensibility of the schema was the primary focus during development. Although only three programming languages have been taken into account, the schema should provide means to add new languages. Especially with respect to Section 3 this requires the schema to allow extensions for programming languages which haven't been created yet.

Despite this extensibility, however, the addition of a new programming language should not interfere with the existing schema. This means that tools, which have been developed with regard to a certain version of the schema, should not have to be changed. To avoid this problem, we decided that extensions should be separated into their own namespaces. This guarantees that the original namespace, as well as all namespaces relevant to a tool during its development time, remains unaffected by extensions.

These namespaces still have to interact and no copy of the complete SrcML schema should be required for the definition of an extension's schema. To this end there is a special `any` element available in the schema definition language which explicitly allows the inclusion of an arbitrary element into instance documents. Consequently the SrcML schema could simply be defined as a single `any` element as shown in Listing 7. This would defeat the very purpose of having a schema though, as any instance document is considered valid for such a schema. So there has to be a trade-off between the extensible parts of a document and the fixed structures of it.

```
<xs:any minOccurs="0" maxOccurs="unbounded"/>
```

Listing 7: any element

Another important point is that storing source code in XML makes it possible to add metainformation to the document which is currently still placed into comments or additional files. Considering that there is basically no end as to what metainformation a user might want to store with her code it is only just to allow every element to have at least one place where arbitrary elements can be inserted. The trade-off mentioned above was therefore achieved by fixing a specific structure for every element and then allowing an arbitrary amount of further child elements.

This is best explained at an example: Consider an if statement which has a then branch, an else branch, and a condition. The condition gets evaluated at runtime and determines which branch is to be executed. This concept of an if statement now leads to a SrcML representation as shown in Listing 8.

```
<if>
    <condition>...</condition>
    <block><!-- the then branch -->...</block>
    <block><!-- the else branch -->...</block>
</if>
```

Listing 8: if statement in SrcML

A special part where the SrcML format abstracts from traditional ASTs can be found here as well. Considering a chain of if-else-if statements as in the arithmetic example, then this is usually represented in an AST as a tree with linear depth in the length of the if-chain. In SrcML we decided to flatten this subtree by extending the representation of an if statement. Instead of only allowing one `condition` and two `block` elements we allow an arbitrary number of `condition`, `block` constructs each of which represents one if-statement's condition and the block to be executed when the condition evaluates to true. These pairs of elements may be followed by a final `block` element representing the last block executed if all conditions evaluate to false. An example for this can be seen in Listing 37 in Appendix D.

Now suppose that the execution of this if statement should only occur during the debugging process and this part of the code should not be contained in the final product. In C++ this is achieved through the preprocessor whereas it is rather difficult to get this separation in Java. With SrcML metainformation like this can instead be stored directly with the if statement which is allowed due to the `any` element. But as the structure of such an if statement is equal in all possible languages – otherwise it should not be represented using this tag – a tool may not be interested in this metainformation and wants to access the condition and the two branches. Therefore an if statement is guaranteed a fixed structure consisting of the `condition` and a `block` for the then branch. The second `block` for the else part is optional and only after those elements, instance documents are free to add arbitrary elements. So marking this statement to be only executed in a debug environment could look like in Listing 9.

```
    <if>
        <condition>...</condition>
        <block><!-- the then branch -->...</block>
        <block><!-- the else branch -->...</block>
        <meta:debug level="INFO"/>
    </if>
```

Listing 9: if statement for debug environment

Apart from extending SrcML documents with new elements there is also the option of smaller extensions which only affect the attributes of existing elements. This can also be used by tools to mark certain parts of the document. An example for such a usage is the `autogenerated` attribute which is used in the implementation for Section 3.5 to mark elements which have been generated by a tool so they can easily be removed later on. As can be seen in Listing 4 line 303, the SrcML schema is very lax by using the `anyAttribute` element for all declared tags. Therefore attributes – like the`autogenerated` attribute – can be freely used without violating the schema.

For a larger extension it is advisable to create an accompanying schema for the same reasons the SrcML schema was created: it allows the automated verification of instance documents and provides other developers with structured information about the extension. Listing 10 is an example for such a schema. The extension is used to add metainformation about the programming languages found in the SrcML document. The first few lines are XML schema specific and declare the involved namespaces. After that a `languages` element is declared which has a `base` attribute and can contain an arbitrary number of `language` elements, each of which has a mandatory name attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://srcml.de/meta"
    xmlns="http://srcml.de/meta"
    xmlns:meta="http://srcml.de/meta"
    xmlns:SrcML="http://srcml.de">
    <!-- ... -->
    <xs:complexType name="Tlanguages">
        <xs:sequence>
```

```
            <xs:element name="language" type="meta:Tlanguage"
                minOccurs="0" maxOccurs="unbounded"/>
            <xs:any namespace="##other"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="base" type="xs:string" use="required"/>
        <xs:anyAttribute processContents="skip" />
    </xs:complexType>
    <xs:element name="languages" type="meta:Tlanguages"/>

    <xs:complexType name="Tlanguage">
        <xs:sequence>
            <xs:any namespace="##other"
                minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:anyAttribute processContents="skip" />
    </xs:complexType>
</xs:schema>
```

Listing 10: example extension schema

Section 3 discusses how multiple languages are used inside a single SrcML document if they can be projected onto a base language. This base language is represented in the `base` attribute of the `languages` element. Furthermore an unlimited list of `language` child elements can be specified for additional programming languages used in the document. The extension is providing its own namespace *http://srcml.de/meta* which properly separates it from the original SrcML schema and it is also created to be extensible itself. There is one problem involved with this extension mechanism: there does not appear to be a way to restrict the position in a document where the `languages` element can be used. While it is meant to be used only as a child element of the outermost `unit` element, the schema allows its use in for example an if statement. Nevertheless this is not a problem, as it was already mentioned earlier that not every validating SrcML document makes sense in terms of being the syntactic representation of source code. It is also noteworthy that extension elements which are positioned in unexpected locations in the DOM are not considered a problem: Tool developers know that the `any` element may result in an arbitrary number of other elements and therefore always have to make sure they are only working with elements they know and generally ignore unknown elements.

The usage of this `any` element also is an additional reason for using the `expr` tag as mentioned in Section 2.3.3. When directly adding all expressions known from the three major languages as individual tags into the schema, there is no way to properly extend this, as every tag consists of a fixed structure after which arbitrary elements can follow. Therefore a new expression introduced by an extension can only be added in places where the `any` element is used. However this means it is mixed with metainformation and many other extensions, making it hard to distinguish it as an expression. Tools which need to find an expression can easily do so for the expressions explicitly covered in the schema, but will fail for new expressions due to combining the expression with metainformation. Using a special `expr` tag for which the very first child node always is the actual expression solves this dilemma: Either it is one of the expressions known from the existing schema or it is an expression introduced by an extension, but in either case a tool knows the element corresponding to the expression. Note that for empty expressions a special `nop` expression is used so that metainformation can still be added without creating ambiguity in such a case.

This ambiguity problem occurs in a more generic way as well. As can be seen in Listing 9, possible non-determinism can occur in XML schemas: When a schema is not restrictive enough, which in this case is bound to happen due to the additional demand for extensibility, an element in an XML instance document may not be deterministically associated with the correct element of the schema. The `meta` namespace used for the `debug` element in Listing 9 is there for a reason: when allowing an arbitrary element to be inserted at such a point one cannot rule out the possibility of this element resulting in non-determinism if it can be in the same namespace as the previous element. When taking a look at Listing 8, one can see the two `block` elements which are supposed to represent the two branches of this *if* statement. Representing an *if* statement which does not have an *else* branch, makes it problematic to add another `block` element. This would be legal though, as arbitrary elements can be added at this point. This is an example of non-determinism, as later-on it cannot be decided if this statement has two branches or one. To avoid this problem the extensions made to instance documents as allowed by the `any` element have been restricted – as shown in Listing 4 line 300 – to only allow elements which use another namespace. This is not really a hard restriction considering that all extensions should have their own namespaces anyways.

2.4 Eclipse platform

After the development of the SrcML schema an exemplary tool was implemented to demonstrate the transformation from Java source code into SrcML. As mentioned in Section 2.1, Eclipse was chosen to base this implementation on. This section provides a short introduction to the Eclipse platform, before discussing the process of plug-in development.

2.4.1 Eclipse platform architecture

The Eclipse platform is based on a very modular architecture which is extensible by plug-ins. An Eclipse plug-in is a collection of Java classes which provide functionalities to the Eclipse platform. As can be seen in Figure 2 from the official Eclipse webpage [ecl] even the platform itself is built with plug-ins. The *Platform Runtime* is required for the plug-in mechanism to work as it contains a registry of available plug-ins as well as an API to access functionality for managing plug-ins.

The choice of using plug-ins allows writing tools with a very small core functionality and a freely selectable set of plug-ins which provide more specific functionalities. In our case the implementation contains a small core responsible for loading parsers and executing them on given documents to retrieve a SrcML document. The actual parser is located in a plug-in. This means that adding new parsers for additional languages is as easy as copying the corresponding plug-ins into the Eclipse plug-in directory. The parser core will automatically scan all registered plug-ins for usable parser plug-ins and provide them to the user.

The *Team* related plug-ins in Figure 2 are responsible for managing the shared access to source code among a team which consists of the Concurrent Versions System (CVS) integration mostly. They are of no further need for the purpose of this work just as well as the *Help* related plug-ins which manage the collecting and presenting of documentation in the Eclipse platform.

The *Workspace* is responsible for managing the available resources including projects, folders, and source code files. Not all of these plug-ins are required for the purpose of this work. The implementation was created such that transforming Java source code also works on the commandline without the need of running a complete workbench window, although transforming single files without any project specific context may lower the quality of the resulting document due to missing context information. Especially type resolving cannot work without setting up an Eclipse project to enable the parser to make use of the other source code files. The most obvious way this lack of information can be seen in SrcML documents are the missing ID and reference attributes linking variables to their declarations.

The *Workbench* related plug-ins are responsible for what an Eclipse user sees on her screen. To this end Eclipse provides the *Standard Widget Toolkit* (SWT) which provides native widget implementations and *JFace* which provides a higher level API for common GUI related tasks. These plug-ins have f.ex. been used for developing the SrcML tree view described in 2.5.

The *Plug-in Developer Environment* (PDE) provides plug-ins which help developers to implement new plug-ins and was used for the accompanying implementation, but is not a requirement for the usage of the developed tools. The *Java Development Tooling* (JDT) consists of plug-ins which provide Java specific functionality including the parser for Java source code files.

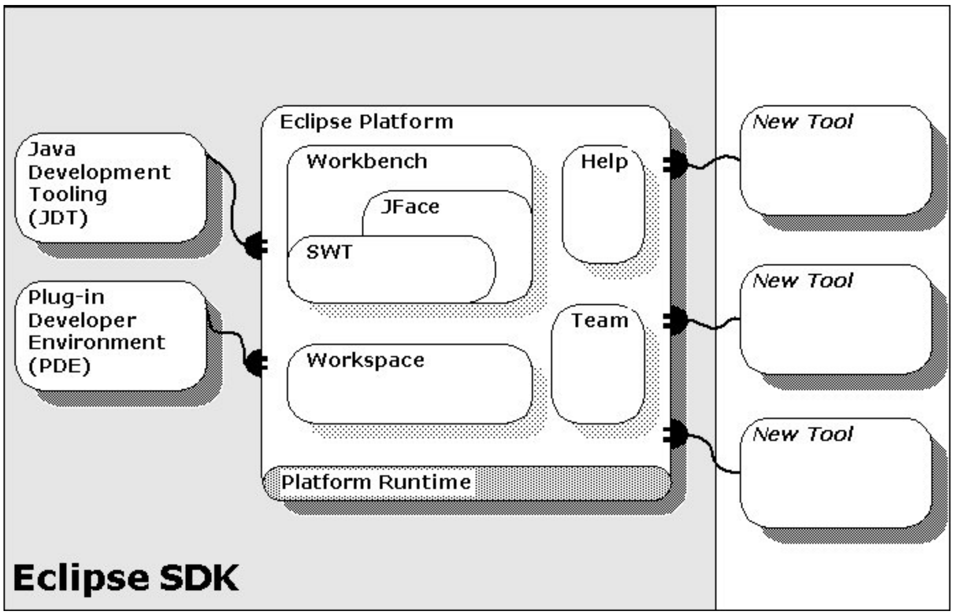


Figure 2: Eclipse platform architecture

2.4.2 Implementation of Eclipse plug-ins

Developing plug-ins for Eclipse is covered in [GE03] and this section will therefore only give a short summary of the process. With the help of the PDE most of the plug-in can be specified in a declarative way which is stored in the manifest or `plugin.xml` files accompanying the plug-in. This covers metainformation like the plug-in's author, version, and technical details like published packages, dependencies on other plug-ins, and so on. This section will focus on the more interesting extensions and extension points instead:

Definition 2.4. For the purpose of this work an *extension point* is an XML schema. It is bundled in an Eclipse plug-in and considered a place where other plug-ins can provide new functionality according to the schema.

A instance of the schema of an extension point is called an *extension.* It is usually accompanied by an implementation of the provided functionality and bundled in an Eclipse plug-in as well.

Remark 2.5. For simplicity we often use the words plug-in and extension synonymously. It is important to note though, that a single plug-in can provide multiple extension points. In fact a plug-in can provide extension points as well as extensions in the form of implementations for the same and/or other extension points. Basically the term plug-in is used for the means of how extensions and extension points are distributed. The core functionality mentioned earlier f.ex. is a plug-in which provides several extensions, but also an extension point for parsers.

Due to Eclipse being highly modular it is itself based on extensions. An example for an extension point is the `org.eclipse.ui.popupMenus` extension point which is used to add new entries to popup menus. Listing 11 shows how a popup menu can be extended in a declarative way in the `plugin.xml` file. In this example the right-click popup menu is affected when a file with the `.srcml` extension is selected. In such a case an additional submenu is added which contains an entry to validate the selected file against the SrcML schema. Note how in line 15 the declarative definition states the class file which contains the implementation for the functionality to be executed when this menu entry is activated.

```
1 <plugin>
2    <extension
```

```
            point="org.eclipse.ui.popupMenus">
        <objectContribution
                adaptable="false"
                id="de.srcml.ui.SrcMLMenuContribution"
                nameFilter="*.srcml"
                objectClass="org.eclipse.core.resources.IFile">
            <menu
                    id="de.srcml.ui.SrcMLMenu"
                    label="%srcml.file.menu">
                <separator name="separator"/>
            </menu>
            <action
                    class="de.srcml.ui.actions.ValidateAction"
                    enablesFor="1"
                    id="de.srcml.ui.Validate"
                    label="%srcml.file.menu.validate"
                    menubarPath="de.srcml.ui.SrcMLMenu/separator"/>
        </objectContribution>
    </extension>
</plugin>
```

Listing 11: Defining an extension in Eclipse

Usually a plug-in is a collection of extensions and extension points centered around a common functionality. For example the main `de.srcml` plug-in developed for this work provides several extensions of `org.eclipse.core.runtime.applications` which makes core functionalities available on the commandline. It also provides extension points for `de.srcml.parser` which allows other plug-ins to provide a means of parsing source code and transforming it into SrcML. The implementation created for transforming Java source code into SrcML also makes use of this extension point. This is one of the generic rules of plug-in development – called the *Fair Play Rule* in [GE03] – which says: "All clients play by the same rules, even me."

2.5 Implementation of SrcML in Eclipse

The goal of this implementation was to achieve the same functionalities the previous SrcML project described in Section 2.1 possessed with the exception of the analyses and API. Especially the Java parser is obsoleted by the parser available through Eclipse and the plug-in mechanism which was implemented manually is replaced by the OSGi framework bundled with Eclipse. This guarantees a more reliable implementation and reduces the amount of future maintenance significantly. As Eclipse is becoming more popular, parsers for various other languages are starting to show up which could similarly be used to parse those languages into SrcML with a minimal effort as well.

Most of the implementation made for this work concentrates around the Intentional Programming idea discussed in Section 3. This section only covers details about the Java parser and the underlying plug-in structure. Especially the transformation from SrcML documents back to normal Java source code is only slightly touched here, as it requires a fundamental feature introduced in Section 3.2.3. A more detailed discussion of this transformation is given in Section 3.5 which details the implementations made specifically for Intentional Programming. As mentioned in Section 2.1 no source code was reused from the previous project due to the changes in the SrcML format and Eclipse replacing the underlying architecture.

After deciding to switch to the Eclipse architecture we want an implementation which helps making the SrcML format available to potential users. To this end it should be possible to transform Java source code into SrcML, as well as transforming it back again. Therefore a lot of the initial implementations are used to perform various transformations. Most importantly the Java parser, built upon the Eclipse Java parser, transforms traditional Java source code into the SrcML format. Another important transformation is creating a readable text representation of a SrcML document, as the XML data itself is not suitable for presenting it directly to developers. This section further discusses using these two kinds of transformations in different ways. Furthermore the plug-ins created this way are integrated into Eclipse to provide for a better user experience.

2.5.1 Java parser

The `de.srcml.java` plug-in which contains classes for transforming Java source code to SrcML is connected to the `de.srcml.parser` extension point mentioned above. Listing 12 shows the declaration of this extension which is very straightforward: It contains the Java class which provides the necessary functionality and specifies Java as the only programming language this parser can handle.

```
<plugin>
    <extension
          id="parser"
          name="%srcml.parser.java"
          point="de.srcml.parser">
       <run class="de.srcml.java.parser.ParserJava"/>
       <language name="java"/>
    </extension>
</plugin>
```

Listing 12: Java parser extension

The class `ParserJava` itself is then implementing the `IParser` interface which is shown in Listing 13. It essentially contains methods to parse source code from different sources: a generic `Reader`-based source or preferably an `ICompilationUnit` allowing Eclipse to calculate type bindings which can improve the quality of the resulting SrcML document. Furthermore it is possible to only parse type declarations or expressions which is often easier for creating a complex DOM subtree than building it manually. A developer can use these methods to call them with a `StringReader` on a String which contains a normal Java expression. The result is a SrcML document which can be connected to another document as a subtree.

```
public interface IParser {
    public static enum Kind {
        UNIT, TYPE_DECL, EXPRESSION;
    }
    public Element parse(Reader reader);
    public Element parse(Reader reader, Kind kind);
    public Element parse(ICompilationUnit unit);
}
```

Listing 13: IParser interface

The actual implementation of the transformation can be found in the `JavaASTVisitor` class which is inheriting from the `ASTVisitor` class. This means the Java parser provided by the Eclipse JDT first parses the source code and creates an abstract syntax tree which is then visited to create the SrcML document from it according to the visitor pattern [GE94].

An exemplary method from `JavaASTVisitor`, which transforms assignments from the abstract syntax tree into SrcML, can be seen in Listing 14. The if statement is only used for a unit test and not important for this discussion. Lines 787-788 initially create elements for the assignment. As an `assignment` tag is only allowed inside an `expr` tag, both elements are created here. Because there are several possible assignment operators, the operator is stored in an attribute of the assignment element in line 789. In this implementation the variable `current` is always holding the current DOM element during the construction process. So when the visitor pattern leads to a call of the method in Listing 14 there will already be a partial tree built to which the subtree of this assignment is attached in line 790. For the left hand and right hand sides of the assignment the current variable is set to the assignment element so that the respective subtrees are correctly attached to it. Lines 793 and 795 then invoke the visitor pattern for the left hand and right hand sides. The method returns false to avoid visiting child elements which have already been included.

```
783     @Override
784     public boolean visit(Assignment node) {
785         if (bVisit)
786             visitedNodes.add(node);
787         Element expr = createElement("expr");
788         Element assign = createElement("assignment");
789         assign.addAttribute("operator",node.getOperator().toString());
790         current.add(expr);
791         expr.add(assign);
792         current = assign;
793         node.getLeftHandSide().accept(this);
794         current = assign;
795         node.getRightHandSide().accept(this);
796         return false;
797     }
```

Listing 14: visit method for assignment

After the whole abstract syntax tree was visited like this the result is a complete DOM representation of the corresponding SrcML document. This DOM representation can then be directly used for further tasks or it can be converted into a string of its XML representation and stored in a file for later use. As Listing 13 shows the parser extensions generally return an Element instance – which is an element from the DOM tree – as those instances already provide an implementation for a straightforward mapping to a string.

The implementation accompanying this work makes use of the parser extension at two places. There is a command line application which converts the DOM into a string and either prints it on the standard output stream or writes it into a file. The other usage appears as part of the Eclipse integration: A *project nature* was created for SrcML which allows users to set a flag for a project to tell Eclipse that this project is considered to be a SrcML project. This in turn means that the SrcML *builder* created for this purpose automatically converts source code into SrcML documents whenever changes are made to the original source code. Initially when the SrcML project nature is set for the first time the complete source code associated with the project will be transformed into SrcML.

The implementation made for this purpose is only useful as a proof of concept so far, because SrcML documents are completely overwritten whenever the original source code is changed. This complicates the addition of for example metainformation to the SrcML document, as the information could easily be erased as a side effect of the rewrite of the document. A better integration – which was not created due to the restricted time available – should make use of the deltas the Eclipse parser offers for changes and thereby provide incremental changes to existing SrcML documents. In the long run it might be preferable to work directly on the SrcML documents through the use of a special editor as discussed in Section 3.3. The Java parser would then only be used to initially transform source code into SrcML and thus prepare it to be used in that special editor.

2.5.2 Presentation of SrcML documents

When displaying a SrcML document in an editor it is not suitable to use the XML format for this. Traditional XML editors work for SrcML documents, but it is cumbersome to write source code directly in XML due to the high verbosity of the format. Therefore we added a transformation which recreates the traditional Java syntax. Apart from being more readable than the XML format this also allows us to reuse existing Java compilers. It is very interesting to point out that creating a plain text representation of the source code always works from the same SrcML representation no matter how the text output is formatted. This allows several developers to see different textual representations of the same source code suited to their personal formatting preferences. It could also help with enforcing a corporate layout for the presentation of source code.

This advantage of SrcML inherently eliminates the need for formatting specific code conventions which are still in use nowadays whenever teams of developers are working on the same source code files. By improving the way source code is stored an implicit improvement of the way it is displayed to developers can therefore be achieved. [Wil04] elaborates on this idea to the point of representing traditional Java source code in a way more familiar to a Lisp developer.

The implementation we made for transforming SrcML to Java is based on the declaration of an extension point. More specifically the extension point is responsible for extensions which transform a SrcML document to a textual representation. It is only through the use of a special extension that the output resembles Java source code. Different syntactical representations can then be achieved by providing several extensions. For the purpose of this work an extension was made which is loosely based on the Java Coding Conventions proposed by SUN. The classes used for such an extension have to implement the `IPresentation` interface which is shown in Listing 15.

```java
public interface IPresentation {
    public void present(PresentationManager mgr,
        IPresentationDestination dest);
}
```

Listing 15: IPresentation interface

The implementation makes use of the adapter pattern as described in Section 3.5.1. For the sake of simplicity it can be assumed that an instance of `IPresentation` is created for every element of the DOM on which the `present` method is called. The decision to use this design is based on the problem of transforming elements which have unknown child elements due to the extensibility of the SrcML schema. As this is a more generic problem it is covered in detail in Section 3.5.1. The `PresentationManager` passed to the `present` method is used to manage the transformation of child elements. It is used to add the string representation without requiring any additional information from the caller. The textual output is made through the `IPresentationDestination` instance which contains methods for outputting tokens, incrementing or decrementing indentation, requesting newlines, etc. For example when creating the presentation for an if statement the `PresentationManager` is used to create a textual presentation of the if statement's condition and blocks and the `IPresentationDestination` is used to create the `if` and `else` strings and output those elements in the correct order.

2.5.3 Combining parser and presentation

Having transformations available in both directions leads to investigating the effects of converting Java source code into SrcML and back into Java source code again. As the SrcML format only stores the information contained in an abstract syntax tree there is no means of recreating the exact Java source code. Note that contrary to similar projects we do not consider this a drawback at all. In fact we consider traditional textual source code to be obsolete and are not required to be downwards compatible to it. Therefore the generated Java source code is syntactically determined by the chosen extension and is not depending on the formatting of the original input document. This may be considered by some as a drawback as it disallows the comparison of those two Java source codes. We believe that a comparison should rather be made on the SrcML level instead which contains the code's syntactic structure. The presentations of this structure are not required to be equal and in fact we want them to be arbitrarily different as to match the developer's personal preferences.

As a side effect of this transformation process we are further rewarded with a pretty-printing tool for free. The initial Java source code can be formatted in any way, as only its structure is contained in the SrcML document. But when the SrcML document is transformed back into Java source code the chosen extension takes care of proper formatting. With the ability of providing multiple extensions with different implementations it is also possible to satisfy various interpretations of the term *pretty* in this pretty-printing process.

With all those possible transformations one might be tempted to investigate the transformation from one programming language into another through the use of SrcML. A first idea might be to transform C++ source code into SrcML, and then have it printed with an extension created for SrcML documents containing Java code. However one has to keep in mind that SrcML is only a different way to store the syntactic structure of a program. It is therefore possible to run the transformation tools on a SrcML document which was generated from a C++ source code and force the output of a Java source code. But with only the syntactic markup being present the Java source code will not be compilable, as constructs like `cout << ‘‘hello world’’;` will not be transformed into the proper output routines of Java.

We have not further investigated the possible usage of such transformations, but they might be useful to perform an initial transformation which then has to be manually verified and adjusted. It is also possible to create specialized transformations which take a source code's semantics into account when transforming to the target language. This presentation mechanism itself though can be used to create better cross-language transformations. As f.ex. the information that the document is written for Java is stored within the source code this information can be used by an extension. It is therefore possible to create extensions which retrieve the document's programming language and perform special transformations for well-known constructs of this language. But this is a whole new subject and beyond the scope of this work.

2.5.4 Transformations

It should be noted that, while the term *transformation* suits the process of transforming a textual representation into a DOM representation or vice versa, the implementation differentiates between two terms: *presentation* and *transformation*. Creating a SrcML document from a traditional textual representation is called *parsing* and does not have anything to do with transformations from our implementation's point of view. The opposite direction of recreating the textual source code is called a textual *presentation* of the SrcML document. While this is some kind of transformation it is still not what we call a *transformation* in our implementation. A *transformation* is instead based on the `ITransformation` interface shown in Listing 16.

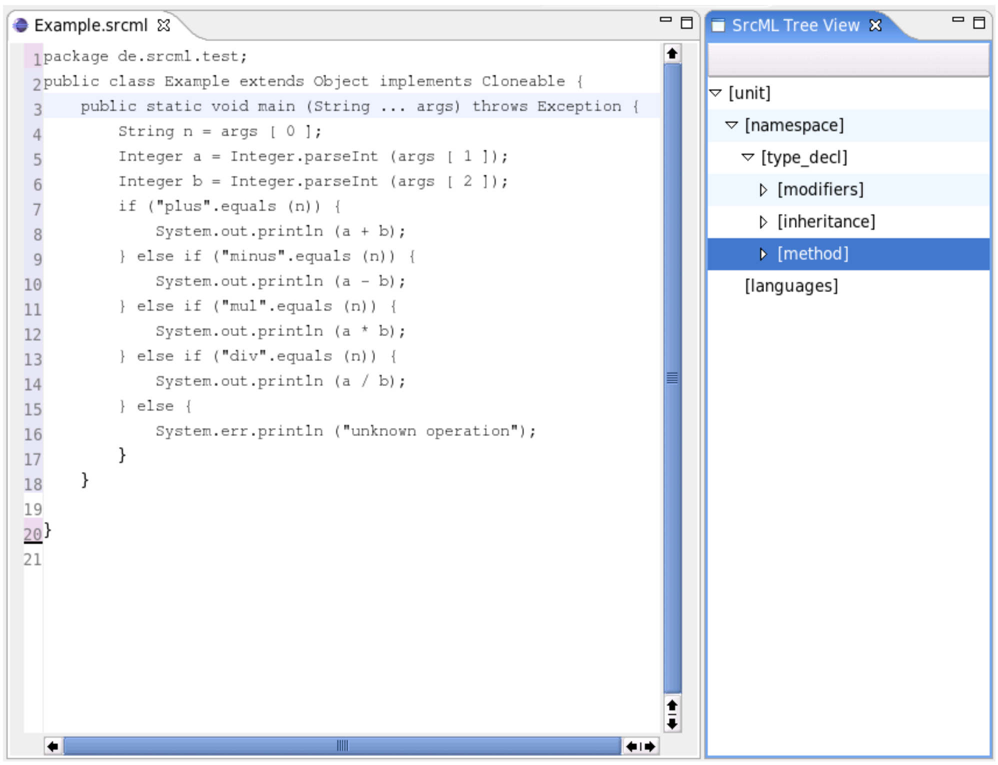
```
public interface ITransformation {
    public void transform(TransformationManager mgr);
}
```

Listing 16: ITransformation interface

Transformations as they are called in the implementation are transforming the DOM tree. This is mainly used for language extensions when new statements can be mapped back to a base language. A transformation then replaces the corresponding node by a subtree providing its implementation in the base language. The transformation is not restricted to local changes. Any amount of changes can be made to the DOM, although in the example of language extensions only additions are made to the DOM. This kind of transformation is explained in more detail in Section 3.4 where we create an extension of the Java programming language to improve the arithmetic example introduced earlier.

2.5.5 SrcML tree view

The implementation was created in a way which only exposes a textual presentation to the developer in Eclipse instead of the XML format. Sometimes though it is useful to take a closer look at the data model one is working on. To this end we added an additional view to Eclipse which we called the *SrcML tree view*. This is a simple tree view displaying the SrcML DOM of the document currently opened in the editor. Due to the time constraints of this work it was not possible to thoroughly integrate some functionalities into Eclipse and the SrcML tree view was used instead to provide access to some of those functionalities. For example the control flow graph



```
package de.srcml.test;
public class Example extends Object implements Cloneable {
    public static void main (String ... args) throws Exception {
        String n = args [ 0 ];
        Integer a = Integer.parseInt (args [ 1 ]);
        Integer b = Integer.parseInt (args [ 2 ]);
        if ("plus".equals (n)) {
            System.out.println (a + b);
        } else if ("minus".equals (n)) {
            System.out.println (a - b);
        } else if ("mul".equals (n)) {
            System.out.println (a * b);
        } else if ("div".equals (n)) {
            System.out.println (a / b);
        } else {
            System.err.println ("unknown operation");
        }
    }

}
```

Figure 3: Screenshot of SrcML tree view

generation discussed thoroughly in Section 4 is accessible through it. Figure 3 shows a screenshot of the SrcML tree view representing the DOM tree for the arithmetic example.

3 Intentional Programming (IP)

In this section we introduce Intentional Programming (IP) and its application to SrcML. This combination comes naturally considering that IP shares several properties with SrcML. For example both emphasize extensibility whereas IP even provides several different types of extensibility. We show that SrcML's extensibility matches the addition of new intentions to an IP environment. The data structure used to represent source code in an IP environment also bares a close resemblance to a SrcML DOM tree. Similar to the language neutrality in the SrcML schema IP is abstracting from specific programming languages through the use of intentions. These intentions again bare a close resemblance to SrcML tags.

Because the terms used for Intentional Programming are not clearly defined in the literature, Section 3.1 establishes definitions of the related terms for the purpose of this work. Section 3.2.1 highlights some of the advantages programmers gain from Intentional Programming. Section 3.2 discusses how the different components of Intentional Programming work in principle and Section 3.3 explains a special editor based on IP. As an application of Intentional Programming we developed an extension of the Java programming language which is presented in Section 3.4. An implementation was made as part of this work which combines SrcML with ideas from Intentional Programming and Section 3.5 describes some details of this implementation.

3.1 Definitions

Definition 3.1. The term *Intentional Programming* is coined in [CE00] where it is defined as an "extendible programming and metaprogramming environment based on active source". *Active source* in turn is defined as "a graph data structure with behavior at programming time". The term *Intentions* appearing in Intentional Programming is defined in [Sim95] as "memes of language features".

These are very informal definitions and [CE00] spends several pages on explaining these terms in more intuitive ways. As it is very unreliable to build upon such vague definitions and large parts of this work are related to Intentional Programming we use the following different definitions for the purpose of this work:

Definition 3.2. Definitions used for Intentional Programming:

- An *Intention* is an abstraction feature of a programming language which closely resembles the intent of the programmer using it.
- *Active Source* is a finite directed graph with intentions as nodes and a mapping which maps each intention to a possibly empty set of operations.
- *Intentional Programming* (IP) is a programming environment for working with active source.

Nevertheless some comments on Definition 3.2 are due as the nature of these terms makes a formal definition very hard to grasp. The meaning of an intention as used in the context of IP is not far from its meaning in the English language. It resembles an idea of the programmer who wants to add a certain intent to her program. Intentions therefore cover well-known constructs of programming languages like class declarations, assignments, or method calls as well as more abstract ideas like the sum of the values of a collection.

One of the ideas behind active source is to use these intentions as a basis for representing source code. The graph structure of this representation originates from abstract syntax trees enriched with additional links. A core concept of IP is the addition of operations to those intentions. It is important to realize that these operations are evaluated during programming time as they are working on the source code itself and are usually not part of the developed program. Although these operations can also be used in the resulting programs to process source code, which is discussed later under the term metaprogramming.

Remark 3.3. Every traditional textual source code `S` can be represented as active source.

An important aspect of compiling source code is the creation of abstract syntax trees during the parsing process. Let `T` be the AST corresponding to `S` as it is created by an arbitrary parser for the programming language `L` the program `S` is written in. Now declare intentions for every kind of node occurring in `T`. These intentions are not abstracting from the language very much, but Definition 3.2 allows intentions to be very specific. In fact the level of abstraction in this case is equal to the level of abstraction available in `L`. By replacing every node in `T` with an instance of the corresponding intention the active source representation of `S` is created. When adding additional edges to the tree, for example from variable usages to their declarations, the tree can further turn into a generic graph. Note that in this case each intention maps to an empty set of operations.

As Remark 3.3 shows, active source graphs are very similar to abstract syntax trees. This allows the combination of the active source idea with SrcML: As SrcML documents are trees as well and the XML format allows adding additional edges through attributes the active source representation can be based on SrcML. Every element available in SrcML – including possible extensions – therefore can be considered an intention. The set of operations associated with these intentions is incrementally created during the following sections and is subject to extension as well. Section 3.5 covers implementation details of this association.

Note that a SrcML element is not necessarily identical with an intention, as the actual intent can still depend on other parameters. In Listing 4 the `type_decl` element is used to declare a class which also is the desired intention. Nevertheless the `type_decl` element can also be used for declaring an interface which is a different intention. So while there is an intuitive correspondence between elements and intentions this does not always have to be a one-to-one correspondence.

The term Intentional Programming which encloses intentions and active source refers to a complete environment for programmers. It can be thought of as an Integrated Development Environment (IDE) like Eclipse with the difference of being based on intentions. Apart from replacing the underlying data model with active source graphs it also provides access to the operations associated with the intentions. For example these operations include an operation which is used for displaying a textual representation of the active source. Because of this operation an IP environment can look very similar to existing IDEs from the outside. The following section explains the advantages programmers gain from an IP environment.

3.2 Properties of IP

3.2.1 Advantages

The Intentional Programming environment provides programmers with several benefits. One major advantage is how IP can display the active source. As active source can provide multiple operations for displaying itself, the program a developer is working on can be shown in several ways. It is possible to recreate a traditional textual presentation of the source code with which programmers are most familiar. Far more radical approaches are possible though, depending on the abstraction level of the intentions used in the active source. Coming back to the earlier example of an intention for the sum of all values in a collection the IP environment is free to display it as Σ. This already shows how an IP environment can benefit the programmers by clearly identifying the intentions. Traditionally summing up the values in a collection could result in source code similar to the one in Listing 17. When the intent of summing up all the values in the collection is properly abstracted into an intention and integrated into the IP environment a programmer might instead see the code in Listing 18. The latter version is a much clearer way of expressing the intent of the programmer.

```java
int sum = 0;
for (Iterator i = collection.iterator(); i.hasNext(); )
  sum += ((Integer)i.next()).intValue();
```

Listing 17: Java source for sum

```
int sum = Σcollection;
```

Listing 18: displaying of sum intention in IP environment

If Java already had a language construct for this purpose, there would be no need for creating a new intention for it. However the intent of a certain part of the source code is often specific to the problem the developer is trying to solve with it and thus can only rarely be covered directly by a feature of a traditional programming language. This leads to a desired increase of available intentions specific for a developer's problem domain. How this is achieved is further discussed in Section 3.4 which introduces the addition of new intentions.

The IP environment itself promotes the development of new intentions, as it is designed to be extensible. A developer working in this environment is not restricted to creating a simple text file which is associated with a given programming language as it is currently done in IDEs. Instead she is able to select a set of intentions she wants to use in the environment and an active source is therefore not written in a certain programming language, but with a set of intentions. We consider this a major improvement over the restrictiveness of traditional programming languages, as it allows a developer to customize her development environment so that it matches her problem domain more closely. This effectively allows developers to adjust the programming language used to the problem domain as opposed to the other way round.

When source code is written with the help of a set of intentions it is possible, that the resulting textual presentation is ambiguous. In fact the likelihood for an ambiguous syntax increases with the number of intentions added to the environment. While this is a real problem when trying to extend traditional programming languages the IP environment itself nullifies it: As the environment is working on active source it is independent from the textual artifacts generated for the underlying data structure. This means that text which is created for the developer does not have to be parsed, because the result of the parsing process has already been used to create that text in the first place. While it may be desired in some situations to create a textual source code presentation which still remains unambiguously parseable to support third party tools, a developer of new intentions generally has a simplified task as neither parsing nor grammars are of concern.

Working with active source in an IP environment means having a graph data structure available and invoking operations on this structure. This is in itself another advantage of the IP environment in that it allows developers easy access to metaprogramming facilities. Writing a tool which modifies another program's source code is much simpler with the help of IP. First of all the other program's source – when already available as active source – does not require the development of a parser. It suffices to load the active source through the means already provided by the IP environment. This means that the very same functionality which is responsible for the developer being able to load the active source of her own tool into the IP environment during programming time can be reused to load another active source from within the tool during execution time. Similarly modifications performed on active source during programming time are based on operations which can be reused as well. Just like the IP environment uses these operations to display a textual presentation of the source code for example a tool can use these operations to create the same output.

Furthermore it is possible to import existing legacy source code into the IP environment with the help of additional parsers. While this parsing process can create active source as explained in Remark 3.3, the intentions of the original programmer are not properly captured like this. Nevertheless a conversion to active source can be used as a basis for refactoring which in the case of IP means replacing basic intentions with more abstract intentions which better resemble the purpose behind that part of the source code. Legacy code can therefore be imported into an IP environment to work with and improve it by refactoring it to use more appropriate intentions.

The Intentional Programming environment gains most of its power from the use of a special IP editor. Normal text-based editors are unsuitable for IP, as they are unaware of active source as the underlying data structure used for the source code. The advantages already discussed in this work could still be utilized with special tools, but it is believed that a seamless integration into an editor comes most natural to developers and offers the most benefits. It also allows a gradual transition to using SrcML and the IP concepts, as the editor could almost be used like a normal text editor and gradually extended to support more advanced features. The IP editor is originally presented in [CE00] and we examine the ideas presented there as well as investigate the usage of SrcML for it in Section 3.3.

A very interesting feature of an IP editor is its ability to take advantage of the high-level abstractions given by active source. This allows ideas like presenting source code in a totally different way. Section 3.5 gives some examples of how this can radically change the way we look at source code. The example mentioned above about adding mathematical notations to source code is just the beginning of what is possible. Especially with regard to intentions which are highly abstracted towards a specific problem domain the IP editor allows to adopt notations of this problem domain.

3.2.2 Comparison to domain-specific languages

Domain-specific languages (DSLs) – also known as *little languages* – are programming languages which are tailored towards a specific problem domain. These languages optimally provide equivalent abstractions to all terms used in that problem domain. Intentional Programming in turn obsoletes the notion of domain-specific languages, as those abstractions can be thought of as intentions. This means that a DSL is realized within an IP environment as a set of intentions which are specific to the problem domain. Contrary to traditional DSLs there is no need of being restricted to a given language when working in an IP environment. It is fully possible and convenient to work with a superset of the intentions given for the problem domain.

This section explains properties of traditional DSLs whereas it is important to notice that all advantages given by DSLs are automatically carried over to the IP environment. It further discusses the disadvantages related to *general purpose languages* (GPLs) like C++, C#, or Java. Domain-specific languages are making up for some of these disadvantages and therefore the same holds true for Intentional Programming. In fact the IP environment can remedy several of these disadvantages completely, because the working set of intentions can work as a superset of the intentions corresponding to a GPL and a DSL and thus leverages the advantages of both.

Domain-specific languages offer the developer an abstraction level which is much closer to the problem domain. This makes expressing problems in that domain easier. Some of the functionalities provided by a DSL can be compared to a library. In fact some developers consider a library as a kind of DSL as well, which is not too far off the truth. There are other ways of providing a domain-specific language besides a library. Sometimes a DSL warrants its own syntax, but often DSLs are added on top of existing languages. For the remainder of this work it does not matter how exactly a DSL is provided.

A major advantage of domain-specific languages is the ability to write very short and concise source code. As the language is tailored towards the problem domain the abstractions it provides are already sufficient, whereas in a GPL the developer is often required to emulate this abstraction on a less abstracted level. The result of this emulation is an increased size of the source code. A simple statement provided through a DSL generally requires several lines of code in a GPL to achieve the same effect. It is also easier for another developer to read source code written with the help of a DSL. While the emulation of a DSL abstraction may be clear to the developer who wrote it it demands from another developer to understand what is being emulated. This means the second developer has to find out the intention of several lines of source code as opposed to a short and concise part of the source code. When a DSL provides domain-specific notation as well this makes it even easier for other developers working in the same problem domain to understand source code written in that notation.

Considering those advantages we may ask ourselves, why general purpose languages still exist. While a DSL is generally favorable to a GPL when working in that problem domain the actual problem lies elsewhere: it isn't always easy to identify the problem domain. This identification requires an expert of the domain who is not always available. Furthermore a lot of time is required for this which could already be used for the implementation in a GPL. Even after the problem domain is identified chances are that there is no existing DSL for the domain. In such a case a domain-specific language has first to be created which is only worthwhile in large scale projects.

Depending on how the DSL is to be used this step requires an expert in language design. And after the DSL is finally available the developers who are supposed to use it still have to be trained as it is after all a new programming language unknown to them. All of this can make the usage of a DSL a very time consuming and expensive affair which generally hinders the development of DSLs.

The IP environment while unable to completely eliminate those problems tries to at least reduce their severity. As a DSL in the IP environment is nothing more than a set of intentions, and intentions can be combined as the developer wishes, the need of exactly identifying the problem domain is reduced. There is no harm done if only part of the problem domain is abstracted via intentions. Usually a DSL is considered incomplete if it does not cover the complete problem domain. For example if a DSL is published for graph algorithms and it contains a depth first search, but not a breadth first search it will not be accepted by the developers. In an IP environment one would instead just publish a set of intentions which happens to include a depth first search. Similarly a developer working in an IP environment wants to include an intention for the depth first search when she requires one. It is not relevant when other intentions are unavailable. Of course once all necessary intentions are available it is still possible to publish those as an intention set for graph algorithms.

Furthermore Intentional Programming makes it easier to integrate new intentions. As the whole IP environment is based on intentions from which the developer selects whichever intention she wants to use, the addition of new intentions is straightforward. Most importantly existing active source is not affected by the addition of new intentions at all. The problem of downward-compatibility is also non-existent. There is no way of breaking existing active source by adding new intentions to the IP environment. With traditional DSLs developers had to be careful, when for example changing the syntax of a programming language. If a DSL adds new keywords to a GPL this can easily break other code in the project which uses these keywords as identifiers. The syntax independent nature of active source, however, completely eliminates such problems. While it is important to traditional programming languages to consist of a syntax which allows for being easily parseable the active source approach already consists of the result of this parsing process. Therefore developers of new intentions don't have to bother with syntax if they don't want to. Section 3.5 gives an example of how radically different the same intentions can be displayed to the developer.

This lowers the boundary for who can add intentions to the IP environment. There is no need of being a complete expert of a problem domain, as it is sufficient to be fluent in those parts of the domain which are required for solving one's problems. Similarly integrating new intentions does not require a language expert as many of the problems which could be caused by the addition are not existing within an IP environment either.

Apart from the advantages of DSLs the general purpose languages themselves also have disadvantages. The IP environment helps reducing the severity of these disadvantages by being able to enhance GPLs with higher level abstractions in the form of additional intentions.

One problem – which at the same time is a major advantage of DSLs – is the loss of design information in source code written in a GPL. As mentioned above the developer is forced to emulate a certain intent through several lines of code. This can make the source code harder to understand up to the point of not even being able to recover the original design informations. For example there are many possibilities for implementing the singleton design pattern from [GE94], but given the source code of a class it is not always possible to determine if the original author intended it to be a singleton. In an IP environment the developer can include this information directly in the active source and thus make her intent clear to everyone reading her source. In fact this makes the intention known to other programs as well which is superior to a simple comment. Similarly the lack of domain-specific programming support in GPLs leads to more cluttered code which is harder to understand and maintain. This is after all the reason, why developers are considering DSLs to be helpful.

When discussing domain-specific languages there seems to be a set of questions which continually reappear and which we want to clarify here in the context of Intentional Programming:

- What about the cost of teaching developers the new domain-specific notations?
- Aren't feature-rich languages more complicated?
- What about notational havoc being created due to everyone being able to extend a language?
- What properties do we know about the interoperability of language extensions?

When creating new intentions it is a necessity to teach those intentions to the developers who are supposed to use them. Depending on the number of developers involved this can be a significant cost factor. But the costs involved with not using those intentions and continue working with general purpose languages may well exceed the teaching costs: With no domain-specific abstractions being available there is a lot more work to be done during the creation of the source code. And after completion the resulting software has to be maintained which is significantly harder the lower the abstraction level of the source code is. So while adding new intentions gives an initially higher cost it reduces other costs in the long run.

Feature-rich languages tend to be complicated. There are many possible expressions a developer has to master and it takes a long time to learn how to read other developers source code. The fear of this complication to get out of hand with ever new intentions being added to an IP environment is therefore justified. Nevertheless it should be considered, that this level of complexity is only achieved when a developer decides to actually make use of all those intentions in her active source. Considering that many intentions are restricted to their specific problem domain this should be unnecessary most of the time. In an IP environment the developer chooses a set of intentions to work with and we can safely assume, that she is familiar with the chosen intentions, or at least plans to learn them in order to solve her programming problems. So on a closer look the complexity of the features available to the developer are always restricted by the chosen intentions, which means that a developer should not be overwhelmed by the available features.

The different notations available in an IP environment can result in documents which contain what could be called a notational havoc. But similarly to the above argumentation one should bear in mind, that the developer chooses the set of intentions and thus the available notations. So restricting the intentions to only the required ones already reduces this problem and the remaining intentions use a notation which is specific to their problem domain. This notation optimally is the best possible notation for this intention considering that it was created specifically for this problem domain. Therefore while there may be a large amount of different notations all those notations should be familiar to the developers working in these problem domains.

The interoperability of intention implementations is not clarified further. None of the existing writings on Intentional Programming delve into details on this topic, thus we consider this a point for further research. While it is possible when given a fixed set of intentions to make them work together in a clearly defined way the question remains what happens when a developer uses intentions developed independently by several other contributors. Considering how the reuse of such intentions is a significant advantage of the IP environment further research in that direction is justified. It is especially interesting to find out if there is a scalable approach which does not require special glue code between each pair of intentions to make them properly work together. Assume that a number of intentions have been created independently and a developer wants to use all of them for her active source document. Are there a lot of special cases to be considered then? When further investigating this issue one also has to clarify first what exactly interoperability refers to in this case.

3.2.3 Adapters for active source operations

After the required terms have been defined, this section goes into more details of how the Intentional Programming environment works. It explains the basic building blocks based on Eclipse and how IP provides tremendous potential for extensibility. The last part is an in-depth coverage of an IP editor which is a very sophisticated editor taking full advantage of the features IP offers.

Intentional Programming is all around active source. When talking about functionality or active source functionality we refer to an Eclipse extension point which essentially is an interface.

These extension points are used to model the active source operations, meaning that operations are made available through extension points and their actual implementation is provided by extensions for that extension point. We leave the details of how to find such extensions to Section 3.5.

Active source operations directly manipulate the AST which in case of SrcML translates to a modification of the DOM. In fact every intention is a node in the DOM. As opposed to [CE00] we are using intention nodes directly instead of linking them to special declaration nodes for those intentions. From our understanding of [CE00] the declaration nodes are used in terms of active source to find and activate operations belonging to the corresponding intention. With SrcML we use an adapter concept instead which does not require this indirection step.

Based on [GE94] we apply the adapter design pattern to active source and SrcML: when a SrcML document is loaded a DOM representation of it will be available in the memory. This DOM representation also acts as our active source data structure. A remaining issue is how to associate operations with individual nodes of the DOM tree. As those nodes are instances of a class which is provided by the XML parser library we cannot make changes to it. Even if we could, we don't want to change it, as adding new extension points would always require a change of the underlying data structure which is unfeasible. Instead we apply the adapter pattern to adapt these nodes to the interface of a required extension point. This basically means wrapping the node instance into a class instance which provides implementations for the interface's methods. Section 3.5 covers the details of how this adapter pattern is brought to use in our implementation.

The important aspect is that given an extension point one also knows the accompanying interface and has an option available to adapt a node of the DOM tree to this interface which in turn allows execution of the associated operations. This allows us for example to print a textual presentation of the document or a subtree by adapting to the `IPresentation` interface and invoking its `present` method.

The process of adapting a node to an interface can be customized further. Often it is not enough to simply create an adapter, but the user wants to use a specific implementation. This comes naturally from the fact that an extension point can provide any number of implementations via extensions. Considering the example of printing a subtree of the document different users may want to create different outputs depending on their personal preferences. This can be achieved by giving the user the ability to further specify which extension to choose for the adapter pattern. In general the user can specify which extensions to use for each individual extension point. To this end our implementation contains an adapter view which lists the available extensions and allows the user to switch each one on or off. The adapter view is discussed in more details in Section 3.5.

This also implies that a situation might occur in which there is no extension available for a given extension point and thus the requested operation cannot be executed. Due to the highly extensible nature of active source and SrcML all developers of tools based on Intentional Programming will have to keep that in mind. In fact this is part of a remaining open problem which is discussed in Section 6.3.

3.2.4 Extensibility

With the active source concept being a very customizable idea this section investigates how extensible it is. To this end we have identified three different types of extensibility: new intentions, new active source operations, and alternative implementations. New intentions are thoroughly discussed in Section 3.4 and are only covered here in terms of extensibility. New active source operations means the addition of a new operation to the operations available on the active source nodes. For example creating textual presentations or compiling code are such operations. Section 4 shows this kind of extensibility by adding functionality to the source code for creating control flow graphs. Finally there is the option of different available implementations for the same operation. A straightforward example for these is the creation of textual presentations which can vary significantly in terms of the syntax created. Section 3.5 provides an example of how these different implementations can affect the developer's experience.

New intentions. The main idea of this work is using the SrcML format as a base for developing domain-specific languages or extending existing languages. This process is discussed in more detail in Section 3.4 while we are concentrating on the implications for active source here. When a new intention is added to a language the developer should also provide adapter implementations for the existing operations. The effort required for this grows linear in the number of available operations. Due to this number being extensible as well this means that adding a new intention with complete support for all active source operations requires a lot of effort. Often it is unnecessary to have all possible operations available to work with the new intention, so that it suffices to implement the set of core operations required in the beginning. New adapter implementations for the remaining operations can be added later if required.

New active source operations. Sometimes one happens to miss functionality which cannot be added by providing new adapters for existing active source operations. Considering that any kind of metainformation can be stored along with the source code this offers quite a large amount of possible operations.

Adding a new active source operation requires an interface which specifies the required methods for this feature. Then an extension point needs to be created where plug-ins which implement the required operation can register at. It is important to make a strict separation between the implementation of the operation and the implementation of the loading mechanism. The idea is that there can always be plug-ins added, but the loading mechanism – which may include some generic helper methods or classes for this operation – should remain unchanged, as any change could potentially break all existing plug-ins.

With the extension point and the loading mechanism being implemented the new active source operation is basically available, although it only becomes useful after the corresponding plug-ins providing the adapter implementations are available. This is a much greater effort, as adapters have to be created for all new intentions as well and so the effort grows linearly in the number of available intentions. Nevertheless it is possible to only implement adapters for those intentions which are actually used and thereby gradually introduce the new active source operation.

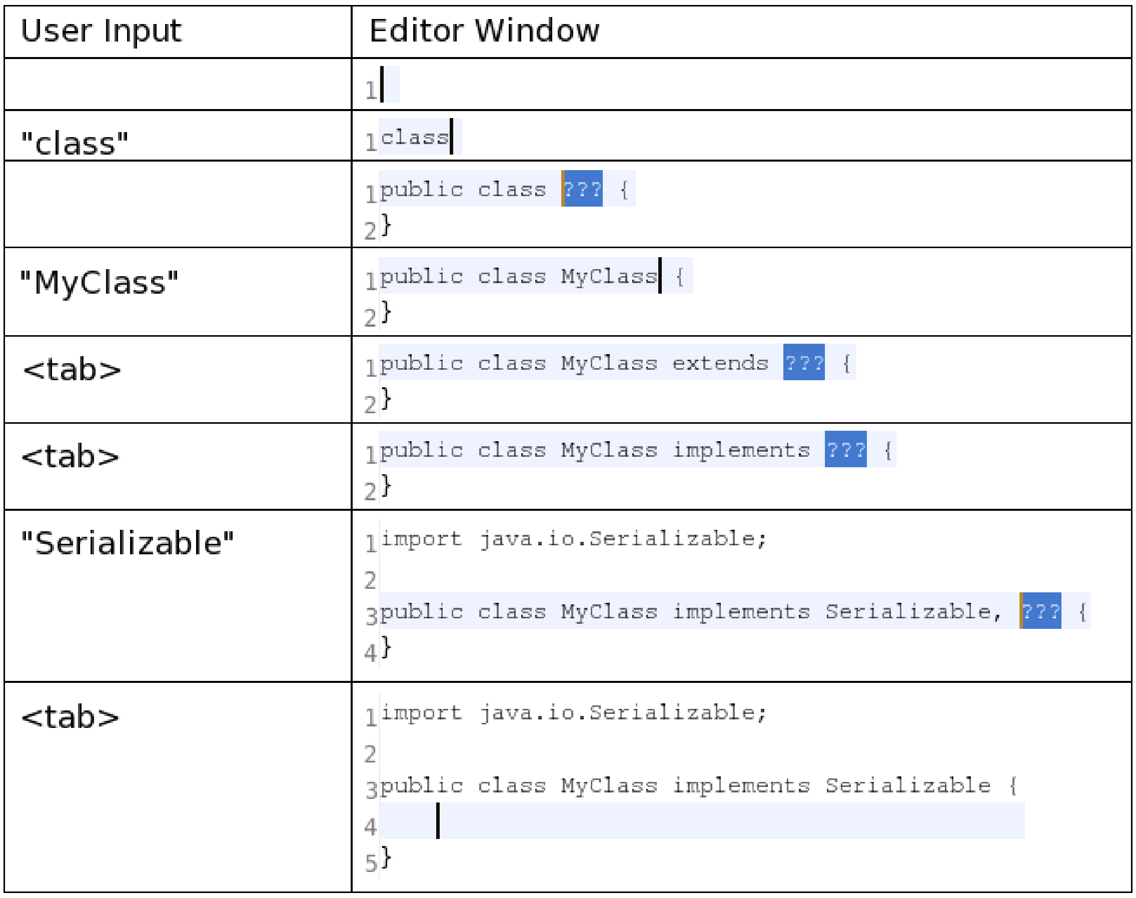
Alternative implementations. After a new active source operation is created it is usually accompanied by a set of adapter implementations. These provide a specific way of implementing the required operation which may not always be desired. It should therefore be possible to exchange some of these adapters with new custom implementations providing an alternative implementation. This kind of extensibility comes naturally from the choice of using the Eclipse architecture, as any number of extensions can be available for an extension point. It is the responsibility of the developer of an extension point to consider that multiple extensions may be available and provide a means of selecting one of them.

3.3 Intentional Programming editor

Intentional Programming develops its full potential by making a special editor available. This Intentional Programming editor knows about the active source data structure and the operations associated with it and gives the developer easy access to those. As the active source data model is no longer a simple text as used by traditional programming languages a special editor supporting efficient editing is also a requirement for a user-friendly IP environment.

Editing

Editing with an IP editor is quite different from using a normal text editor. Naturally with a graph data structure in memory developers should not be required to edit program source code as graphs, or with SrcML as pure XML data. Instead it should be possible to directly manipulate the underlying tree data structure based on intentions. As the editing process works with intentions it is unnecessary to require developers to enter their program token by token.



User Input	Editor Window
"class"	`class`
	`public class ??? {`
	`}`
"MyClass"	`public class MyClass {`
	`}`
<tab>	`public class MyClass extends ??? {`
	`}`
<tab>	`public class MyClass implements ??? {`
	`}`
"Serializable"	`import java.io.Serializable;`
	`public class MyClass implements Serializable, ??? {`
	`}`
<tab>	`import java.io.Serializable;`
	`public class MyClass implements Serializable {`
	`}`

Figure 4: Editing with IP editor

A purpose of such an editor is to reduce the amount of necessary typing by providing high level editing capabilities. Figure 4 shows an example for creating a Java class. Note how the editing of tokens no longer happens in the syntactic order as given for traditional Java source code. Instead the order is determined by the intentions of the developer. In this example the developer wants to declare a class, so she types `class`. The IP editor recognizes this as an intention and executes the corresponding operations which insert the necessary nodes into the active source, thus modifying the underlying SrcML document. In this case a `type_decl` node is created for the class. Additionally these operations can take language specific semantics into account which leads to the default `public` modifier being created for the class here. As the class requires a name the IP editor positions the cursor on a predefined field to let the developer enter the name `MyClass` for the class.

After the class name has been entered the developer tells the IP environment that she finished typing that name by hitting the `tab` key. The IP editor then executes another operation of the newly created `type_decl` node to find out what else the developer might want to input. In the example in Figure 4 this operation involves semantics of the Java programming language again by providing two types of inheritance with the corresponding keywords, beginning with the `extends` keyword to declare a baseclass. As the developer does not want to declare any baseclasses another hit of the `tab` key leads to an input possibility for implemented interfaces.

This time the developer enters `Serializable` as an interface to implement in this class. At this point the actions of the IP editor involve semantics again when recognizing this interface and adding the required `import` statement to the beginning of the document. If the interface had methods declared, then the IP editor could create placeholder methods in this class declaration as well. Another hit of the `tab` key notifies the IP editor, that no more interfaces should be implemented, the editor positions the cursor in the main block of the class waiting for further input.

Note that the functionality of this editor is based on active source operations. The addition of a new `type_decl` node into the SrcML document – including the `public` modifier and the class body – is the result of invoking an operation on an existing active source node which in this case is the root node of the document. As mentioned earlier these operations are very important for metaprogramming , as they allow developers to write tools which invoke the same operation to add a class declaration to an active source data structure. Due to the time restriction of this work we have not implemented a complete IP editor, but a developer of such an editor should take special care not to intertwine the core editor functionality and the active source operations, because a proper separation provides maximal metaprogramming capabilities.

So far the IP editor looks just like an improved editor of traditional IDEs , but we have not yet begun to take advantage of Intentional Programming as such. The main benefit we found for both SrcML and IP was the potential for a very high abstraction level. The IP editor can take advantage of these abstractions by providing equally high level ways of displaying them. The example given in the introduction of Intentional Programming already shows how an intention for summing up values in a collection may be shown as Σ. Yet the IP editor goes even further to the point where source code is no longer consisting of mere textual artifacts. In [CE00] the authors bring up the example of a function which calculates a boolean circuit. Using appropriate intentions in the active source data structure the IP editor can display this circuit in a graphical way as well.

In general when adding domain-specific languages to an IP environment the IP editor also allows to adopt the domain's notations . As these special notations are provided by extensions for certain extension points the developer can also choose which one to use. For example the developer might be coming from a background with Java programming and thus prefer to modify the boolean circuit with a Java-like syntax. Nevertheless she can show the circuit to colleagues in the standard notation used for circuits simply by selecting another extension responsible for displaying the active source.

The semantics the editor takes advantage of mentioned earlier are of course language specific. They should therefore be properly separated into extensions and the extension point implementation should preselect the set of extensions depending on the target language of the active source. Furthermore the IP editor can also work without having any semantics available by allowing the developer to create any well-formed document such that it validates against the SrcML schema. For an implementation of an IP editor the language specific extensions can gradually be enhanced thus allowing for a faster adoption of the editor.

Finally it is rather hard for experienced developers to keep track of their editing position in relation to the SrcML DOM tree. This is difficult because of the various different ways the IP editor can display source code. The above example where the input and the textual display look very much like traditional Java is rather out of the ordinary once more sophisticated intentions are in use. This is also reflected by the various possibilities of selecting a part of the active source. As selection refers to the DOM tree there are many more selection types available than the linear selection model of traditional editors. For example a single node, a node and the subtree of which it is the root of, a node and all of its siblings, or a node and all the following siblings are just a few ways to select parts of a DOM tree. We therefore suggest to use the SrcML tree view in parallel to the IP editor to give experienced developers the means to access the underlying data structure directly.

Integrating an IP editor into Eclipse

With Eclipse already providing the architecture required for customizable editors it is possible to add an IP editor as well. Due to the time restriction of this thesis only a small prototype implementation was made, but we provide a short explanation how such an editor could be properly integrated into Eclipse.

The intentions available for the editing can be added to the system through the help of plug-ins with each extension specifying the keywords for its activation. The editor itself can then delegate the actual work to these extensions once the correct intention has been identified. It is important to be able to use an extension without the editor as well to support metaprogramming capabilities . To support undo operations it is useful to apply the command pattern [GE94] to the activation of intentions.

Additionally the usage of SrcML allows the integration with other Eclipse-based SrcML tools. Storing the data used by the IP editor in an XML format even allows the usage of existing XML tools, as opposed to binary formats as proposed by [CE00].

3.4 Applying IP to extend the Java programming language

After discussing Intentional Programming in general we provide an example here for using the IP environment to extend the Java programming language. To keep things simple the intention will only add syntactic sugar to the language. The arithmetic example presented in 2.3 is then modified to make use of the new feature. Section 4 discusses extending the IP environment by a new active source operation and shows how implementations can be provided for new intentions, such that the intention created in this section can be displayed as a control flow graph too.

In the arithmetic example in Listing 3 a chain of if-statements is used, each of which performs a string comparison in its condition. The original intention, however, was to execute different parts of the code depending on the value of a variable. Usually this is achieved by a switch statement, but Java does not support switch statements over a variable of type `String`. This is where we add a new intention which allows switching over strings. Note that we could have reused the existing `switch` intention for this, but instead we show how an extension can freely reuse the names of existing SrcML tags. The new intention is represented by a SrcML tag with the name `switch` as well and is separated from the main SrcML schema through its namespace *http://srcml.de/ext/java* .

In order to be able to use this new intention in a SrcML document nothing more is required. The schema is designed to be freely extensible and so allows for the addition of the new `switch` element into instance documents. Only a few restrictions explained in Section 2.3.4 need to be considered, like the requirement of a different namespace. This results in the SrcML document given in Listing 38 in Appendix D. An excerpt containing the new switch statement is given in Listing 19. The IP environment now “supports” switching over strings.

```
    <ext:switch>
        <expr>
          <identifier name="n" idref="..."/>
        </expr>
        <group>
          <expr>
            <constant value="plus">
              <type name="string"/>
            </constant>
          </expr>
          <block>
    ...
```

Listing 19: new switch intention

Considering that nothing has been implemented so far it is a bit harsh to speak of supporting an intention. Nevertheless the IP system accepts the new element as representing an intention as the documents are validating against the schema. The XML parser requires no further knowledge and creates a DOM which is used as active source by the IP environment. At this point it becomes clear that active source is only useful because of the operations associated with the intention nodes.

Implementing the necessary operations is the main part of providing a new intention. As mentioned earlier though this can be restricted to only implementing the required operations. In this case we want to be able to see the switch statement within Eclipse when viewing the document. This means an extension has to be provided implementing the `IPresentation` interface. There is nothing special about this implementation, as it delegates most of the work to its child nodes. We skip details about how this implementation is added to the system in the form of a plug-in, as this is covered in Section 3.5. When activating this extension and opening the arithmetic example with the new switch statement in Eclipse the code in Listing 20 is shown. This version of the code is already much more readable than the original version.

```
public class Example extends Object implements Cloneable {
    public static void main (String ... args) throws Exception {
        String op = args [ 0 ];
        Integer a = Integer.parseInt (args [ 1 ]);
        Integer b = Integer.parseInt (args [ 2 ]);
        switch (op) {
        case "plus" :
            System.out.println (a + b);
            break;
        case "minus" :
            System.out.println (a - b);
            break;
        case "mul" :
            System.out.println (a * b);
            break;
        case "div" :
            System.out.println (a / b);
            break;
        default :
            System.err.println ("unknown operation");

        }
    }

}
```

Listing 20: new switch intention in Java

Now the active source correctly displays the new intention, although it still remains fairly unusable, as the Java compiler will not accept the source code given in Listing 20 as valid input. At this point we either need an operation which translates the above code to f.ex. Java bytecode or an operation which transforms the above code into equivalent Java source code. As creating bytecode requires a similar operation to be available for all other intentions in use in that document this essentially means implementing a compiler backend. While this is possible to do, the limited time available for this work urged us to choose the latter variant.

The idea for transforming this special switch statement into statements in traditional Java is very simple: we use a `HashMap` to map strings to integers and replace the new switch with a traditional switch statement. The traditional switch statement then switches over the value stored in the `HashMap` for the argument string. To this end the `ITransformation` interface needs to be implemented as the second operation available for the new switch intention. This transformation creates the required `HashMap` and fills it with values for each of the strings used in the switch statement. It then replaces the switch statement by a traditional switch statement as discussed above. When applying this operation to the active source the result looks like Listing 21.

```
package de.srcml.test;
public class Example extends Object implements Cloneable {
 3      private static java.util.Map < String, Integer > __strswitch1;
 4      static {
 5          __strswitch1 = new java.util.HashMap < String, Integer > ();
 6          __strswitch1.put ("plus", 1);
 7          __strswitch1.put ("minus", 2);
 8          __strswitch1.put ("mul", 3);
 9          __strswitch1.put ("div", 4);
10      }
11      public static void main (String ... args) throws Exception {
12          String op = args [ 0 ];
13          Integer a = Integer.parseInt (args [ 1 ]);
14          Integer b = Integer.parseInt (args [ 2 ]);
15          {
16              java.lang.String tmpStr = op;
17              int idx = ((__strswitch1.containsKey (tmpStr)) ?
18                          __strswitch1.get (tmpStr) : 0);
19              switch (idx) {
20              case 1 :
21                  System.out.println (a + b);
22                  break;
23              case 2 :
24                  System.out.println (a - b);
25                  break;
26              case 3 :
27                  System.out.println (a * b);
28                  break;
29              case 4 :
30                  System.out.println (a / b);
31                  break;
32              default :
33                  System.err.println ("unknown␣operation");
34
35              }
36          }
37      }
38
39  }
```

Listing 21: New switch intention in Java after transformation

Listing 21 shows the declared `HashMap` in line 3. The name used for this variable should be chosen according to the semantics of the underlying programming language. As discussed earlier it is advisable to offer such semantics as active source operations to provide for optimal metaprogramming capabilities. This would allow a tool to simply *ask* the node declaring the `Example` class for an unused variable name by calling the appropriate operation. Due to lack of time the transformation implemented for this work contains hardcoded Java semantics at this point to find a free variable name. It is also important to realize, that the developer does not work with the code given in Listing 21. Therefore when the developer adds a new variable – which happens to have the same name as one of the generated variables in this listing – there will not be a problem, as the transformation dynamically determines a new unused variable name.

Lines 4-10 show the static initialization of the `HashMap` with the strings used in the switch statement. This is a straightforward numbering and we implicitly reserve the number `0` for strings which do not have cases associated with them and which are covered by the `default` case of the switch statement. Lines 11-14 remain unchanged and Lines 15-36 contain the transformation to the traditional switch statement. The variable `idx` is assigned the previously stored value from the

HashMap or 0 if the string is not one of the given cases. After that a traditional switch statement is used, which switches over the cases with the numbers previously set up in the initialization of the HashMap.

Finally we have to deal with the case of hash collisions. It is possible, that different strings evaluate to the same position in the HashMap which would result in the wrong code to be executed. This was not taken care of in our implementation due to the lack of time, but this problem is easy to solve: To this end, it is important that we can detect hash collisions during transformation time, as the different strings are all statically available at that time. If a hash collision is detected a similar approach to overflow buckets as used by some hash implementations can be taken. This means, that the colliding strings are mapped to the same integer and in the case block, corresponding to that integer, code is inserted which manually checks the string once more with the help of the equals method. In a worst case scenario where all strings yield the same hash value the transformation will generate code which is almost the same as the original chain of if statements. Nevertheless using this intention helps the developer by displaying code like in Listing 20.

Transforming to intentions which use an equals method is specific to the Java programming language and therefore breaks the language neutrality we want to achieve. This is a prime example where complete language neutrality is not easily achievable. An advantage of the IP environment is the ability to introduce further high-level intentions, which allows us to declare an intention for string comparison which can then be used instead of the equals method. This string comparison intention then needs to be transformed to the equals method for Java. As can be seen in this example IP allows us to raise the abstraction level to a level with which we are comfortable, while still giving us the ability to influence the low-level intentions to the point of creating fine-grained language specific constructs. Extending this example leads to the addition of a kind of pseudo programming language which could be used as the basis for many similar transformations. We have not implemented such a pseudo programming language due to the lack of time. Nevertheless we suggest to investigate some amount of time into its development before creating other transformations, as it helps improving the language neutrality of metaprogramming and other transformations tremendously.

3.5 IP implementation using SrcML

This section discusses the implementation which was created for the theory on Intentional Programming provided in the previous sections. Most notably it consists of integrating the adapter concept into Eclipse to provide for active source operations as explained in Section 3.2. The implementation makes heavy use of the design patterns presented in [GE94] and it is assumed that the reader is familiar with those.

The implementation created for this work contains several parts of an Intentional Programming environment: loading active source, executing active source operations, and editing active source. Loading active source is equivalent to loading the corresponding SrcML representation and can be easily performed by a traditional XML parser. Section 3.5.1 explains the design behind managing and executing the active source operations. The IP editor discussed in Section 3.3 was not fully implemented due to the restricted time of this thesis. Only a small prototype was created which is based on a new SWT widget which was added to Eclipse for the upcoming 3.2 release. This widget is an extended text widget which allows for the seamless inclusion of arbitrary widgets in the text. Therefore the IP editor can easily provide combo boxes with possible selections, text input fields, and arbitrary complex representations of intentions. However the prototype only makes use of combo boxes and text input fields.

The principles of plug-in development in Eclipse are discussed in Section 2.5. At this point it is relevant to know that every presentation of active source is realized as an extension point in Eclipse. Usually this also implies that there is a plug-in responsible for this extension point, although sometimes a plug-in may contain several extension points. The implementations which provide the active source operation are realized as extensions to those extension points. They are generally contained in different plug-ins. Functionality which is specific to a certain programming language can be bundled in a plug-in corresponding to that language. For this work a de.srcml.java plug-in has been created, which contains several extensions. All these extensions contain implementations for operations which are specific to the Java programming language, like parsing Java source code or creating textual presentations of active source.

Separating the plug-ins for extension points and extensions must always be possible as this is the foundation for the extensibility of the IP environment. New plug-ins can easily be added to Eclipse by developers and it should be possible to install new active source operations as well as different implementations for those operations individually. This separation also allows companies to provide their products in logical bundles. For example the extension points could be freely published with their source code to give other developers a better idea of what is expected from the extensions. At the same time the company could sell their own extensions – implementations of the active source operations – as binary plug-ins.

3.5.1 Adapter concept used for active source implementation

The concept of active source operations is implemented using the adapter pattern from [GE94] as outlined earlier. The basic idea of this pattern is to extend existing datatypes with new functionality in cases where the original datatype should not be modified. Therefore an additional class is responsible for adapting the datatype to a given interface. The motivation for these restrictions is given in Section 3.2 and is mainly based on the extensibility of the IP environment.

The objects for which we require adapters are the nodes found in the DOM representation of the active source. This is one of the cases where the modification of the original datatype is infeasible, as the DOM representation is generated by an XML parser. One might argue that a copy of the resulting parse tree could be made with new objects which extend from the original DOM elements while also implementing the required interfaces. This approach was turned down due to limiting the addition of new active source operations. Section 3.2.4 explains the various types of extensibility, but it is clear at this point that the addition of a new active source operation would require modifications of the underlying DOM data structure which makes this approach suboptimal.

The different operations are modelled through interfaces providing the methods which determine the operation. The example mentioned above for generating a textual presentation of the DOM uses the `IPresentation` interface as shown in Listing 15. The remaining objective is finding or creating an object which implements `IPresentation` when given an element of the DOM. This is the point where the adapter pattern comes in: When an element of the DOM should be displayed in a textual form an adapter has to be created for this element, which implements the necessary `IPresentation` interface. The interfaces used for describing the active source operations are important to the declaration of the corresponding extension point. As we are using an interface for each extension point responsible for an active source operation we use the terms interface and extension point synonymously in this context. Similarly the extensions which provide implementations for these extension points contain adapters which implement the required interfaces.

Figure 5 shows an UML sequence diagram describing how an adapter instance is created. The user process depicted in that figure is a process which runs during programming time, but it could also be a process used for metaprogramming. As a textual representation shall be created the user process is talking to the `PresentationManager`. For this example there are two factories which provide the necessary adapter implementations: `FactoryJava` and `FactoryJavaExt`. We are using factories to bundle adapters for several intentions to reduce the search time, because factories are able to check their applicability for a whole range of intentions. To this end we provide every factory with an applicability check to determine if it should be used for a given node. This allows for example a factory which covers all intentions of the traditional Java programming language and for applicability simply checks if the node is in the main SrcML namespace. For this discussion it is irrelevant though, how these factories determine if they are applicable, or what the adapter implementation consists of. The `SrcMLElementAdapterManager` is a special class which behaves similarly to the `AdapterManager` provided by Eclipse in that it is responsible for looking up the correct adapter factory. It is used only because the lookup mechanism provided by Eclipse is unsatisfactory for our purpose as is explained in more detail later. Note that Eclipse is already making use of the adapter pattern in a similar way and hence using its `AdapterManager` is a seamless integration of active source operations into Eclipse. It allows developers to provide adapters required for the Eclipse platform in the same way as active source operations.

During the initial loading process of the system the `PresentationManager` is already registering a set of default factories with the `SrcMLElementAdapterManager`, which in turn registers itself with Eclipse's `AdapterManager`. The set of default factories is stored with the Eclipse plug-in and is automatically adjusted whenever the user chooses another set of factories in the adapters view as described in Section 3.5.2.

Initially the user process invokes a method of the `PresentationManager` to create a presentation for a given SrcML element – this happens for example when a `.srcml` file is opened in the editor. From a developer's point of view this system is very easy to use because of the manager class and from the point of view of the developer of such a manager class the implementation is straightforward as well: Listing 22 shows the `createPresentation` method which is called to initiate this process, and the `present` method's implementation which only consists of a couple of lines. Most of the functionality required is already contained in the base class `AdapterFactoryManager`.

```
public class PresentationManager extends AdapterFactoryManager {
    public PresentationManager() { /**/ }

    public static PresentationManager getDefault() { /**/ }

    public void createPresentation(
            IPresentationDestination dest, Element e) { /**/ }

    public void createPresentation(
            IPresentationDestination dest, Element e,
            List<IAdapterFactory> factories) { /**/ }

    public void present(
            IPresentationDestination dest, Element e) {
        if (e == null) return ;
        // do not visit nodes which have been transformed:
        if (e.attribute("transformed") != null) return ;
        // get the corresponding adapter:
        Object o = Platform.getAdapterManager().getAdapter(
            e, IPresentation.class);
        if (o instanceof IPresentation) {
            fireElementAboutToBePresented(e);
            ((IPresentation)o).present(this, dest);
            fireElementPresented(e);
            return ;
        }
        SrcMLPlugin.logger.fine(
            "No matching present method found for element: " + e);
    }

    public boolean addPresentationListener(
            PresentationListener listener) { /**/ }

    public boolean removePresentationListener(
            PresentationListener listener) { /**/ }

    protected void fireElementAboutToBePresented(
```

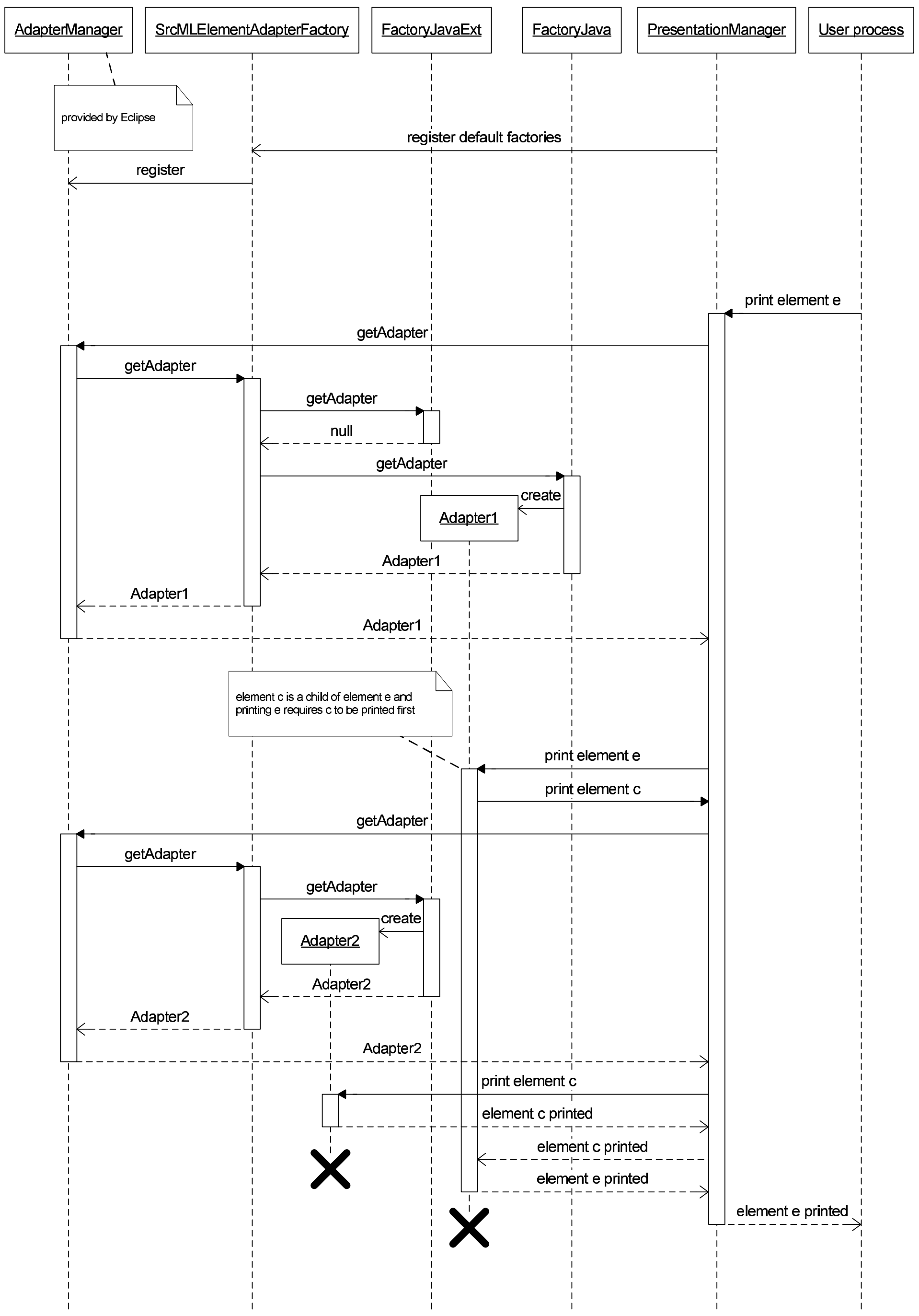


Figure 5: Example of adapter concept

```
38               final Element e) { /**/ }
39
40       protected void fireElementPresented(
41               final Element e) { /**/ }
42
43       public void load() { /**/ }
44
45       public void save() { /**/ }
46   }
```

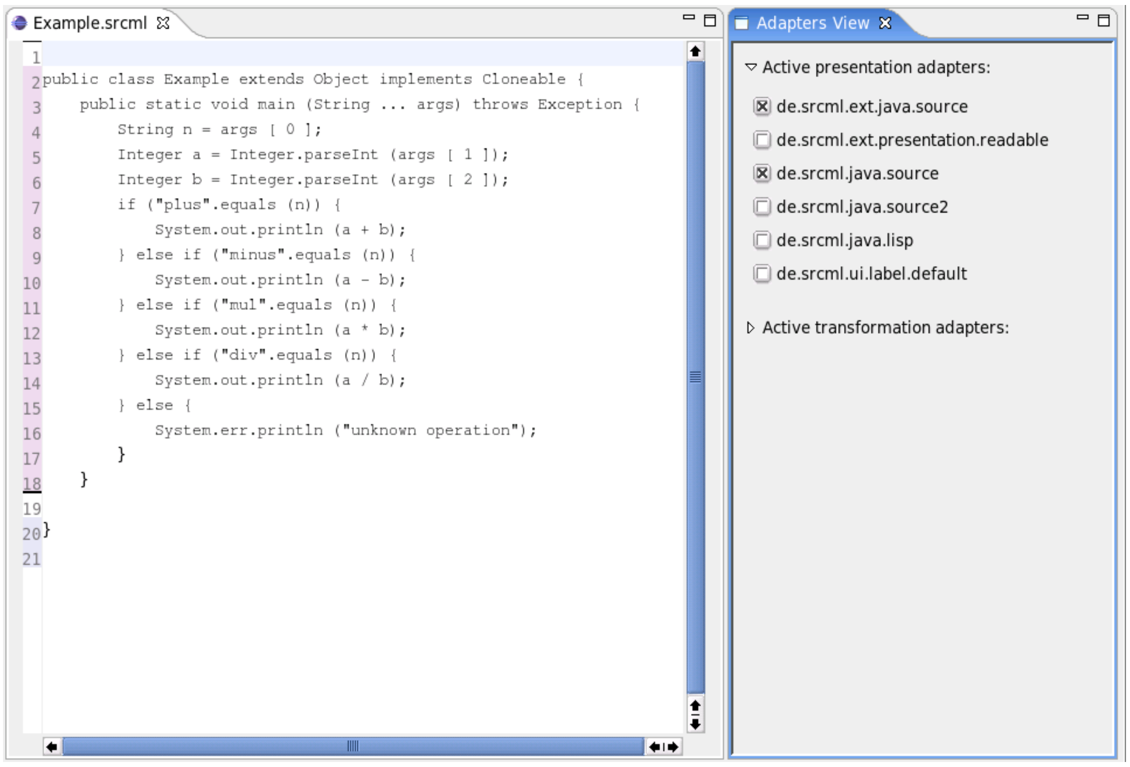
Listing 22: PresentationManager class

After the call to the `createPresentation` method – which in turn calls the `present` method – the `AdapterManager`'s `getAdapter` method is invoked. This implies a request for an adapter implementation for the `IPresentation` interface as specified in Listing 22 lines 19-20. Next the registered factories are checked for their applicability. Unfortunately the `AdapterManager` provided by Eclipse only accepts one factory for each interface. This is the reason why the `SrcMLElementAdapterManager` was introduced, as we specifically want to provide multiple factories. Therefore it allows multiple factories to be registered for an interface and checks them in reverse registration order for their applicability. In this example it would be unsuitable to require a single factory to provide adapter implementations for all possible SrcML elements. Especially when considering the extensibility of the format this cannot suffice. For example consider the previously mentioned factory for all nodes representing intentions of the traditional Java programming language and add the new switch statement created in Section 3.4. As we want the factory responsible for Java to remain unmodified we can simply provide an additional factory for our extension. With multiple factories registered an applicability check should be included in each factory to quickly check if the element to adapt to is one of its supported elements. As mentioned earlier this improves the search time when multiple adapters are handled by a single factory which is usually the case when providing extension for sets of domain-specific intentions.

Next the `getAdapter` call leads to a similar method in the `SrcMLElementAdapterManager` class, which in turn calls this method on all factory classes registered for the requested interface as just mentioned. Figure 5 shows that the first such call involves `FactoryJavaExt` which returns `null`. Therefore the factory failed the applicability check and is not suitable to provide an adapter implementation for the given element and the next registered factory has to be checked. In the case of `FactoryJava` the applicability test succeeds and an adapter implementation is instantiated which is handed through to the `PresentationManager`.

The `PresentationManager` uses a method provided by the adapter to finally create the textual representation of the element it adapts to. As Figure 5 shows the adapter might in turn require textual presentations from the child nodes of its element. This leads to another call of the `PresentationManager`'s `present` method which triggers another round of searching for an applicable adapter implementation. Figure 5 shows how this time another factory gets used to provide the adapter implementation.

The execution time required for this process may seem very high at first. We have to keep in mind though, that the problem involves a high amount of extensibility as well. This approach is guaranteed to work for any number of extensions. It should also be noted, that one has to be careful with ideas like caching adapters. The applicability check of an adapter factory is of a very dynamic nature. There is no guarantee that a node results in the same adapter instance when the adapter is requested a second time, as the DOM tree might have changed in such a way that the corresponding factory is no longer applicable. It could also happen that a factory which in the previous request was not applicable becomes applicable due to the change of the DOM tree and should therefore be preferred. This also hinders us from caching the adapters and simply evaluating their factory's applicability once more. There was not enough time to further investigate more sophisticated caching methods considering that the execution of our implementation performed alright.



```

public class Example extends Object implements Cloneable {
    public static void main (String ... args) throws Exception {
        String n = args [ 0 ];
        Integer a = Integer.parseInt (args [ 1 ]);
        Integer b = Integer.parseInt (args [ 2 ]);
        if ("plus".equals (n)) {
            System.out.println (a + b);
        } else if ("minus".equals (n)) {
            System.out.println (a - b);
        } else if ("mul".equals (n)) {
            System.out.println (a * b);
        } else if ("div".equals (n)) {
            System.out.println (a / b);
        } else {
            System.err.println ("unknown operation");
        }
    }

}

```

Active presentation adapters:

- [✓] de.srcml.ext.java.source
- [ ] de.srcml.ext.presentation.readable
- [✓] de.srcml.java.source
- [ ] de.srcml.java.source2
- [ ] de.srcml.java.lisp
- [ ] de.srcml.ui.label.default

Active transformation adapters:

Figure 6: Screenshot of adapters view

3.5.2 Adapters view

The order in which the `SrcMLElementAdapterManager` tests the registered factories and the set of factories to test determine the semantics of evaluating an active source operation. Therefore a developer should be able to influence this process. To this end the implementation contains an additional view called the adapters view which is shown in Figure 6. This view provides a list of the available active source operations and their respective extensions. This means that a new plug-in providing an extension for an existing operation is automatically added to the list. In fact a single plug-in can be seen in the list multiple times depending on how many extensions it provides. The extensions all have unique identifiers which are used for the display in the adapters view. It is also possible to extend the declaration of extension points which are used for modeling active source operations such that the adapters view can automatically determine the available operations.

The adapters view allows the developer to turn specific extensions on or off. Every time an extension is turned on its factory is registered at the `SrcMLElementAdapterManager`. This has two effects: First it is used as a default factory. So all parts of the code which do not request explicit extensions will use the selected extension by default. This is relevant for example in the editor which displays the active source as text. It is not depending on a specific extension to be present, but will happily use whichever extension provides an applicable factory. The second effect is that the factories are checked in reverse order of registration. Therefore a newly selected extension will be given the highest priority thus allowing the developer to determine the order in which the factories are checked. It is possible to reflect this order in the order of the appearance of the extensions in the adapters view, which was not implemented due to the restricted time for this work.

The adapters view implementation is further based on the observer design pattern . This allows other components to register listeners which are notified whenever the selection changes. One such component is the editor which, after being notified of a change, regenerates its textual representation using the latest selection of extensions. Therefore developers can quickly see different presentations of their active source. The current editor implementation also takes activated transformation extensions into account, thus the developer can also see how intentions are transformed to the base language.

With the help of the adapters view the listings in Section 3.4 can be gained by a few mouse clicks. To see the new switch statement in the editor the corresponding extension has to be activated. In order to see the transformed code including the `HashMap` it is sufficient to simply activate the respective extension providing the transformation implementation. Note that in the current implementation it is also possible to hide parts or all of the active source by deselecting all extensions which provide the necessary `IPresentation` adapters. Another restriction is that the displaying of transformed active source is only for convenience. No changes should be made to the active source while it is in a transformed state, as those changes might be lost when the active source is reverted back to its normal state as discussed in the following section.

3.5.3 Transformations

The transformation extensions which implement `ITransformation` shown in Listing 16 are used for implementing domain-specific intentions. As discussed above one way of implementing domain-specific languages is transforming them back to equivalent code in a base language, which is the purpose of the `ITransformation` interface.

From the point of view of a tool the transformation can be tried when no adapter is available for an intention in the hope that there may be a transformation to other intentions for which the required adapters are available. For example an active source operation which compiles an intention might not be available for the new switch statement developed in 3.4, although after the transformation to the traditional Java switch statement the necessary adapters might be present. Developers have to be careful when performing transformations, to ensure that the results use intentions with a lower abstraction level. As newly added intentions can be transformed again this could result in a possible termination problem otherwise.

These transformations are restricted to only perform additions to the active source. They should not make changes, or delete parts of the active source. If such a behavior is wanted a separate active source operation should be created. With this restriction the transformations offer the possibility of reusing the existing DOM. After transforming an intention the corresponding node is marked with a `transformed` attribute. Similarly the newly added nodes are marked with an `autogenerated` attribute. For complete subtrees it is sufficient to mark the root node.

These marks have the advantage, that the unaffected parts of the active source need not to be copied to create a transformed version. The process of recovering the untransformed active source is also simplified: It is sufficient to remove all subtrees with nodes which are marked with the `autogenerated` attribute and remove the `transformed` attribute from nodes. This approach resembles that the transformations should only be used to create intermediate representations of the active source.

Developers have to be careful when working with such intermediate representations, because the nodes with a `transformed` attribute should be considered deleted. They have already been replaced by equivalent intentions and should not be processed as long as the active source remains in this intermediate representation. Convenience methods are available to clear the DOM tree of the artifacts of these transformations after work on the intermediate representation is finished.

Macros. At first one might mistake these transformations for traditional macros, as found in C++, providing a simple textual replacement, but these transformations always have the complete DOM available for deriving necessary information. Therefore a transformation result generally depends on the state of the DOM as opposed to a macro which always performs the same replacement. Furthermore transformations are free to add new nodes at any number of places in the DOM whereas traditional macros only perform a local replacement.

The previously discussed transformation for the new switch statement for example works at three different places in the DOM: In one place the variable for the `HashMap` is added. In another place the static initializer block is added and additionally a local replacement is made. Furthermore finding the free variable names depends on the state of the DOM at the execution time of the transformation.

3.6 Example Lisp-style presentation of Java source code

As a proof of concept the implementation we created for this work contains different extensions for the `IPresentation` interface. The previously mentioned extension creates a traditional Java source code representation. To show how the same active sources can be displayed in a very different way we also provide an extension which represents the active source in a Lisp-like syntax. While this may be an extreme difference it does show the potential of different representations of the same source.

```
public class Example extends Object implements Cloneable {
    public static void main (String ... args) throws Exception {
        String op = args [ 0 ];
        Integer a = Integer.parseInt (args [ 1 ]);
        Integer b = Integer.parseInt (args [ 2 ]);
        if ("plus".equals (op)) {
            System.out.println (a + b);
        } else if ("minus".equals (op)) {
            System.out.println (a - b);
        } else if ("mul".equals (op)) {
            System.out.println (a * b);
        } else if ("div".equals (op)) {
            System.out.println (a / b);
        } else {
            System.err.println ("unknown operation");
        }
    }

}
```

Listing 23: arithmetic example in Java

Reconsider the Java source code for our arithmetic example as given in Listing 23. This listing is the output from the first extension. The result of using the Lisp-like syntax when activated through the adapters view is shown in Listing 24. Note that due to a lack of time we have not polished the Lisp extension as well as the Java extension and thus it might not display all intentions.

```
((
(class Example (public) extends (Object) implements (Cloneable)
    (fun main (public static) (void) (var (String) (args ...)) (
        (var (String) (op (= ((args) ((0))))))
        (var (Integer) (a (= (parseInt (Integer) (((args) ((1))))))))
        (var (Integer) (b (= (parseInt (Integer) (((args) ((2))))))))
        (if (equals (("plus")) (op)) (
            (println (System.out) ((+ a b)))

        ) (equals (("minus")) (op)) (
            (println (System.out) ((- a b)))

        ) (equals (("mul")) (op)) (
            (println (System.out) ((* a b)))

        ) (equals (("div")) (op)) (
            (println (System.out) ((/ a b)))

        ) (
            (println (System.err) (("unknown operation")))

        ))
    ))

)))
```

Listing 24: arithmetic example in Java using Lisp-like syntax

This example effectively shows how the same active source can have very different representations. Note that this example only uses very low-level intentions as well – namely those provided by the Java programming language. Due to those low-level intentions it is still possible to tell that the code in Listing 24 resembles source code for the Java programming language. High-level intentions would provide far more unusual representations which makes it very hard to determine what programming language the active source is written in. In fact one might argue that such an active source is not written in a traditional programming language anymore as Simonyi does in [Sim95].

4 Control Flow Graphs (CFG)

After discussing the SrcML format in Section 2 and Intentional Programming in Section 3 this section presents a comprehensive application. We extended the IP environment with a means to create control flow graphs from active source, whereas the definition of a control flow graph is given in due time. Section 4.1 details the construction process of these graphs in the context of Intentional Programming, Section 4.2 provides a formal basis and correctness proof for the construction algorithm, and Section 4.3 details aspects of our implementation of the algorithm and its integration into Eclipse and our IP environment. Finally Section 4.4 applies the construction of control flow graphs to the arithmetic example and its extended version introduced in Section 3.4. This section assumes that the reader has a basic working knowledge of graph theory. For an introduction to graph theory see [CLR89] or [Sch01]. For the terminology used throughout this section a graph consists of *vertices* and *edges* whereas *vertex* and *node* are used interchangeably.

A *control flow graph* (CFG) is a directed graph representing the possible flows of control in a part of the source code, with a vertex for each primitive statement according to [WM97]. It contains labeled edges from one vertex to another if and only if the corresponding statements can be executed directly after each other. For statements, like a typical if statement, which have multiple other directly reachable statements the edge labels are used for distinction. Therefore the two edges going out of the node representing the conditional expression of an if statement might be labeled with *TRUE* and *FALSE*, like in the example CFG in Figure 7. Every CFG further contains a designated ENTRY vertex where the control flow begins and a corresponding EXIT vertex. A more formal definition of a control flow graph – without the notion of edge labels as they do not affect any of the following theorems – is given in Section 4.2.

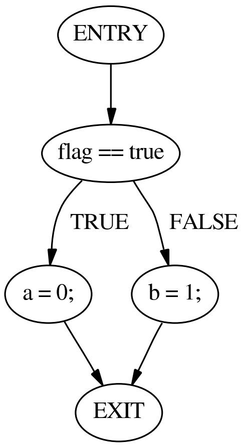


Figure 7: Example of a control flow graph

Control flow graphs have a wide number of applications in compiler design and source code analyses. The most important usages are in optimizations where CFGs help with loop detection, unreachable code detection, register allocation, or instruction alignment [WM97, Muc97]. Control flow graphs are also often used in the construction of other graphs tailored more specifically to the required needs, like data flow graphs or program-dependence graphs [Muc97].

4.1 Control flow graphs in an extensible environment

The construction of control flow graphs is usually very straightforward once an abstract syntax tree of the corresponding source code is available. In the case of SrcML and IP the intrinsic ability of extending the language makes it harder to implement a CFG construction for the following reasons:

- SrcML documents only using elements from the main SrcML schema can already contain statements from three different languages.
  If a specific DSL is considered the construction algorithm must also be able to create a proper control flow graph for the DSL's added statements.
- If another intention is created it has not been known at the time of the development of the CFG construction algorithm, yet such an algorithm should easily be extensible to also create a control flow graph for any new intentions.
- SrcML documents may contain metadata which has no influence at all on the resulting control flow graph, but a construction algorithm must be able to distinguish such metadata from statement intentions which have to be shown in the CFG.

The adapter design pattern described in Section 3.2 can be used very effectively to solve these problems. Each time a new language is developed and thereby new elements get added to SrcML documents, an adapter for an `ICFGCreation` interface, as shown in listing 25, should be supplied.

```
public interface ICFGCreation {
    public void createEdges(CFGCreationManager mgr);
    public CFG createGraph(CFGCreationManager mgr);
    public boolean hasCFG();
}
```

Listing 25: ICFGCreation interface

The `hasCFG` method allows the construction algorithm to distinguish metadata from elements which are to be represented in the CFG. As the generic construction algorithm cannot possibly know how to arrange the vertices in the graph and what edges to create when a new intention gets added it is left to the adapter implementation to create a subgraph with `createGraph`, which can then be used by the main algorithm. The `CFGCreationManager` is necessary, because the very same problems described above also apply during the construction of the subgraph. With the help of the `CFGCreationManager` the adapter implementation can delegate the task of creating a subgraph for its required elements. This construction process is a kind of recursion where the recursive function is dynamically determined each time from a set of functions provided by several adapter classes.

Note that termination of this process is not guaranteed, as the adapter pattern also allows an adapter class to be used in which the `createGraph` method consists of an endless loop. If all adapter classes behave as intended though, then they will only delegate further subgraph constructions for their child elements which is a terminating process due to the SrcML tree being finite.

After a subgraph has been recursively created for an element and is used in the graph currently being constructed, this graph will temporarily be consisting of vertices which can be complete graphs themselves. So there is an additional step necessary, in which this hierarchy of graphs is flattened into a graph which no longer contains any graph vertices. This final graph will then be the requested control flow graph for that specific part of the source code.

As control flow graphs are intended to contain a designated ENTRY and EXIT node these nodes will also appear in all of the subgraphs. During the flattening of these graphs new graphs are created which contain multiple ENTRY and EXIT nodes, so that the flattening procedure will also have to take care, that ENTRY and EXIT nodes in subgraphs get correctly replaced. How this is done is shown in detail in Section 4.2.

This process involves some more problems in the construction of the graph's edges:

- for a jump instruction the target node, which by construction is another graph, may not have been created by the time the `createGraph` method for the element containing the jump instruction is called.

- Some jumps also require edges going from a vertex in the currently created subgraph to a vertex in another graph.

The first problem is solved straightforward by ensuring that all graphs have been created, before edges for jump instructions are added. The `CFGCreationManager` therefore allows adapter classes to queue themselves if they require to add such a jump edge. After all graphs have been recursively created the `createEdges` method will be called for all queued adapter classes thus eliminating the first problem.

From a formal point of view edges crossing graph boundaries are a bad idea. Even defining a new graph in which those edges could be drawn gets complicated by the fact that the adjacent vertices may reside in different hierarchical subgraph levels. One possible solution which solves this problem in an intuitive and non-intrusive way is the addition of so-called `ReplacementNode` instances. Each time an edge crossing graph boundaries should be created a `ReplacementNode` is instead inserted into the destination graph with an edge from that node to the target vertex of the intended edge. Additionally a `ReplacementNode` contains information about the node it is going to be replaced with which is the source vertex of the intended edge. During the above mentioned flattening process it will then be checked if a `ReplacementNode` and the vertex it is to be replaced with reside in the same graph. If this is the case all edges adjacent to the `ReplacementNode` will be moved to the node it is replaced with and the `ReplacementNode` is removed from the graph.

4.2 Formal examination of CFG construction using IP

In this section a formal definition of control flow graphs and the functions required for their construction will be given. The flattening algorithm introduced in the previous section will then be properly defined and proved to correctly create control flow graphs under the assumption, that the implementation provided by each of the adapter classes is correct.

As described above, the algorithm which constructs the control flow graph consists of creating several other subgraphs first. Those subgraphs should not be in the final CFG, but for the construction process they will have to be considered. The same holds true for the `ReplacementNode` instances. To this end, the notion of a temporary control flow graph is introduced in Definition 4.1 for the graphs which can be created during the construction process.

Definition 4.1. For a SrcML document represented as a DOM tree $S = (V_s, E_s)$ a *temporary control flow graph* (or *TCFG*) $G = (V, E, v_I, v_O)$ consists of *vertices* (or *nodes*) V, *edges* E, and two special nodes $v_I, v_O \in V - V_s$ with neither of them being a `ReplacementNode` or a TCFG. A function *nodes* which maps a TCFG to the set of all vertices contained in it is defined as well.

The set $\mathbb{TCFG}$ of all TCFGs and the function *nodes* are inductively defined as follows:

$$G_0 = (\{v_I, v_O\}, \emptyset, v_I, v_O) \in \mathbb{TCFG} \qquad (4.2.1)$$
$$nodes(G_0) = \{v_I, v_O\}$$

$$G = (V, E, v_I, v_O) \in \mathbb{TCFG} \Rightarrow H = (V, E', v_I, v_O) \in \mathbb{TCFG} \qquad (4.2.2)$$
$$\text{with } E \subseteq E' \subseteq (V \times V) - \{(v_I, v_I), (v_O, v_O)\}$$
$$nodes(H) = nodes(G)$$

$$G = (V, E, v_I, v_O) \in \mathbb{TCFG} \Rightarrow H = (V \cup \{v\}, E, v_I, v_O) \in \mathbb{TCFG} \qquad (4.2.3)$$
$$\text{for } v \in V_s - nodes(G)$$
$$nodes(H) = nodes(G) \cup \{v\}$$

$$G = (V, E, v_I, v_O) \in \mathbb{TCFG} \Rightarrow H = (V \cup \{v_r^e\}, E, v_I, v_O) \in \mathbb{TCFG} \qquad (4.2.4)$$
$$\text{for a } \texttt{ReplacementNode}\ v_r^e \text{ replacing a node } v_e \in V_s$$
$$nodes(H) = nodes(G) \cup \{v_r^e\}$$

$$G = (V, E, v_I, v_O) \in \mathbb{TCFG} \text{ and } H = (V', E', v_I', v_O') \in \mathbb{TCFG} \qquad (4.2.5)$$
$$\text{with } nodes(G) \cap nodes(H) = \emptyset \Rightarrow I = (V \cup \{H\}, E, v_I, v_O) \in \mathbb{TCFG}$$
$$nodes(I) = nodes(G) \cup nodes(H) \cup \{H\}$$

Remark 4.2. The set $nodes(G)$ is usually a superset of the nodes in V. When considering an if statement and how a control flow graph for it is created it can be seen that there will be three different subgraphs for the condition, the then-, and else-block. Each of these three graphs is found in $nodes(G)$ as well as in V, but all nodes found in one of those blocks for example are also contained in $nodes(G)$ because of (4.2.5).

Remark 4.3. The exclusion of the edges (v_I, v_I) and (v_O, v_O) is to avoid loops which do not involve actual statements from this part of the source code. This does not yet rule out loops of the form $(v_I, v_O), (v_O, v_I)$ which will be considered valid for the purpose of the construction of the CFG. In Definition 4.6 such loops will then be disallowed for the control flow graph and will be removed during the flattening process.

Definition 4.4. The following sets can be defined for a SrcML document S:

- $\mathbb{V} := \bigcup\limits_{(V,E,v_I,v_O)\in\mathbb{TCFG}} V$
- $\mathbb{V}_r := \{v_r^e \in \mathbb{V} \mid v_r^e$ is a `ReplacementNode` replacing a node $v_e \in V_s\}$

Corollary 4.5. *Every 4-tuple $G = (V, E, v_I, v_O)$ for a SrcML document $S = (V_s, E_s)$ which adheres to the following conditions is a TCFG:*

- $V = \{v_I, v_O\} \uplus \tilde{V}_s \uplus \tilde{V}_r \uplus \tilde{V}_{TCFG}$
  - $\tilde{V}_s \subseteq V_s$
  - $\tilde{V}_r = \{v_r^e \mid v_r^e$ *is a* `ReplacementNode` *replacing a node* $v_e \in V_s\}$
  - $\tilde{V}_{TCFG} \subseteq \mathbb{TCFG}$
- $E \subseteq (V \times V) - \{(v_I, v_I), (v_O, v_O)\}$

Proof. G can be constructed using the inductive definition of a TCFG starting with G_0 from (4.2.1) and successively adding all nodes from $\tilde{V}_s$ (4.2.3), $\tilde{V}_r$ (4.2.4), and $\tilde{V}_{\mathrm{TCFG}}$ (4.2.5). Finally the edge set is added as described in (4.2.2). According to Definition 4.1 each of those steps results in a valid TCFG again. □

After defining the algorithm's domain $\mathbb{TCFG}$, the set $\mathbb{CFG}$ of control flow graphs which will be the algorithm's range can now be defined:

Definition 4.6. For a SrcML document represented as a DOM tree $S = (V_s, E_s)$ a *control flow graph* (or *CFG*) $G = (V, E, v_I, v_O)$ consists of *vertices* (or *nodes*) V, *edges* E, and two special nodes $v_I, v_O \in V - V_s$ with neither of them being a `ReplacementNode` or a TCFG. The set $\mathbb{CFG}$ of CFGs is inductively defined as follows:

- $G_0 = (\{v_I, v_O\}, \emptyset, v_I, v_O) \in \mathbb{CFG}$
- $G = (V, E, v_I, v_O) \in \mathbb{CFG} \Rightarrow H = (V, E', v_I, v_O) \in \mathbb{CFG}$
  with $E \subseteq E' \subseteq V \times V$ if the following conditions hold for E':
  - $E' \cap (V \times \{v_I\}) = \emptyset$
  - $E' \cap (\{v_O\} \times V) = \emptyset$
- $G = (V, E, v_I, v_O) \in \mathbb{CFG} \Rightarrow H = (V \cup \{v\}, E, v_I, v_O) \in \mathbb{CFG}$ for $v \in V_s$

Remark 4.7. The conditions given in Definition 4.6 are based on the following observations:

- As the special vertex v_O is not associated with a statement of the represented program it does not make sense for it to have outgoing edges. Similarly the vertex v_I should not have any incoming edges, as the control flow is always going to another statement and not to a placeholder vertex like v_I – unless it's the end of the control flow implying an edge going to v_O.
- The vertex v_I is a special marker for the beginning of the control flow and thus must only have one outgoing edge, as it is not representing a branching statement. For the purpose of easier construction of CFGs graphs in which the out-degree of v_I is zero are allowed. This effectively is a dead-end for the control flow, but nevertheless other nodes in such a graph could be reached by jump statements.
- The restriction of E' in Definition 4.6 implies that $\{(v_I, v_I), (v_O, v_O)\} \cap E' = \emptyset$.
- Every CFG is also a TCFG, but not vice versa. Apart from the stronger restrictions on the edge set the difference is that CFGs may not contain `ReplacementNode` instances or TCFGs.

Definition 4.8. For the process of transforming a TCFG into a CFG the following functions are used:

- $\mathrm{TC}_I : \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v, (V, E, v_I, v_O)) \mapsto (V, E_1 - E_2, v_I, v_O)$
  - $E_1 = E \cup \{(v_i, v_j) \mid v_i, v_j \in V : v \neq v_i \wedge v \neq v_j \wedge (v_i, v) \in E \wedge (v, v_j) \in E\}$
  - $E_2 = (V \times \{v\}) \cup \{(v_I, v_I), (v_O, v_O)\}$
- $\mathrm{TC}_O : \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v, (V, E, v_I, v_O)) \mapsto (V, E_1 - E_2, v_I, v_O)$
  - $E_1 = E \cup \{(v_i, v_j) \mid v_i, v_j \in V : v \neq v_i \wedge v \neq v_j \wedge (v_i, v) \in E \wedge (v, v_j) \in E\}$
  - $E_2 = (\{v\} \times V) \cup \{(v_I, v_I), (v_O, v_O)\}$
- $\mathrm{REP} : \mathbb{V}_r \to \mathbb{TCFG} \to \mathbb{TCFG}, (v_r^e, (V, E, v_I, v_O)) \mapsto (V', E', v_I, v_O)$
  for a `ReplacementNode` v_r^e replacing a node $v_e \in V_s$.
  - if $v_e \in V$:
    * $V' = V - \{v_r^e\}$
    * $E' = E_1 - E_2$
      $E_1 = E \cup \{(v_e, v_i) \mid (v_r^e, v_i) \in E\} \cup \{(v_i, v_e) \mid (v_i, v_r^e) \in E\}$
      $E_2 = \{(v_i, v_j) \mid v_i \in V \wedge v_j \in V \wedge (v_i = v_r \vee v_j = v_r)\} \cup \{(v_I, v_I), (v_O, v_O)\}$

- if $v_e \notin V$:
    - $V' = V$
    - $E' = E$

- $\text{REDIR}_I : \mathbb{V} \to \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v_{\text{src}}, v_{\text{dst}}, (V, E, v_I, v_O)) \mapsto (V, E_1 - E_2, v_I, v_O)$
    - $E_1 = E \cup \{(v_i, v_{\text{dst}}) \mid (v_i, v_{\text{src}}) \in E\}$
    - $E_2 = (V \times \{v_{\text{src}}\}) \cup \{(v_I, v_I), (v_O, v_O)\}$

- $\text{REDIR}_O : \mathbb{V} \to \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v_{\text{src}}, v_{\text{dst}}, (V, E, v_I, v_O)) \mapsto (V, E_1 - E_2, v_I, v_O)$
    - $E_1 = E \cup \{(v_{\text{dst}}, v_i) \mid (v_{\text{src}}, v_i) \in E\}$
    - $E_2 = (\{v_{\text{src}}\} \times V) \cup \{(v_I, v_I), (v_O, v_O)\}$

- $\text{COLLAPSE} : \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v, (V, E, v_I, v_O)) \mapsto (V', E', v_I, v_O)$
    - if $v \in \{v_I, v_O\}$ then $V' = V$ and $E' = E$
    - otherwise: $V' = V - \{v\}$ and $E' = E_1 - E_2$
      $E_1 = E \cup \{(v_i, v_j) \mid (v_i, v) \in E \wedge (v, v_j) \in E\}$
      $E_2 = (V \times \{v\}) \cup (\{v\} \times V) \cup \{(v_I, v_I), (v_O, v_O)\}$

- $\text{REMOVE} : \mathbb{V} \to \mathbb{TCFG} \to \mathbb{TCFG}, (v, (V, E, v_I, v_O)) \mapsto (V', E', v_I, v_O)$
    - if $v \in \{v_I, v_O\}$ then $V' = V$ and $E' = E$
    - otherwise: $V' = V - \{v\}$ and $E' = E - ((V \times \{v\}) \cup (\{v\} \times V))$

- $\text{INCLUDE} : \mathbb{TCFG} \to \mathbb{TCFG} \to \mathbb{TCFG}, ((V^*, E^*, v_I^*, v_O^*), (V, E, v_I, v_O)) \mapsto (V \cup V^*, E \cup E^*, v_I, v_O)$

Definition 4.9. The above functions now allow the function

$$\text{MERGE} : \mathbb{TCFG} \to \mathbb{TCFG} \to \mathbb{TCFG}, (G, H) \mapsto I$$

to be defined with $G = (V, E, v_I, v_O), H = (V', E', v_I', v_O'), I = (V^*, E^*, v_I, v_O)$ as:
$\text{MERGE}(G, H) =$
$\quad \text{COLLAPSE}(v_O') \circ \text{COLLAPSE}(v_I') \circ \text{REMOVE}(H) \circ$
$\quad \text{REDIR}_O(H, v_O') \circ \text{REDIR}_I(H, v_I') \circ \text{INCLUDE}(H, G)$

Remark 4.10. Definition 4.9 and the definitions of the functions used in MERGE make use of a technique called *currying* known from functional programming [WM97] used here for improved readability.

Definition 4.11. A function which converts a TCFG into a CFG can now be defined:

$$\text{FLATTEN} : \mathbb{TCFG} \to \mathbb{CFG}, G \mapsto H$$

FLATTEN is defined through the following recursive algorithm for $G = (V, E, v_I, v_O)$:

- FOREACH I IN $(V \cap \mathbb{TCFG})$ DO $I \leftarrow \text{FLATTEN}(I)$
- FOREACH I IN $(V \cap \mathbb{TCFG})$ DO $G \leftarrow \text{MERGE}(G, I)$
- FOREACH v_r^e IN $(V \cap \mathbb{V}_r)$ DO $G \leftarrow \text{REP}(v_r^e, G)$
- $G \leftarrow \text{TC}_I(v_I, G)$
- $G \leftarrow \text{TC}_O(v_O, G)$

Assuming that the construction of the recursive subgraphs has been successful the final control flow graph will be given by applying the FLATTEN function to the main graph which is still a TCFG at this point. The following lemmas and theorems will therefore have to prove now that the function FLATTEN does indeed return a CFG according to Definition 4.6.

Lemma 4.12. *For every TCFG $G = (V, E, v_I, v_O), H = (V', E', v'_I, v'_O)$, and $I = (V^*, E^*, v_I, v_O) = MERGE(G, H)$ it is guaranteed that $H \notin V^*$.*

Proof. The results of the function calls $\text{REDIR}_O(H, v'_O)$, $\text{REDIR}_I(H, v'_I)$, and $\text{INCLUDE}(G, H)$ will be ignored. Instead it can be seen that the function $\text{REMOVE}(H)$ will remove the vertex H from the vertex set of the resulting TCFG, as $H \notin \{v_I, v_O\}$ for any TCFG. It remains to be shown, that a COLLAPSE function cannot reintroduce H into the result TCFG: As the definition of COLLAPSE only changes the vertex set of a TCFG by either removing the vertices v'_I, v'_O or not changing it at all, this will never be the case. □

Lemma 4.13. *For every TCFG $G = (V, E, v_I, v_O), H = (V', E', v'_I, v'_O)$, and $I = (V^*, E^*, v_I, v_O) = MERGE(G, H)$ it is guaranteed that $\forall J \in (V' \cap \mathbb{TCFG}) : J \in V^*$.*

Proof. First $\text{INCLUDE}(G, H)$ creates a new vertex set $\tilde{V} = V \cup V'$ and thus $\forall J \in (V' \cap \mathbb{TCFG}) : J \in \tilde{V}$. It remains to be shown, that none of the included TCFGs J gets removed by the remaining function calls:
The REDIR functions leave the vertex set unmodified, so that no TCFG can be removed.
The REMOVE call removes only the vertex given as its first argument and thus removes the vertex H. However this does not remove the TCFGs J which have been copied from V' into $\tilde{V}$ because $\forall J \in (V' \cap \mathbb{TCFG}) : J \neq H$, as a TCFG cannot include itself by (4.2.5).
Analogously the COLLAPSE function only removes the vertex given as its first argument. So only vertices v_I and v_O get removed which are no TCFGs and thus $\forall J \in (V \cap \mathbb{TCFG}) : J \notin \{v_I, v_O\}$. □

Definition 4.14. Let the nesting depth of a TCFG $ndepth : \mathbb{TCFG} \to \mathbb{N}$ be inductively defined – using the notations from Definition 4.1 – as:

- (4.2.1): $ndepth(G_0) = 0$
- (4.2.2), (4.2.3), (4.2.4): $ndepth(H) = ndepth(G)$
- (4.2.5): $ndepth(I) = \max\{ndepth(G), ndepth(H) + 1\}$

Remark 4.15. $ndepth(G) = \max(\{0\} \cup \{ndepth(H) + 1 \mid H \in (V \cap \mathbb{TCFG})\})$

Lemma 4.16. *Let $G = (V, E, v_I, v_O)$ and $H = (V', E', v'_I, v'_O)$ be TCFGs with $\#(V \cap \mathbb{TCFG}) > 0$. After FOREACH I IN $(V \cap \mathbb{TCFG})$ DO $H \leftarrow MERGE(H, I)$ is executed with initially $H = G$ it holds that $ndepth(H) < ndepth(G)$*

Proof.

$$\begin{aligned} ndepth(G) &\overset{\text{Rem. 4.15}}{=} \max(\{0\} \cup \{ndepth(I) + 1 \mid I \in (V \cap \mathbb{TCFG})\}) \\ &\overset{\#(V\cap\mathbb{TCFG})>0}{=} \max\{ndepth(I) + 1 \mid I \in (V \cap \mathbb{TCFG})\} \end{aligned}$$

Let $\hat{I} = (\hat{V}, \hat{E}, \hat{v_I}, \hat{v_O})$ be any TCFG for which $ndepth(G) = ndepth(\hat{I}) + 1$. With Remark 4.15, it follows that $\forall J \in (\hat{V} \cap \mathbb{TCFG}) : ndepth(J) < ndepth(\hat{I})$. After the execution of the MERGE functions Lemma 4.12 implies that $\hat{I} \notin V'$ and Lemma 4.13 implies that $\forall J \in (\hat{V} \cap \mathbb{TCFG}) : J \in V'$. As thus V' no longer contains $\hat{I}$ or any other TCFG K with $ndepth(K) = ndepth(\hat{I})$ it follows that:

$$\begin{aligned} ndepth(H) &= \max(\{0\} \cup \{ndepth(I) + 1 \mid I \in (V' \cap \mathbb{TCFG})\}) \\ &< ndepth(\hat{I}) + 1 = ndepth(G) \end{aligned}$$

□

Theorem 4.17. *For a TCFG $G = (V, E, v_I, v_O)$ the function call FLATTEN(G) always creates a TCFG $H = (V', E', v_I, v_O)$ with ndepth$(H) = 0$.*

Proof. First of all it should be pointed out that the REP and TC calls in FLATTEN do not change the value of *ndepth*: As the functions TC_I and TC_O use the same vertex set V for their result graphs they will not modify the value of *ndepth*, as according to Remark 4.15 it only depends on the vertex set. The REP function only modifies the vertex set by removing `ReplacementNode` instances which themselves cannot be TCFGs. Therefore the REP function also does not change the value of *ndepth*. Now a proof by induction over *ndepth*(G) can be made:
Let *ndepth*$(G) = 0$: With Remark 4.15 this means $\#(V \cap \mathbb{TCFG}) = 0$ (otherwise *ndepth*$(G) \geq 1$) and therefore also *ndepth*$(H) = 0$ because there are executions of the MERGE function.
Let *ndepth*$(G) = k > 0$:
Case 1: $\#(V \cap \mathbb{TCFG}) > 0$
With Rem. 4.15 it follows that $\forall I \in (V \cap \mathbb{TCFG}) : ndepth(I) < k$ (otherwise $ndepth(G) > k$)
Applying the induction hypothesis yields $ndepth(\text{FLATTEN}(I)) = 0$
Using Lemma 4.16 after FOREACH I IN $(V \cap \mathbb{TCFG})$ DO $H \leftarrow \text{MERGE}(H, I)$ with initially $H = G$ it holds that $ndepth(H) < ndepth(G) = \max\{ndepth(I) + 1 \mid I \in V \cap \mathbb{TCFG}\} = 1$ and thus $ndepth(H) = 0$.
Case 2: $\#(V \cap \mathbb{TCFG}) = 0$
This case cannot occur, as from Remark 4.15 follows that $ndepth(G) = 0$ which contradicts $ndepth(G) = k$. □

Theorem 4.18. *For a TCFG $G = (V, E, v_I, v_O)$ the function call FLATTEN(G) creates a CFG if for all* `ReplacementNode` *instances v_r^e replacing a node v_e: $v_e \in nodes(G)$.*

Proof. As FLATTEN results in a 4-tuple (V, E, v_I, v_O) according to Corollary 4.5 the result is a TCFG. Therefore the following properties of G remain to be proved:

- $E \cap (V \times \{v_I\}) = \emptyset$
- $E \cap (\{v_O\} \times V) = \emptyset$
- $V \cap \mathbb{TCFG} = \emptyset$
- $V \cap \mathbb{V}_r = \emptyset$

For all these proofs only the initial FLATTEN call is considered, assuming the recursion has already completed. Note that the graph G is changed in place during the FLATTEN algorithm which is why it has to be taken into account at what point of the algorithm exactly G is referred to.

1. $E \cap (V \times \{v_I\}) = \emptyset$:
   Let $(v_k, v_I) \in E$ after the FOREACH loops, then $\text{TC}_I(v_I, G)$ will create a new CFG $(V, E_1 - E_2, v_I, v_O)$ with $E_1 = E \cup \{(v_i, v_j) \mid v_i, v_j \in V : v_I \neq v_i \wedge v_I \neq v_j \wedge (v_i, v_I) \in E \wedge (v_I, v_j) \in E\}$ and $E_2 = (V \times \{v_I\}) \cup \{(v_I, v_I), (v_O, v_O)\}$. As $(v_k, v_I) \in (V \times \{v_I\})$ it follows that $(v_k, v_I) \notin E_1 - E_2$. Therefore it remains to be shown, that $\text{TC}_O(v_O, G)$ does not reintroduce any edge (v_k, v_I):
   As $\text{TC}_O(v_O, G)$ will create a TCFG $(V, E_1' - E_2', v_I, v_O)$ with $E_1' = E \cup \{(v_i, v_j) \mid v_i, v_j \in V : v_O \neq v_i \wedge v_O \neq v_j \wedge (v_i, v_O) \in E \wedge (v_O, v_j) \in E\}$ and $E_2' = (\{v_O\} \times V) \cup \{(v_I, v_I), (v_O, v_O)\}$ the reintroduction of (v_k, v_I) requires that $v_O \neq v_k \wedge (v_k, v_O) \in E \wedge (v_O, v_I) \in E$. However the execution of $\text{TC}_I(v_I, G)$ would have already removed (v_O, v_I) if it was in E so that this condition will never be satisfied.

2. $E \cap (\{v_O\} \times V) = \emptyset$:
   Let $(v_O, v_K) \in E$. The final call of $\text{TC}_O(v_O, G)$ will create a new TCFG $(V, E_1' - E_2', v_I, v_O)$ with $E_1' = E \cup \{(v_i, v_j) \mid v_i, v_j \in V : v_O \neq v_i \wedge v_O \neq v_j \wedge (v_i, v_O) \in E \wedge (v_O, v_j) \in E\}$ and $E_2' = (\{v_O\} \times V) \cup \{(v_I, v_I), (v_O, v_O)\}$. As $(v_O, v_k) \in (\{v_O\} \times V)$ it follows that $(v_O, v_k) \notin (E_1' - E_2') = E$ which is a contradiction.

3. $V \cap \mathbb{TCFG} = \emptyset$:
$ndepth(FLATTEN(G)) = \max(\{0\} \cup \{ndepth(I) + 1 \mid I \in (V \cap \mathbb{TCFG})\}) = 0$ by Theorem 4.17 we get $\#(V \cap \mathbb{TCFG}) = 0$.

4. $V \cap \mathbb{V}_r = \emptyset$:
Let $v_r^e \in (V \cap \mathbb{V}_r)$ and $v_e \in nodes(G) \wedge v_e \in V_s$.
As $\#(V \cap \mathbb{TCFG}) = 0$ it holds that $nodes(G) = V$. Therefore the function call REP(v_r^e, G) will find that $v_e \in V$ and thus the resulting vertex set V will be $V - \{v_r^e\}$, which contradicts $v_r^e \in (V \cap \mathbb{V}_r)$.

□

4.3 Implementation of CFG construction using IP and SrcML

As part of the implementation accompanying this work an implementation of the functions given above has been made. This section will cover some of the details concerning this implementation, describe the design criteria made, and present a short usage manual.

The source code responsible for the control flow graphs is assembled into the `de.srcml.cfg` Eclipse plug-in. Its main components are the `ICFGCreation` interface seen in Listing 25, the `CFGCreationManager` which will be covered in more detail later, and a `CFGHelper` class which is used after the CFG construction to output a viewable PostScript version of the graph.

In order to avoid reinventing the wheel for the underlying data structure used for representing the graph, several Java graph libraries have been evaluated with JGraphT [Nav05] being the library of choice. Most other graph libraries were tightly coupled with the visual presentation of graphs whereas JGraphT "provides mathematical graph-theory objects and algorithms." [Nav05] Another reason for the choice of JGraphT was the explicit support for using a complete graph as a vertex which allows a very direct implementation of the algorithm described earlier.

The `CFGCreationManager` primarily contains the function `flattenCFG` which is a direct implementation of the FLATTEN function from Section 4.2 and can be seen in Listing 26. The complete source code of the `CFGCreationManager` class which contains implementations of all methods discussed above is available in Appendix C, Listing 36. It is also used to allow `ICFGCreation` adapter classes to delegate the creation of subgraphs and it handles the queue for the creation of jump edges as discussed in Section 4.1.

```java
protected static void flattenCFG(CFG cfg) {
    // first we recursively flatten all subgraphs
    for (Object o : cfg.vertexSet()) {
        if (o instanceof CFG) {
            flattenCFG((CFG)o);
        }
    }
    // now we have to find all flattened subgraphs and flatten
    // them into the main graph:
    List<CFG> subCFGs = new ArrayList<CFG>();
    for (Object o : cfg.vertexSet()) {
        if (o instanceof CFG) {
            subCFGs.add((CFG)o);
        }
    }
    // this loop performs the merge:
    for (CFG subcfg : subCFGs) {
            // now add all the nodes and edges from that subcfg
            // into our current CFG:
            cfg.addAllVertices(subcfg.vertexSet());
            cfg.addAllEdges(subcfg.edgeSet());
```

```
                // as we want to delete the subgraph node we first have to
                // redirect the edges connected to it:
                redirectEdgeTarget(cfg, subcfg, subcfg.ENTRY);
                redirectEdgeSource(cfg, subcfg, subcfg.EXIT);
                // now we can remove the subgraph's node:
                cfg.removeVertex(subcfg);
                // next we need to remove the ENTRY/EXIT nodes copied
                // from the subgraph
                collapseVertex(cfg, subcfg.ENTRY);
                collapseVertex(cfg, subcfg.EXIT);
        }

        checkReplacementNodes(cfg);
        // and finally calculate closure for ENTRY/EXIT nodes
        edgeTransitivityIncoming(cfg, cfg.ENTRY);
        edgeTransitivityOutgoing(cfg, cfg.EXIT);
}
```

Listing 26: FLATTEN implementation

There was no proof made for the correctness of the implementation of the FLATTEN function mainly because it is already very close to the description given in Section 4.2 and therefore an additional proof offers only limited gains. Therefore our implementation of the FLATTEN function and the `ICFGCreation` adapter classes for Java have been created through test driven development (TDD).

Besides the FLATTEN function the main part of the implementation consists of adapter classes implementing `ICFGCreation`. Given the availability of recursively constructing subgraphs from child elements the implementation proved to be very intuitive. Listing 27 shows the source code used for creating a control flow graph for `while`-loops. The variables `cond` and `blk` contain the `loop` element's `condition` and `block` child elements respectively. As can be seen the source code is easily readable and a straightforward implementation of how one expects a control flow graph for a `while`-loop to look like.

```
// add ENTRY->condition->EXIT
CFG subCond = mgr.createSubgraph(cond);
cfg.addVertex(subCond);
cfg.addEdge(cfg.ENTRY, subCond);
cfg.addEdge(subCond, cfg.EXIT);
// add condition->block->condition
CFG subBody = mgr.createSubgraph(blk);
cfg.addVertex(subBody);
cfg.addEdge(subCond, subBody);
cfg.addEdge(subBody, subCond);
```

Listing 27: construction of CFG for a while-loop

In order to be able to visualize control flow graphs the SrcML tree view from Section 2.5 was extended to allow the creation of a CFG for the selected element. Therefore an Eclipse user can now load a SrcML document and right-click an element in the corresponding tree view. After selecting the menu entry for the CFG creation a PostScript viewer will be opened which displays the generated graph for the selected element.

The conversion from the JGraphT representation of the CFG to the PostScript version is performed by the `CFGHelper` class with the help of the Graphviz [Res06] library. The language used for creating graphs with Graphviz is very simple and documented on the corresponding web site. Listing 28 shows a small example for the code needed to create the graph shown in Figure 7.

```
digraph "Auto-generated CFG" {
18235855 [label="ENTRY"];
"18235855" -> "4537415";
4537415 [label="flag == true"];
"4537415" -> "27350423" [label="TRUE"];
"4537415" -> "11576600" [label="FALSE"];
27350423 [label="a = 0;\n"];
"27350423" -> "3845057";
11576600 [label="b = 1;\n"];
"11576600" -> "3845057";
3845057 [label="EXIT"];
}
```

Listing 28: Graphviz description for CFG of if statement

The source code used to produce this output from the previously created control flow graphs only requires about 30 lines. The numbers used for the vertices are the graph objects' `hashCode()` result which was used for simplicity, as it does not matter what name is given for a vertex in the description of the graph as long as every vertex has a proper label. If no label was specified Graphviz would use the name of the vertex given in the description as the label.

4.4 Application to the arithmetic example

The construction algorithm described in Section 4.1 can now be applied to the arithmetic example. The new `switch` intention for strings is a prime example of how a fixed construction algorithm cannot know how to create a control flow graph for every intention. According to the adapter concept an implementation of the `ICFGCreation` interface is required to support the generation of control flow graphs for documents containing the new `switch` intention.

The implementation of the `createGraph` method is straightforward and requires the creation of a subgraph which contains vertices for the string expression and all `switch` target blocks. These have to be complete subgraphs again, as they can be arbitrary complex. Additionally the visual presentation of the `switch` statement can be improved by labeling the edges going from the expression to the various blocks with their respective string constants. The complete implementation of the adapter class can be found in Listing 35.

Another point of interest is the occurrence of non-local jumps via eventual `break` statements inside of the `switch`-blocks. These are an example for the need of `ReplacementNode` instances. As mentioned above the construction of such a block takes part at a time the subgraph for the `switch` statement has not been fully constructed yet. Even worse: the adapter class for the `break` statement would have to create an edge which crosses graph boundaries, as it connects the vertex of the `break` statement to the EXIT vertex of the `switch` statement's subgraph.

The first problem is solved as mentioned in Section 4.1 by delaying the construction of jump edges until all subgraphs have been created. This can be seen in the adapter class responsible for the `break` statement in Listing 29. Only the vertex for the `break` statement itself is created and connected to the ENTRY node, then a call to the `PresentationManager`'s method responsible for delaying the edge construction is made.

```java
protected CFG createGraphBreak(CFG cfg, CFGCreationManager mgr) {
    assert "break".equals(element.getName());
    cfg.addVertex(element);
    cfg.addEdge(cfg.ENTRY, element);
    mgr.queueForEdgeCreation(this);
    return cfg;
}
```

Listing 29: createGraph implementation for break

After this delay the subgraph of the `switch` statement is existing and can be found by the adapter class responsible for the `break` statement. After it has been found a `ReplacementNode` is inserted into this subgraph and connected to its EXIT vertex. This `ReplacementNode` is marked to be replaced by the vertex responsible for the `break` statement, such that after the flattening there will be an edge going from the `break` vertex directly to the EXIT vertex of the `switch` subgraph. If the intended CFG is covering more source code than only the `switch` statement, then the edge will in fact go to the statement immediately following the `switch` statement because the subgraph's EXIT node will be removed as described in Section 4.2.

It is also noteworthy to point out, that for the purpose of our arithmetic example the existing implementation of the `ICFGCreation` adapter class for the `break` statement could be reused because the new statement is also residing in a tag with the name "`switch`". The existing implementation for finding the target of `break` statement jumps does not take namespaces into account and thus will accept the new `switch` statement as the desired jump target. Though generally a new `break` statement would be necessary for the purpose of breaking from blocks of the new string-based `switch` intention.

After having control flow graphs available for the arithmetic example the advantages of this new `switch` intention can be visualized. Figure 8 shows the CFG for a standard chain of if-else-if expressions. It can be easily seen how at runtime the various `equals` calls will have to be executed in order until the correct branch is found. In fact the worst-case for the runtime is linear in the size of all compared strings. Comparing this to Figure 9 – which shows the CFG of the same program rewritten with the new string-based `switch` – directly reveals the performance advantage as the new CFG has a constant depth. This also implies that at runtime even a worst-case run should only require a constant amount of time.

The attentive reader might point out that a CFG for the if-else-if expressions could also be created such that it looks similar to Figure 9 showing a constant depth. Nevertheless the runtime would be as bad as before. This is an important point when discussing control flow graphs which have been created by additional plug-ins and is in fact an inherent problem in any plug-in architecture. There is no guarantee that all plug-ins provided by any third party developers will work as assumed by its user.

There is another way to verify the runtime behavior of the new string-based `switch` intention: As this intention is transformed into Java source code which can then be executed one might also take a look at the CFG of the resulting Java source code. Figure 10 shows the CFG for the previous program after transforming the string-based `switch` intention into its equivalent Java source code. This CFG also shows the usage of the underlying `Map` data structure and once again presents a graph with constant depth. It can now be better argued that the runtime of the new version is indeed an improvement over a simple if-else-if chain.

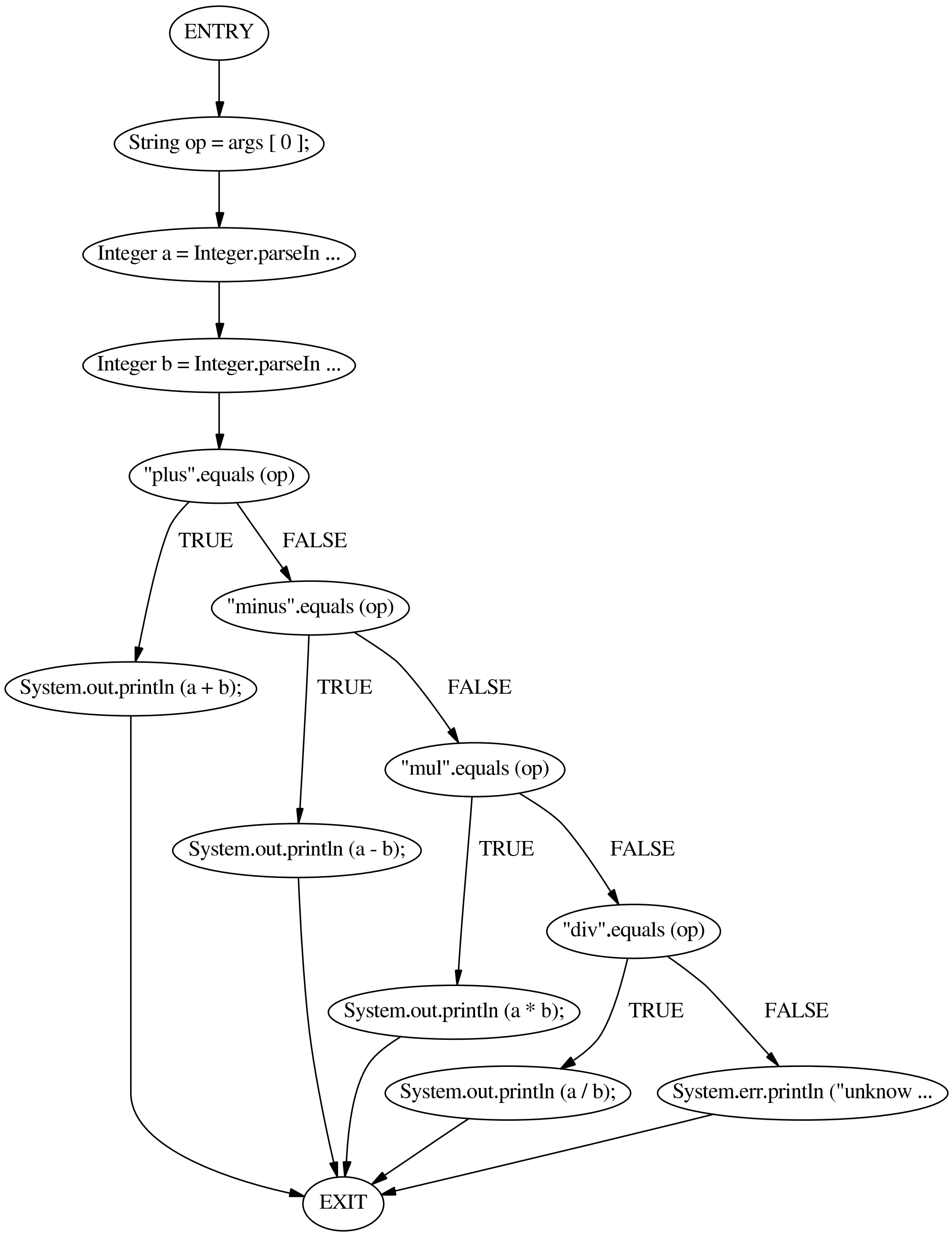

ENTRY
String op = args [0];
Integer a = Integer.parseIn ...
Integer b = Integer.parseIn ...
"plus".equals (op)
TRUE
FALSE
"minus".equals (op)
System.out.println (a + b);
TRUE
FALSE
"mul".equals (op)
System.out.println (a - b);
TRUE
FALSE
"div".equals (op)
System.out.println (a * b);
TRUE
FALSE
System.out.println (a / b);
System.err.println ("unknow ...
EXIT

Figure 8: CFG for if-chain

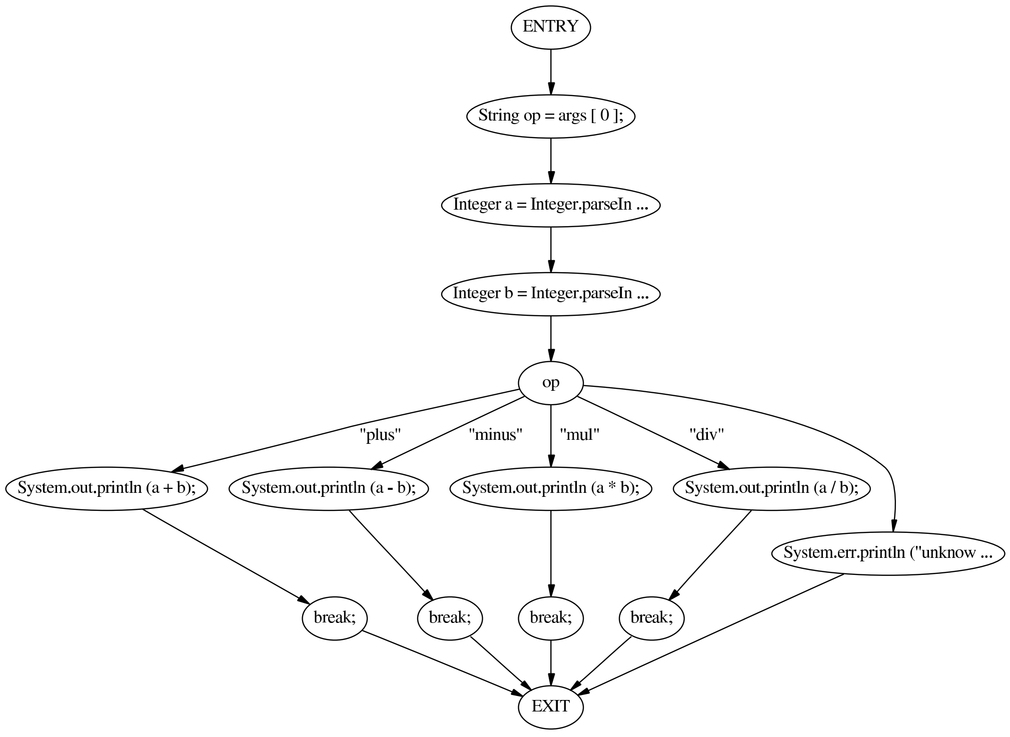

ENTRY
String op = args [0];
Integer a = Integer.parseIn ...
Integer b = Integer.parseIn ...
op
"plus"
"minus"
"mul"
"div"
System.out.println (a + b);
System.out.println (a - b);
System.out.println (a * b);
System.out.println (a / b);
System.err.println ("unknow ...
break;
break;
break;
break;
EXIT

Figure 9: CFG for extended switch

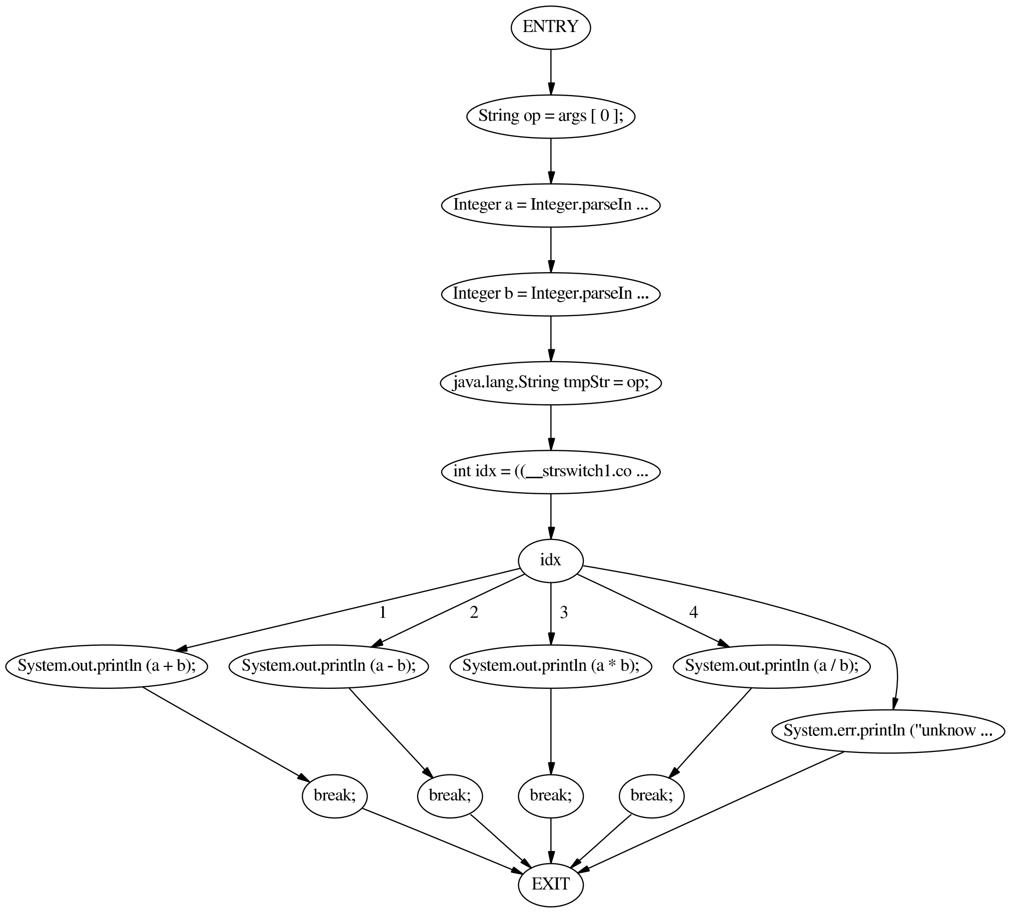

ENTRY
String op = args [0];
Integer a = Integer.parseIn ...
Integer b = Integer.parseIn ...
java.lang.String tmpStr = op;
int idx = ((__strswitch1.co ...
idx
1
2
3
4
System.out.println (a + b);
System.out.println (a - b);
System.out.println (a * b);
System.out.println (a / b);
System.err.println ("unknow ...
break;
break;
break;
break;
EXIT

Figure 10: CFG for extended switch after transformation

5 Overview of the implementation

This section provides an overview of the complete implementation created for this work. As the implementation consists of almost 20000 lines of Java code we decided not to include it in this work in its entirety. The complete source code is available through the SrcML project webpage [Rai04] via CVS . The following sections give a short introduction on the installation of our implementation. Furthermore the API is explained which allows using the SrcML and IP features from a programming point of view. This section does not provide details on how the provided implementations work. Those can be found in the corresponding sections of this work.

5.1 Installation and usage

The installation requires an existing installation of the 3.2 version of the Eclipse platform. A Java installation for Java 1.5 has to be present as well. Once these requirements are fulfilled Eclipse can be used to retrieve a copy of the project source code via CVS. This process is explained in detail in the Eclipse documentation, so that we only list the available plug-ins here:

- `de.srcml` – provides high-level access to SrcML and IP features
- `de.srcml.cfg` – provides the implementations related to Section 4
- `de.srcml.dependencies` – provides class files on which the remaining plug-ins depend
- `de.srcml.ext.java` – contains the Java extension created for Section 3.4
- `de.srcml.java` – contains Java specific functionality like the parser and presentations to display Java source code
- `de.srcml.ui` – is responsible for the integration into the Eclipse user interface
- `de.srcml.ui.editor` and `de.srcml.ui.editor.java` contain the prototype IP editor, which is deactivated by default as it is missing vital features

The next step is to export these plug-in projects into an existing Eclipse installation. The same installation used to retrieve the plug-ins can be used for that. The export process – described in [GE03] – adds the plug-ins to the Eclipse plug-in repository the next time Eclipse is started. So after a restart of Eclipse the SrcML and IP functionality can be accessed and the installation is complete. Note that for changes to the source code the plug-ins need to be exported again and the Eclipse environment needs to be restarted.

The installation can be verified by opening the *About Dialog* of Eclipse and select the *Plug-in Details* button. This should display a list of available plug-ins including the plug-ins mentioned above. Once the plug-ins are successfully loaded the *Preferences* dialog of Eclipse provides an additional tab for configuring the installation. Default values are already set at this point, but on a new installation the preferences should be checked and if needed updated by the developer. The preferences currently include file system locations for additional programs needed to create and display the control flow graphs. Additionally the location of a local copy of the SrcML schema file can be supplied which speeds up the schema verifications, because the file does not have to be downloaded for a verification.

Using SrcML is currently only possible with Java projects, as there are no parsers available for other programming languages yet. It is advisable to first create a test project and add a few Java source code files to it. A right-click on the project entry in the package explorer of Eclipse provides a menu labelled *SrcML*. Selecting the entry *Toggle project nature* in this menu activates the automatic parsing for this project. After selecting the entry all Java source code files are transformed into SrcML and stored within the project. Furthermore whenever a Java source code is modified and saved the corresponding SrcML document is updated as well.

The SrcML documents can be opened in Eclipse which invokes the required active source operations for creating a textual display. The developer can now open the SrcML tree view and adapters view through `Window>Show View>Other...` This allows using those views to adjust the representation of the source code displayed in the editor as discussed earlier. The implementation of Section 4 can also be accessed through the SrcML tree view to display control flow graphs of the active source, assuming that the programs required for creating and displaying the graphs have been configured in the preferences first. Right-clicking on nodes in the SrcML tree view then displays a popup menu which contains an entry for the control flow graph generation.

5.2 Programming with SrcML and IP

Developers can use the SrcML plug-ins in their own programs. The simplest way to do so is to write the program in Eclipse, thus being able to make it depend on the SrcML plug-ins. Otherwise Eclipse's plug-in loading mechanism has to be included in the program. The following sections discuss some typical tasks in which our implementation can be used.

In order to make the implementation easier to use for other developers we included a `Facade` class in the `de.srcml` plug-in. This class hides the underlying concepts discussed in earlier sections and provides a simple interface to use them. Some important methods of the `Facade` class are explained in the following sections.

5.2.1 Parsing source code into SrcML

Before being able to work with SrcML documents one usually has to transform traditional source code documents into SrcML first. To this end, the `Facade` in the `de.srcml` plug-in provides the methods given in Listing 30. Using these methods the developer does not have to worry about how the corresponding parser plug-ins are found.

```
public static Element parseByParser(String str, String parserID)
    {/**/}
public static Element parseByParserAndKind(String str,
        String parserID, IParser.Kind kind) { /**/ }
public static Element parseByLanguage(String str, String language)
    { /**/ }
public static Element parseByLanguageAndKind(String str,
        String language, IParser.Kind kind) { /**/ }
```

Listing 30: Methods in Facade used for parsing

All four of the parser methods expect the source code as their first argument. The `Facade` only provides methods for which the source code is of type `String`. For more fine-grained control an experienced developer can also load the parser plug-in directly and provide source code through a `Reader` instance. The reason for only using strings in the `Facade` is that this allows developers to also create small SrcML subtrees for metaprogramming purposes. If for example a developer wants to add an assignment to the SrcML document, she would have to create several nodes for the assignment subtree manually. With the help of the `Facade`, however, the assignment can be written in a traditional programming language and parsed into a complete tree. The `IParser.Kind` variants of the methods are used to this end, as they can be further customized by providing the parser with information on what kind of source code is contained in the argument string. Listing 31 shows the different supported kinds of source code. Note that it is also possible to use these methods to create a tree which is larger than needed, in order to extract only a subtree from it. For example parsing a complete declaration of a class is a simple way to obtain the tree of a method which is a subtree of this declaration.

```
public static enum Kind {
    UNIT, TYPE_DECL, EXPRESSION;
}
```

Listing 31: Source code kinds for parser

The parser methods provided in Listing 30 are separated into two groups for two corresponding usage scenarios: the `parseByLanguage`-based methods allow the developer to tell the `Facade` in which traditional programming language the source code string is written in. The `Facade` automatically tries to find an appropriate parser plug-in then. For the second scenario it is assumed, that the developer already knows about a specific parser plug-in and wants to perform the parsing with exactly that plug-in. To this end the `parseByParser`-based methods allow specifying the unique ID of this plug-in. For those methods the `Facade` will not try different parsers.

The return values of these methods always contain the root element of the resulting SrcML tree. This root element can differ depending on the choice of the `IParser.Kind` value used. For example when parsing source code with `IParser.Kind.UNIT`, which is used to parse complete source code files, the resulting element after a successful parse will be a `unit` element.

Note that due to the time restrictions of this work the error handling is very restricted. An error in the parsing process is identified by a `null` return value, but it is not possible to determine the error cause, unless a specific parser plug-in is used which provides these errors. There is no further error handling included in the current implementation besides a generic logging mechanism.

After successfully parsing a source code file the developer retrieves its DOM representation which can be stored in a file after being used. The `Facade` provides the `writeSrcML` method to this end which uses a generic `Writer` to store the document.

5.2.2 Loading SrcML documents

When developing a program which can expect SrcML documents to already be available, these documents have to be loaded to make their DOM representations accessible in the memory space. To this end the `Facade` provides the methods given in Listing 32. The `IFile`-based method can be used when working within the Eclipse environment, as it is the interface Eclipse uses for representing files in projects. The more generic `Reader`-based method can be used in all other cases.

```
public static Element loadSrcML(Reader reader) { /**/ }

public static Element loadSrcML(IFile file) { /**/ }
```

Listing 32: Load methods in Facade

5.2.3 Active source operations

Active source operations are generally made available through the implementations of the plug-ins providing the corresponding extension point. As the presentation and transformation operations are already defined in the `de.srcml` plug-in, the `Facade` can also provide the shortcuts found in Listing 33 for their usage.

```
public static String createStringPresentation(Element e) { /**/ }

public static String createStringPresentation(Element e,
        String[] presentIDs) {/**/ }

public static String createStringPresentation(
        Element e, String[] presentIDs, String[] transformIDs)
    { /**/ }
```

Listing 33: Facade methods for creating presentations

The easiest way to retrieve a textual presentation of a SrcML subtree is to pass its root node to the `createStringPresentation(Element)` method. The `Facade` then uses the default factories to produce the output. These default factories can be set using the adapters view as described in Section 3.5.2. The other two methods provide a more fine-grained control to produce certain outputs by specifying an explicit set of presentation and transformation plug-ins to use.

In the more generic case the operations have to be accessed through the plug-ins which provide the extension point. An example for this can be seen in the `de.srcml.cfg` plug-in which provides the extension point for creating control flow graphs as discussed in Section 4. Note that extension points can be provided by any third party and therefore accessing the corresponding operations is not always identical. For our implementation we use manager classes with methods similar to the `Facade` methods. For example the `CFGCreationManager` class provides a method `createCFG(Element)` which returns a `CFG` instance. In order to generate a control flow graph for a method a developer passes the root element of the method's subtree to this method and gets the corresponding control flow graph as a result. The result graph is only a memory representation – based on JGraphT [Nav05] – and not yet visualized. The visualization provided as part of the Eclipse integration is created with the help of the Graphviz [Res06] package. To this end, the graph representation has to be brought into a suitable text format as discussed in Section 4. This transformation is available through the `createDotRepresentation(CFG)` method in the `CFGHelper` class.

5.3 Unit tests

As much as we would have liked to prove the correctness of our implementation it is unfeasible to do so. It could be proved that every compilable Java source code can be represented in SrcML, and that after transforming it back to a textual representation it is compilable again and results in the same bytecode sequence, thus remaining semantically identical. Such a proof requires a tremendous amount of time and even a successful proof would not add anything significant. Therefore we decided to rely on empirical verification in terms of unit tests.

The implementation has been created in parallel to those unit tests. This process is referred to as test driven development (TDD). In theory it is based upon small cycles each of which consists of creating a test case, verifying it fails, adding the necessary code, and verifying the test case succeeds with the new code. This approach is slightly problematic with larger tests, but it performed extraordinarily well on the unit test for the implementation of the control flow graph creation algorithm.

For the unit tests the well-known JUnit [Var06] package is used. We supplied unit tests for all major parts of the implementation. Namely this includes a unit test to verify, that the SrcML document remains unchanged, when represented as Java source code and parsed again. To this end Java source code is first parsed into SrcML. Then the active source operations for recreating a textual presentation of the Java source code are invoked. Finally the result is parsed once more and compared to the first SrcML tree such that the test fails if there is any discrepancy. Note that we had to make some adjustments for this test to work. For example we specifically excluded comments, as we have not implemented a heuristic for associating them with an element in the SrcML tree and so comments might be causing non-semantic differences in the trees.

The attentive reader might notice that such a unit test can easily be fulfilled by always making the parser output the same constant well-formed SrcML document. After discovering this inherent flaw in the unit test we added a second unit test which manually checks that after the parser has finished its work it has visited and handled every single node of the original AST. While this does not eliminate the theoretical possibility above it is a fairly strong requirement considering the parser was not developed to bypass the unit test. Furthermore a third unit test verified that the resulting SrcML documents always adhere to the SrcML schema. These three tests have been run successfully on the entire Eclipse code base, which consists of roughly 12000 Java files, as well as on the implementation itself.

The part of the implementation which generates the control flow graphs as discussed in Section 4 was developed with test driven development as well. In this case it was infeasible to provide an automated test, so we chose a set of specific control flow graphs. The test then uses certain control structures and invokes the implementation to create their corresponding control flow graphs which are then compared to the expected graph. Note that this is not a generic graph isomorphism problem, because in this case the mapping of nodes is known: A node of the graph which corresponds to the condition node of the SrcML tree has to be the same node in both graphs with the same edges. Thus there is no need for performing a search and the two graphs can be compared very quickly. The roughly 20 tests include several control structures and various different usages, for example different kinds of `for` loops.

6 Conclusion

This section presents a summary of this thesis. It discusses which goals have been met as well as the remaining work including open problems. As there is a lot of activity on XML representations of source code at the time of writing, we also present an overview over the related work done in the field including short comparisons to our approaches.

6.1 Summary

We have developed SrcML, a format for storing source code in XML. This format provides a large amount of language neutrality and allows for easy, yet powerful, querying of source code. The format was further developed with a focus on extensibility. As an application of this extensibility the accompanying implementation provides an exemplary extension to the Java programming language.

Furthermore we succeeded in using SrcML as a base for the implementation of an Intentional Programming environment in Eclipse. We developed a prototype implementation of an IP environment including active source operations based on the adapter pattern [GE94]. This implementation supports all three types of extensibility offered by IP: new intentions, new active source operations, and different implementations of active source operations. Exemplary extensions have been created for each of these types: a new switch statement for the Java programming language, an active source operation to create control flow graphs, and a presentation of Java source code in a Lisp-style syntax.

For the creation of control flow graphs we developed an algorithm based on a formal description of the required graphs. This algorithm is based on the adapter pattern as well and works for the possible extensions of an IP environment. It contains the interesting property that the creation algorithm is split up into a generic part and adapter parts such that the generic part remains unchanged despite all three types of extensibility. We further made use of the formal description to prove the correctness of the generic part of the algorithm under the assumption that the adapter parts work correctly.

These implementations are written in Java and integrated into the Eclipse platform. The complete source code consists of roughly 20.000 lines of code including the code required for the unit tests. Our work further contains the development of the SrcML schema, the application of the adapter pattern to active source operations, the IP extensions, and the CFG construction algorithm.

6.2 Related work

This section covers other works with similar topics to those discussed in this thesis. There has been a lot of momentum in the development of XML representations of source code recently, resulting in a large amount of publications on that matter. Kostas Kontogiannis has been very active in this field [MK00, ZK01, AEK05]. He is discussing the idea of directly mapping a language's grammar to a document type definition (DTD) which results in XML documents being identical to abstract syntax trees. On top of this format he proposes another XML format called Object-Oriented Markup Language (OOML) . With SrcML we have taken the approach to directly create a format similar to OOML. While Kontogiannis removes the expression level from this format we believe this information can easily be ignored if not required and does not justify a second format which simply resembles language dependant ASTs. Kontogiannis further proposed various different high-level XML formats which could be gained from OOML. We have not pursued this idea further, but it is clear, that the same formats can be derived from SrcML documents as well. [AEK05] further delves into the topic of language neutral analysis tools. While we have not concentrated on the analysis of source code we always tried to support language neutral metaprogramming which also covers the analysis of source code.

Marko Topolnik also investigates the issue of representing source code in XML [ST03, Top05]. He emphasizes the importance of querying source code similar to our approach. He further argues that the direct mapping of ASTs to XML as proposed by Kontogiannis negatively affects queries. Topolnik further discusses the concept of extensibility through namespaces to provide orthogonal extensions and add metadata to source code documents. We have kept these ideas in mind when developing the SrcML schema. Topolnik also provides an Eclipse plug-in for creating his XML format from Java source code. As we integrate Intentional Programming concepts into our implementation and work with a different schema, we have not reused his implementation. We further found that his idea to use recursive forms – representing nested binary operations as one operation accepting a list of operands – is already implemented in this way in the Eclipse parser. We extended this idea by applying it to if statements as well.

Jonathan Maletic and Andrian Marcus have developed an XML representation for C++ at the Kent University [MCM02, CMM02, CKM03]. As opposed to the other XML formats above, their format – coincidentally called srcML – is added as an additional layer on top of the source code. To this end the XML tags are inserted into the source code without making any other changes to it. This means it is non-intrusive and the original source code can be recovered by simply removing the XML tags. However this format is restricted in having to adjust to the original order of tokens in the source code. It can therefore not restructure the XML to provide the benefits of the other formats.

Greg Badros is working on JavaML [Bad00, ADB04] , another popular XML representation for Java source code and he suggests it as a base for the design of a more generalized markup language similar to Kontogiannis' suggestion of OOML. Badros further added the idea of cross-referencing program symbols in the XML format, for example by providing links from variables to their declarations. This cross-referencing is an essential part of the graph data structure used for Intentional Programming in this work. JavaML has been used for aspect and component composition in Stefan Schonger's diploma thesis [Sch02]. One of his results was the insufficiency of XSLT for more complex transformations. We therefore never tried to base our transformations on XSLT and integrated them into the IP approach as active source operations instead. Nevertheless it is possible to apply XSL transformations on the SrcML format if required.

The diploma thesis from Heiko Lorenz [Lor05] similarly to SrcML tries to identify common language constructs to create a so-called *Unificated Abstract Syntax Tree* (UAST). We found this structure to be unsuitable for this work, as it is overly simplified and does not even cover the expression level. Furthermore no argumentation is provided on how the UAST structure was chosen.

Our discussion of Intentional Programming is based on the corresponding section of [CE00] where it is treated as a subconcept of Generative Programming. Charles Simonyi orginially introduced Intentional Programming in [Sim95]. We use the results provided in both works and combine them with approaches to store source code in SrcML. Nevertheless we found the available information about Intentional Programming to be lacking at times, such that it warrants future work which is discussed in the next section.

The PhD thesis from Donovan Michael Kolbly on Extensible Language Implementation [Kol02] further influenced our approach taken for the Intentional Programming environment. Kolbly's work concentrates on modifications in parsers mainly because he argues that no tree-like structure of source code is available. In our case, SrcML provides this structure and eliminates the need for parser modifications. Further Kolbly's argumentation about macros influenced our transformations. His conclusion, that it is advantageous to include implementations for extensible languages directly in compilers, can also be made for SrcML. As SrcML provides a representation of a compiler's data as it is usually found to be passed from frontend to backend of the compiler it is perfectly suited for adding language extensions. Language extensions which have to be included in compiler backends are also supported, as a compiler backend can be created as an active source transformation as well.

6.3 Future work

There is still a lot of future work to be done. Due to the given time constraints we have not been able to polish the implementation and this section details some of the possible improvement points.

Additionally a few theoretic problems have not been further investigated and are mentioned here as well.

6.3.1 Eclipse integration

With Eclipse being a very large and matured platform a lot of effort is required to fully integrate an idea like SrcML and Intentional Programming into it. Our implementation does integrate with Eclipse quite well, but there are still points for improvement. For example there is currently no fine-grained synchronization of the Java source code and its SrcML representation. Instead a change in the Java source code leads to a complete rewrite of the SrcML document possibly destroying any additional metainformation the developer added to the SrcML document. This situation can be improved by a more fine-grained control of Eclipse's parser by making use of the provided deltas which mark the modified positions in the source code. This allows only partially rewriting the SrcML document in the affected places. However the effort required for such an implementation is rather high and ideally this effort should be invested into improving the IP editor, thus eliminating the need for developers to work on traditional Java source code after it has been transformed to SrcML.

Features like auto-completion or refactoring, provided by Eclipse are initially unavailable when working with SrcML. Eclipse itself uses an AST representation of the source code to implement these features and it is possible to implement the interfaces required for this AST model to project changes directly to the SrcML DOM. This is complicated by the fact that Eclipse is specialized on development with the Java programming language and other programming languages use different representations in Eclipse or provide a smaller amount of features. While being able to use those features is an advantage the required implementation effort is very high and the result is breaking the language neutrality of our approach.

6.3.2 Intentional Programming

There is still a lack of information about Intentional Programming, as it has not become very popular since Simonyi proposed it in [Sim95]. His ideas are included in [CE00], but there is still no formal definition of IP available at this time. We have therefore tried to define the required terms in regard of the scope of this work, but we are aware that our definitions may not be sufficient for formalizing Intentional Programming in general.

An interesting topic which seems to have been neglected in IP so far is the semantics of how intentions are working together. As mentioned earlier it might be interesting to investigate if the IP environment is a scalable environment. Assuming a growing success of Intentional Programming the number of published intentions increases significantly. It is unclear how this affects the required efforts for adding new intention adapters. Considering the IP environment consists of a number of intentions does this require developers of new active source operations to take a quadratic number of special cases into account – or even more depending on the number of child elements? For example an operation which compiles intentions to bytecode might depend heavily on the intentions used as child or parent elements.

Similarly with the growing number of intentions adding a new active source operation at least requires adapters for all existing intentions. As discussed earlier this is not a strict requirement as it might suffice to only have adapters for a specific set of intentions, although this results in another problem in the long run: Is the IP environment still usable, when adapters are not available for some intentions? We hope that the growing popularity of an IP environment could help here as it also increases the amount of developers who can provide additional adapters. Nevertheless the original scalability question remains, as adding new intentions involves significantly less effort than creating the adapter implementations for them.

The implementations created for the IP environment for this work also offer many points for improvements. For example the previously mentioned caching of adapter methods could be further investigated to improve the search time for adapter implementations. The adapters view could also show the order in which extensions are searched to provide more transparency to the developer.

Due to the time restrictions we sometimes included language specific semantics into the implementation. These implementations should be changed and the semantics properly refactored from them. For example this includes determining an unused variable name which is required for the transformation of the string-based switch statement. This transformation currently does not handle hash collisions and could be updated to use the previously discussed solution for this problem.

Furthermore we only provide a small set of active source operations and adapters. Due to the extensibility of the IP environment there is a huge potential here to provide more sophisticated intentions, operations, and adapters.

Intentional Programming editor

Most of the remaining work is to be done for the IP editor. There is still a lot of research to do in order to determine how to design the editor implementation. It is still unclear for example what kinds of plug-ins the editor requires. As discussed earlier, the IP editor uses active source operations to perform the editing process, but there are many different types of editing – like addition, modification, and removal of intentions. To make the editor more comfortable to use, semantics should be integrated as well to provide for example auto-completion. As the IP editor can also provide very different visual presentations of the active source it is also an open question whether or how it will be possible to directly edit those presentations.

One of those problems for example is the keyword recognition: After the user types in a word the IP editor has to match the input to an active source operation. Usually this can be dealt with by well-known approaches from compiler design, but in the case of an IP editor the keywords which trigger active source operations can change dynamically. Depending on the state of the DOM, the current editing position, and the available adapter implementations a keyword may or may not be acceptable. It is questionable, if a naive approach like recalculating an Aho-Corasick search tree each time keywords are entered, is feasible. There might also be relations to the caching of adapter methods, such that results could be reused to provide a similar caching of keywords.

Some more research should also be made for the presentation of the active source. For example the current active source operations used for presentation of active source are by themselves not sufficient. When a part of the document is edited the IP editor should only be required to recreate the displaying of this part. With the current implementation though this results in problems, because the active source operation for creating presentations is not taking a possible indentation depth into account. This gets more complicated once more radical display methods are considered which can contain arbitrary widgets.

References

[ADB04] Ademar Aguiar, Gabriel David, and Greg Badros. JavaML 2.0: Enriching the Markup Language for Java Source Code. In *XATA'2004*, XML: Aplicacoes e Tecnologias Associadas, 2004.

[AEK05] Raihan Al-Ekram and Kostas Kontogiannis. An XML-based Framework for Language Neutral Program Representation and Generic Analysis. In *CSMR'05*, Ninth European Conference on Software Maintenance and Reengineering, 2005.

[Bad00] Greg J. Badros. JavaML: A Markup Language for Java Source Code. In *WWW9*, Ninth International World Wide Web Conference, 2000.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: methods, tools, and applications*. Addison-Wesley, 2000.

[CKM03] M.L. Collard, H. Kagdi, and J.I. Maletic. An XML-Based Lightweight C++ Fact Extractor. In *IWPC 2003*, IEEE International Workshop on Program Comprehension, pages 124–143, 2003.

[CLR89] Thomas Cormen, Charles Leiserson, and Ronald Rivest. *Introduction to Algorithms*. The MIT Press, 1989.

[CMM02] M.L. Collard, J.I. Maletic, and A. Marcus. Supporting Document and Data Views of Source Code. In *DocEng'02*, 2nd ACM Symposium on Document Engineering, pages 34–41, 2002.

[Con96] World Wide Web Consortium. Extensible Markup Language, 1996. `http://www.w3.org/XML/`.

[Con98] World Wide Web Consortium. Document Object Model (dom) level 1 specification, 1998. `http://www.w3.org/TR/REC-DOM-Level-1/`.

[Con99] World Wide Web Consortium. XML Path Language, 1999. `http://www.w3.org/TR/xpath`.

[Con01] World Wide Web Consortium. XML Schema, 2001. `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`.

[Cos05] Roger L. Costello. XML Schemas: Best Practices, 2005. `http://www.xfront.com/BestPracticesHomepage.html`.

[csh] C# Language Specification. "`http://msdn.microsoft.com`".

[ecl] Eclipse.org homepage. "`http://www.eclipse.org`".

[G$^+$05] James Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 2005.

[GE94] Johnson R. Vlissides J. Gamma E., Helm R. *Design Patterns : elements of reusable object-oriented software*. Addison-Wesley, 1994.

[GE03] Beck K. Gamma E. *Contributing to Eclipse: Principles, Patterns, and Plug-ins*. Addison-Wesley, 2003.

[Kol02] Donovan Michael Kolbly. *Extensible Language Implementation*. PhD thesis, University of Texas, September 2002.

[Lor05] Heiko Lorenz. Redesign eines Transformators für Refactoringmengen auf mehrsprachigen Quellcodes. Diploma thesis, University of Ulm, July 2005.

[MCM02] J.I. Maletic, M. Collard, and A. Marcus. Source Code Files as Structured Documents. In *IWPC 2002*, IEEE International Workshop on Program Comprehension, pages 289–292, 2002.

[MK00] Evan Mamas and Kostas Kontogiannis. Towards Portable Source Code Representation Using XML. In *WCRE'00*, IEEE Working Conference on Reverse Engineering, 2000.

[Muc97] Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1997.

[Nav05] Barak Naveh. JGraphT - a free java graph library, 2003-2005. `http://jgrapht.sourceforge.net`.

[Rai04] Frank Raiser. SrcML, 2004. `http://srcml.de`.

[Res06] AT&T Research. Graphviz - Graph Visualization Software, 2006. `http://www.graphviz.org`.

[Sch01] Uwe Schöning. *Algorithmik.* Spektrum Akademischer Verlag, 2001.

[Sch02] Stefan Schonger. Aspect and Component Composition using XML Representations of Abstract Syntax Trees. Diploma thesis, University of Ulm, February 2002.

[Sim95] C. Simonyi. The death of computer languages, the birth of intentional programming, 1995.

[ST03] Hrvoje Simic and Marko Topolnik. Prospects of encoding Java source code in XML. In *ConTel 2003*, 7th International Conference on Telecommunications, 2003.

[Str98] Bjarne Stroustrup. *The C++ Programming Language, Third Edition.* Addison-Wesley, 1998.

[Top05] Marko Topolnik. An improved XML syntax for the Java programming language. In *ConTel 2005*, 8th International Conference on Telecommunications, 2005.

[Var06] Various. JUnit, 2006. `http://www.junit.org`.

[Wil04] Dr. Gregory Wilson. Extensible Programming Systems for the 21st Century, 2004.

[WM97] R. Wilhelm and D. Maurer. *Übersetzerbau.* Springer-Verlag, Berlin, 2. Auflage, 1997.

[ZK01] Ying Zou and Kostas Kontogiannis. Towards a Portable XML-based Source Code Representation. In *XSE2001*, XML Technologies and Software Engineering, 2001.

Listings

List of Figures

Index

A SrcML schema

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" targetNamespace="http://srcml.de"
xmlns="http://srcml.de" xmlns:SrcML="http://srcml.de">
  <xs:annotation>
    <xs:documentation xml:lang="en">SrcML

    SrcML is an XML format which makes syntactic and semantic information
    in source code explicit through XML tags. Any compilable Java/C++/C#
    program can be represented in SrcML.

    This schema does not try to match the grammar of a language but to
    contain it, i.e. all valid programs can be represented in SrcML, but not
    all valid SrcML documents are valid programs.

    www.srcml.de

    $Id: SrcML.xsd,v 1.30 2006/07/19 12:42:08 crashchaos Exp $
    </xs:documentation>
  </xs:annotation>

  <xs:element name="unit">
    <xs:complexType>
      <xs:sequence>
        <xs:group ref="SrcML:basics" />

        <xs:element name="namespace" type="SrcML:Tnamespace" />

        <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>

      <xs:anyAttribute processContents="skip" />
    </xs:complexType>
  </xs:element>

  <xs:attributeGroup name="basicattrs">
    <xs:attribute name="name" />

    <xs:attribute name="namespace" />

    <xs:attribute name="id" />

    <xs:attribute name="idref" />

    <xs:attribute name="label" />
  </xs:attributeGroup>

  <xs:group name="basics">
    <xs:sequence>
      <xs:group ref="SrcML:comments" />

      <xs:element name="annotation" type="SrcML:Tannotation" minOccurs="0"
      maxOccurs="unbounded" />
    </xs:sequence>
  </xs:group>

```

```
<xs:group name="comments">
  <xs:sequence>
    <xs:element name="comment" type="SrcML:Tcomment" minOccurs="0"
    maxOccurs="unbounded" />
  </xs:sequence>
</xs:group>

<xs:complexType name="Tmodifiers">
  <xs:sequence>
    <xs:element name="modifier" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:attribute name="name" />

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:anyAttribute processContents="skip" />
</xs:complexType>

<xs:element name="modifiers" type="Tmodifiers" />

<xs:complexType name="Tvariables">
  <xs:sequence>
    <xs:group ref="SrcML:basics" />

    <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

    <xs:element name="variable" type="SrcML:Tvariable" minOccurs="1"
    maxOccurs="unbounded" />

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:attributeGroup ref="SrcML:basicattrs" />

  <xs:anyAttribute processContents="skip" />
</xs:complexType>

<xs:element name="variables" type="Tvariables" />

<xs:complexType name="Tcomment">
  <xs:sequence>
    <xs:element name="text" type="xs:string" minOccurs="0" />

    <xs:element name="tag" type="SrcML:Ttag" minOccurs="0"
    maxOccurs="unbounded" />

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
```

```

    <xs:attribute name="type" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="comment" type="Tcomment" />

  <xs:complexType name="Ttag">
    <xs:sequence>
      <xs:group ref="SrcML:comments" />

      <xs:element name="text" type="xs:string" minOccurs="0"
      maxOccurs="unbounded" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="name" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="tag" type="Ttag" />

  <xs:complexType name="Tannotation">
    <xs:sequence>
      <xs:element name="type" type="SrcML:Ttype" />

      <xs:element name="arguments" type="SrcML:Targuments" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="annotation" type="Tannotation" />

  <xs:complexType name="Tnamespace">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="imports" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="import" type="SrcML:Timport"
            maxOccurs="unbounded" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:choice minOccurs="0" maxOccurs="unbounded">
```

```
        <xs:element name="type_decl" type="SrcML:Ttype_decl" />

        <xs:element name="block" type="SrcML:Tblock" />
      </xs:choice>

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:attribute name="alias" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="namespace" type="Tnamespace" />

  <xs:complexType name="Timport">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

      <xs:element name="expr" type="Texpr" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="name" />

    <xs:attribute name="alias" />

    <xs:attribute name="type" />

    <xs:attribute name="on_demand" type="xs:boolean" default="false" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="import" type="Timport" />

  <xs:complexType name="Ttype_decl">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

      <xs:element name="parameterization" type="SrcML:Tparameterization"
      minOccurs="0" />

<!-- used for enum constants -->
      <xs:element name="arguments" type="SrcML:Targuments" minOccurs="0" />

      <xs:element name="inheritance" type="SrcML:Tinheritance"
      minOccurs="0" />

      <xs:element name="values" minOccurs="0">
        <xs:complexType>
```

```
          <xs:sequence>
            <xs:element name="type_decl" type="SrcML:Ttype_decl"
            minOccurs="1" maxOccurs="unbounded" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:element name="alias" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="type" type="SrcML:Ttype" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="variables" type="SrcML:Tvariables" />

        <xs:element name="method" type="SrcML:Tmethod" />

        <xs:element name="type_decl" type="SrcML:Ttype_decl" />

        <xs:element name="block" type="SrcML:Tblock" />

        <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
      </xs:choice>
    </xs:sequence>

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:attribute name="type" />

    <xs:attribute name="anonymous" type="xs:boolean" default="false" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="type_decl" type="Ttype_decl" />

  <xs:complexType name="Tinheritance">
    <xs:sequence>
      <xs:element name="inherits" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="modifiers" type="SrcML:Tmodifiers"
            minOccurs="0" />

            <xs:element name="type" type="SrcML:Ttype" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:attribute name="type" />

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="inheritance" type="Tinheritance" />

  <xs:complexType name="Ttype">
    <xs:sequence>
      <xs:element name="type" type="SrcML:Ttype" minOccurs="0"
      maxOccurs="unbounded" />

      <xs:element name="parameterization" type="SrcML:Tparameterization"
      minOccurs="0" />

      <xs:element name="type_array_size"
          minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="expr" type="Texpr" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:attribute name="dimension" type="xs:nonNegativeInteger" />

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:element name="pointer" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:attribute name="const" type="xs:boolean" />

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
```

```

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:attribute name="dimensions" type="xs:positiveInteger" />

    <xs:attribute name="reference" type="xs:boolean" />

    <xs:attribute name="typename" type="xs:boolean" />

    <xs:attribute name="jagged" type="xs:boolean" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="type" type="Ttype" />

  <xs:complexType name="Tparameterization">
    <xs:sequence>
      <xs:element name="type" type="SrcML:Ttype" maxOccurs="unbounded" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="type" default="generic" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="parameterization" type="Tparameterization" />

  <xs:complexType name="Tvariable">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

      <xs:element name="type" type="SrcML:Ttype" minOccurs="0" />

      <xs:element name="init" type="SrcML:Tinit" minOccurs="0" />

      <xs:element name="block" type="SrcML:Tblock" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:attribute name="type" />

    <xs:attribute name="varargs" type="xs:boolean" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="variable" type="Tvariable" />

  <xs:complexType name="Tmethod">
    <xs:sequence>
    <xs:group ref="SrcML:basics" />

    <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

    <xs:element name="parameterization" type="SrcML:Tparameterization"
    minOccurs="0" />

    <xs:element name="type" type="SrcML:Ttype" minOccurs="0" />

    <xs:element name="variables" type="SrcML:Tvariables" minOccurs="0" />

    <xs:element name="value" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="expr" type="Texpr" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:element name="baseinit" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="expr" type="Texpr" maxOccurs="unbounded" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:element name="throws" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="type" type="SrcML:Ttype"
          maxOccurs="unbounded" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:element name="block" type="SrcML:Tblock" minOccurs="0" />

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:attributeGroup ref="SrcML:basicattrs" />

```

```
    <xs:attribute name="type" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="method" type="Tmethod" />

  <xs:complexType name="Tblock">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="modifiers" type="SrcML:Tmodifiers" minOccurs="0" />

      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element name="method" type="SrcML:Tmethod" />

        <xs:element name="variables" type="SrcML:Tvariables" />

        <xs:element name="block" type="SrcML:Tblock" />

        <xs:element name="type_decl" type="SrcML:Ttype_decl" />

        <xs:element name="expr" type="Texpr" />

        <xs:any namespace="##other" />
      </xs:choice>
    </xs:sequence>

    <xs:attribute name="type" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="block" type="Tblock" />

  <xs:complexType name="Tinit">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="expr" type="Texpr" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="init" type="Tinit" />

  <xs:complexType name="Tassignment">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="expr" type="Texpr" minOccurs="2" maxOccurs="2" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="operator" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="assignment" type="Tassignment" />

  <xs:group name="expressions">
    <xs:choice>
      <xs:element name="array_access" type="SrcML:Tarray_access" />

      <xs:element name="assert" type="SrcML:Tassert" />

      <xs:element name="assignment" type="SrcML:Tassignment" />

      <xs:element name="break" type="SrcML:Tbreak" />

      <xs:element name="call" type="SrcML:Tcall" />

      <xs:element name="constant" type="SrcML:Tconstant" />

      <xs:element name="continue" type="SrcML:Tcontinue" />

      <xs:element name="field_access" type="SrcML:Tfield_access" />

      <xs:element name="fixed" type="SrcML:Tfixed" />

      <xs:element name="goto" type="SrcML:Tgoto" />

      <xs:element name="identifier" type="SrcML:Tidentifier" />

      <xs:element name="if" type="SrcML:Tif" />

      <xs:element name="import" type="SrcML:Timport" />

      <xs:element name="list" type="SrcML:Tlist" />

      <xs:element name="loop" type="SrcML:Tloop" />

      <xs:element name="namespace" type="SrcML:Tnamespace" />

      <xs:element name="new" type="SrcML:Tnew" />

      <xs:element name="nop" type="SrcML:Tnop" />

      <xs:element name="operation" type="SrcML:Toperation" />

      <xs:element name="return" type="SrcML:Treturn" />

      <xs:element name="switch" type="SrcML:Tswitch" />

      <xs:element name="synchronized" type="SrcML:Tsynchronized" />

      <xs:element name="throw" type="SrcML:Tthrow" />

      <xs:element name="try" type="SrcML:Ttry" />

      <xs:element name="value_pair" type="SrcML:Tvalue_pair" />
    </xs:choice>
  </xs:group>

  <xs:complexType name="Texpr">
    <xs:sequence>
      <xs:group ref="expressions" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="expr" type="Texpr" />

  <xs:complexType name="Targuments">
    <xs:sequence>
      <xs:element name="expr" type="Texpr" maxOccurs="unbounded" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="arguments" type="Targuments" />

  <xs:complexType name="Tarray_access">
    <xs:sequence>
      <xs:element name="array">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="expr" type="Texpr" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:element name="index">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="expr" type="Texpr" maxOccurs="unbounded" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="array_access" type="Tarray_access" />

  <xs:complexType name="Tassert">
    <xs:sequence>
      <xs:element name="condition" type="SrcML:Tcondition" />

      <xs:element name="expr" type="Texpr" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="assert" type="Tassert" />

  <xs:complexType name="Tbreak">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="label" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="break" type="Tbreak" />

  <xs:complexType name="Tcall">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="parameterization" type="SrcML:Tparameterization"
      minOccurs="0" />

      <xs:element name="callee" minOccurs="0">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="expr" type="Texpr" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:element name="arguments" type="SrcML:Targuments" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:attribute name="type" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="call" type="Tcall" />

  <xs:complexType name="Tcondition">
    <xs:sequence>
      <xs:element name="expr" type="Texpr" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="condition" type="Tcondition" />

  <xs:complexType name="Tconstant">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="type" type="SrcML:Ttype" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="value" type="xs:string" />

    <xs:attribute name="type" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="constant" type="Tconstant" />

  <xs:complexType name="Tcontinue">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="label" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="continue" type="Tcontinue" />

  <xs:complexType name="Tfield_access">
    <xs:sequence>
      <xs:element name="identifier" type="SrcML:Tidentifier" />

      <xs:element name="expr" type="Texpr" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="struct" type="xs:boolean" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="field_access" type="Tfield_access" />

  <xs:complexType name="Tfixed">
    <xs:sequence>
      <xs:element name="variable" type="SrcML:Tvariable" />

      <xs:element name="block" type="SrcML:Tblock" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="fixed" type="Tfixed" />

  <xs:complexType name="Tgoto">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="label" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="goto" type="Tgoto" />

  <xs:complexType name="Tidentifier">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attributeGroup ref="SrcML:basicattrs" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="identifier" type="Tidentifier" />

  <xs:complexType name="Tif">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:sequence maxOccurs="unbounded">
        <xs:element name="condition" type="SrcML:Tcondition" />

        <xs:element name="block" type="SrcML:Tblock" />
      </xs:sequence>

<!-- optional last else block -->
```

```
      <xs:element name="block" type="SrcML:Tblock" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="type" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="if" type="Tif" />

  <xs:complexType name="Tlist">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="expr" type="Texpr" minOccurs="0"
      maxOccurs="unbounded" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="type" type="xs:string" default="array" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="list" type="Tlist" />

  <xs:complexType name="Tloop">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="variables" type="SrcML:Tvariables" minOccurs="0" />

      <xs:element name="condition" type="SrcML:Tcondition" minOccurs="0" />

      <xs:element name="block" type="SrcML:Tblock" maxOccurs="3" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="type" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="loop" type="Tloop" />

  <xs:complexType name="Tnop">
    <xs:sequence>
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="nop" type="Tnop" />
```

```

<xs:complexType name="Tnew">
  <xs:sequence>
    <xs:group ref="SrcML:basics" />

    <xs:element name="parameterization" type="SrcML:Tparameterization"
    minOccurs="0" />

    <xs:element name="type" type="SrcML:Ttype" />

    <xs:element name="placement" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="expr" type="Texpr" maxOccurs="unbounded" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:element name="arguments" type="SrcML:Targuments" minOccurs="0" />

    <xs:element name="init" type="SrcML:Tinit" minOccurs="0" />

    <xs:element name="type_decl" type="SrcML:Ttype_decl" minOccurs="0" />

    <xs:element name="base" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="expr" type="Texpr" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:anyAttribute processContents="skip" />
</xs:complexType>

<xs:element name="new" type="Tnew" />

<xs:complexType name="Toperation">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="unbounded">
      <xs:element name="expr" type="Texpr" />

      <xs:element name="type" type="SrcML:Ttype" />
    </xs:choice>

```

```
      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="operator" type="xs:string" use="required" />

    <xs:attribute name="affix" type="xs:string" default="infix" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="operation" type="Toperation" />

  <xs:complexType name="Treturn">
    <xs:sequence>
      <xs:group ref="SrcML:basics" />

      <xs:element name="expr" type="Texpr" minOccurs="0" />

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:attribute name="type" type="xs:string" />

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="return" type="Treturn" />

  <xs:complexType name="Tswitch">
    <xs:sequence>
      <xs:element name="expr" type="Texpr" />

      <xs:element name="group" minOccurs="0" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="expr" type="Texpr" minOccurs="0"
            maxOccurs="unbounded" />

            <xs:element name="block" type="SrcML:Tblock" minOccurs="0" />

            <xs:any namespace="##other" minOccurs="0"
            maxOccurs="unbounded" />
          </xs:sequence>

          <xs:attribute name="default" type="xs:boolean" />

          <xs:anyAttribute processContents="skip" />
        </xs:complexType>
      </xs:element>

      <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>

    <xs:anyAttribute processContents="skip" />
  </xs:complexType>

  <xs:element name="switch" type="Tswitch" />

```

```
<xs:complexType name="Tsynchronized">
  <xs:sequence>
    <xs:element name="expr" type="Texpr" />

    <xs:element name="block" type="SrcML:Tblock" />

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:anyAttribute processContents="skip" />
</xs:complexType>

<xs:element name="synchronized" type="Tsynchronized" />

<xs:complexType name="Tthrow">
  <xs:sequence>
    <xs:element name="expr" type="Texpr" />

    <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>

  <xs:anyAttribute processContents="skip" />
</xs:complexType>

<xs:element name="throw" type="Tthrow" />

<xs:complexType name="Ttry">
  <xs:sequence>
    <xs:group ref="SrcML:basics" />

    <xs:element name="block" type="SrcML:Tblock" />

    <xs:element name="catch" minOccurs="0" maxOccurs="unbounded">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="variables" type="SrcML:Tvariables"
          minOccurs="0" />

          <xs:element name="block" type="SrcML:Tblock" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
      </xs:complexType>
    </xs:element>

    <xs:element name="finally" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="block" type="SrcML:Tblock" />

          <xs:any namespace="##other" minOccurs="0"
          maxOccurs="unbounded" />
        </xs:sequence>

        <xs:anyAttribute processContents="skip" />
```

```
          </xs:complexType>
        </xs:element>

        <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>

      <xs:anyAttribute processContents="skip" />
    </xs:complexType>

    <xs:element name="try" type="Ttry" />

    <xs:complexType name="Tvalue_pair">
      <xs:sequence>
        <xs:element name="expr" type="Texpr" minOccurs="2" maxOccurs="2" />

        <xs:any namespace="##other" minOccurs="0" maxOccurs="unbounded" />
      </xs:sequence>

      <xs:anyAttribute processContents="skip" />
    </xs:complexType>

    <xs:element name="value_pair" type="Tvalue_pair" />
</xs:schema>
```

Listing 34: SrcML schema

B Documentation of SrcML schema tags

This section provides a list of the available tags in the SrcML Schema along with a description of their intented usage as available in [Rai04]:

1. alias
2. annotation
3. arguments
4. array
5. array_access
6. assert
7. assignment
8. base
9. baseinit
10. block
11. break
12. call
13. catch
14. comment
15. condition
16. constant

17. continue
18. expr
19. field_access
20. finally
21. fixed
22. goto
23. group
24. identifier
25. if
26. import
27. imports
28. index
29. inheritance
30. inherits
31. init
32. list
33. loop
34. method
35. modifier
36. modifiers
37. namespace
38. new
39. nop
40. operation
41. parameterization
42. pointer
43. return
44. switch
45. synchronized
46. tag
47. throw
48. throws
49. try

50. type
51. type_array_size
52. type_decl
53. unit
54. value
55. value_pair
56. values
57. variable
58. variables

alias

Used in type_decl.

Structure

See type_decl.

Language specific notes

See type_decl.

annotation

An annotation can be used to annotate program constructs. These annotations can usually be queried with programs at runtime through some kind of reflection API. Note that this tag is not used for declaring a new kind of annotation. This is done with the type_decl tag instead.

Structure

An annotation tag consists of a type element specifying the type of annotation. Additionally an arguments tag is used for passing arguments to the annotation.

Language specific notes

C# annotations are called attributes

C++ not supported

arguments

An arguments tag is a generic way of passing a list of arguments.

Structure

The given arguments are each represented by an individual expr tag. It is not allowed to have an arguments tag without any expr child.

Language specific notes

None

array

This is used and described in array_access

Structure

See array_access

Language specific notes

See array_access

array_access

This tag is used for accessing an array. It is also used for a multi-dimensional array. Note that the linear representation is not equal to the execution order of the index expressions, as those depend on the language semantics. Instead it just represents the linear occurance in the original source code.

Structure

The first child is an array tag specifying which array is accessed. This is followed by an index tag specifying the access index. If multiple dimensions are accessed individually all index expressions will be listed linearly in the index element to reduce the tree depth.

Language specific notes

None

Example

```
arr[i][j] => <array_access>
    <array><identifier name="arr"/></array>
    <index>
      <expr><identifier name="i"/></expr>
      <expr><identifier name="j"/></expr>
    </index>
  </array_access>
```

assert

An assertion statement which checks a condition at runtime usually causing an exception to be thrown it the condition does not hold.

Structure

Always has a condition child, possibly followed by an expr tag which usually specifies an error message to print if the condition does not hold.

Language specific notes

None

assignment

An assignment is used, when an expression is evaluated and the result is then assigned to one or more variables. An assignment therefore consists of a left hand side (which the result is assigned to) and a right hand side (which is evaluated). Additionally some languages allow the developer to perform arithmetic assignments like +=.

Structure

The left hand side and right hand side of an assignment are represented as two expr tags, with the first one being the left hand side. As assignments can include operations there is an additional operator attribute set to the string representation of the operator.

Language specific notes

None

Example

```
a += 10; =>
          <expr>
            <assignment operator="+=">
              <expr>
                <identifier name="a"/>
              </expr>
              <expr>
                <constant value="10"/>
              </expr>
            </assignment>
          </expr>
```

base

Used in new.

Structure

See new.

Language specific notes

See new.

baseinit

Used in method.

Structure

See method

Language specific notes

See method

block

A block is a collector for a sequence of statements. In most languages a block is syntactically shown with corresponding begin and end tokens.

Structure

A block can have an optional modifiers element (f.ex. static initializer blocks in Java). The main part of the block however is made up of elements in any order. This includes the any elements used for extensions, so that you have to be very careful when going through a block's content. The elements given for a block's main content in the SrcML schema are method, variables, block, type_decl, and expr.

Some blocks serve a special purpose beyond the mere collecting of statements which can be set in the type attribute.

Language specific notes

Java (static) initializer blocks are set to @type="initializer".

break

A break statement is usually used to terminate repetition of a loop or finish a block in a switch statement (if no fall-through is wanted.)

Structure

In some languages it is possible to continue program execution at a certain label after the break statement was executed. Such a label can be stored in the label attribute.

Language specific notes

None

call

A method (or function, procedure) call or invocation.

Structure

The name attribute holds the name of the called method. The object to which the method belongs could be dynamically specified by an expression which is represented as a expr child in the callee element. If arguments are given to this method they are represented with their own arguments element.

In some languages methods can be generic too in which case a call to such a method needs additional type arguments which are handled by the parameterization element.

Special types of calls can be explicitly marked up using the type attribute (f.ex. constructor calls)

Language specific notes

None

Example

```
a.b(c) =>
    <call name="b">
        <callee><expr><identifier name="a"/></expr></callee>
        <arguments><expr><identifier name="c"/></expr></callee>
    </call>

this(...) =>
    <call name="this" type="constructor">
        ...
    </call>

BaseClass.super(...) =>
    <call name="super" type="supermethod" namespace="BaseClass">
        ...
    </call>
```

catch

Used in try.

Structure

See try.

Language specific notes

See try.

comment

A generic comment which may contain tags.

Structure

type attribute can be used for special language-specific comments (like javadoc) and is usually used to distinguish single-line and multi-line comments. Javadoc or doxygen and other special commenting techniques allow the addition of tags to a comment, which is directly supported by the tag element. The actual comment goes into a text element.

Language specific notes

Java Javadoc comments set @type=”javadoc”.

condition

A boolean condition for use in various other expressions. Basically a condition is just an expression which is supposed to evaluate to a boolean value.

Structure

Simply contains an expr tag.

Language specific notes

None

constant

A constant value. A constant should always have a value and can optionally have a type if available.

Structure

The value of the constant is stored in the value attribute and optionally there's a type child element. There's also a type attribute used for type constants in Java.

Language specific notes

Java type constants like SomeClass.class are represented as:

```
<constant type="class">
    <type name="SomeClass"/>
</constant>
```

Example

```
<constant value="10"/>
<constant value="hello world">
    <type name="string"/>
</constant>
```

continue

A continue statement, which interrups a loop run and starts the next run of the loop (after eventually executing any step code). In some languages continuation of the execution can be directed to a label given with the continue expression.

Structure

If the continue statement is given a label it is set in the label attribute.

Language specific notes

None

expr

A generic expression.

Structure

Expressions are very specific to the underlying programming language and provide a mean of abstraction for new languages. So if you want to work with expressions in a generic way don't go deeper than this tag. If you restrict your program to a specific language you can specify a set of valid child element expressions.

Language specific notes

field_access

An object-oriented field access. In most languages such a statement looks like obj.something.

Structure

A field_access requires an identifier for the accessed field, as well as an expr for the object to which the field belongs.

Language specific notes

C++ for struct access through a pointer the struct attribute is set to true.

Example

```
foo().bar =>
    <field_access>
        <identifier name="bar"/>
        <expr><call name="foo">...</call></expr>
    </field_access>

Java: Classname.super.field =>
    <field_access>
        <identifier name="field"/>
        <expr><identifier name="super" namespace="Classname"/></expr>
    </field_access>

C++: struct access through pointer:
p->field =>
    <field_access struct="true">
        <identifier name="field"/>
        <expr><identifier name="p"/></expr>
    </field_access>
```

finally

Used in try.

Structure

See try.

Language specific notes

See try.

fixed

A fixed statement which pins objects to their memory location.

Structure

A fixed tag contains a variable and a block during which that variable remains fixed.

Language specific notes

C++ not available

Java not available

goto

A goto statement. In some languages goto statements are allowed and are used to jump to a certain position in the code marked with a label.

Structure

The jump target is specified with the label attribute.

Language specific notes

Java not available

group

Used in switch

Structure

See switch.

Language specific notes

See switch.

identifier

An identifier tag is used to represent simple identifier usages. This does not apply to more complicated constructs like method calls f.ex.

Structure

The name of the identifier is set in the name attribute.

Language specific notes

C++ the construct ::x to access a variable in the global namespace is represented with a special value of the namespace attribute: ".global." As '.' is a reserved character for a namespace name and is used to separate namespaces this is guaranteed to not occur in a document otherwise.

if

The well-known if statement consisting of a condition, a then block and an else block.

To reduce the tree depth, statements of the form if (..) then {..} else if (..) then {..} else if (..) .... can all be included in one if tag with multiple pairs of condition and block elements. See the example below for a demonstration.

Ternary conditions (usually syntactically denoted with the ?: operator) are a special kind of if statements and are represented by if elements with @type="ternary" (tree flattening is usually not performed in such cases, as such statements are very hard to read already and it is preferable to clearly see the structure especially as the tree of ternary conditions usually isn't anywhere as deep as that of nested else-if statements.)

Structure

A list of condition, block pairs followed by an optional block.
The type attribute is used for ternary ifs.

Language specific notes

None

Example

```
if (cond1) then expr1;
else if (cond2) then expr2;
else expr3;

<if>
  <condition><!-- cond1 --></condition>
  <block><!-- expr1 --></block>
  <condition><!-- cond2 --></condition>
  <block><!-- expr2 --></block>
  <block><!-- expr3 --></block>
</if>
```

import

Import statements, which import new names into the parent namespace. Imports exist in various types, distinguishable by attributes (see the examples).

Structure

import tags make use of the attributes on_demand, alias, and type. additionally import elements may have a modifiers or expr child element.

Language specific notes

C++ using imports take an additional expression for what is imported. no static, single type or aliased imports.

C# using imports like in C++. no static imports.

Java no using imports. no aliased imports.

Example

```
import de.srcml.*; =>
  <import name="de.srcml" on_demand="true"/>

import de.srcml.SrcMLPlugin; =>
  <import name="de.srcml.SrcMLPlugin"/>

import static java.lang.Math.abs; =>
  <import name="java.lang.Math.abs">
    <modifiers><modifier name="static"/></modifiers>
  </import>
```

```
import static java.lang.Math.*; =>
  <import name="java.lang.Math" on_demand="true">
    <modifiers><modifier name="static"/></modifiers>
  </import>
```

imports

Groups together various import elements.

Structure

List of import elements.

Language specific notes

None

index

This is and described in array_access

Structure

See array_access

Language specific notes

See array_access

inheritance

General concept of a type inheritance. Used for implementation inheritance as well as for pure type inheritance.

Structure

A type can inherit several other types. Some languages also allow several types of inheritance, or allow modifications of the inheritance process. Therefore every inherited type is contained in an inherits tag, which can have a type attribute and a modifiers child node.

Language specific notes

Java distinguishes between subclassing and implementation of interfaces. The type attribute uses the two corresponding values “implementation” and “type”.

C# same as Java.

C++ doesn’t know the concept of interfaces, making the type attribute redundant. allows modifiers to affect visibility of baseclass components which is represented by the modifiers tag.

inherits

Used in inheritance

Structure

See inheritance

Language specific notes

See inheritance

init

A kind of initializer which can be used at various places, where some kind of initialization is available (f.ex. variable initialization during declaration.)

Structure

Consists of an expr tag which holds the initializing expression.

Language specific notes

None

list

A list expression creates some kind of a list of expressions. This is most commonly used to create arrays in initialization statements, but can also be used for other kinds of lists, tuples, etc.

As the most common usage seems to be arrays the type attribute is set to default to "array"

Structure

Contains a list of expr tags and a type attribute specifying the type of list.

Language specific notes

None

loop

Represents all types of loops. The actual loop type is stored in the type attribute and currently possible values are: for, foreach, while, until

Every loop can have a condition determining when the loop will be exited again and a loop body which contains the expressions to execute each time the loop is executed. Some loops additionally allow for initializing code and/or code responsible for computing the steps (f.ex. variable incrementation after each loop run).

Structure

In a loop element the first occuring block represents the loop body, while a second block should have type="loop_initializer" or type="loop_step" set. If both - initializer and step - blocks occur it is recommended to place the initializer block before the step block, though this is not strictly necessary due to the unique type attribute.

In some languages it is also possible to declare variables in the initializer part of the loop and those variables are valid throughout the other parts of the loop. This can be represented, by moving the declaration from the loop initializer block to a variables child element of loop.

Due to the various possible types of loops most elements are optional. The only exception to this is the loop body represented with the first block element, as every loop has to have a body (as looping over nothing doesn't make much sense).

Language specific notes

None

Example

```
for (int i = 0; i <= 10; i += 1) { ... } =>
    <loop type="for>
        <variables>
            <variable name="i">
                <type name="int"/>
                <init>
                    <expr><constant value="0"/></expr>
                </init>
            </variable>
        </variables>
        <condition>
            <expr><operation operator="&lt;=">
                <operand>
                    <expr><identifier name="i"/></expr>
                </operand>
                <operand>
                    <expr><constant value="10"/></expr>
                </operand>
            </operation></expr>
        </condition>
        <block>...</block>
        <block type="loop_step">
            <expr><assignment operator="+=">
                <lvalue>
                    <expr><identifier name="i"/></expr>
                </lvalue>
                <rvalue>
                    <expr><constant value="1"/></expr>
                </rvalue>
            </assignment></expr>
        </block>
    </loop>
```

In Java 1.5 enhanced for loops have been introduced for easier looping over Iterable objects:

```
for (Object o : list_of_objects) { ... } =>
    <loop type="foreach">
        <variables>
            <variable name="o">
                <type name="Object"/>
                <init>
                    <expr><identifier name="list_of_objects"/></expr>
                </init>
            </variable>
        </variables>
        <block>
            ...
        </block>
```

```
</loop>
```

method

A method is a sequence of statements. A method has a name, return type and can have any number of parameters.

Structure

The statements are kept in a separate block. The parameters of a method are represented as variables (with idref set to the method's id if available). The type element represents the method's return type (optional, because untyped languages do not have return types of course, also constructors don't have return types)

The value tag can be used when a certain value can be attached to a method in some way. Currently the only known usage for this seems to be the default value of an annotation type's element in Java.

The throws element which contains a list of type elements is used for exceptions which can be thrown by the method.

As in some languages methods can be parameterized the parameterization element will be able to store that information.

The type attribute is used to highlight special methods like constructors or destructors.

The baseinit element is used for C++/C# style initializations: foo(string name) : base(name)

Language specific notes

C# Indexers are represented as methods with name="this"

modifier

Used in modifiers

Structure

See modifiers.

Language specific notes

See modifiers.

modifiers

This tag is used for all kinds of modifiers, but primarily for the typical access modifiers public, private, and protected.

Structure

All modifiers are grouped together by this tag and each modifier is represented by its own modifier tag which contains a name attribute containing the actual modifier.

Language specific notes

None

namespace

A namespace contains all kinds of declarations and imports.

Structure

A namespace consists of several imports followed by a collection of type_decl and block elements in any order.

Language specific notes

Java

- only one namespace element under unit should be present in java.
- the name attribute should be set to the given package.

C++ the alias attribute can be used for aliasing namespace statements like: namespace NS1 = NS2;

new

Creation of a new instance of a type. The type can be a generic type, in which case a parameterization can be specified.

Structure

Arguments passed to the type constructor are handled in the arguments element accordingly. The init element can be used in cases where the instance creation is also taking some initializer expression as well (f.ex. creating a new array).

For anonymous type declaration the type_decl element is used with its anonymous attribute set to "true".

Language specific notes

Java for non-static inner classes there is a seldomly occuring construct:

```
class A { class B {} }
void f() { A a = new A(); a.new B(); }
```

The representation of the expression a in this new statement is stored in the base element.

C++ The placement element is used for the special new statements in C++ including placement arguments.

nop

The typical no-operation operation. For some languages this is required to be used syntactically for some reasons, so we also represent it in SrcML.

Structure

No real content.

Language specific notes

None

operation

A generic operation element for all kinds of operations involving arguments and an operator.

Structure

The affix attribute defaults to “infix”, but could also be “prefix” or “postfix”. Some operations take an actual type as argument, which is why we extend the content with possible type elements here. Type casts are one example for such an operation, in which case the first child element is the type element and the second child element the expr which is to be casted to the given type. The operator attribute is set to “cast” in such a case (or for C++ also “static_cast”, “dynamic_cast” and so on.)

Usually however there are only expr child elements involved for the operands of the operation.

Language specific notes

Java instanceof expressions are represented as operation with the attribute operator set to “instanceof”, the first child element being the expression to test and the second the type.

C# for statements like int* a = stackalloc int[10]; the stackalloc is represented as an operation with operator=”stackalloc”.

parameterization

A parameterization can be used in all kinds of types. Also used for parameterizing classes, interfaces, etc.

Structure

See type for some usage examples. The type attribute distinguishes between different forms of parameterization with the following types currently being in use: extends, super, constraint, method

Language specific notes

C++ type=”class” is used for constructs of the kind template<class T>... type=”method” is used for function pointer types (see type)

pointer

Used in type.

Structure

See type.

Language specific notes

See type.

return

A return statement which can return an expr to the caller of the method.

Structure

Some programming languages also allow for a method to continue processing after such a statement when the method is called again. As such a feature is usually called a yield statement we will set the type attribute to yield in those cases.

Language specific notes

None

switch

A switch statement which takes an expr argument and consists of groups of case conditions and a block each.

Structure

Each group element can have several expr child elements denoting the various cases this group accounts for. After that a block child element is used for the actual statements to execute in those cases.

A group can be marked as the default group with the boolean default attribute.

Language specific notes

None

synchronized

A synchronization block which takes an object to synchronize over.

Structure

Contains an expr tag for the object to synchronize over and the actual block which is synchronized.

Language specific notes

C++ not available

C# not available

tag

A tag occuring inside a comment (javadoc, doxygen, ...).

Structure

Tags consist of a name attribute and text descriptions.

Language specific notes

None

throw

A throw statement takes an expression which evaluates to some exception object.

Structure

Only needs the expr element describing the exception thrown.

Language specific notes

None

throws

Used in method

Structure

See method.

Language specific notes

See method.

try

A try/catch statement, i.e. a block in which exceptions may be thrown and which cause the corresponding catch block to be executed.

Structure

There can be multiple catch elements for one try block to catch different exceptions. A catch element declares a variables element for the exception caught and a block for the statements to execute in such a case.

A try/catch block sometimes also supports a finally construct which is always executed, no matter if an exception was caught or not. As a finally element is also legal without any catch elements both catch and finally elements are optional.

Language specific notes

C# allows catch with no exception type given (defaults to catch(System.Exception)) hence the variables element is optional.

type

A generic representation for a type.

Structure

The intentions of the structure are best explained by the examples below.

Language specific notes

C# jagged arrays are represented like rectangular arrays with an additional jagged="true" set.

C++ Types explicitly marked up with the typename keyword should set the typename attribute to "true".

Example

```
int => <type name="int"/>

int[] => <type name="int" dimensions="1"/>

int[10][3] => <type name="int" dimensions="2">
    <type_array_size dimension="0">
      <expr><constant value="10"/></expr>
    </type_array_size>
    <type_array_size dimension="1">
      <expr><constant value="3"/></expr>
    </type_array_size>
    </type>

de.srcml.MyClass => <type name="MyClass" namespace="de.srcml"/>

List<Integer> => <type name="List">
    <parameterization>
        <type name="Integer"/>
    </parameterization>
    </type>

some::name::space::MyClass<GenType>::MyType[3] =>
    <type name="MyType" dimensions="1">
    <type name="MyClass" namespace="some.name.space">
        <parameterization>
            <type name="GenType"/>
        </parameterization>
    </type>
    <parameterization>
        <type name="GenType"/>
    </parameterization>
    <type_array_size dimension="0">
      <expr><constant value="3"/></expr>
    </type_array_size>
    </type>

Map<SomeClass, SomeInterface> => <type name="Map">
    <parameterization>
        <type name="SomeClass"/>
        <type name="SomeInterface"/>
    </parameterization>
    </type>

List<? extends Something> => <type name="List">
    <parameterization>
        <type name="?">
            <parameterization type="extends">
                <type name="Something"/>
            </parameterization>
        </type>
    </parameterization>
    </type>

List<? super Object> => <type name="List">
    <parameterization>
        <type name="?">
            <parameterization type="super">
                <type name="Object"/>
            </parameterization>
        </type>
    </parameterization>
    </type>
```

```
List<List<String>> => <type name="List">
    <parameterization>
        <type name="List">
            <parameterization>
                <type name="String"/>
            </parameterization>
        </type>
    </parameterization>
    </type>

SomeClass<T> where T: IOne, ITwo => ... <parameterization>
    <type name="T">
        <parameterization type="constraint">
            <type name="IOne"/>
            <type name="ITwo"/>
        </parameterization>
    </type>
    </parameterization>

char * * const ppc =>
<type name="char">
    <pointer/>
    <pointer const="true"/>
</type>

char *&rpc =>
<type name="char" reference="true">
    <pointer/>
</type>

int (*fp)(char*) =>
<type name="fp">
    <parameterization type="method">
        <type name="int"/>
        <type name="char">
            <pointer/>
        </type>
    </parameterization>
    <pointer/>
</type>
```

type_array_size

Used in type.

Structure

See type.

Language specific notes

See type.

type_decl

This tag represents a type declaration. The various possible type declarations available can be specified by the type attribute. A declaration of a type can use other types: as a means of parameterizing the new type, or by declaring a new subtype based on other types.

Structure

In some languages types can be declared as anonymous (in that they are not affiliated with a name). Usually this is done somewhere inside of expressions. In those cases the anonymous tag should be set to true.

A type declaration can have several of the following child elements:

modifiers	visibility modifiers, language specific modifiers like abstract, static, ...
parameterization	allows type to be declared generically (f.ex. container classes)
inheritance	A type can inherit certain features from other types. How the inheritance actually influences the semantic of the newly declared type depends highly on the underlying language and its semantic.
values	Some types can be declared by naming all their possible values (enums). These values have similar declaration possibilities to the general type declarations, which is the reason, why the value element is chosen to be a type_decl element.
alias	A type declaration can be used to define an alias for another type
variables	A type can contain variables (most commonly found in classes)
method	A type can define methods. For the matter of the syntactic markup it doesn't matter if a method is only declared, or also implemented. The need for implementations is dependent on the underlying language.
type_decl	Types can declare new sub-types recursively. (f.ex. inner classes in Java)
block	blocks of code usually responsible for creating an initial state of the declared type (marked as block[@type="initializer"])

The following types of type declarations have already been considered:

"class"	the well-known class type which can have baseclasses, implement interfaces and which contains variable and method declarations
"interface"	similar to the class type on a syntactical level. methods should only be declared, not implemented (not enforced by schema)
"enum"	an enumeration type, which has properties similar to "class", but also contains the enumeration values. initialization arguments of enum values are specified in the init element of the respective value element.
"annotation"	a special annotation interface (in the Java language) or an attribute declaration (in C# language). Such a type can be used to annotate other constructs. (see annotation)
"struct"	similar to the class type. structs don't have access modifiers for their members (not enforced by schema)
"union"	similar to struct. semantics differ, but schema does not include any semantics whatsoever.
"typedef"	declares an alias type for a given type
"method"	declares a method type (used for C# delegates)
"value"	not really a type, but a single value of a type (possibly including it's own behavior in form of variables, methods, ..)
"event"	special event type available in C#

Language specific notes

Java elements of annotation types are represented with method (possible default values can be represented with method/value)

C#

- attributes are represented like classes except that the type attribute is set to “annotation”.
- delegates are represented with a method. The name of the delegate is stored in name of the type_decl node (type_decl/method[@name] needs not to be used)

unit

A compilation unit which usually is equal to one source code file.

Structure

This element only contains the global namespace.

Language specific notes

Java a package statement will modify namespace/@name to match the given package

value

Used in method

Structure

See method.

Language specific notes

See method.

value_pair

A value pair is an assignment pair of two expressions. In some languages it is used to initialize Maps with key and value pairs.

Structure

Contains two expr elements.

Language specific notes

Java value pairs are used in annotations: @annotation(member=value)

values

Used in type_decl.

Structure

See type_decl.

Language specific notes

See type_decl.

variable

Declaration of a variable. This includes *variables* like method parameters, catch parameters, etc.

Structure

As untyped languages should be supported too the specification of a type is optional. Several languages also allow initializations of newly declared variables, which is supported by the optional init tag.

Language specific notes

Java Representation of variable method arguments:

```
method(int ... a) =>
 ...
 <variables>
     <variable name="a" varargs="true">
         <type name="int" dimensions="1"/>
     </variable>
 </variables>
```

C# Properties are represented as variables with @type=”property” and a block element. The corresponding getters and setters are represented as method declarations with no arguments and no return type element.

variables

Can be used for anything declaring a variable for some namespace. This concept includes normal class/member variables, as well as method parameters or catch parameters.

Structure

The idref attribute (if available) points to the namespace or element in which the declared variables are valid.

The modifiers element holds the modifiers which are valid for all of the variable child elements.

Language specific notes

None

C Source code examples

```
public class CFGCreationAdapter implements ICFGCreation {
    private Element element;

    public CFGCreationAdapter(Element e) {
        element = e;
    }

    public void createEdges(CFGCreationManager mgr) {
        SrcMLPlugin.logger.warning(
            "This␣method␣should␣not␣be␣called.");
    }

    public CFG createGraph(CFGCreationManager mgr) {
        assert "switch".equals(element.getName());
        CFG cfg = new CFG();
        Element expr = CoreHelper.selectSingleXPath(
                CoreHelper.SRCML_NAMESPACE,
                "SrcML:expr[1]", element);
        CFG subcfg = mgr.createSubgraph(expr);
        CFG initcfg = subcfg;
        cfg.addVertex(subcfg);
        cfg.addEdge(cfg.ENTRY, subcfg);
        subcfg = null;
        for (Element grp : CoreHelper.selectXPath(
                CoreHelper.SRCML_NAMESPACE,
                "SrcML:group/SrcML:block", element)) {
            CFG cfg1 = mgr.createSubgraph(grp);
            cfg.addVertex(cfg1);
            String label = null;
            for(Element ex : CoreHelper.selectXPath(
                    CoreHelper.SRCML_NAMESPACE,
                    "../SrcML:expr", grp)) {
                if (label == null)
                    label = Facade.createStringPresentation(ex);
                else
                    label += "␣/␣"
                        + Facade.createStringPresentation(ex);
            }
            if (grp.attributeValue("default") != null) {
                if (label == null) label = "default";
                else label += "␣/␣default";
            }
            cfg.addEdge(new LabeledEdge(initcfg, cfg1, label));
            if (subcfg != null) cfg.addEdge(subcfg, cfg1);
            subcfg = cfg1;
        }
        if (subcfg != null) cfg.addEdge(subcfg, cfg.EXIT);
        // add direct edge to EXIT if there is no default block:
        Element def = CoreHelper.selectSingleXPath(
                CoreHelper.SRCML_NAMESPACE,
                "SrcML:group[@default]", element);
        if (def == null) {
            cfg.addEdge(initcfg, cfg.EXIT);
        }
        return cfg;
    }

    public boolean hasCFG() {
        if ("switch".equals(element.getName())) return true;
        return false;
    }
```

```

}
```

Listing 35: ICFGCreation adapter for new switch statement

```
package de.srcml.cfg;

import java.util.*;

import org._3pq.jgrapht.edge.DirectedEdge;
import org.dom4j.Element;
import org.eclipse.core.runtime.Platform;

import de.srcml.AdapterFactoryManager;
import de.srcml.SrcMLPlugin;
import de.srcml.cfg.CFG.ReplacementNode;
import de.srcml.cfg.edge.LabeledEdge;

/**
 * The CFG creation manager supervises the construction of a CFG from
 * a SrcML document.
 *
 * @author Frank Raiser
 */
public class CFGCreationManager extends AdapterFactoryManager {

    /**
     * A mapping from dom4j elements to their respective CFGs
     */
    private Map<Element, CFG> mapElement2CFG;
    /**
     * A list of dom4j elements which still have dangling edges to be
     * created.
     */
    private List<ICFGCreation> edgeCreationQueue;

    private static CFGCreationManager instance;

    public static CFGCreationManager getDefault() {
        if (instance == null) {
            // create default instance:
            instance = new CFGCreationManager();
            // set default ICFGCreation factories:
            for (String id : CFGHelper.getCFGCreationFactoryIDs(null))
            {
                instance.addDefaultFactory(
                    id, CFGHelper.loadFactory(id));
            }
            // TODO: not a good solution. maybe add another method for
            // outside calls to call begin/endUsingFactories there ?
            instance.beginUsingFactories(instance.defaultFactories);
        }
        return instance;
    }

    /**
     * This method expects a dom4j element for which to create a CFG
     * using the default ICFGCreation factories.
     *
     * @param element a dom4j element to create the CFG for
     */
    public CFG createCFG(Element element) {
        mapElement2CFG = new HashMap<Element, CFG>();
        edgeCreationQueue = new ArrayList<ICFGCreation>();
        if (!"method".equals(element)) {
            SrcMLPlugin.logger.fine(
             "Request for creating a CFG for a non-method element. " +
             "Ignore this if it's occuring during the unit testing");
        }
        CFG cfg = createSubgraph(element);
        if (cfg == null) {
            // ops.. something bad happened, no need to
            // continue construction
            return null;
        }
        // now create any queued edges
        for (ICFGCreation cfgc : edgeCreationQueue) {
            cfgc.createEdges(this);
        }
        edgeCreationQueue.clear();
        // finally merge any unnecessary entry/exit nodes:
        flattenCFG(cfg);
        mapElement2CFG.clear();
        return cfg;
    }

    /**
     * This method creates a new CFG instance for a subgraph for the
     * given element. The mapping from elements to CFGs will also be
     * updated accordingly.
     *
     * @param e the element for which to create the subgraph
     * @return the CFG for the subgraph
     */
    public CFG createSubgraph(Element e) {
        Object o = Platform.getAdapterManager().getAdapter(
            e, ICFGCreation.class);
        if (o instanceof ICFGCreation) {
            ICFGCreation creation = (ICFGCreation)o;
            if (!creation.hasCFG()) return null;
            CFG res = creation.createGraph(this);
            if (res != null) {
                // register the graph for this element:
                mapElement2CFG.put(e, res);
                return res;
            }
            SrcMLPlugin.logger.info(
                    "Invalid CFG created for element: " + e +
                    " by ICFGCreation: " + creation);
        }
        SrcMLPlugin.logger.fine(
            "No ICFGCreation adapter available for element: " + e);
        return null;
    }

    /**
     * Returns the CFG for a given element during the construction of
     * the CFG. This method should primarily be called from the
     * {@see ICFGCreation#createEdges(CFGCreationManager)} methods, as
     * for earlier calls the CFGs might not exist for all required
     * elements and after the flattening the Element-&gt;CFG
     * association is lost.
     *
     * @param e the element for which to find the CFG
     * @return the CFG for the given element or <code>null</code>
     */
    public CFG getCFGForElement(Element e) {
        return mapElement2CFG.get(e);
    }

    /**
     * When a CFG is initially created some edges may point to places
     * no CFG has been created for yet in which case such an element
     * can register for being called back later, when all CFGs are
     * created.<br/>
     * After constructing the complete CFGs all queued
     * {@see ICFGCreation#createEdges(CFGCreationManager)} methods
     * will be called.
     *
     * @param cfgc the ICGFCreation to be called back later
     */
    public void queueForEdgeCreation(ICFGCreation cfgc) {
        if (!edgeCreationQueue.contains(cfgc)) {
            edgeCreationQueue.add(cfgc);
        }
    }

    /**
     * Helper method to redirect targets of all matching edges. Every
     * edge with target <code>oldTarget</code> will then point to
     * <code>newTarget</code> instead.
     *
     * @param cfg the CFG in which to redirect the edges
     * @param oldTarget edges which point to this get redirected
     * @param newTarget the new target for the redirected edges
     */
    protected static void redirectEdgeTarget(
            CFG cfg, Object oldTarget, Object newTarget) {
        // first run: create edges to new target
        for (Object o : cfg.incomingEdgesOf(oldTarget)) {
            Object label = null;
            if (o instanceof LabeledEdge)
                label = ((LabeledEdge)o).getLabel();
            if (o instanceof DirectedEdge) {
                DirectedEdge de = (DirectedEdge)o;
                cfg.addEdge(new LabeledEdge(
                    de.getSource(), newTarget, label));
            }
        }
        // second run: remove edges to old target:
        Object [] edges = cfg.incomingEdgesOf(oldTarget).toArray();
        for (Object e : edges) {
            if (e instanceof DirectedEdge) {
                cfg.removeEdge((DirectedEdge)e);
            }
        }
    }

    /**
     * Helper method to redirect sources of all matching edges.
     * Every edge with source <code>oldSource</code> will then start
     * from <code>newSource</code> instead.
     *
     * @param cfg the CFG in which to redirect the edges
     * @param oldSource edges which start from this get redirected
     * @param newSource the new source for the redirected edges
     */
    protected static void redirectEdgeSource(
            CFG cfg, Object oldSource, Object newSource) {
        // first run: create edges from new source:
        for (Object o : cfg.outgoingEdgesOf(oldSource)) {
            Object label = null;
            if (o instanceof LabeledEdge)
                label = ((LabeledEdge)o).getLabel();
            if (o instanceof DirectedEdge) {
                DirectedEdge de = (DirectedEdge)o;
                cfg.addEdge(new LabeledEdge(
                    newSource, de.getTarget(), label));
            }
        }
        // second run: remove edges from old source:
        Object [] edges = cfg.outgoingEdgesOf(oldSource).toArray();
        for (Object e : edges) {
            if (e instanceof DirectedEdge) {
                cfg.removeEdge((DirectedEdge)e);
            }
        }
    }

    /**
     * Helper method to collapse a vertex, such that edges pointing to
     * it will instead point to all vertices directly reachable from
     * the collapsed vertex and all edges targeting the collapsed
     * vertex will be copied to point to all vertices reachable from
     * the collapsed vertex. After creating the new edges the vertex
     * will be removed from the CFG.
     *
     * @param cfg the CFG in which to collapse the vertex
     * @param v the vertex to collapse
     */
    protected static void collapseVertex(CFG cfg, Object v) {
        // phase 1 : incoming edges
        for (Object o : cfg.incomingEdgesOf(v)) {
            Object label1 = null;
            if (o instanceof LabeledEdge)
                label1 = ((LabeledEdge)o).getLabel();
            if (o instanceof DirectedEdge) {
                DirectedEdge de = (DirectedEdge)o;
                // now create new edges for every vertex
                // reachable from v:
                for (Object o2 : cfg.outgoingEdgesOf(v)) {
                    Object label2 = null;
                    if (o2 instanceof LabeledEdge)
                        label2 = ((LabeledEdge)o2).getLabel();
                    if (o2 instanceof DirectedEdge) {
                        DirectedEdge de2 = (DirectedEdge)o2;
                        Object label =
                            label1 != null ? label1 : label2;
                        if (label1 != null && label2 != null) {
                            // oh oh.. we cannot preserve both labels
                            // here, so we just concat their strings
                            label = label1.toString() + "<>" +
                                    label2.toString();
                        }
                        cfg.addEdge(new LabeledEdge(
                            de.getSource(), de2.getTarget(), label));
                    }
                }
            }
        }
        // finally remove the vertex v (and all connected edges)
        cfg.removeVertex(v);
    }

    /**
     * Helper method which creates new edges for every edge going to
     * <code>target</code> such that the new edges go to all edges
     * directly reachable from <code>target</code> instead. All
     * incoming edges for <code>target</code> will be removed.
     *
     * @param cfg the CFG on which to perform the operation
     * @param target the target vertex
     */
    protected static void edgeTransitivityIncoming(
            CFG cfg, Object target) {
        assert !cfg.containsEdge(target, target);
        for (Object o : cfg.incomingEdgesOf(target)) {
            Object label1 = null;
            if (o instanceof LabeledEdge)
                label1 = ((LabeledEdge)o).getLabel();
            if (o instanceof DirectedEdge) {
                DirectedEdge de = (DirectedEdge)o;
                for (Object o2 : cfg.outgoingEdgesOf(target)) {
                    Object label2 = null;
                    if (o2 instanceof LabeledEdge)
                        label2 = ((LabeledEdge)o2).getLabel();
                    if (o2 instanceof DirectedEdge) {
                        DirectedEdge de2 = (DirectedEdge)o2;
                        Object label =
                            label1 != null ? label1 : label2;
                        if (label1 != null && label2 != null) {
                            // oh oh.. we cannot preserve both labels
                            // here, so we just concat their strings
                            label = label1.toString() + "<>" +
                                    label2.toString();
                        }
                        cfg.addEdge(new LabeledEdge(
                            de.getSource(), de2.getTarget(), label));
                    }
                }
            }
        }
        // now remove all incoming edges
        Object[] edges = cfg.incomingEdgesOf(target).toArray();
        for (Object o : edges) {
            if (o instanceof DirectedEdge) {
                cfg.removeEdge((DirectedEdge)o);
            }
        }
    }

    /**
     * Helper method which creates new edges for every edge leaving
     * from <code>target</code> such that the new edges all start at
     * edges directly reaching <code>target</code> instead. All
     * outgoing edges for <code>target</code> will be removed.
     *
     * @param cfg the CFG on which to perform the operation
     * @param target the target vertex
     */
    protected static void edgeTransitivityOutgoing(
            CFG cfg, Object target) {
        assert !cfg.containsEdge(target, target);
        for (Object o : cfg.outgoingEdgesOf(target)) {
            Object label1 = null;
            if (o instanceof LabeledEdge)
                label1 = ((LabeledEdge)o).getLabel();
            if (o instanceof DirectedEdge) {
                DirectedEdge de = (DirectedEdge)o;
                for (Object o2 : cfg.incomingEdgesOf(target)) {
                    Object label2 = null;
                    if (o2 instanceof LabeledEdge)
                        label2 = ((LabeledEdge)o2).getLabel();
                    if (o2 instanceof DirectedEdge) {
                        DirectedEdge de2 = (DirectedEdge)o2;
                        Object label =
                            label1 != null ? label1 : label2;
                        if (label1 != null && label2 != null) {
                            // oh oh.. we cannot preserve both labels
                            // here, so we just concat their strings
                            label = label1.toString() + "<>" +
                                    label2.toString();
                        }
                        cfg.addEdge(new LabeledEdge(
                            de2.getSource(), de.getTarget(), label));
                    }
                }
            }
        }
        // now remove all incoming edges
        Object [] edges = cfg.outgoingEdgesOf(target).toArray();
        for (Object o : edges) {
            if (o instanceof DirectedEdge) {
                cfg.removeEdge((DirectedEdge)o);
            }
        }
    }

    protected static void flattenCFG(CFG cfg) {
        // first we recursively flatten all subgraphs
        for (Object o : cfg.vertexSet()) {
            if (o instanceof CFG) {
                flattenCFG((CFG)o);
            }
        }
        // now we have to find all flattened subgraphs and flatten
        //  them into the main graph:
        List<CFG> subCFGs = new ArrayList<CFG>();
        for (Object o : cfg.vertexSet()) {
            if (o instanceof CFG) {
                subCFGs.add((CFG)o);
            }
        }
        // this loops performs the merge:
        for (CFG subcfg : subCFGs) {
                // now add all the nodes and edges from that subcfg
                // into our current CFG:
                cfg.addAllVertices(subcfg.vertexSet());
                cfg.addAllEdges(subcfg.edgeSet());
                // as we want to delete the subgraph node we first
                // have to redirect the edges connected to it:
                redirectEdgeTarget(cfg, subcfg, subcfg.ENTRY);
                redirectEdgeSource(cfg, subcfg, subcfg.EXIT);
                // now we can remove the subgraph's node:
                cfg.removeVertex(subcfg);
                // next we need to remove the ENTRY/EXIT nodes copied
                // from the subgraph
                collapseVertex(cfg, subcfg.ENTRY);
                collapseVertex(cfg, subcfg.EXIT);
        }

        checkReplacementNodes(cfg);
        // and finally calculate closure for ENTRY/EXIT nodes
        edgeTransitivityIncoming(cfg, cfg.ENTRY);
        edgeTransitivityOutgoing(cfg, cfg.EXIT);
    }

    /**
     * Helper method to check for any ReplacementNodes and if their
     * required element is already in the graph in which case the
     * replacement will occur.
     *
     * @param cfg the CFG in which to check
     */
    protected static void checkReplacementNodes(CFG cfg) {
        // finally find any remaining ReplacementNodes:
        List<CFG.ReplacementNode> repnodes =
            new ArrayList<CFG.ReplacementNode>();
        for (Object o : cfg.vertexSet()) {
            if (o instanceof CFG.ReplacementNode) {
                repnodes.add((ReplacementNode)o);
            }
        }
        // and now try to replace them if possible:
        for (CFG.ReplacementNode n : repnodes) {
            // check if the vertex for the replacement exists:
            if (cfg.containsVertex(n.getElement())) {
                redirectEdgeSource(cfg, n, n.getElement());
                redirectEdgeTarget(cfg, n, n.getElement());
                cfg.removeVertex(n);
            }
        }
    }
}
```

Listing 36: CFGCreationManager class

D SrcML examples

```
<?xml version="1.0" encoding="UTF-8"?>

<unit
  xmlns='http://srcml.de'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:meta='http://srcml.de/meta'
>
  <namespace>
    <type_decl
      id='CExample;'
      type='class'
      name='Example'
    >
      <modifiers>
        <modifier
      name='public'
     />
</modifiers>
<inheritance>
   <inherits
      type='implementation'
   >
      <type
         name='Object'
        />
   </inherits>
   <inherits
      type='type'
   >
      <type
         name='Cloneable'
        />
   </inherits>
</inheritance>
<method
   id='MExample;main(java.lang.String[])'
   name='main'
>
   <modifiers>
      <modifier
         name='public'
        />
      <modifier
         name='static'
        />
   </modifiers>
   <type
      name='void'
     />
   <variables>
      <variable
         dimensions='1'
      id='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
         varargs='true'
         name='args'
      >
         <type
            name='String'
           />
      </variable>
   </variables>
   <throws>
      <type
         name='Exception'
        />
   </throws>
   <block>
      <variables>
         <variable
            id='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
            name='op'
         >
            <type
        name='String'
       />
      <init>
        <expr>
          <array_access>
            <array>
              <expr>
                <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                  name='args'
                 />
              </expr>
            </array>
            <index>
              <expr>
                <constant
                  value='0'
                 />
              </expr>
            </index>
          </array_access>
        </expr>
      </init>
    </variable>
  </variables>
  <variables>
    <variable
      id='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
      name='a'
    >
      <type
        name='Integer'
       />
      <init>
        <expr>
          <call
            idref='Mjava.lang.Integer;parseInt(java.lang.String)'
            name='parseInt'
          >
            <callee>
              <expr>
                <identifier
                  idref='Cjava.lang.Integer;'
                  name='Integer'
                 />
              </expr>
            </callee>
            <arguments>
              <expr>
                <array_access>
                  <array>
                    <expr>
                      <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                        name='args'
                       />
                    </expr>
                  </array>
```

```
                    <index>
                      <expr>
                        <constant
                          value='1'
                         />
                      </expr>
                    </index>
                  </array_access>
                </expr>
              </arguments>
            </call>
          </expr>
        </init>
      </variable>
    </variables>
    <variables>
      <variable
        id='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
        name='b'
      >
        <type
          name='Integer'
         />
        <init>
          <expr>
            <call
              idref='Mjava.lang.Integer;parseInt(java.lang.String)'
              name='parseInt'
            >
              <callee>
                <expr>
                  <identifier
                    idref='Cjava.lang.Integer;'
                    name='Integer'
                   />
                </expr>
              </callee>
              <arguments>
                <expr>
                  <array_access>
                    <array>
                      <expr>
                        <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                          name='args'
                         />
                      </expr>
                    </array>
                    <index>
                      <expr>
                        <constant
                          value='2'
                         />
                      </expr>
                    </index>
                  </array_access>
                </expr>
              </arguments>
            </call>
         </expr>
      </init>
   </variable>
</variables>
<expr>
   <if>
      <condition>
         <expr>
            <call
               idref='Mjava.lang.String;equals(java.lang.Object)'
               name='equals'
            >
               <callee>
                  <expr>
                     <constant
                        value='plus'
                     >
                        <type
                           name='string'
                          />
                     </constant>
                  </expr>
               </callee>
               <arguments>
                  <expr>
                     <identifier
  idref='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
                        name='op'
                       />
                  </expr>
               </arguments>
            </call>
         </expr>
      </condition>
      <block>
         <expr>
            <call
               idref='Mjava.io.PrintStream;println(int)'
               name='println'
            >
               <callee>
                  <expr>
                     <identifier
                        namespace='System'
                  idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                        name='out'
                       />
                  </expr>
               </callee>
               <arguments>
                  <expr>
                     <operation
                        operator='+'
                     >
                        <expr>
                           <identifier
  idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
```

```
                   name='a'
                  />
               </expr>
               <expr>
                 <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                   name='b'
                  />
               </expr>
             </operation>
           </expr>
         </arguments>
       </call>
     </expr>
   </block>
   <condition>
     <expr>
       <call
         idref='Mjava.lang.String;equals(java.lang.Object)'
         name='equals'
       >
         <callee>
           <expr>
             <constant
               value='minus'
             >
               <type
                 name='string'
                />
             </constant>
           </expr>
         </callee>
         <arguments>
           <expr>
             <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
               name='op'
              />
           </expr>
         </arguments>
       </call>
     </expr>
   </condition>
   <block>
     <expr>
       <call
         idref='Mjava.io.PrintStream;println(int)'
         name='println'
       >
         <callee>
           <expr>
             <identifier
               namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
               name='out'
              />
           </expr>
         </callee>
```

```
                        <arguments>
                          <expr>
                            <operation
                              operator='-'
                            >
                              <expr>
                                <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                                  name='a'
                                 />
                              </expr>
                              <expr>
                                <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                                  name='b'
                                 />
                              </expr>
                            </operation>
                          </expr>
                        </arguments>
                      </call>
                    </expr>
                  </block>
                  <condition>
                    <expr>
                      <call
                        idref='Mjava.lang.String;equals(java.lang.Object)'
                        name='equals'
                      >
                        <callee>
                          <expr>
                            <constant
                              value='mul'
                            >
                              <type
                                name='string'
                               />
                            </constant>
                          </expr>
                        </callee>
                        <arguments>
                          <expr>
                            <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
                              name='op'
                             />
                          </expr>
                        </arguments>
                      </call>
                    </expr>
                  </condition>
                  <block>
                    <expr>
                      <call
                        idref='Mjava.io.PrintStream;println(int)'
                        name='println'
                      >
                        <callee>
                          <expr>
                            <identifier
                              namespace='System'
                          idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                              name='out'
                             />
                          </expr>
                        </callee>
                        <arguments>
                          <expr>
                            <operation
                              operator='*'
                            >
                              <expr>
                                <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                                 name='a'
                                />
                              </expr>
                              <expr>
                                <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                                 name='b'
                                />
                              </expr>
                            </operation>
                          </expr>
                        </arguments>
                      </call>
                    </expr>
                  </block>
                  <condition>
                    <expr>
                      <call
                        idref='Mjava.lang.String;equals(java.lang.Object)'
                        name='equals'
                      >
                        <callee>
                          <expr>
                            <constant
                              value='div'
                            >
                              <type
                                name='string'
                               />
                            </constant>
                          </expr>
                        </callee>
                        <arguments>
                          <expr>
                            <identifier
           idref='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
                              name='op'
                             />
                          </expr>
                        </arguments>
                      </call>
                    </expr>
```

```
   </condition>
   <block>
     <expr>
       <call
         idref='Mjava.io.PrintStream;println(int)'
         name='println'
       >
         <callee>
           <expr>
             <identifier
               namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
               name='out'
              />
           </expr>
         </callee>
         <arguments>
           <expr>
             <operation
               operator='/'
             >
               <expr>
                 <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                   name='a'
                  />
               </expr>
               <expr>
                 <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                   name='b'
                  />
               </expr>
             </operation>
           </expr>
         </arguments>
       </call>
     </expr>
   </block>
   <block>
     <expr>
       <call
         idref='Mjava.io.PrintStream;println(java.lang.String)'
         name='println'
       >
         <callee>
           <expr>
             <identifier
               namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;err;'
               name='err'
              />
           </expr>
         </callee>
         <arguments>
           <expr>
             <constant
               value='unknown operation'
```

```
                        >
                          <type
                            name='string'
                           />
                        </constant>
                      </expr>
                    </arguments>
                  </call>
                </expr>
              </block>
            </if>
          </expr>
        </block>
      </method>
    </type_decl>
  </namespace>
  <meta:languages
    base='java'
   />
</unit>
```

Listing 37: arithmetic example in SrcML

```
<?xml version="1.0" encoding="UTF-8"?>

<unit
  xmlns='http://srcml.de'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:meta='http://srcml.de/meta'
  xmlns:ext='http://srcml.de/java/ext'
>
  <namespace>
    <type_decl
      id='CExample;'
      type='class'
      name='Example'
    >
      <modifiers>
        <modifier
          name='public'
         />
      </modifiers>
      <inheritance>
        <inherits
          type='implementation'
        >
          <type
            name='Object'
           />
        </inherits>
        <inherits
          type='type'
        >
          <type
            name='Cloneable'
           />
        </inherits>
      </inheritance>
      <method
```

```
          id='MExample;main(java.lang.String[])'
          name='main'
        >
          <modifiers>
            <modifier
              name='public'
             />
            <modifier
              name='static'
             />
          </modifiers>
          <type
            name='void'
           />
          <variables>
            <variable
              dimensions='1'
            id='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
              varargs='true'
              name='args'
            >
              <type
                name='String'
               />
            </variable>
          </variables>
          <throws>
            <type
              name='Exception'
             />
          </throws>
          <block>
            <variables>
              <variable
                id='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
                name='op'
              >
                <type
                  name='String'
                 />
                <init>
                  <expr>
                    <array_access>
                      <array>
                        <expr>
                          <identifier
         idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                            name='args'
                           />
                        </expr>
                      </array>
                      <index>
                        <expr>
                          <constant
                            value='0'
                           />
                        </expr>
                      </index>
                    </array_access>
                  </expr>
                </init>
              </variable>
            </variables>
            <variables>
              <variable
                id='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                name='a'
              >
                <type
                  name='Integer'
                 />
                <init>
                  <expr>
                    <call
                      idref='Mjava.lang.Integer;parseInt(java.lang.String)'
                      name='parseInt'
                    >
                      <callee>
                        <expr>
                          <identifier
                            idref='Cjava.lang.Integer;'
                            name='Integer'
                           />
                        </expr>
                      </callee>
                      <arguments>
                        <expr>
                          <array_access>
                            <array>
                              <expr>
                                <identifier
         idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                                  name='args'
                                />
                              </expr>
                            </array>
                            <index>
                              <expr>
                                <constant
                                  value='1'
                                />
                              </expr>
                            </index>
                          </array_access>
                        </expr>
                      </arguments>
                    </call>
                  </expr>
                </init>
              </variable>
            </variables>
            <variables>
              <variable
                id='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                name='b'
              >
                <type
                  name='Integer'
                 />
                <init>
                  <expr>
                    <call
                      idref='Mjava.lang.Integer;parseInt(java.lang.String)'
                      name='parseInt'
                    >
                      <callee>
                        <expr>
                          <identifier
                            idref='Cjava.lang.Integer;'
                            name='Integer'
                           />
                        </expr>
                      </callee>
                      <arguments>
                        <expr>
                          <array_access>
                            <array>
                              <expr>
                                <identifier
         idref='VMExample;main(java.lang.String[])#Cjava.lang.String[];args;'
                                  name='args'
                                />
                              </expr>
                            </array>
                            <index>
                              <expr>
                                <constant
                                  value='2'
                                />
                              </expr>
                            </index>
                          </array_access>
                        </expr>
                      </arguments>
                    </call>
                  </expr>
                </init>
              </variable>
            </variables>
            <expr>
              <ext:switch>
                <expr>
                  <identifier
             idref='VMExample;main(java.lang.String[])#Cjava.lang.String;op;'
                    name='op'
                   />
                </expr>
                <group>
                  <expr>
                    <constant
                      value='plus'
                    >
                      <type
                        name='string'
```

```
               />
          </constant>
       </expr>
       <block>
          <expr>
             <call
                idref='Mjava.io.PrintStream;println(int)'
                name='println'
             >
                <callee>
                   <expr>
                      <identifier
                         namespace='System'
                idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                         name='out'
                        />
                   </expr>
                </callee>
                <arguments>
                   <expr>
                      <operation
                         operator='+'
                      >
                         <expr>
                            <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                               name='a'
                             />
                         </expr>
                         <expr>
                            <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                               name='b'
                             />
                         </expr>
                      </operation>
                   </expr>
                </arguments>
             </call>
          </expr>
          <expr>
             <break />
          </expr>
       </block>
    </group>
    <group>
       <expr>
          <constant
             value='minus'
          >
             <type
                name='string'
               />
          </constant>
       </expr>
       <block>
          <expr>
             <call
```

```
           idref='Mjava.io.PrintStream;println(int)'
           name='println'
         >
           <callee>
             <expr>
               <identifier
                 namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                 name='out'
                />
             </expr>
           </callee>
           <arguments>
             <expr>
               <operation
                 operator='-'
               >
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                     name='a'
                    />
                 </expr>
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                     name='b'
                    />
                 </expr>
               </operation>
             </expr>
           </arguments>
         </call>
       </expr>
       <expr>
         <break />
       </expr>
     </block>
   </group>
   <group>
     <expr>
       <constant
         value='mul'
       >
         <type
           name='string'
          />
       </constant>
     </expr>
     <block>
       <expr>
         <call
           idref='Mjava.io.PrintStream;println(int)'
           name='println'
         >
           <callee>
             <expr>
               <identifier
```

```
                 namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                 name='out'
                /> 
             </expr>
           </callee>
           <arguments>
             <expr>
               <operation
                 operator='*'
               >
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                     name='a'
                    />
                 </expr>
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                     name='b'
                    />
                 </expr>
               </operation>
             </expr>
           </arguments>
         </call>
       </expr>
       <expr>
         <break />
       </expr>
     </block>
   </group>
   <group>
     <expr>
       <constant
         value='div'
       >
         <type
           name='string'
          />
       </constant>
     </expr>
     <block>
       <expr>
         <call
           idref='Mjava.io.PrintStream;println(int)'
           name='println'
         >
           <callee>
             <expr>
               <identifier
                 namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;out;'
                 name='out'
                /> 
             </expr>
           </callee>
           <arguments>
             <expr>
               <operation
                 operator='/'
               >
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;a;'
                     name='a'
                    />
                 </expr>
                 <expr>
                   <identifier
idref='VMExample;main(java.lang.String[])#Cjava.lang.Integer;b;'
                     name='b'
                    />
                 </expr>
               </operation>
             </expr>
           </arguments>
         </call>
       </expr>
       <expr>
         <break />
       </expr>
     </block>
   </group>
   <group
     default='true'
   >
     <block>
       <expr>
         <call
          idref='Mjava.io.PrintStream;println(java.lang.String)'
           name='println'
         >
           <callee>
             <expr>
               <identifier
                 namespace='System'
           idref='VCjava.lang.System;#Cjava.io.PrintStream;err;'
                 name='err'
                />
             </expr>
           </callee>
           <arguments>
             <expr>
               <constant
                 value='unknown operation'
               >
                 <type
                   name='string'
                  />
               </constant>
             </expr>
           </arguments>
         </call>
       </expr>
```

```
                    </block>
                  </group>
                </ext:switch>
              </expr>
            </block>
          </method>
        </type_decl>
      </namespace>
      <meta:languages
        base='java'
      />
    </unit>
```

Listing 38: arithmetic example in SrcML using extended switch

Ich erkläre, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die hier angegebenen Quellen und Hilfsmittel verwendet habe.

I affirm that I composed this diploma thesis independently and did not make use of any resources other than those indicated herein.