JOHANNES KEPLER
UNIVERSITY LINZ

Submitted by
Peter Feichtinger

Submitted at
Institute for System Software

Supervisor
Dipl.-Ing. Dr. Herbert Prähofer

September 2017

Dependency Browser for Program Comprehension and Change Impact Analysis of PLC Programs

Master Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Computer Science

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at
DVR 0093696

Abstract

A *System Dependence Graph* (SDG) encodes control and data dependencies in a whole program, and can be used, for example, for change impact analysis in software evolution. However, an SDG for a non-trivial program can quickly become very large, with several tens or hundreds of thousands of nodes.

The tool developed in the course of this thesis, the *SDG Browser*, is an attempt at making large SDGs accessible to the user by providing an interactive view for exploring different aspects of an SDG. The tool was developed with a number of use cases in mind, but it can be easily extended to support others. Rather than producing the underlying SDG in every detail, abstractions are made for better understanding.

This thesis was created in the course of our cooperation with Keba AG at the Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems (CDL MEVSS). It uses the SDG developed by Florian Angerer in the course of his PhD work.

Kurzfassung

Ein *System Dependence Graph* (SDG) stellt die Kontroll- und Datenabhängigkeiten in einem gesamten Programm dar und findet z. B. für Change Impact Analyse in der Sofwareevolution Verwendung. Allerdings sind SDGs für nicht-triviale Programme von erheblicher Größe und können einige zehn- oder hunderttausend Knoten beinhalten.

Das im Rahmen dieser Masterarbeit entwickelte Tool, der *SDG Browser* ist ein Versuch, dem Benutzer große SDGs durch eine interaktive Ansicht zugänglich zu machen, mit der unterschiedliche Aspekte eines SDG untersucht werden können. Das Tool wurde basierend auf einigen konkreten Use Cases entwickelt, lässt sich aber leicht um zusätzliche Anwendungsfälle erweitern. Anstatt den zugrunde liegenden SDG in allen Details wiederzugeben, werden an einigen Stellen entsprechende Vereinfachungen vorgenommen, die für leichtere Verständlichkeit sorgen sollen.

Diese Arbeit entstand im Zuge des Christian Doppler Labors für Monitoring and Evolution of Very-Large-Scale Software Systems (CDL MEVSS) in Zusammenarbeit mit Keba AG. Der verwendete SDG wurde von Florian Angerer im Rahmen seiner Doktorarbeit entwickelt.

Acknowledgment

I would like to thank my supervisor Herbert Prähofer for his valuable feedback on this thesis. Thanks also to my colleagues Florian Angerer, Andreas Grimmer, and Lukas Linsbauer for their input and their time for discussions.

Thanks also go to Alois Mühleder for his contributions to the project, and to Paul Grünbacher, head of the CDL MEVSS.

This thesis was performed at the Christian Doppler Laboratory for Monitoring and Evolution of Very-Large-Scale Software Systems and was supported by the Christian Doppler Forschungsgesellschaft and Keba AG.

Contents

1 Introduction **1**

1.1 The Tool 2

1.2 Context 3

1.3 Outline 3

2 The System Dependence Graph **5**

2.1 Construction and Implementation 5

2.1.1 SDG Classes 6

2.2 Structure 8

2.2.1 Overview 8

2.2.2 Calls 10

2.2.3 Instances 12

2.2.4 Simplifications and Additions 12

3 Use Cases **15**

3.1 Executions 15

3.2 Variable Assignment 16

3.3 Change Impact 17

3.4 Change Cause 19

4 User's Manual **21**

4.1 Overview 21

4.2 Using the SDG Browser 25

4.3 Additional Features 30

4.3.1 Dead Code Detection 33

4.3.2 Zoom 33
4.4 Examples 36

5 Implementation **43**
5.1 Overview 44
5.2 Model 46
5.2.1 Display Graph 47
5.2.2 Use Cases 51
5.3 Controller 52
5.4 View 53
5.4.1 Draw2D View 53
5.5 Commands 55
5.5.1 External Command Interface 55
5.6 Extensions 56

6 Summary **59**
6.1 Experiences 60

References **61**

Curriculum Vitae **63**

1 Introduction

A *system dependence graph* (SDG), introduced by Horwitz et al. [9], encodes all control and data dependencies in a whole program. The SDG represents procedures by their *program dependence graphs* (PDGs). A PDG consists of a node for the procedure itself, a number of formal parameter nodes for parameters and global variables used in the procedure, and nodes for every statement in the procedure. The nodes are connected by edges representing the control and data dependencies inside the procedure. The SDG also represents the dependencies between procedures in the whole program, based on one or more entry points into the program. Further, the SDG is instance-based, which means in object-oriented terms, that a method called on different instances of a class will be represented by distinct instances of the method's PDG.

In configurable software systems that need to meet a wide range of customer requirements, variability is of great importance. Understanding variable software is essential for its evolution and there are several approaches for dealing with the analysis of variable software systems. One important application of program analysis is *change impact analysis* (CIA), the identification of the potential consequences of a change. An approach for change impact analysis of configurable software was developed by Angerer et al. [3]. This approach uses a system dependence graph to represent the whole program. An extension of that is the *conditional system dependence graph* (CSDG) introduced by Angerer [4], which also represents variability information.

System dependence graphs are constructed based on the source code of a whole program through static program analysis. This means it is not necessary to run a program for analyzing it, but it also means that the analysis cannot use any information which might only be available at runtime.

In the concrete case of building an SDG, the major downside of using static analysis is that information about object instances needs to be extracted from the source code alone, which might not be easy or at all possible. For example, in languages like Java that support dynamic allocation of objects, complicated points-to analysis is necessary to determine which object instances might exist, and then the result may still be inaccurate.

This is, however, not a problem with the IEC 61131-3 family of languages [12] used in this thesis. The IEC 61131-3 standard defines languages for programmable logic controllers used in automation systems. Vendors implement the standard and provide tools, e.g. the *Kemro IEC* implementation provided by our industry partner Keba AG. An important property of those languages is that they don't support dynamic allocation. Therefore, an object instance is statically allocated by defining a variable of a particular type. This makes it possible to statically determine all instances of objects that will exist at runtime, making it possible to build an instance-based SDG.

1.1 The Tool

Since SDGs can get very large—many hundreds of thousands of nodes [4, sec. 4.3]—using them for manual analysis in their entirety is impossible. There are, however, a number of use cases where one would want to explore the SDG of a system manually, for example, the aforementioned change impact analysis.

The tool developed in this thesis, named the *SDG Browser*, provides a means for interactive exploration of an SDG based on a chosen entry point. An entry point can be—depending on the use case—a single statement, a procedure, or a variable. Based on the selected entry point, control and data edges in the graph can be followed interactively. The tool was developed with a number of use cases in mind.

Executions Determine through which paths a statement or procedure can be executed. This effectively gives a graph which contains all possible stack traces that could be observed for any execution.

Variable Assignment Determine all statements that write to a particular variable (including procedure parameters), and for those statements show their *Executions*.

Change Impact For any program element (statement, procedure, variable) determine other statements or variables possibly affected by its change or execution. This can be used to determine whether a particular change will have any unforeseen consequences.

Change Cause This is basically an inverse change impact analysis. It determines for any program element other statements or variables which might have an impact on its value or execution.

The versions of the SDG and SDG Browser introduced in this thesis work with the Kemro IEC languages only, because this was the initial target of Florian Angerer's PhD work. However there is ongoing work to support analysis across language boundaries as well [2].

1.2 Context

The SDG Browser was developed in the course of our cooperation with Keba AG at the *Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems* (CDL MEVSS for short).

Keba AG is a company that makes tools, hardware, and software for the industrial automation domain. One of their major products is the KePlast platform, a solution for the automation of injection molding machines. At its core KePlast has a configurable control software framework implemented in Keba's proprietary dialect of the IEC 61131-3 language (see also [4, sec. 3.1]).

Module 2 of the lab investigates development and evolution in large software ecosystems such as those developed by Keba.[1] Key challenges in the development of such software systems are:

- dealing with different variants and version of the software system;
- assessing the impact of changes to different product variants; and
- providing support to developers in evolving the system.

Our approach to these challenges is the FORCE environment [8], which provides feature-oriented and role-specific views of the software ecosystem. Feature-oriented views help in breaking the system into manageable pieces to work on, while role-specific views are useful to distinguish features managed by different teams.

1.3 Outline

The remainder of the thesis is organized as follows. Chapter 2 will cover the SDG in more detail and explain its structural properties. In Chapter 3 the use cases targeted by the SDG Browser implementation will be introduced. A tutorial and user's manual for the SDG Browser, based on the use cases defined earlier, will be given in Chapter 4. Chapter 5 will explain some of the implementation details and serve as an introduction to the system for making future extensions. Finally, Chapter 6 will conclude the thesis.

[1] See `http://mevss.jku.at/?page_id=654` for an overview of the motivation, goals, and results of CDL MEVSS Module 2.

2 The System Dependence Graph

A *system dependence graph* (SDG) is a directed graph used to represent the control and data dependencies of a program in its entirety. Note, however, that the control flow itself is not encoded in the SDG. Consider, for example, two statements in a procedure, which are control-dependent on that procedure (meaning they will be executed whenever the procedure is executed). The SDG does not contain any information about the order in which those two statements will be executed when the procedure is called, this information is only available in the control flow graph.

The *conditional system dependence graph* (CSDG) used here, in addition to control and data dependencies, also encodes variability in the form of conditions on edges. Those so-called *presence conditions* are Boolean expressions that encode for which combinations of enabled features an edge is present in the graph. A feature in this context is, for example, a configuration option that is stored in a global variable. Since these conditions are of no concern to the visualization and exploration of the SDG itself, I will ignore them most of the time. They may, however, be useful in future additions to the SDG Browser.

The following two sections will describe the implementation and construction of the SDG and give a detailed explanation of its structural properties.

2.1 Construction and Implementation

The construction of the SDG is covered in great detail in Andreas Grimmer's master thesis [7]. The following section will only give a brief overview.

Figure 2.1 shows an overview of the analysis toolchain. The input to the SDG construction is a PLC program written in the IEC 61131-3 family of languages [12], specifically, a proprietary dialect of the language used by KEBA. The program is first parsed to build an *abstract syntax tree* (AST) which is based on the OMG ASTM [5] and was extended to handle features of the IEC languages [7, ch. 4].

The ASTM representation serves as an input to generate Jimple code, an intermediate language representation used by the Soot analysis framework [10]. Soot is used to

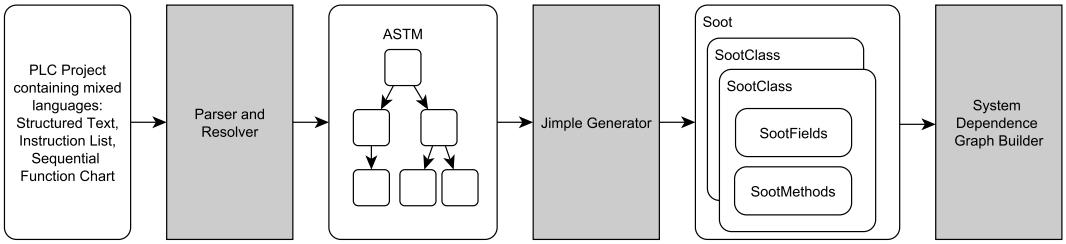

Figure 2.1: Overview of the analysis toolchain for building the SDG (taken from [7])

perform control flow and points-to analyses, and it provides def-use information and post-dominator trees used in the construction of the SDG.

The SDG is then created in two steps [7, ch. 6].

1. In the first step, a program dependence graph (PDG) is built for each procedure in the program. The nodes in this graph represent statements and formal parameters, the edges represent the intra-procedural control and data dependencies. The PDGs are object-insensitive, which means they are computed once for each procedure rather than based on object instances.
2. The object-sensitive SDG is created by instantiating each PDG according to the object instances it is used in. Information about object instances is provided by Soot's points-to analysis. The PDGs are linked by introducing control edges that link calls to called procedures, and data edges between actual and formal parameter nodes. Since a PLC program may have several entry points, an artificial procedure node is created for the `main` procedure, which links to every possible entry point of the program.

2.1.1 SDG Classes

The SDG is represented by classes `DGNode` and `DGEdge`, which are the abstract base classes for different kinds of nodes and edges, and by class `SystemDependenceGraph`, which has the main entry point. Figure 2.2 shows a class diagram of all SDG classes.

- Each `DGNode` has a list of successors and predecessors, edges which link to and from other nodes, respectively.
- A `DGNode` may have a seed condition associated with it, a Boolean expression that specifies for what feature combinations dependent nodes are reachable.
- A `PDGNode` represents a statement in the program, whether it be conditional or not (to determine that, information from the AST is necessary).
- There are three subclasses of `SDGNode`, which represent procedure instances and their interactions. Each `SDGNode` has an owning `Procedure` associated with it which represents the instance and links to the corresponding AST node.

  - A `ProcedureNode` represents an instance of a procedure and may have zero or more formal parameter nodes associated with it.
  - `ExprNode` represents formal or actual parameters, as well as global variables. In the case of a formal parameter node for a global variable accessed by the procedure, the node also refers to the actual node representing the global variable itself.
  - Calls between procedures are represented by `ActivationNodes`. An activation node refers to the statement that represents the call, has one or more procedures that are called, and has all actual parameters for the call attached to it.

- Some nodes have an AST node associated with them, for others the AST node can be accessed indirectly.
- An edge may have a presence condition associated with it, a Boolean expression that specifies for what feature combinations the edge does exist.
- A `ControlEdge` has an associated label, which in the case of conditional statements specifies whether the control dependency is for the **`true`** or **`false`** branch of the condition.

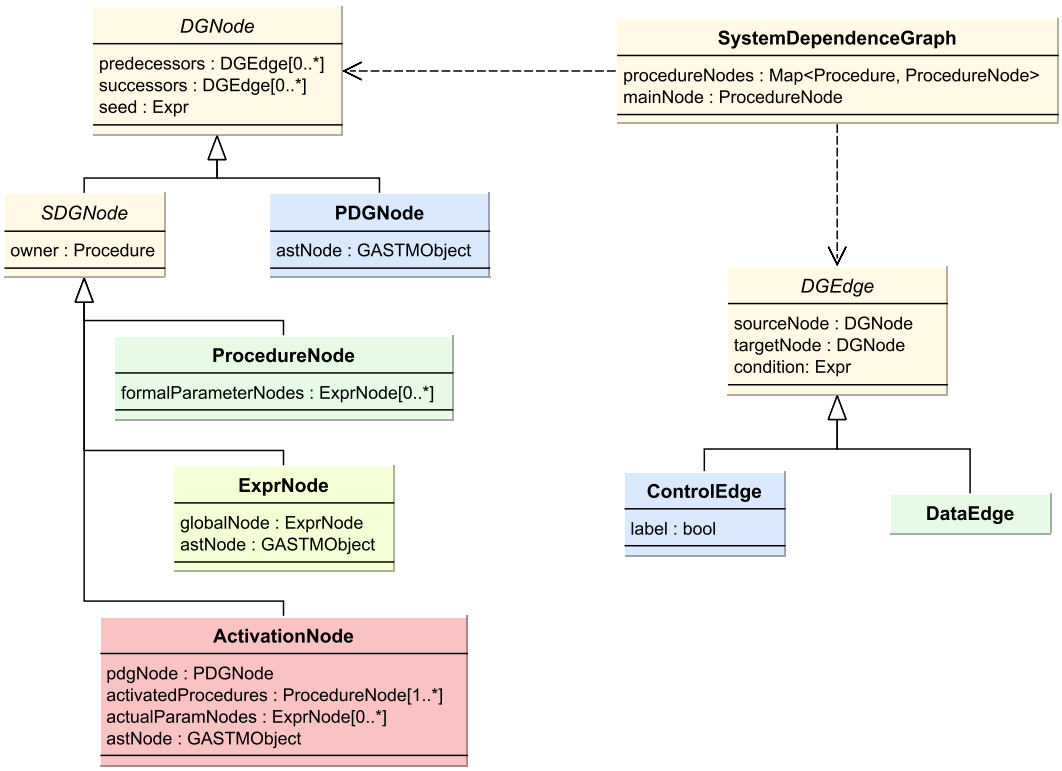

Figure 2.2: Class diagram of the SDG classes

2.2 Structure

This section will explain the structure of the SDG in detail, and also the changes and simplifications made for display to the user. Because not everything one might want to explore is represented in the SDG (for example, local variables don't have a representation in the SDG), additions are also mentioned.

2.2.1 Overview

The rough structure of the SDG is best explained by means of an example: the program in Listing 2.1 uses two load-time configuration options. It also contains conditional statements and method calls. The CSDG corresponding to the program can be seen in Figure 2.3; note that this SDG is greatly simplified, but will suffice for a rough overview.

The procedure nodes in green correspond to the two methods in the program, `main` and `foo`. Control edges in blue and red (for the **`true`** and **`false`** branch of a conditional, respectively) connect control dependent statements to procedures and conditional statements. Note that conditional statements are represented by statement nodes just like ordinary statements; the SDG itself does not contain any information about whether a statement is a condition or not, this information is taken from the AST nodes linked to each statement node. Dashed edges show the flow of data between different nodes, from the writing to the reading node.

In the CSDG, edges are annotated with their respective presence conditions, which depend on the two configuration options. Seed conditions (not shown in the figure), as mentioned in Section 2.1.1, are present on the three conditional nodes and serve as the initial conditions for the propagation of presence conditions. A detailed explanation of the process of finding configuration options and augmenting the SDG with presence conditions can be found in [3]. Note that in this example only some of the presence conditions are shown.

This example also illustrates how the CSDG may be used for configuration-aware change impact analysis: say we want to know whether a change to the highlighted return statement in method `foo` would affect the print statement. Then we can see that the conditions of the data edge from the return to the print statement ($\neg c_1 \wedge c_0$) and the print statement's incoming control edge ($\neg c_0$) are disjoint, they cannot both be true (i.e. $(\neg c_1 \wedge c_0) \wedge \neg c_0 \implies$ False). This means that a change to the return statement does not affect the print statement, while we would have concluded that it does, had we ignored variability information provided by the configuration options.

Listing 2.1: Program using load-time variability resulting in the SDG in Figure 2.3

```
class Main {
  static final boolean c0 = getConfiguration("c0");
  static final boolean c1 = getConfiguration("c1");

  public static void main(String[] args) {
    int res = 0;
    if(c0) {
      res = foo();
      return;
    } else if(c1)
      res = foo();
    System.out.println(res);
  }

  private static int foo() {
    if(!c1)
      return 1;
    return 0;
  }
}
```

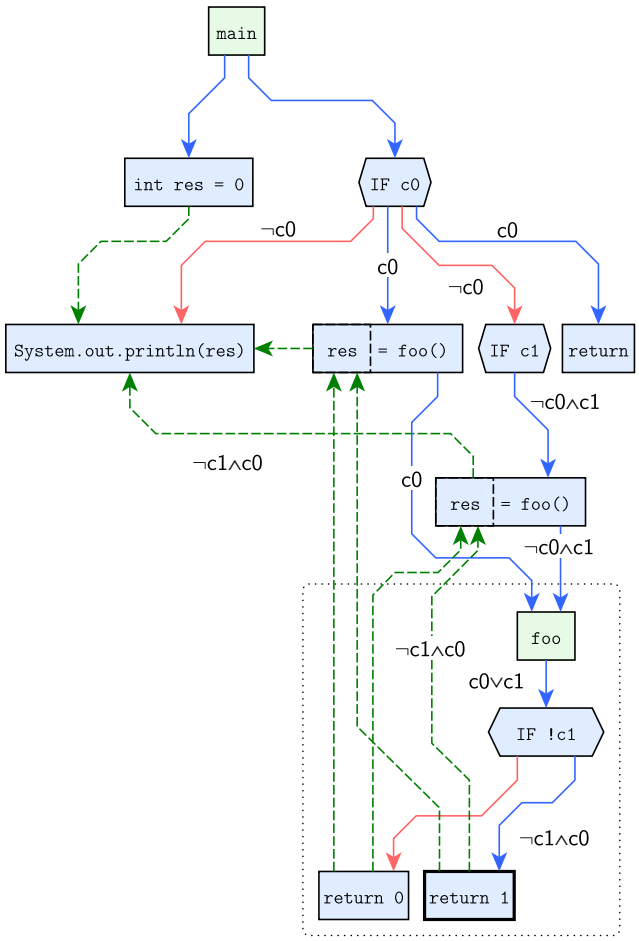

Figure 2.3: Simplified CSDG for the sample program in Listing 2.1

2.2.2 Calls

Let's now look at an example of function calls: the program in Listing 2.2 calls two methods with one parameter each; one of the methods uses a global variable as well. Figure 2.4 show the corresponding SDG. The SDG now contains two additional node types: activation nodes (call nodes) are shown in red, while three different kinds of expression nodes are shown in orange.

Global variables are represented in the SDG by expression nodes attached to the main procedure node via control edges. Semantically these are *variable nodes*.

The parameters to procedures are represented by expression nodes (semantically those are *formal parameter nodes*) attached to their procedure node via control edges (which semantically are *parameter edges*, shown dotdashed in the figure). The procedure `foo` has an additional formal parameter node that represents the global variable, since that can be seen as a hidden parameter.

A call is represented by an activation node and the parameters to the call are again represented by expression nodes (which semantically are now *actual parameter nodes*), attached via parameter edges. The statement causing a call is linked to the activation node via a control edge, the activation node in turn is also linked to the activated procedures (which may be more than one) via control edges (which can be seen semantically as *call edges*). Note, however, that one statement calling more than one procedure (e.g. `bar(foo(42));`) is still represented by multiple activation nodes in the SDG; a single activation node for multiple activated procedures happens only in case of, IEC events, where one expression may cause a number of event handlers to be executed. If a procedure returns one or more values (more than one value may be returned via output parameters in IEC procedures), these will be represented by formal parameter nodes as well.

Generally, the SDG Browser will not make a visual distinction between different kinds of control edges or parameter nodes, only variable nodes will be distinguished from other expression nodes for clarity.

Data flow, instead of going directly in and out of a procedure's PDG, now takes an indirection over the procedure's formal parameter nodes. Generally, there will be data edges connecting some nodes to an actual parameter node, which in turn connects to the formal parameter node, which then connects to statements inside the procedure. Data flow out of the procedure works much the same, with data edges connecting statements to formal parameter nodes for output parameters. Note that in the IEC 61131-3 language there are also in-out parameters, which may have data flowing in both directions. Similarly, reading from and writing to global variables takes an indirection over the corresponding formal parameter node.

Listing 2.2: Program with calls resulting in the SDG in Figure 2.4

```
class Main {
  static int global = 1;

  public static void main(String[] args) {
    boolean res1 = foo(42);
    int res2 = bar(res1);
    System.out.println(res2);
  }

  private static boolean foo(int n) {
    if(n < global)
      return false;
    gloabl = n;
    return true;
  }

  private static int bar(boolean b) {
    if(b)
      return 2;
    return 3;
  }
}
```

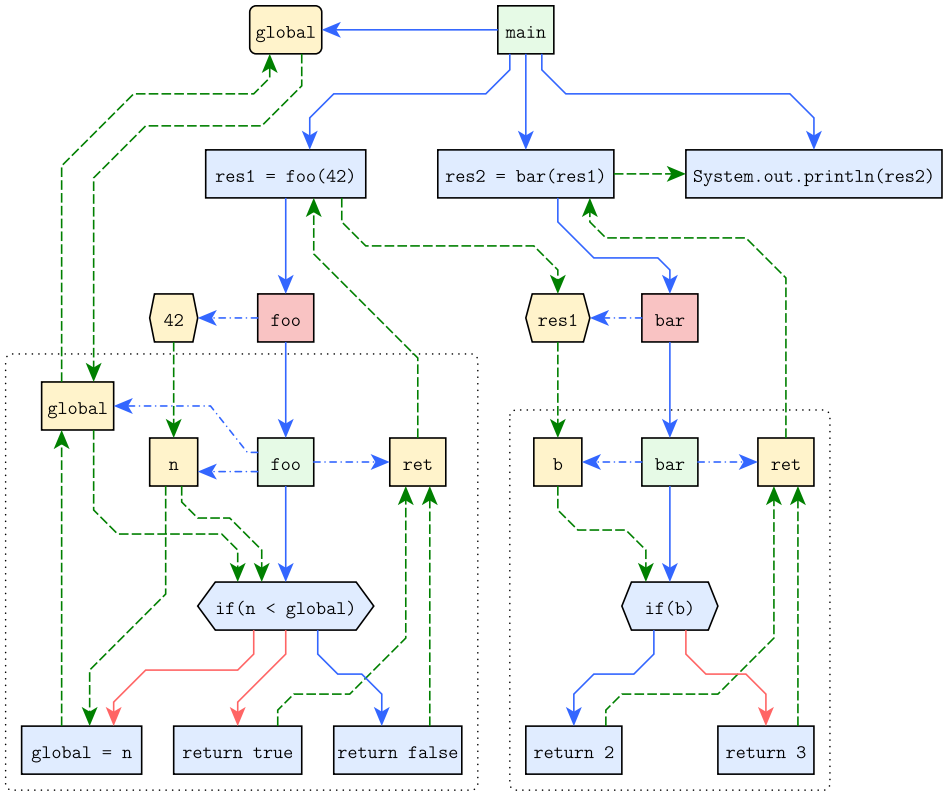

Figure 2.4: SDG for the sample program in Listing 2.2

2.2.3 Instances

In this next example the concept of an object-sensitive SDG will be illustrated: the program in Listing 2.3 calls instance methods on two different objects. Figure 2.5 shows the corresponding SDG.

Up until now the examples only contained static variables and methods, there were no object instances involved. In this example the class `Main` contains a member variable, and there are two instances of that class (which for simplicity are declared as static variables).

The object-sensitive nature of the SDG means that there are two instances of method `foo`'s PDG, one for each instance of the class. Because of this, calls to the same method on different objects can be distinguished, since they activate distinct procedures in the SDG. Member variables are represented by expression nodes attached to the main node via control edges, semantically they are equivalent to global variables (so they also have a corresponding formal parameter when used in a procedure).

2.2.4 Simplifications and Additions

The following simplifications and additions are made when displaying the SDG to the user, which is accomplished by having an intermediate *display graph* that wraps elements of the original SDG (see Chapter 5 for details).

Local Variables In all examples so far the program contained local variables, but they are not represented in the SDG. Instead, data flows directly between statements which make use of a variable or assign to it. Because in the *Variable Assignment* use case the SDG Browser should show assignments to local variables as well, the model used for display contains an additional kind of virtual node that wraps the AST definition node for the local variable together with a procedure node for the PDG instance. Assignments to the local variable are then located in the AST and virtual data edges are added from the corresponding statements to the virtual variable node.

Global Variables As illustrated in Figures 2.4 and 2.5, global and member variables are never written to or read from directly, instead there is always an indirection through a formal parameter node. Since this leads to lots of unnecessary formal parameter nodes which need to be displayed when a global or member variable is used in many procedures, the SDG Browser does not display those. Instead, a synthetic data edge is introduced which links directly from a statement to the expression node representing the variable itself. The corresponding formal parameter is then just excluded from display.

Listing 2.3: Program illustrating instances resulting in the SDG in Figure 2.5

```
class Main {
  static Main m1 = new Main();
  static Main m2 = new Main();

  int member;

  public static void main(String[] args) {
    int x = m1.foo(23);
    x += m1.foo(2) + m2.foo(3);
    System.out.println(x);
  }

  public int foo(int n) {
    member += n;
    return member;
  }
}
```

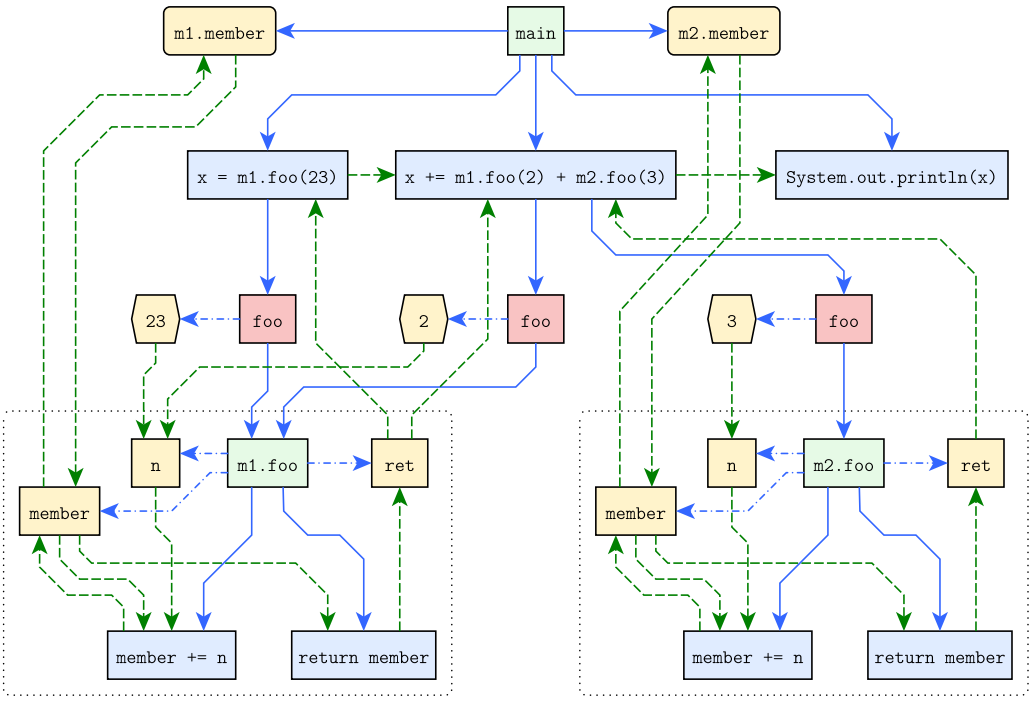

Figure 2.5: SDG for the sample program in Listing 2.3 (constructor calls omitted for brevity)

Container Nodes The SDG itself does not have any kind of hierarchy, i.e. a node does not contain other nodes. In the context of IEC programs, however, showing hierarchical relationships may be quite important. The SDG Browser therefore introduces artificial container nodes, which can represent procedures or functional units (which are similar to packages in Java). There are container nodes representing each procedure instance (except for the main procedure), which contain all nodes in the corresponding PDG, as well as formal parameter nodes and the procedure node itself. These in turn are contained in nodes representing functional units.

Container nodes are connected by artificial edges which are added whenever there are any edges between nodes within the container nodes.

Dummy Nodes Since the user selects source code elements which may have more than one corresponding node in the SDG (e.g. more than one instance of a procedure), the SDG Browser introduces dummy nodes indicating that there is more than one instance of something. They only serve as a means to group multiple instances of the same source code element for easier display and have no semantic meaning in the displayed SDG.

Again, dummy nodes connect to the actual nodes via artificial edges.

3 Use Cases

To guide and simplify the interactive exploration of the SDG, the SDG Browser implements four concrete *use cases*, which were briefly explained in Section 1.1 already. Those use cases were identified by Keba engineers to provide potentially helpful insights when investigating the code, trying to fix certain problems, or when making changes. This chapter will give a description of each of the four use cases (*Executions*, *Variable Assignment*, *Change Impact*, and *Change Cause*) and illustrate them with examples.

A use case in this context is basically a strategy for deciding which parts of the whole SDG should be presented to the user, together with a set of element types which are valid *targets* (starting points) it can apply to (e.g. statements, variables, procedures). The *result* of a use case is that part of the SDG which would be visible to the user if they were to expand every node in the graph (similar to a *slice* of the program [14]).

The SDG Browser implements use cases in a modular way, which makes it easy to add new ones or add custom behavior to existing ones. See Chapter 5 for a detailed description on that.

3.1 Executions

The targets of this use case are executable elements, i.e. statements and procedures. The goal is to find out how a statement or procedure can be executed, that is, which execution paths in the program may lead to it being executed. The result is a graph of all possible stack traces one could observe for the target, but also including conditional statements between calls.

Figure 3.1 shows an example for the Executions of the highlighted statement for one particular object instance (source code in Listing 2.3, page 13). The nodes and edges in the SDG which are not part of the result (i.e. will not be displayed) are grayed out. Figure 3.2 shows the same example for all instances of the statement (now grouped by a *dummy node*), which is usually the result when a statement is selected in the source code.

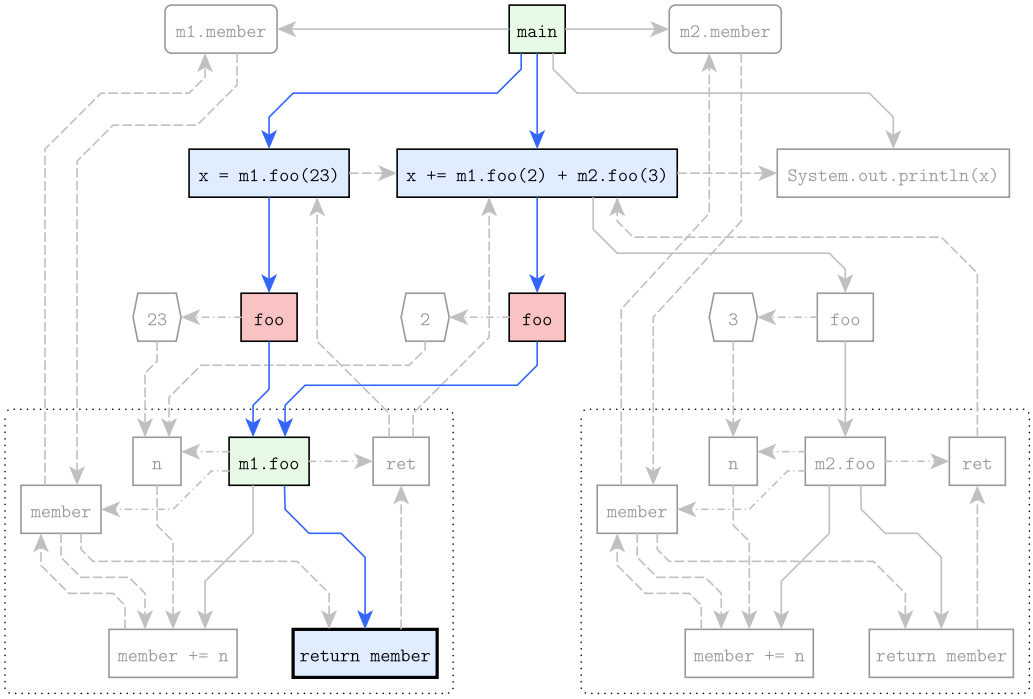

Figure 3.1: Executions for one instance of a statement

The resulting graph contains all nodes the target is directly or indirectly control-dependent on. Those nodes are found by including all predecessor nodes which connect via a control edge to any node already in the graph. In figures 3.1 and 3.2 we can see that the highlighted target statement depends on the execution of the respective procedure containing it, which in turn depends on one or more activation nodes (this means the procedure will be executed only if any activation node is executed). The activation nodes then depend on the respective statement containing the call, which in turn depend on their containing procedure, `main` in this case.

3.2 Variable Assignment

The targets of this use case are variables of any kind (global, member, and local variables; formal parameters). The goal is to find out which sequences of statements could cause a variable to change its value. The result is a graph that contains the variable node (which may be an artificial node in case of a local variable), all statements directly writing to that variable, and for those statements their *Executions*.

Figure 3.3 shows an example for the Variable Assignment of the highlighted instance of a member variable (source code in Listing 2.3, page 13). There is one statement that writes to the variable, connected via an incoming data edge. Note that the formal parameter node for the member variable is not displayed, instead an artificial edge is added directly from the writing statement to the variable node. The rest of

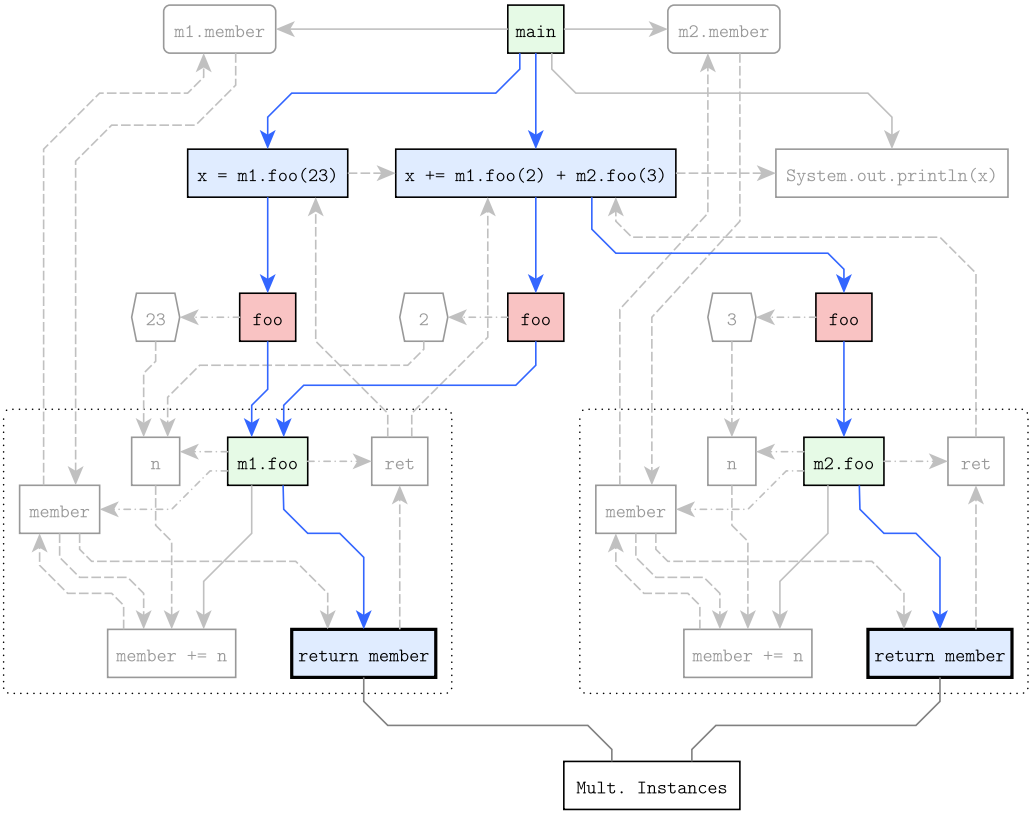

Figure 3.2: Executions for all instances of a statement, grouped by a dummy node

the graph is the same as for the *Executions* for that writing statement, i.e. statements and procedures which the statement is control-dependent on.

For variables that are represented in the SDG (member and global variables, parameters to procedures), constructing the graph for this use case is straight forward: just find predecessors which are statement nodes connected via data edges (possibly ignoring intermediate formal parameter nodes), and from there continue as for *Executions*. Local variables, on the other hand, are not represented in the SDG. To be able to include local variables as well, the SDG Browser introduces an artificial node representing the local variable for each instance of the procedure declaring it. The AST is then searched for each assignment, or use as a reference or output parameter, of the variable. From the resulting AST nodes the corresponding statement nodes in the SDG are found and connected to the artificial node via synthetic data edges.

3.3 Change Impact

This use case targets all nodes in the SDG. The goal is to find all statements and variables which might be influenced (i.e. whether a statement or procedure is executed, and changes to a variable's value). The result is a graph which contains all nodes that

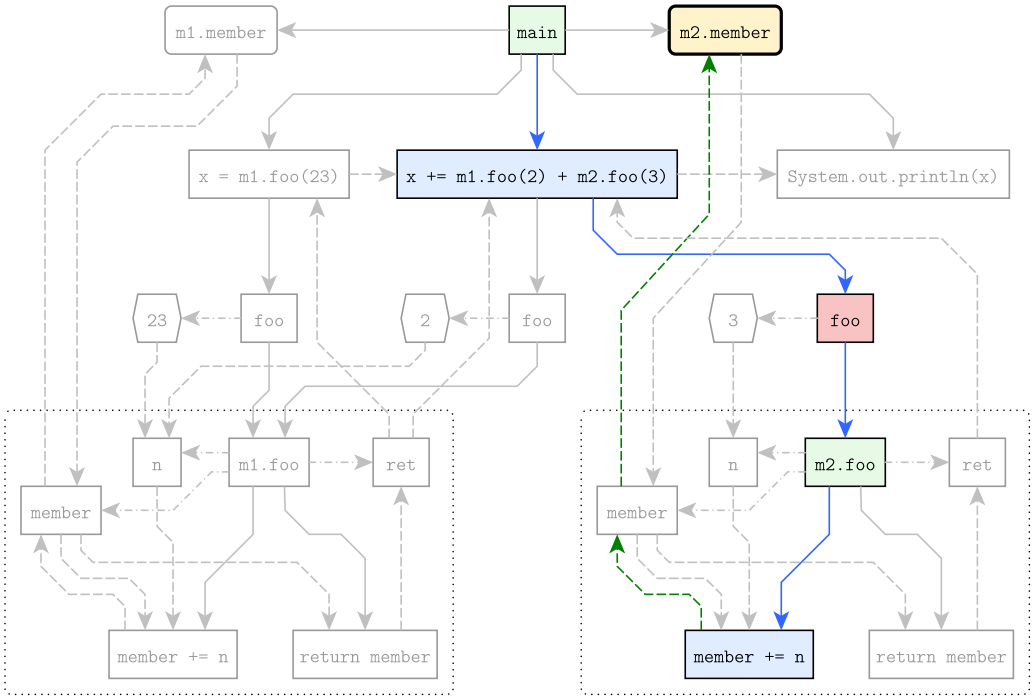

Figure 3.3: Variable Assignment for one instance of a member variable

are directly or indirectly influenced by the execution of a statement or procedure, or by a variable's value.

The example in Figure 3.4 shows a change impact for the highlighted return statement (source code in Listing 2.2, page 11). The return statement directly influences the procedure's return value, which in turn influences the value of variable `res2` and the print statement (via the highlighted edges). Execution of the statement `res2 = bar(res1)` influences the execution of procedure `bar` (via the corresponding activation node). The value of parameter `b` depends on the value of `res2` and influences the execution of the conditional statement, which in turn determines which of the return statements get executed.

The change impact excludes certain edges, such as control edges from procedure nodes to their parameters, because parameters should not be included if they don't have any incoming data edges. Also excluded are any outgoing edges from the main procedure (which connects to all global and member variables, but doesn't actually influence their values).

We can see in the example that the return statement actually only influences the return value, which in turn influences the value of variable `res2` and the print statement (via the highlighted edges). The change impact, however, also contains a lot of unrelated nodes. This is caused by the SDG Browser following all outgoing data and control edges, regardless of whether they are actually influenced by incoming edges. In this example, the statement influenced by the return value of method `bar`

is not a conditional statement, so we know that the outgoing control edge is not influenced by the incoming data edge. The SDG, however, does not encode this kind of information.

This issue with change impacts is caused in part by the SDG Browser itself, in part by the SDG. The SDG Browser already has information on whether a statement is a conditional statement or not (because they are drawn differently), which could be used to exclude certain control edges from the change impact. It is, however, unknown at the time a node is expanded by the user whether or not there will be any incoming control edges from other nodes, so just filtering such control edges unconditionally is not an option.

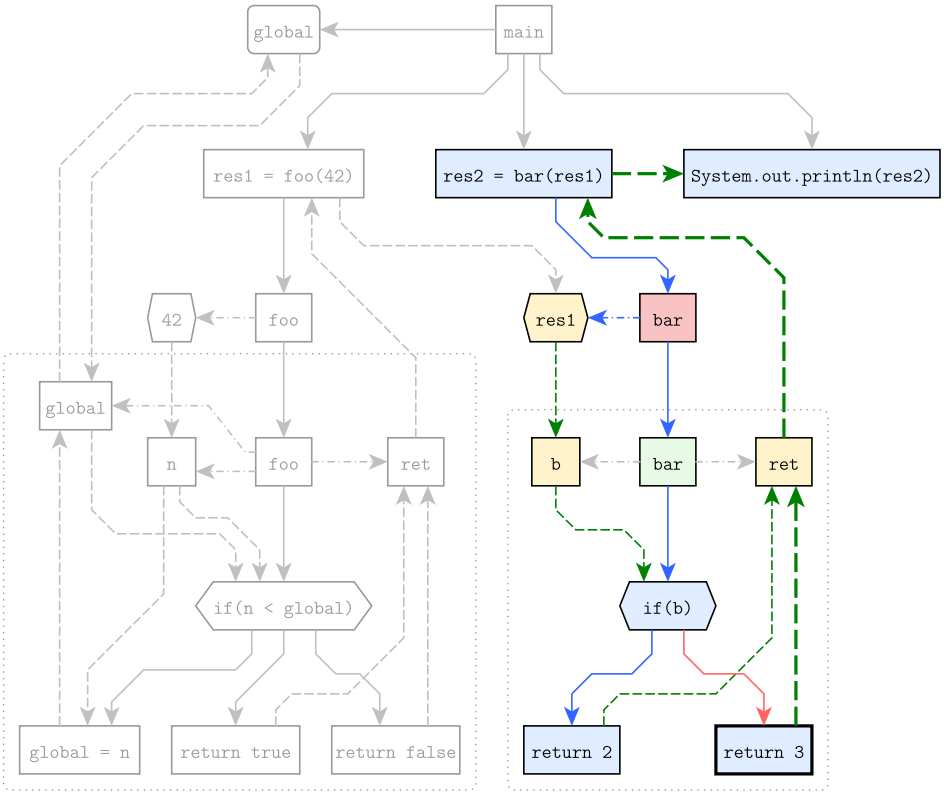

Figure 3.4: Change Impact for a return statement

3.4 Change Cause

This use case is basically an inverse *Change Impact*. It again targets all nodes in the SDG, the goal this time being to find all statements and variables which might influence the value of a variable or whether a statement gets executed. The result, therefore, is a graph which contains alls nodes directly or indirectly influencing the target node.

The example in Figure 3.5 shows a change cause for the highlighted assignment statement (source code in Listing 2.2, page 11). The statement is directly influenced by the conditional, because it only gets executed if the conditional yields **`false`**, and of course by the parameter which is assigned from. The conditional statement in turn depends on the value of the global variable and the parameter, as well as the execution of procedure `foo`. The value of the formal parameter depends on the actual parameter, both are only relevant if their procedure or activation node, respectively, gets executed. Note that, as in the *Variable Assignment* example, the formal parameter node for the global variable is not displayed, instead an artificial edge is added directly from the variable node to the reading statement.

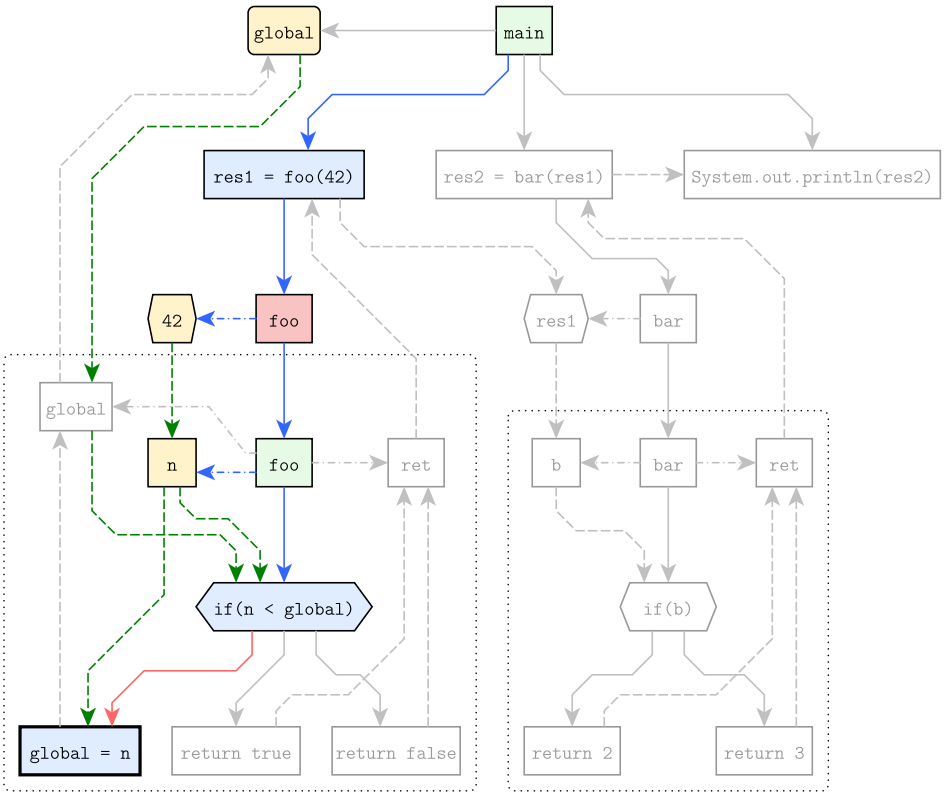

Figure 3.5: Change Cause for an assignment

4 User's Manual

This chapter will describe the usage and features of the SDG Browser in detail and serve as a user's manual. Section 4.1 will give an overview of the IEC editor and the process of creating the IEC project. Section 4.2 will cover the steps needed to use the SDG Browser. A detailed description of the SDG Browser features will be given in Section 4.3. Finally Section 4.4 will cover each use case based on example problems.

4.1 Overview

After installing the SDG Browser plugins and opening the *SDG Browser* perspective, the Eclipse workbench should have a layout as shown in Figure 4.2. On the left and right are the standard Eclipse Project Explorer and Outline views, respectively. On the bottom is the new SDG Browser view, below the Project Explorer is the Instance Hierarchy view. Those two views can be opened in any other perspective using the *Show View* dialog (Alt+Shift+Q, Q by default) as shown in Figure 4.1.

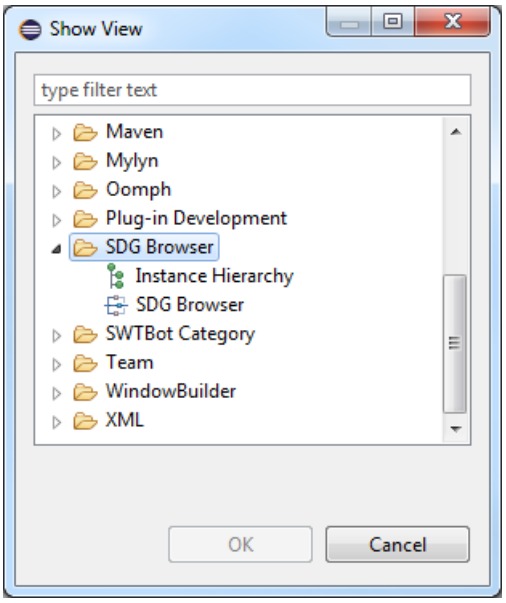

Figure 4.1: Entries in the *Show View* dialog for the SDG Browser views

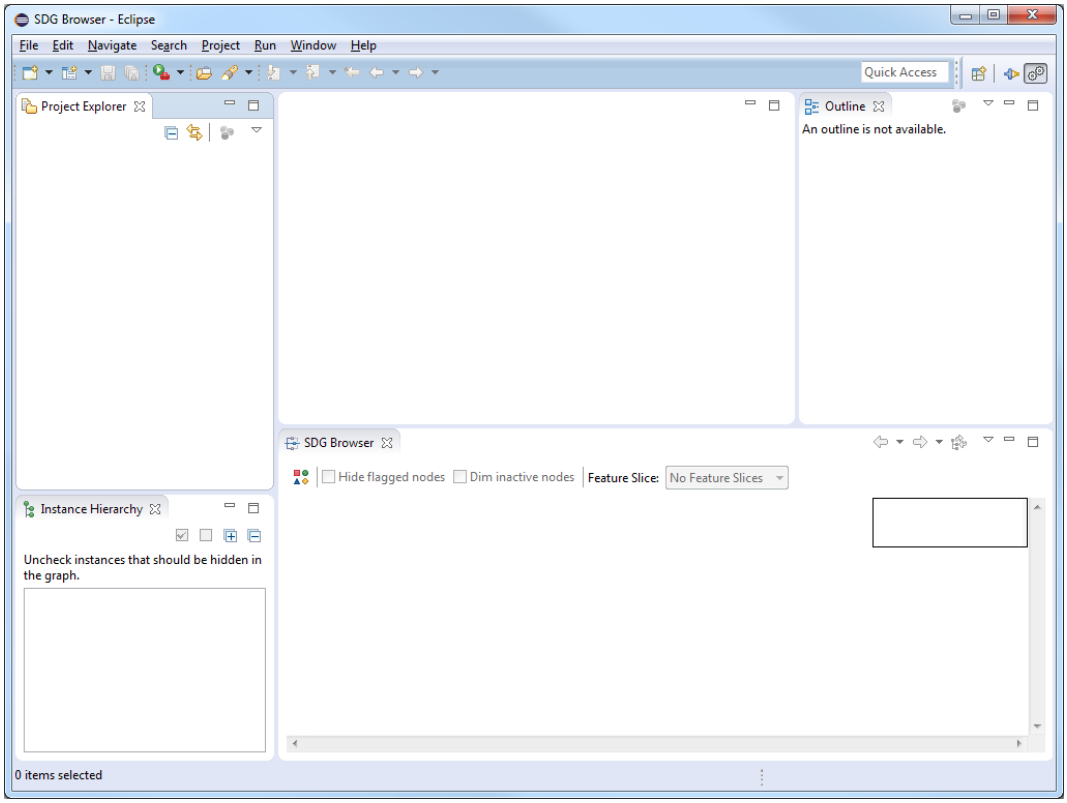

Figure 4.2: Default layout of the *SDG Browser* perspective

In order to use the SDG Browser with IEC source code, you first need to build it in IecEdit (Keba's IDE for Kemro IEC code). A project also needs to be rebuilt in IecEdit whenever it is moved to a different location. Then the project needs to be imported as an Eclipse project as follows.

1. Create a new Eclipse project and select *IEC Project* (see Figure 4.3).
2. On the next page, enter a name for the project and select the location of the IEC project files (Figure 4.4). This is the location where the Eclipse project file will be created. The IEC source code may be located in the same directory or in any subdirectory.
3. In case you have a standard project layout with HMI, the default source code locations may be used (`/prj/Source/IMM/ieccontrol` for IEC code and `/prj/Source/IMM/view/viewKVS` for view code). Otherwise, uncheck the checkbox and specify the paths (relative to the project location selected earlier). If you don't have HMI code, leave that text box empty.

After the project has been created, the Project Explorer will show all files and folders under the project directory, with some exceptions. Files recognized by any Eclipse editor can be opened directly, including IEC source files. Support for IEC files is based on contributions by Alois Mühleder during his bachelor thesis [11].

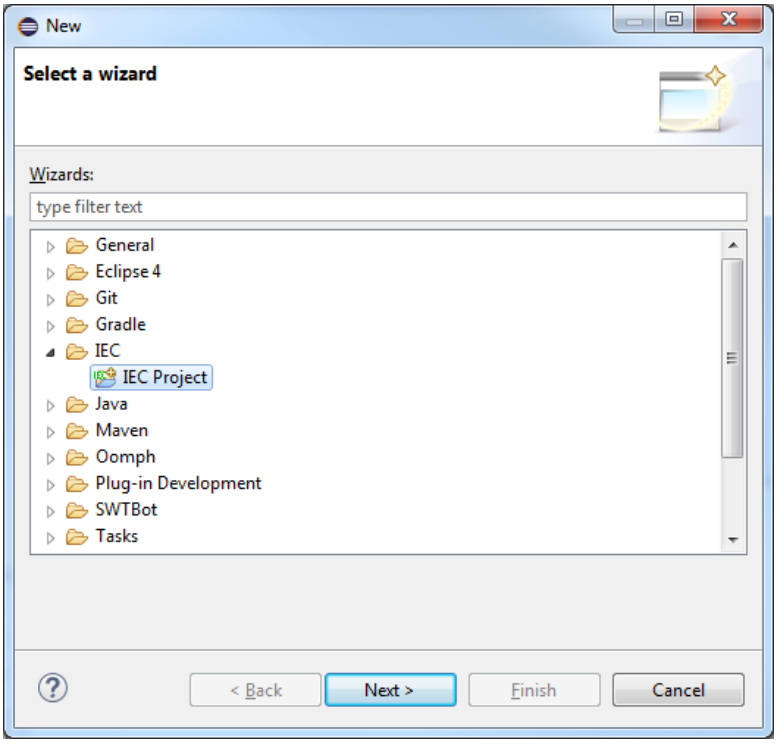

Figure 4.3: Create a new IEC project...

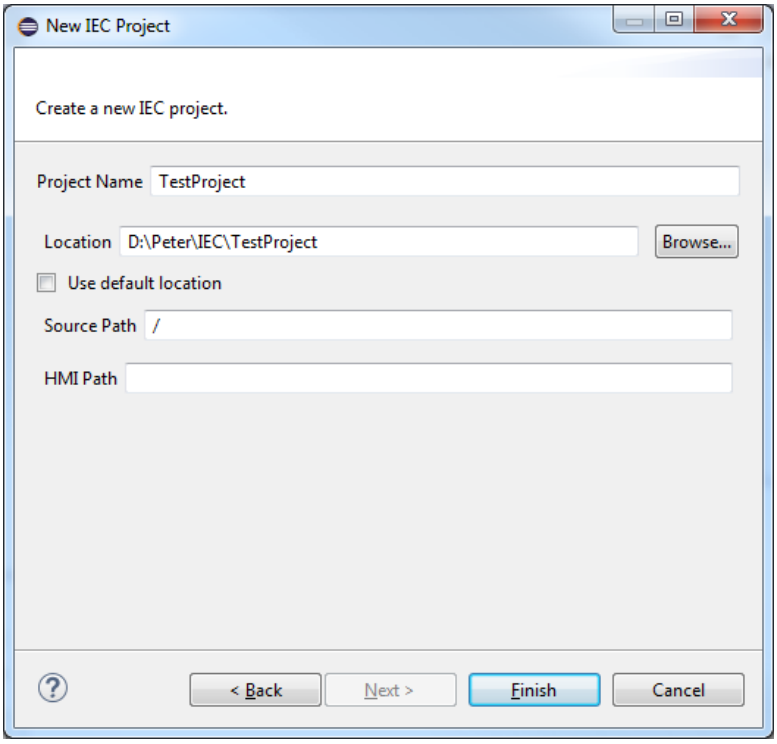

Figure 4.4: ...and specify the source code location

Note, however, that the IEC editor as it is currently implemented doesn't allow editing. The reason for this is that the IEC source files may also contain Keba's proprietary representation of the source code (the so called *UFF code*), in addition to the textual IEC code, which would get corrupted if not edited properly.

By default, IEC binaries and files related to Keba's IecEdit will not be shown in the Project Explorer. This may be changed by disabling the respective filters in the Project Explorer (via the *Customize View...* menu item).

Figure 4.5 shows an IEC file opened in the Eclipse editor. Note the file outline on the right hand side. The IEC editor shows only the textual IEC code, the UFF code is truncated from the output.

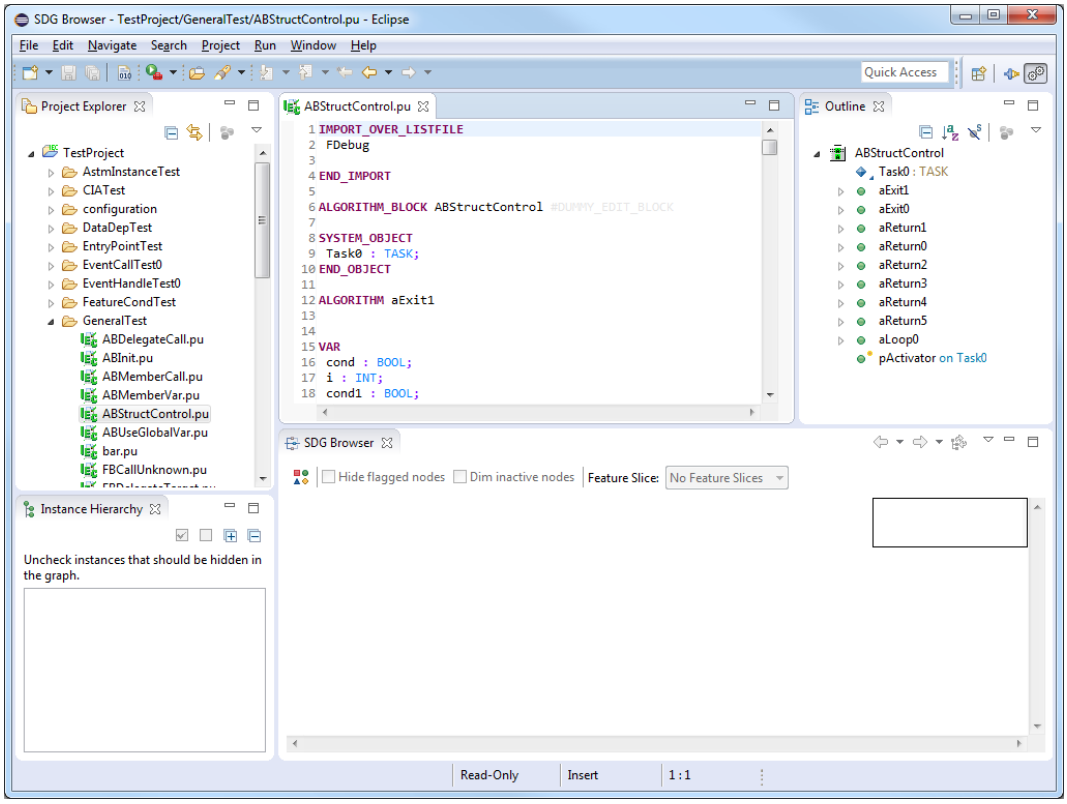

Figure 4.5: IEC file opened in Eclipse, an outline of the file is shown to the right

The IEC editor allows navigation through the IEC code in a number of ways.

- The outline view can be used to navigate within a file to procedures or variables.
- The editor also supports hyperlinks which may be used to go to the definition of a variable or procedure (via Ctrl+Click on the name).
- Furthermore, searching for all references to a particular source code element in the whole project is implemented as well. This can be accessed in the Eclipse *Search* menu or via the editor context menu (Figure 4.12, page 29).

4.2 Using the SDG Browser

Everything shown so far works without manually building the Eclipse project, since the source code is automatically parsed once any of the project's files are opened. However, the SDG must be built before the SDG Browser can be used. For large projects this can take several minutes.

To build the SDG, open any file within the project and then hit Ctrl+B or select *Build Project* from the *Project* menu (Figure 4.6). If you try to use any of the SDG Browser operations before the project has been built, you will be asked whether you want to build the project.

Important: only work on one project at a time; limitations in Soot prevent us from having more than one SDG in memory.

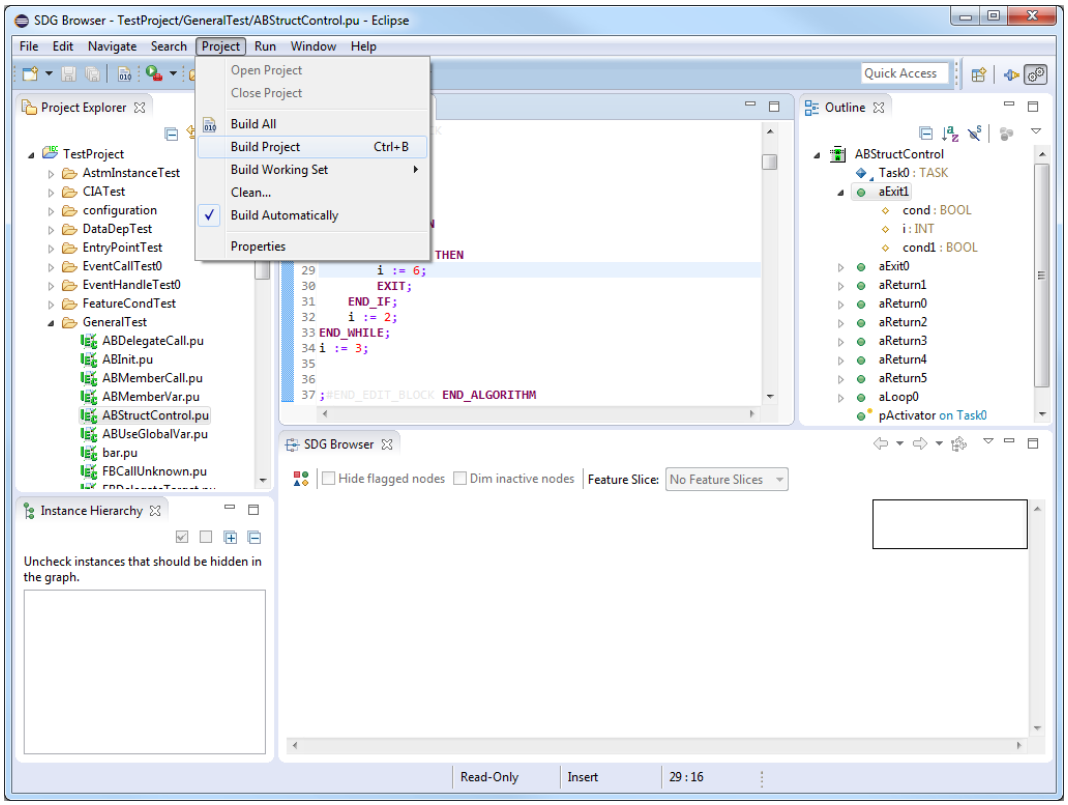

Figure 4.6: Building the IEC project before using SDG Browser operations

The SDG Browser operations can be accessed from the IEC editor context menu (see Figure 4.7). Usually you would select a source code element of interest (variable, statement, procedure) in the editor and then choose the desired operation from the context menu. In case nothing is selected, the whole line will be selected automatically for convenience.

Figure 4.8 shows the screen after choosing *Show Executions* from the context menu.

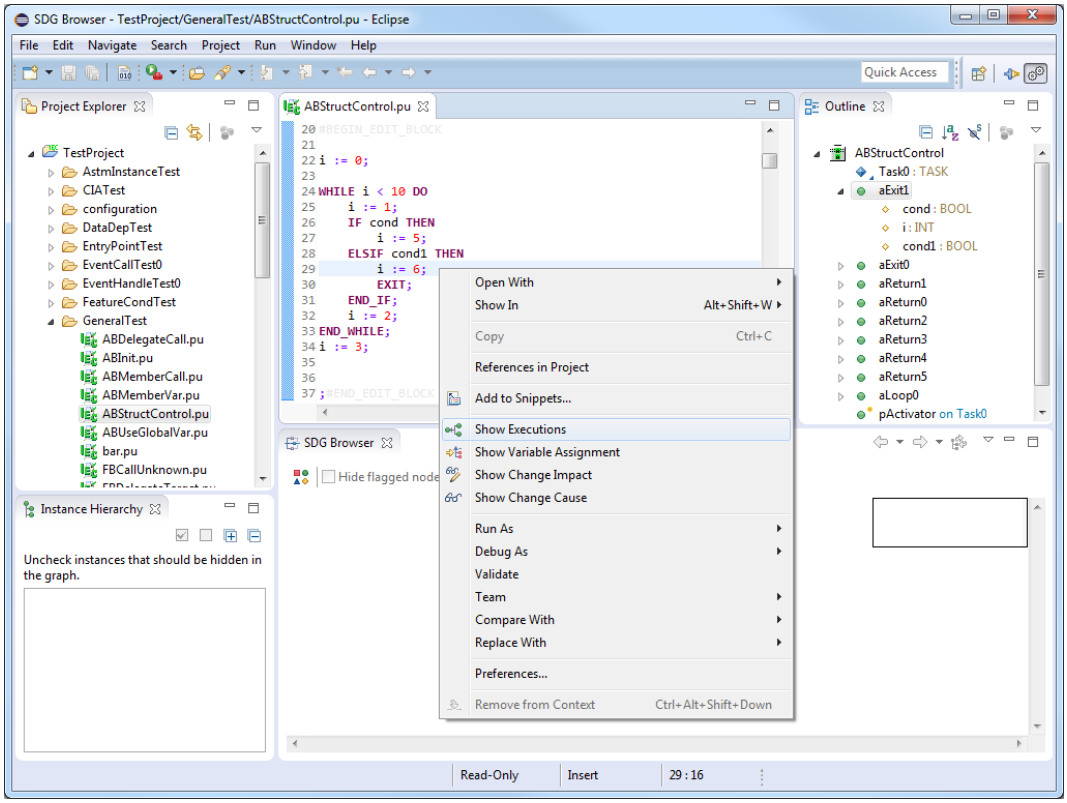

Figure 4.7: IEC editor context menu entries for SDG Browser operations

Notice the whole statement is selected now. The SDG Browser view shows a node for the selected statement `i := 6`, and nodes for all immediate predecessors (only `cond1` in this case). The nodes are contained by a node that represents the procedure those statements belong to (the algorithm `aExit1`).

You can navigate by panning the view: click and then drag with the middle mouse button anywhere in the graph. At the top right corner of the SDG Browser view you can see an overview thumbnail that shows the whole graph. This can also be used to navigate large graphs: click or drag the thumbnail to view that point.

The SDG Browser includes a legend as a quick reference of what the different colors and shapes of nodes and edges represent. Click on the leftmost icon in the toolbar of the SDG Browser view to show/hide it (see Figure 4.9).

For details on the types of nodes and edges in the SDG, as well as its structure, refer to Chapter 2. There are the following types of nodes in the SDG Browser.

- *Procedure nodes* represent executable procedures like functions, event handlers, process algorithms, etc. Procedure nodes may also be containers of the statements and variables within.
- *Variable nodes* represent all kinds of variables: local, global, and member variables, as well as return values.

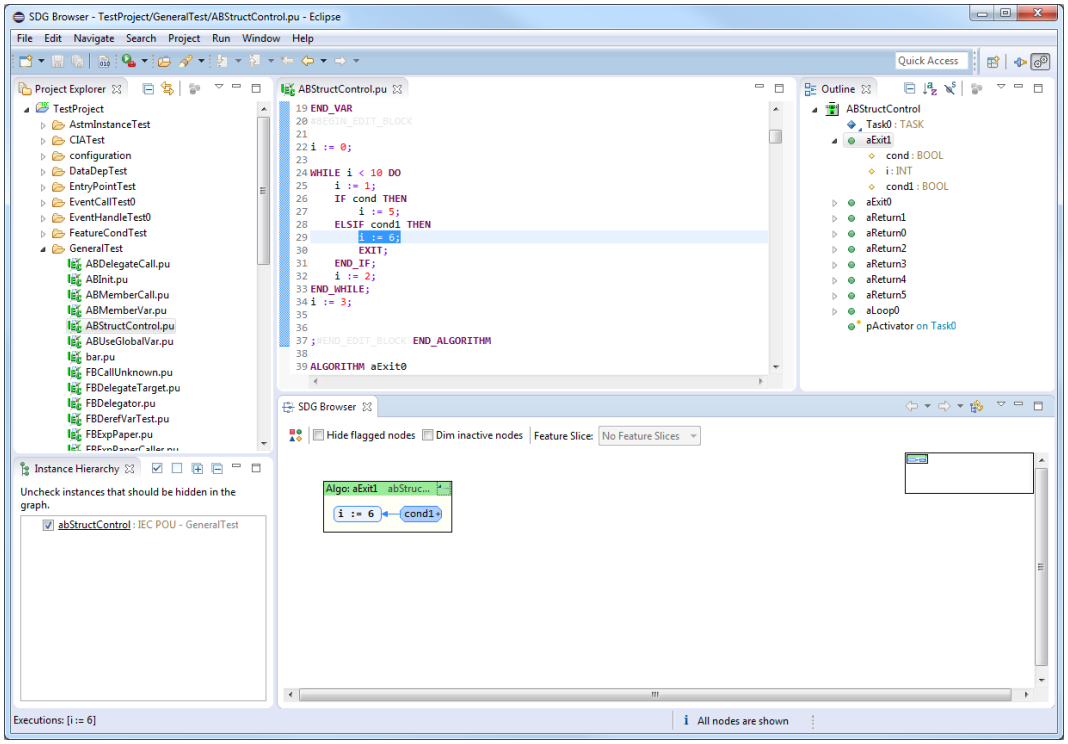

Figure 4.8: Screen after choosing an SDG Browser operation

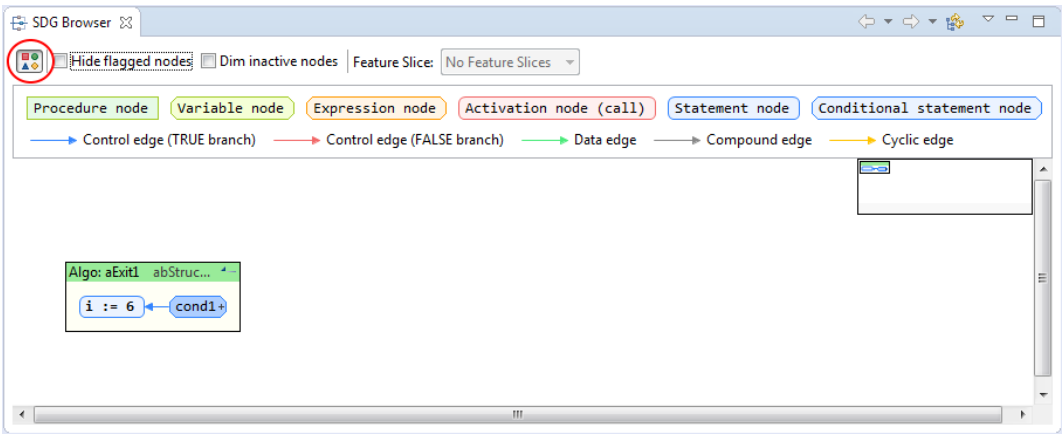

Figure 4.9: A legend shows different node and edge types

- *Expression nodes* represent formal parameters and actual parameter expressions.
- *Activation nodes* represent calls and other kinds of activation (for example, setting an event will cause all its event handlers to be executed, but this is not a call in the traditional sense).
- *Statement nodes* and conditional statement nodes represent ordinary and conditional statements, respectively.

The SDG may contain two types of edges, two additional kinds are introduced by the SDG Browser.

- *Control edges* represent a control dependency, i.e. the target node will only be executed if the source node is executed as well. In case the source node is a conditional statement, the edge color will indicate whether the target node is executed if the condition is true or false.
- *Data edges* represent a data flow from the source node (e.g. a write to a variable) to the target node (e.g. a read from a variable).
- *Compound edges* between container nodes represent more than one edge.
- A *cyclic edge* indicates an edge that introduces a cycle in the graph, i.e. it always leads to a node that has already been discovered earlier.

You will have noticed that the node for the conditional statement `cond1` is of a darker shade than the statement node (and shows a plus sign on its right side). This means that it can be expanded to reveal nodes connecting to it. Click on the node to expand it, the result should look like Figure 4.10.

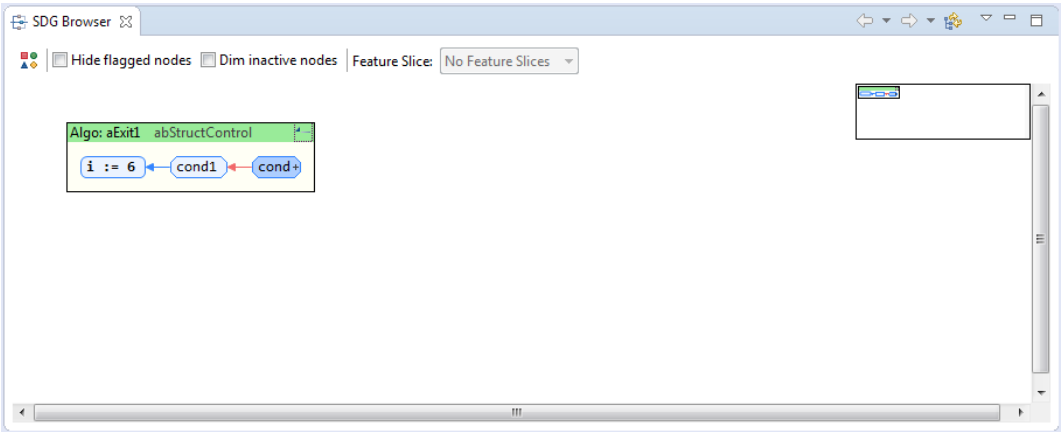

Figure 4.10: Nodes that can be expanded are of a darker shade and marked by a little plus sign on their right side

After expanding all nodes, the graph will look like Figure 4.11. Notice that there are now two procedures involved, algorithm `aExit1` is called by algorithm `pActivator`,

which is in turn called by the artificial `main` procedure. You can ignore the artificial main procedure most of the time, it is just an artifact of the analysis. In this case it signals that `pActivator` is an entry point of the program (all entry points are called by the artificial `main` procedure).

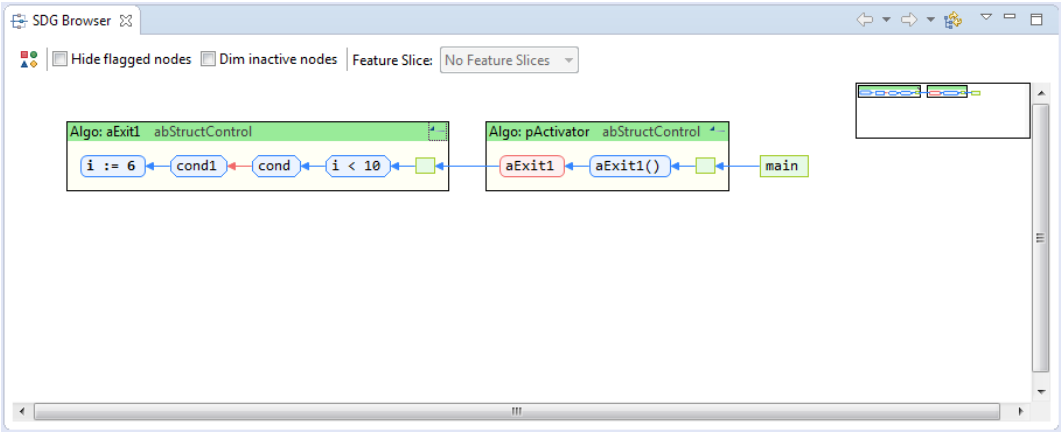

Figure 4.11: This graph shows one procedure calling another, the main procedure signals an entry point. Notice the small nodes, which represent the entry point of the procedure they are contained in.

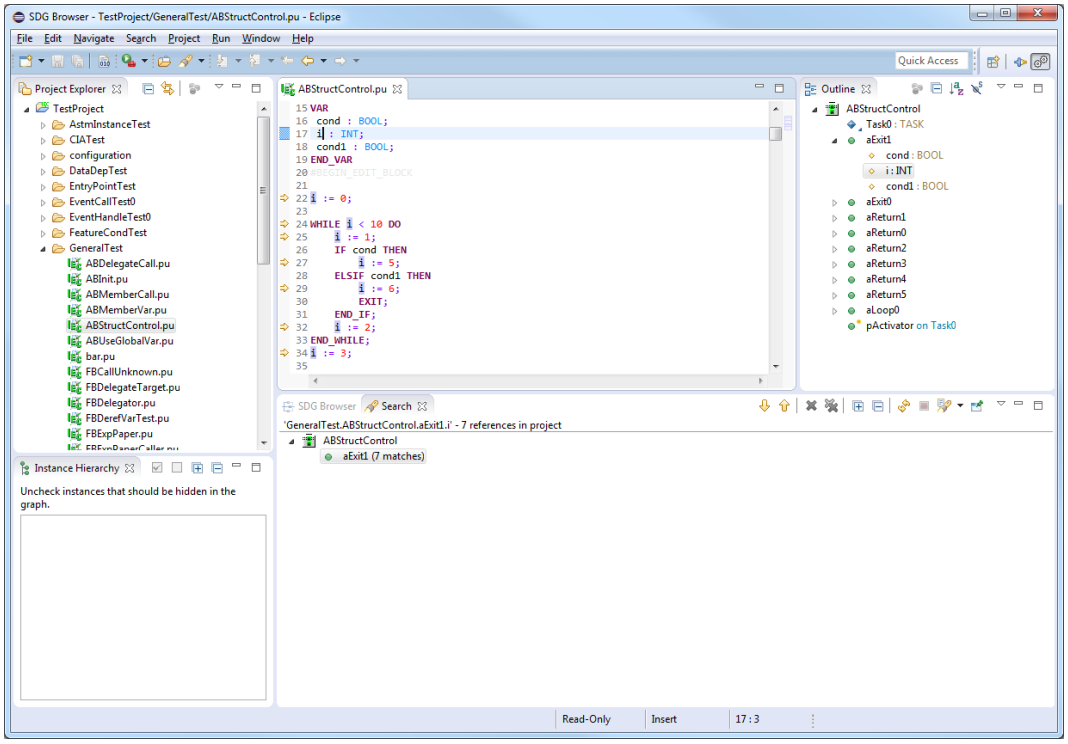

Figure 4.12: Search results for a local variable using *References in Project*

4.3 Additional Features

There are a number of additional feature which may prove helpful, especially when working with larger graphs.

Hovering over a node or edge opens a tooltip, which shows some additional information about the element (see Figure 4.13 for an example). For nodes of large statements (which are truncated so the node doesn't get too large) the tooltip also contains the entire statement.

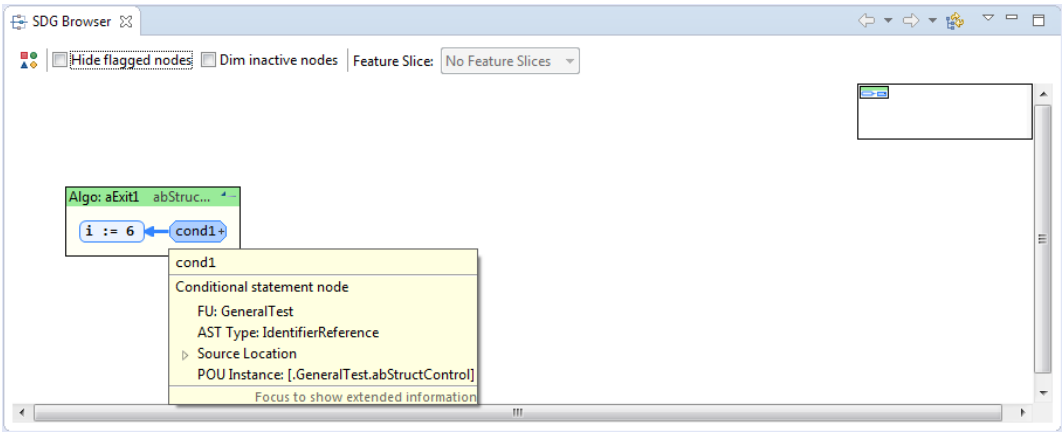

Figure 4.13: Tooltips provide further information about nodes and edges

Right clicking on a node opens a context menu as seen in Figure 4.14, which contains several additional actions.

Highlight Edges highlights all edges connected to the node. This may be helpful for following long edges that don't fit on the screen.

Show Source opens the source code location associated with the node in the IEC editor, if possible (Figure 4.17). Note that not all nodes may have a source location associated with them. This can also be invoked by clicking on a node while holding the control key.

Show Instance Declaration opens the source code location defining the object instance the node belongs to.

Set/Remove Flag sets or removes the node flag (Figure 4.15). Flagged nodes may be hidden by activating the corresponding checkbox (see Figure 4.16).

Collapse All collapses a node and all its successors at once. Note that this is different from simply collapsing a node, because the expansion state of all its successors is reset as well.

Expand Next Level expands all successors of the node.

The context menu also provides entries for executing a use case on the node.

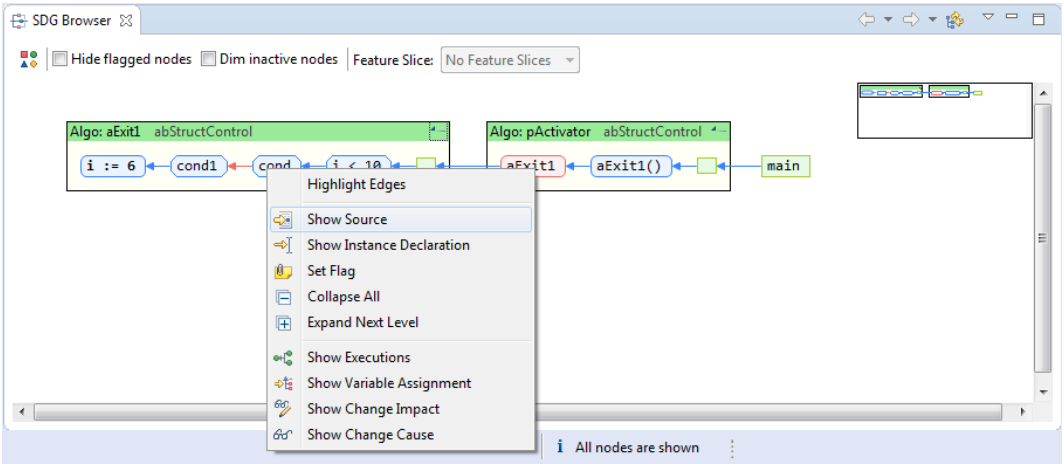

Figure 4.14: A context menu provides actions on nodes

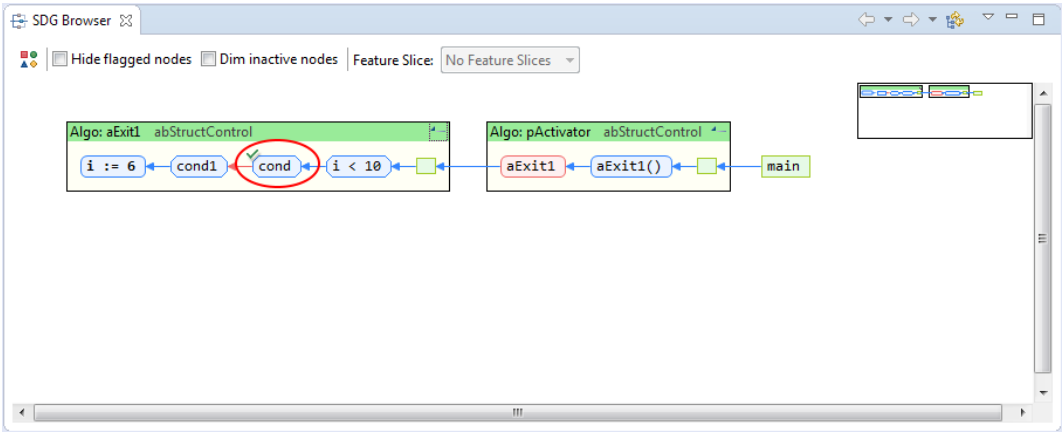

Figure 4.15: Alt+Click on a node to set/remove its flag

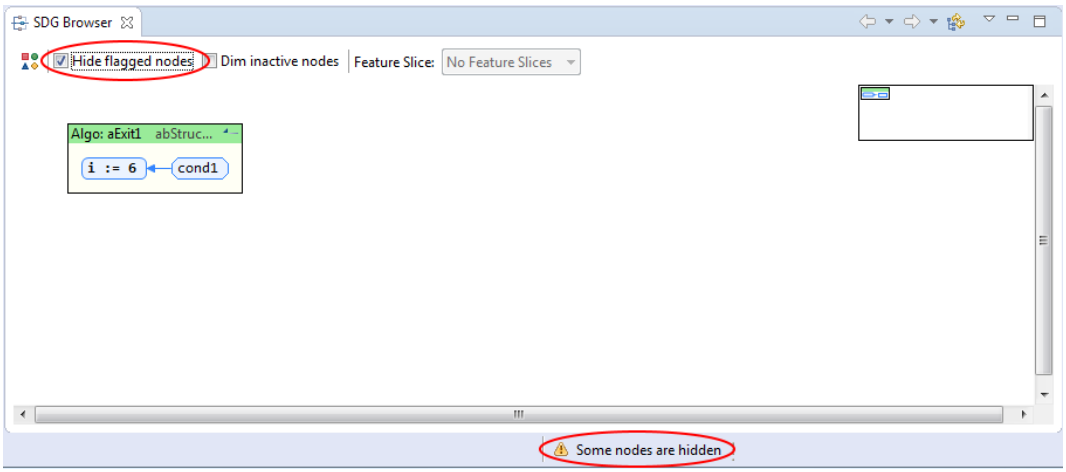

Figure 4.16: Flagged nodes can be hidden to reduce visual clutter

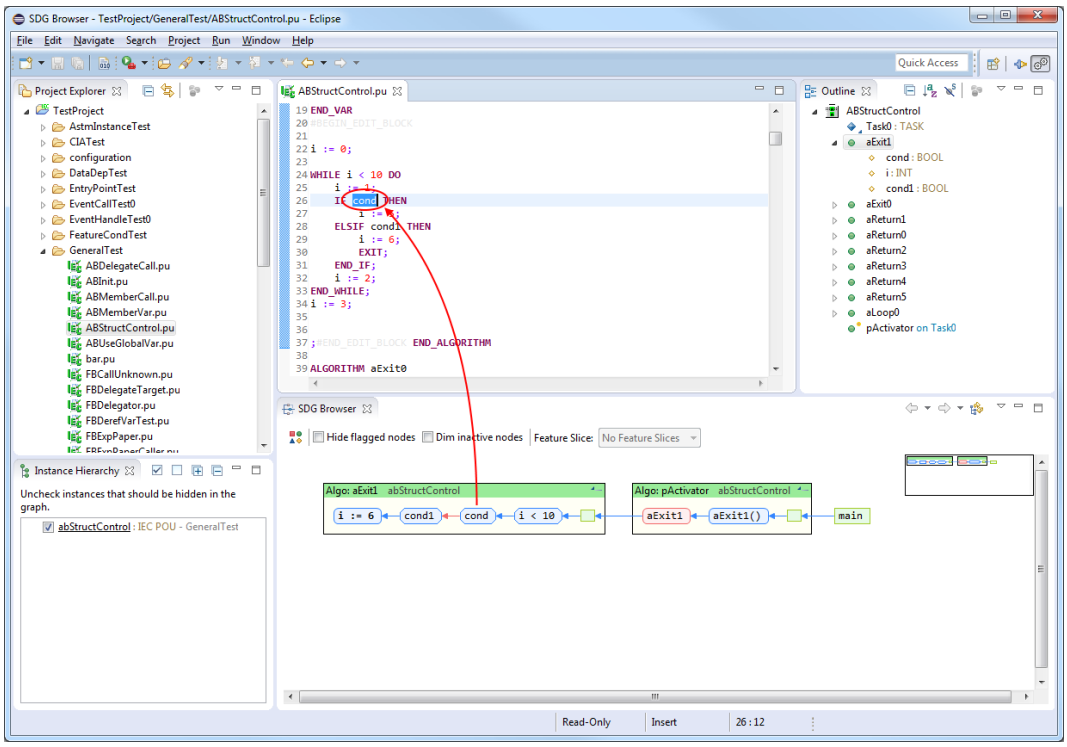

Figure 4.17: Navigation from SDG nodes to their source code is easily accessible (Ctrl+Click on a node, or from the context menu)

4.3.1 Dead Code Detection

Dead code detection is possible by using the hardware configuration of a project to *prune* the SDG (see [1] and [4]). The system expects the hardware configuration to reside in the directory `/prj/Source/IMM/configuration` (which is not configurable yet). To perform pruning, right click on an IEC project that has been built and select *Perform Pruning* from the *Configure* menu as shown in Figure 4.18.

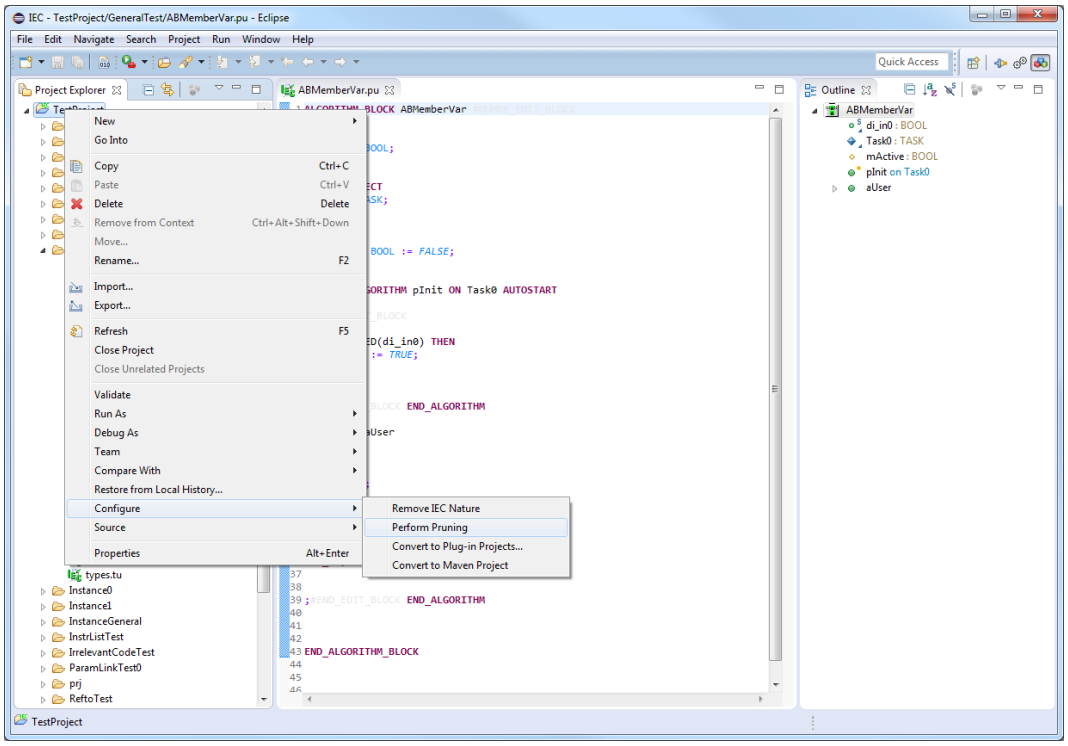

Figure 4.18: Run dead code detection for a project by *pruning* the SDG

After dead code detection has been executed, folders and files containing dead code will be decorated with a gray bubble. Inside files, dead code will be highlighted as in Figure 4.19, using a configurable Eclipse annotation. To configure the annotation style, go to *Preferences—General—Editors—Text Editors—Annotations* (Figure 4.20).

4.3.2 Zoom

The SDG Browser supports a few discrete semantic zooming levels. The default level shows nodes with their contents and one level of containers (the containing procedure). By scrolling down using the mouse wheel, additional zoom levels can be accessed. The first level minimizes nodes by hiding their contents, to better show the overall structure of the graph. The second level additionally groups nodes by the functional unit (FU) they are contained in. An illustration of the zoom levels can be seen in Figure 4.21.

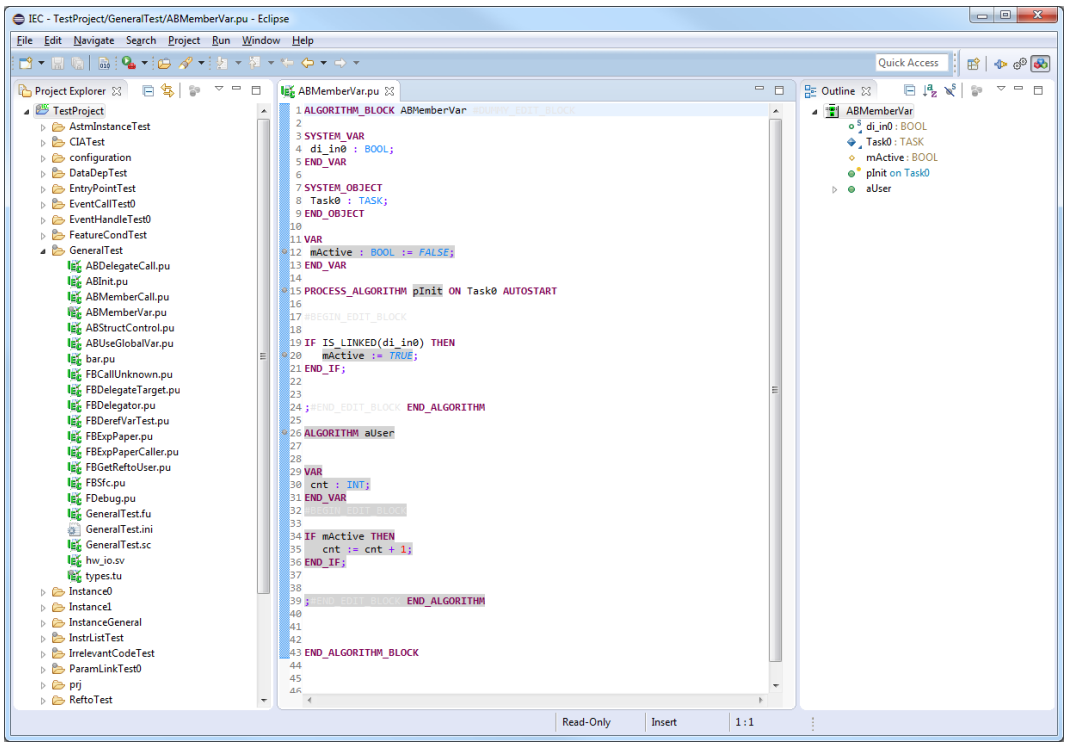

Figure 4.19: Editor and Project Explorer showing highlighted dead code

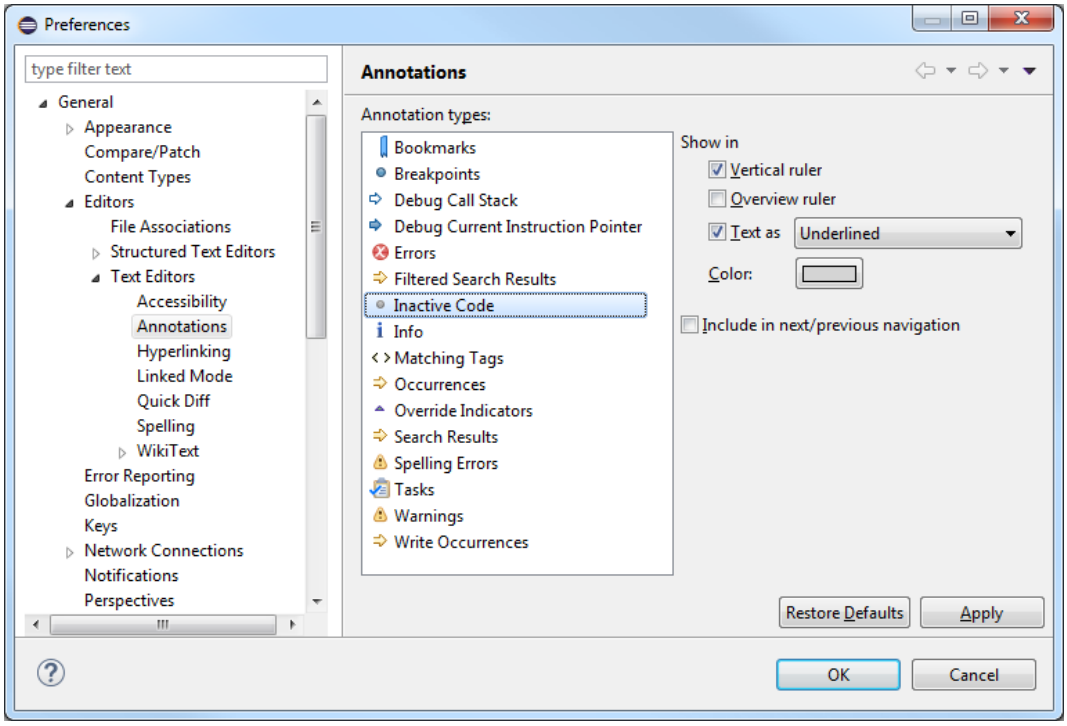

Figure 4.20: Preference page for customizing annotation styles

Figure 4.21: The same graph shown in different zoom levels

4.4 Examples

This section will provide examples for using the SDG Browser for some simple tasks. The examples are based on a test project that covers most of the language features of Keba's IEC 61131-3 implementation. This test project was chosen because it is far more manageable than one of Keba's real systems, plus it does not contain any proprietary code.

The resulting graph for each use case will only be outlined, refer to Chapter 3 for the exact semantics of each use case.

Executions

The first task will be to find out what causes an event handler to be called. The file `EventCallTest0/AB1.pu` defines the event algorithm `MyEventAlgo` which handles events of type `tevEvent1`. Since there is no explicit call to `MyEventAlgo`, we cannot find its activations by searching for references to its name. Instead, the algorithm is called whenever an event of type `tevEvent1` is set using `SET_EVENT`.

This is a case where the SDG Browser can be used for program comprehension rather than solving a specific problem. After selecting the event algorithm's name and selecting *Show Executions* from the editor context menu, the SDG Browser will show a graph similar to Figure 4.22.

The graph shows control dependencies and follows them backwards. This means it will show all conditional statements on which the target directly or indirectly depends, and all procedure calls that lead to their execution.

Notice that there is a call to `MyEventAlgo` from the main procedure. This is caused by the SDG being built in a conservative way regarding entry points. This means it is assumed that an event algorithm might be called, even if no event of the right type is set in code (i.e. the SDG is an over approximation of the real program semantics). In other words, every event handler is considered as an entry point, as are postupdate algorithms and autostarted process algorithms.

We can see that besides the call from `main` there is one other activation in the function block `FBEventTrigger`. Expanding the activation node will show that this activation is caused by a call to `SET_EVENT`. If we were interested in what could lead to the execution of `FBEventTrigger`, we can just expand the graph further. This will show two calls from two different algorithms, `MyAlgo0` and `MyAlgo1` (Figure 4.23).

Remember that you can navigate to the code for a node at any time by holding the control key and clicking on the node.

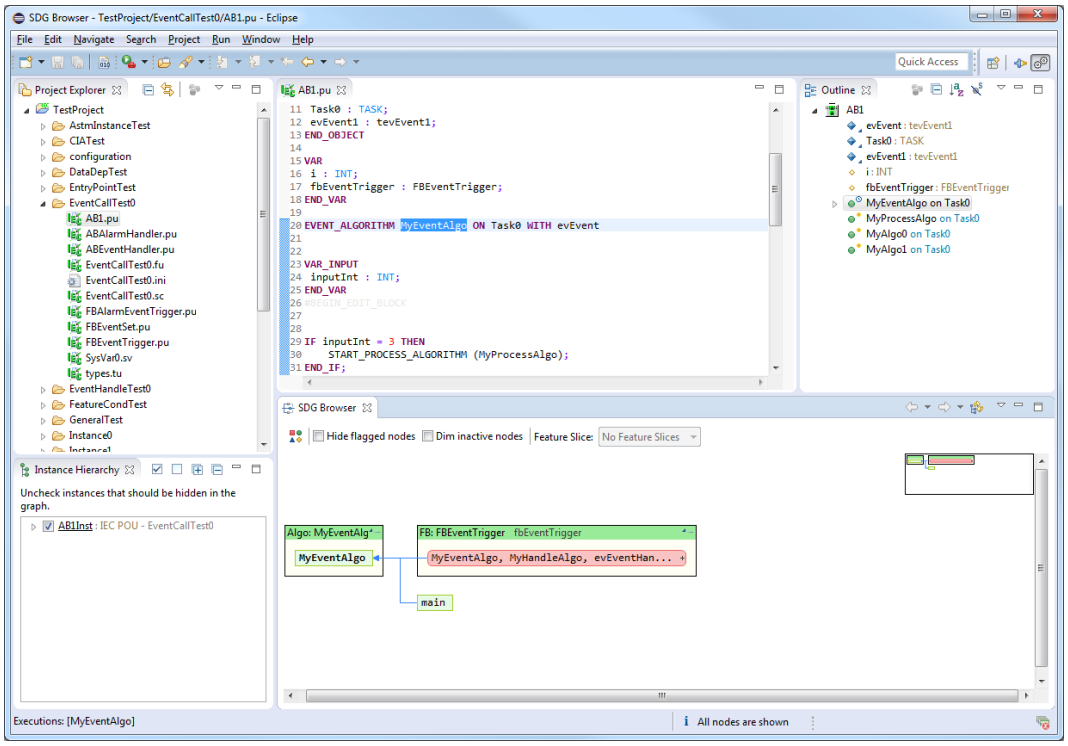

Figure 4.22: SDG for an event algorithm showing two calls

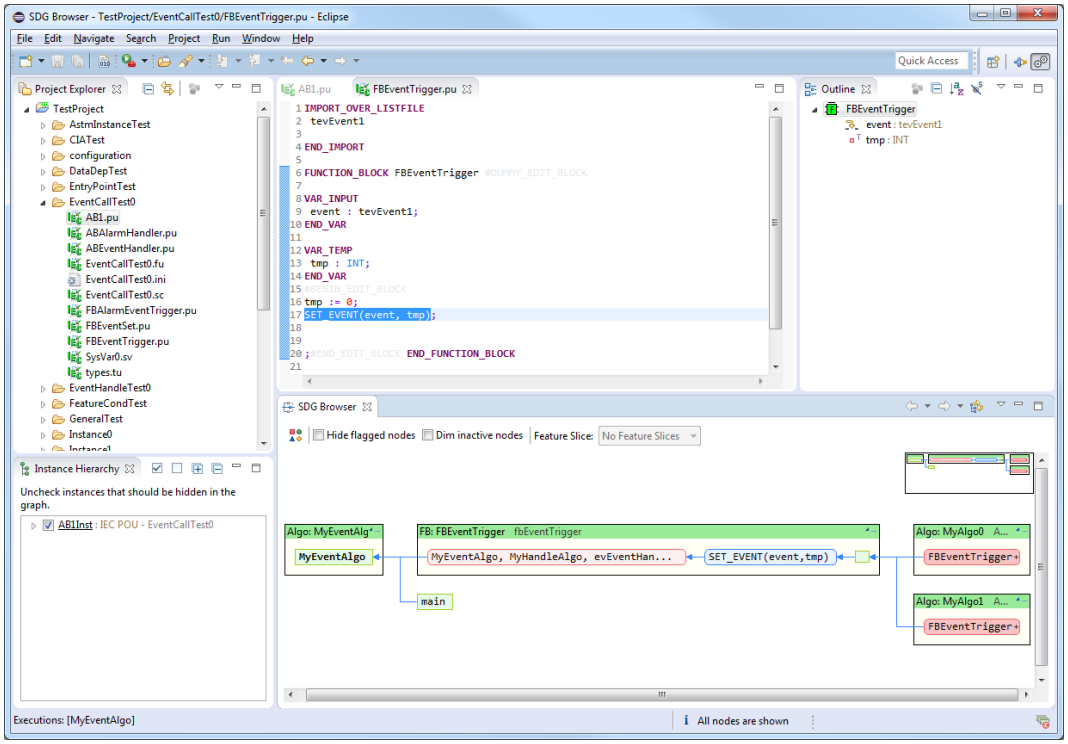

Figure 4.23: SDG showing two calls to the function block that sets the event

Variable Assignment

In this task we want to find all statements that modify the variable `i` in algorithm `ABStructControl.aExit1`. We already saw earlier that we can use *References in Project* to look for its references (Figure 4.12). But now we would like a more convenient way of showing which statements not only reference but modify the variable, and also why those statements are executed.

Figure 4.24 shows the graph after selecting *Show Variable Assignments* from the editor context menu, with the cursor placed in line 17. In this simple case all assignments are in the same procedure, but they might as well be scattered around the program (e.g. for global variables). They might not even be obvious, e.g. in the case of system variable aliases which are bound to different system variables depending on the configuration.

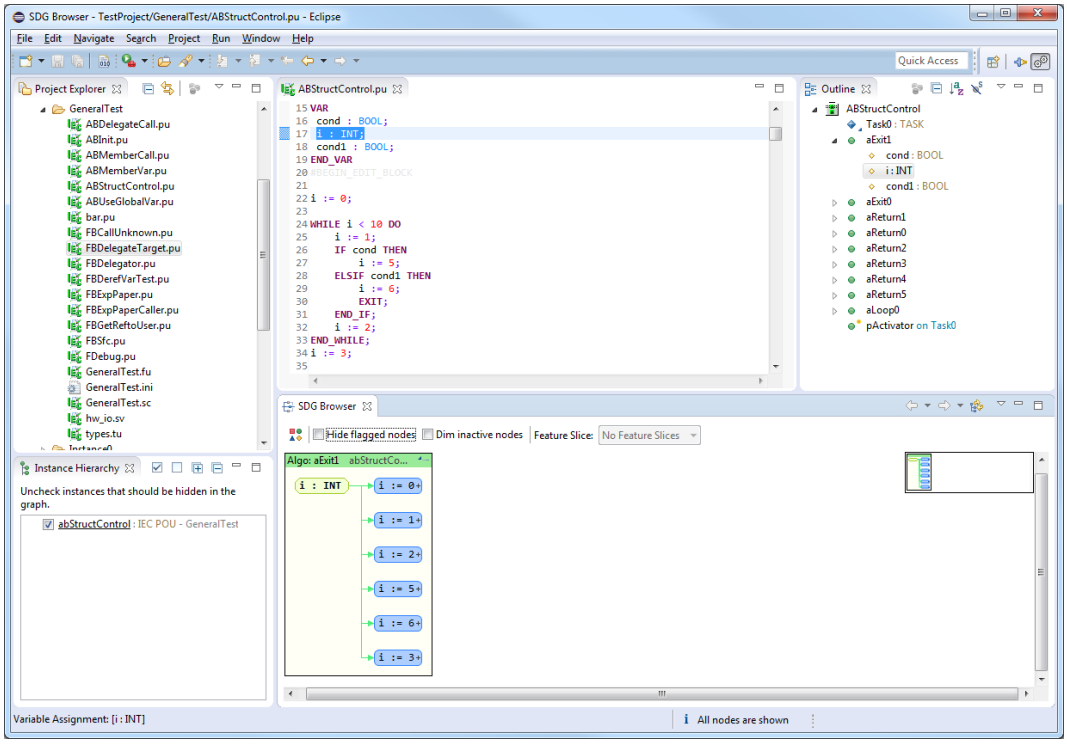

Figure 4.24: SDG showing all assignments of a variable

Since this use case just shows the *Executions* for each of the modifying statements, we can investigate what leads to their execution the same as before. Figure 4.25 shows the graph with some of the nodes expanded, illustrating the conditions that lead to the assignment of the value 5 or 6 to `i`.

We can also see that the algorithm `aExit1` is called from `pActivator`, which is an entry point to the program. Notice also that `aExit1` is always called, since there are no conditional statements between `main` and its activation node.

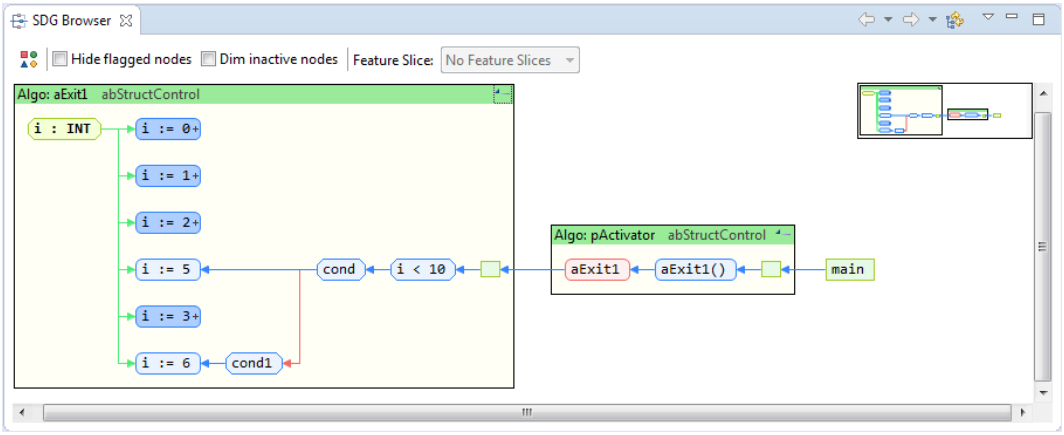

Figure 4.25: SDG showing the conditions that lead to two assignment statements

Change Impact

We now want to know how the value of the variable `ABDelegateCall.var0` can influence the program. Figure 4.26 shows the graph after selecting *Show Change Impact* from the editor context menu. There are some redundant data edges (as we will see when exploring the graph further). We can see two edges to statements `tmp := (in^ * 10)`, which will be explained shortly; one to a call where the address of `var0` is used as a parameter; and one edge to the corresponding actual parameter of that call.

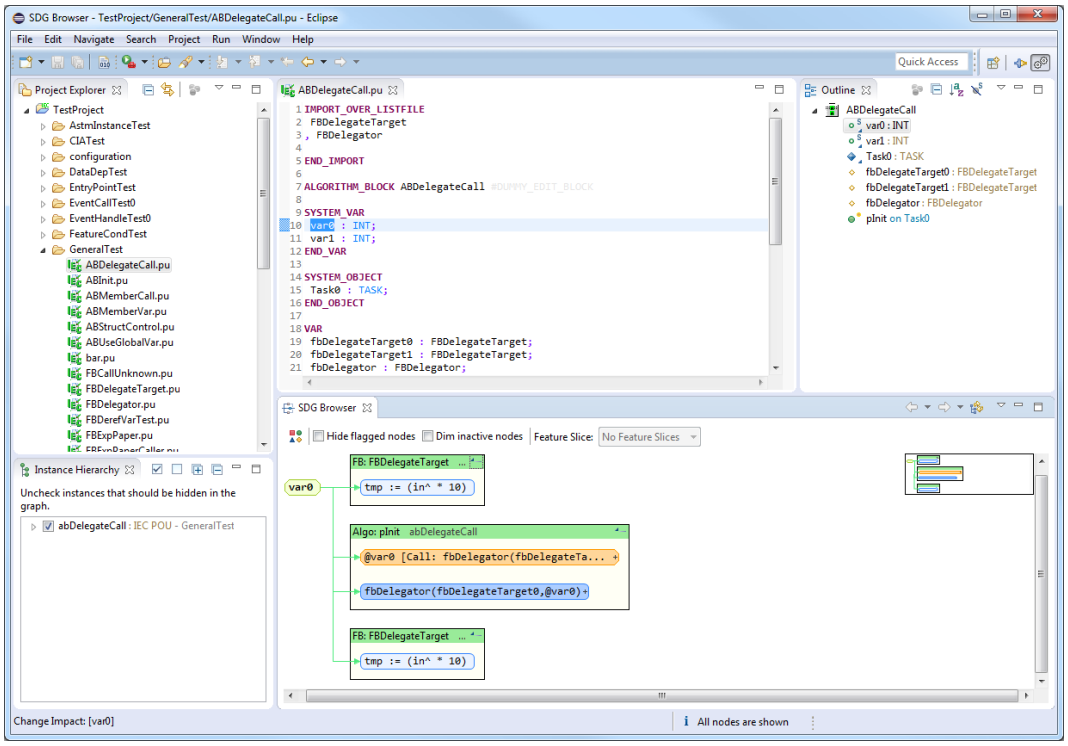

Figure 4.26: SDG showing the change impact for a variable

Since data is carried by parameters the next step will be to expand the actual parameter node, resulting in the graph shown in Figure 4.27. We can see that data flows from the actual parameter to formal parameter `ref` in procedure `FBDelegator`, which is then used in a statement. Examining the code reveals that the two actual parameters inside `FBDelegator` result from the fact that the called procedure `fbTarget` might have two values (i.e. sort of a function pointer).

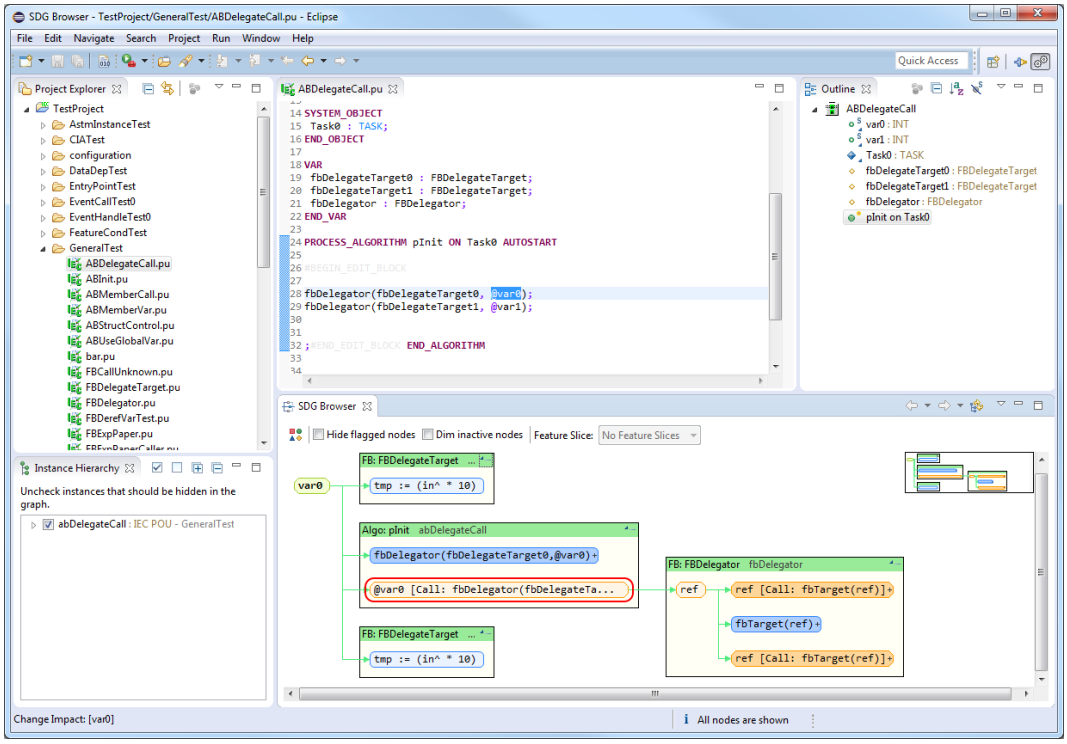

Figure 4.27: SDG showing data flow from an actual to a formal parameter

Expanding one of them results in a graph like Figure 4.28. We can again see data flowing to a procedure, from an actual to a formal parameter, where it is then used in a statement. Notice that this is one of the statements directly connected to the initial variable node. This shows that the SDG may contain some redundant data edges, which stem from the fact that the SDG is an over approximation of all possible dependencies.

Industrial Example

This last example just shows the SDG Browser being used with an application of an industrial scale. Figures 4.29 and 4.30 show a change impact for a conditional statement. Figure 4.31 shows the same graph zoomed out, giving a better overview of the overall structure of the graph.

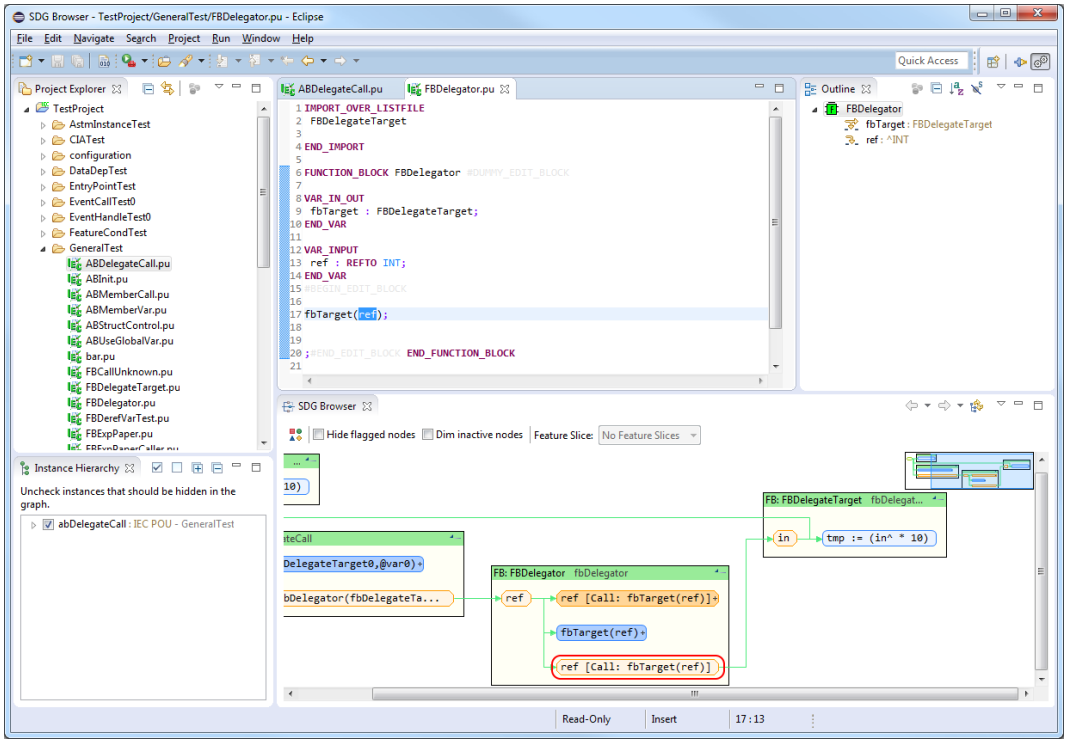

Figure 4.28: SDG showing redundant data edges to statement `tmp := (in^ * 10)`

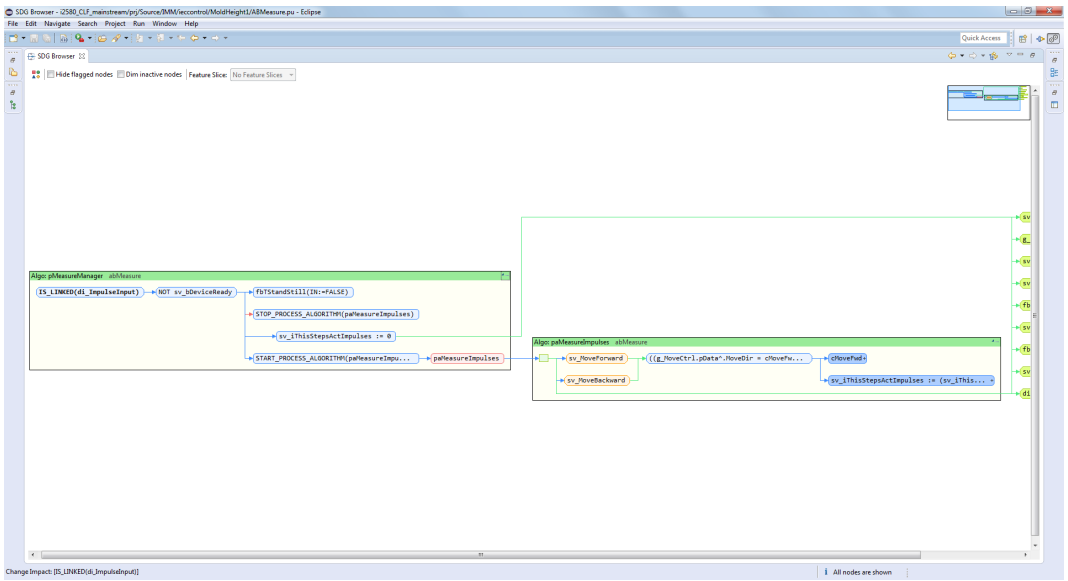

Figure 4.29: SDG showing a big change impact

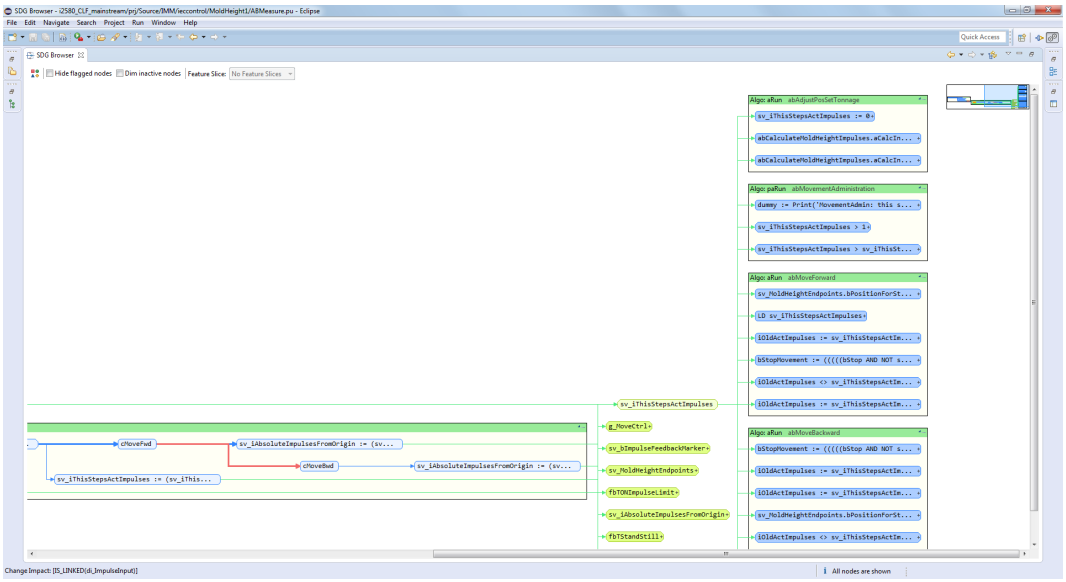

Figure 4.30: SDG showing a big change impact

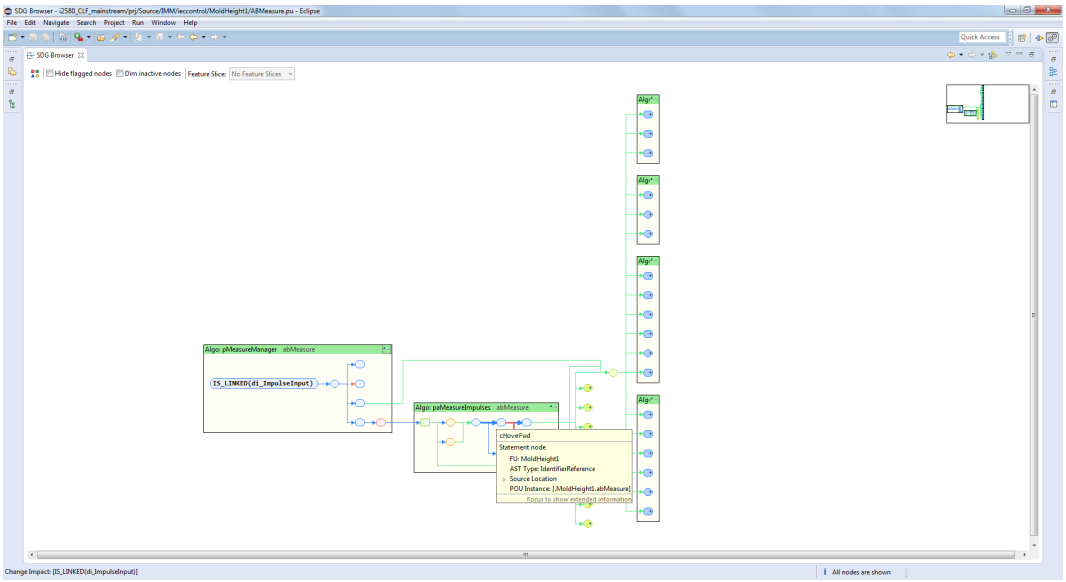

Figure 4.31: SDG showing a big change impact, zoomed out for better overview

5 Implementation

This chapter will describe the implementation of the system and will serve as an introduction for future developers.

The SDG Browser is implemented as a plugin for the Eclipse workbench. It depends on several other MEVSS plugins which provide the AST classes and infrastructure for building the SDG, as well as an Eclipse editor for viewing IEC files. The SDG Browser plugin provides two views (the *SDG Browser* view and the *Instance Hierarchy* view) and contributes items to the IEC editor context menu. Figure 5.1 illustrates the plugin architecture and interactions.

The plugin is implemented largely according to the *model-view-controller* (MVC) pattern, with the model being a use case which contains a so-called *display graph*. The display graph wraps the SDG to adapt it to the view and to augment it with artificial elements. Since the Eclipse view is currently a singleton view, it instantiates the controller when it is shown. The controller instantiates a model when it is asked to display a certain use case (by an Eclipse command or from IecEdit), it also keeps a history of previous models so the user can navigate between them.

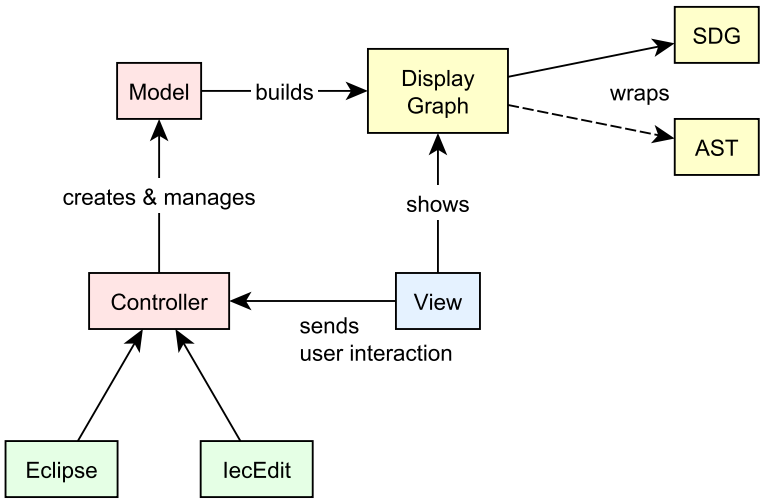

Figure 5.1: Overview of the plugin architecture

5.1 Overview

This section will give an overview of the packages containing the SDG Browser code and introduce important terminology along the way. All packages are part of a common root package `at.jku.mevss.featureide.browserview`.

The root package contains the `DependencyBrowserPlugin` class, which represents the plugin instance and provides methods for showing AST and SDG nodes in the IEC editor. Also part of the root package is the `DependencyBrowserController` class, which is the main entry point into the SDG Browser, as well as the `SDGBrowserSourceProvider` class, which makes the current model available across the workbench.

`model2` This package contains the classes implementing the use cases which derive from the `AbstractModel` base class. A concrete use case is implemented by deriving from that base class and providing strategies for selecting nodes and edges to be displayed. The `AbstractModel` instance then builds a display graph based on these strategies.

`model2.graph` Classes in this package represent the display graph, the model that abstracts from the underlying SDG and allows the addition of container nodes (for representing hierarchy) and artificial graph elements. This abstraction also allows the SDG Browser to work with different SDG types without having to change the view implementation. All nodes in the display graph can be in an *expanded* or *collapsed* state, meaning that their logical successors will be shown or hidden, respectively. Furthermore, container nodes can be *closed* so their children are hidden to reduce visual clutter.

All display graph elements derive from the common base class `DisplayElement` which provides support for attaching arbitrary properties. Classes `DisplayNode` and `DisplayEdge` represent nodes and edges, respectively, and also store layout information (position, size, bend points). The `DisplayGraph` class stores all nodes and edges representing the graph and provides support for adding new elements. This class also produces graph change events and provides a means for getting visible graph elements (which are *reachable* through expanded nodes).

`model2.hierarchy` The instance hierarchy is represented by the `HierarchyTree` and `HierarchyTreeNode` classes in this package. Every `AbstractModel` exposes an instance hierarchy tree for its current graph contents.

`model2.layout` This package defines the classes used for the graph layout. The only implementation of interface `GraphLayouter` is `KLayLayouter`, which uses the *KLay Layered*[1] layout algorithm.

[1] KLay Layered is a layer-based layout algorithm and part of the KIELER project, it is currently

`event` This package defines listener interfaces and event objects for several events: view changes (a new model being displayed), selection of a node in the view, changes to a `DisplayGraph`, and changes to `AbstractModel` properties.

`command` This package contains classes for interfacing an editor selection (path and source code range) with the SDG Browser controller, as well as the class `SDGBrowserCommand`, which represents a command that operates on display graph elements. Subpackage `eclipse` defines Eclipse command handlers that plug into the IEC editor context menu, subpackage `pipe` defines handlers that receive commands via a Windows named pipe.

`command.feature` The interfaces in this package allow an arbitrary graph to be displayed in the SDG Browser by providing a *feature* which may contain a number of *slices* (sets of nodes to include in the graph). This will be used by the FORCE platform [8] to display a graph of those parts of the program that belong to a certain feature.

`views` This package contains the views contributed to the Eclipse workbench. Class `InstanceHierarchyView` and supporting classes implement the *Instance Hierarchy* view, which displays the current model's hierarchy tree. The `BrowserView` interface is used by the controller to render a display graph and is partially implemented by the `AbstractBrowserView` base class. This class is an Eclipse *view part* that provides several controls for setting model properties, but does not implement graph rendering itself.

`views.draw2d` This package and its subpackages contain the classes which implement the `BrowserView` interface using Draw2D[2]. The implementation class `Draw2DBrowserView` extends `AbstractBrowserView` and uses the `GraphCanvas` class for rendering a display graph. The `Resources` class exposes all the colors and dimensions used for the displayed graph elements via static methods, so its implementation may be changed in the future to allow Eclipse settings to be taken into account.

The `GraphCanvas` class implements most of the heavy lifting to support transitions when graph elements enter or exit, implements zooming and hit testing, and handles events on displayed graph figures. The `animation` subpackage contains classes for supporting animations of Draw2D figures; the `overview` and `tooltip` packages implement the scrollable overview thumbnail and tooltip controls, respectively; all figures and decorations used for rendering are part of the `figures` package.

being integrated into Eclipse as part of the Eclipse Layout Kernel (ELK). For more information on KLay Layered see `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/KLay+Layered` and [13].

[2]Draw2D is a lightweight rendering toolkit on top of SWT, the UI toolkit used by Eclipse.

5.2 Model

The model part of the MVC architecture is mainly split in two parts, the display graph and the use cases. Each use case is represented by a class which inherits from `AbstractModel` and implements the strategy pattern [6, pp. 315–323] for adding elements to the graph when nodes are expanded by the user. Each model instance creates a display graph and populates it according to the implemented strategy.

An overview of the model classes can be found in the class diagram in Figure 5.2. It shows roughly what the structure of a model looks like. The subclasses of `AbstractModel`, `DisplayNode`, and `DisplayEdge` are not shown and will be introduced in detail later on.

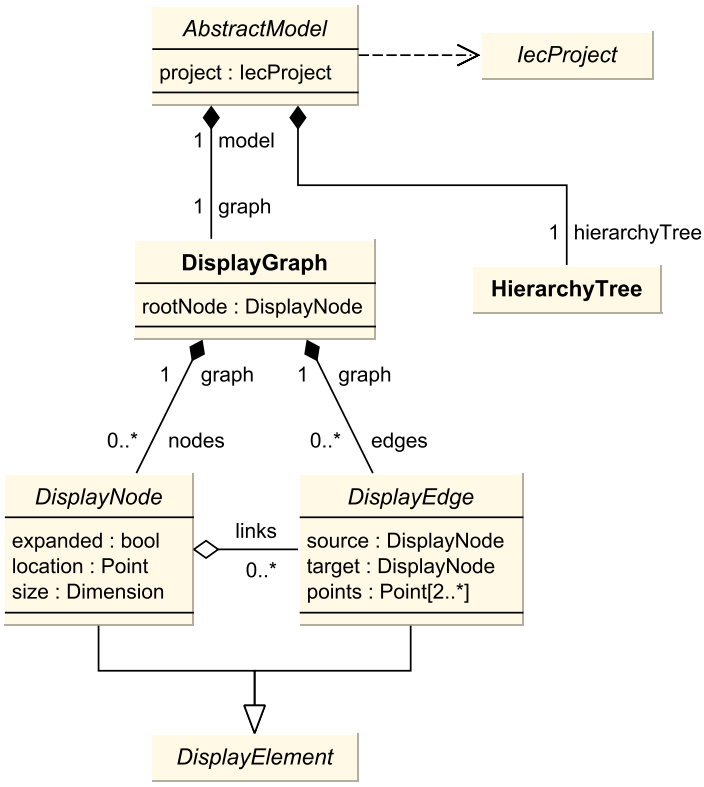

Figure 5.2: Class diagram overview of the model classes

The SDG Browser uses build infrastructure provided by an Eclipse plugin for viewing IEC files, which was developed in the course of Alois Mühleder's bachelor thesis [11] at the CDL MEVSS. The plugin is used to build the SDG from an IEC project and also provides ways to navigate the IEC code (e.g. *Open Declaration*, search for IEC elements like types and variables). The `IecProject` interface is part of that plugin and provides access to the AST and SDG after a build is completed.

5.2.1 Display Graph

As mentioned in the overview, the display graph is a model that wraps the SDG and abstracts from it to provide a uniform interface for the view.

Class `DisplayGraph` contains all nodes and edges and maintains maps from keys to nodes and edges, respectively. Keys are objects that uniquely identify a graph element and are used to determine whether a node or edge is in the graph. For example, a `DGDisplayNode` is uniquely identified by the `DGNode` it wraps, so that is its key; a `VariableDisplayNode` is uniquely identified by a `DataDefinition` (the AST node representing the local variable) and the containing SDG procedure node, so its key is a tuple of those two objects. Furthermore, each display graph has one root node, which is the entry point of the graph (the code element being investigated, or a group thereof).

The `DisplayGraph` class also provides methods for adding new nodes to the graph, instances of node classes are created internally as they are needed. This encapsulates creation of new nodes to make sure they are initialized only once and to make sure there is only one instance of a node for a particular key. That means in case a node already exists in the graph it is just returned. Edges are added to the graph only through nodes, which provide different methods to add edges, depending on the node type. Since every edge in the graph is owned by a node there is no way to add edges to the graph directly.

Finally, class `DisplayGraph` provides method `collectReachableGraph`, together with various convenience methods wrapping it, which allows the currently reachable nodes and edges in the graph to be collected into provided sets. A predicate may also be specified which is used to filter nodes and edges to be collected. A node in the graph is reachable if there is a path to it from the root node through expanded nodes and edges that are not filtered only.

Both display graph nodes and edges inherit from the abstract class `DisplayElement`. This class declares several abstract methods that all classes for display elements need to implement, two of which are `getKey()` and `accept(GraphVisitor<T>)`. The former returns the unique key for any graph element, the latter must be implemented in every leaf class of the hierarchy and implements the visitor pattern [6, pp. 331–344] for graph elements. Furthermore, class `DisplayElement` provides support for properties and tooltip data. Properties are used to store arbitrary information with a graph element (e.g. whether a node is flagged or hidden) that is not meant to be viewed by the user. Tooltip data, on the other hand, is meant to be shown to users in a tooltip for any graph element.

Nodes

A diagram showing all classes of display graph nodes can be seen in Figure 5.3. Abstract class `DisplayNode` represents a node in the graph. It stores layout information (location and size), a set of links (edges owned by the node), the parent node, and whether or not the node is expanded. It also defines abstract methods `getName()` and `getType()` for the view.

`DisplayNode` has two abstract subclasses: `VirtualDisplayNode` represents an artificial node that has no corresponding SDG node; `SDGDisplayNode` represents a node backed by an actual SDG node. The abstract class `ContainerDisplayNode` represents a container for other nodes. It adds insets for the layout and stores a set of child nodes.

The only subclass of `SDGDisplayNode` at the moment is `DGDisplayNode`, which is backed by a `DGNode`. It implements `getName()` and `getType()` in terms of utility methods provided by class `GraphUtil`. To add support for other SDG types, alternatives need to be added for three classes only: `DGDisplayNode`, `DGContainerDisplayNode`, and `VariableDisplayNode`. The latter could instead be changed to store the procedure instance using `AstmInstance` instead of a procedure node.

Edges

A diagram showing all classes of display graph edges can be seen in Figure 5.4. Abstract class `DisplayEdge` represents an edge in the graph. It stores the source and target display nodes and layout information (start and end points, optional bend points). It also defines the abstract method `getType()` for the view.

`DisplayEdge` has two abstract subclasses: `VirtualDisplayEdge` represents an artificial edge that has no corresponding SDG edge; `SDGDisplayEdge` represents an edge backed by an actual SDG edge.

The only subclass of `SDGDisplayEdge` at the moment is `DGDisplayEdge`, which is backed by a `DGEdge`. To add support for other SDG types, alternatives need to be added for two additional classes only: `DGDisplayEdge`, and `SyntheticDisplayEdge`. The latter connects two arbitrary `DGNodes` and only needs an alternative in case such edges should exist for nodes in the new SDG as well.

Links are added to the graph via the protected method `addLink` provided by class `DisplayNode`. Subclasses of `DisplayNode` may provide concrete methods for creating edges from SDG edges or for creating artificial edges. `DGDisplayNode`, for example, provides a method for adding an edge from a `DGEdge`, and another method for adding a synthetic edge of arbitrary type to any other `DGNode`.

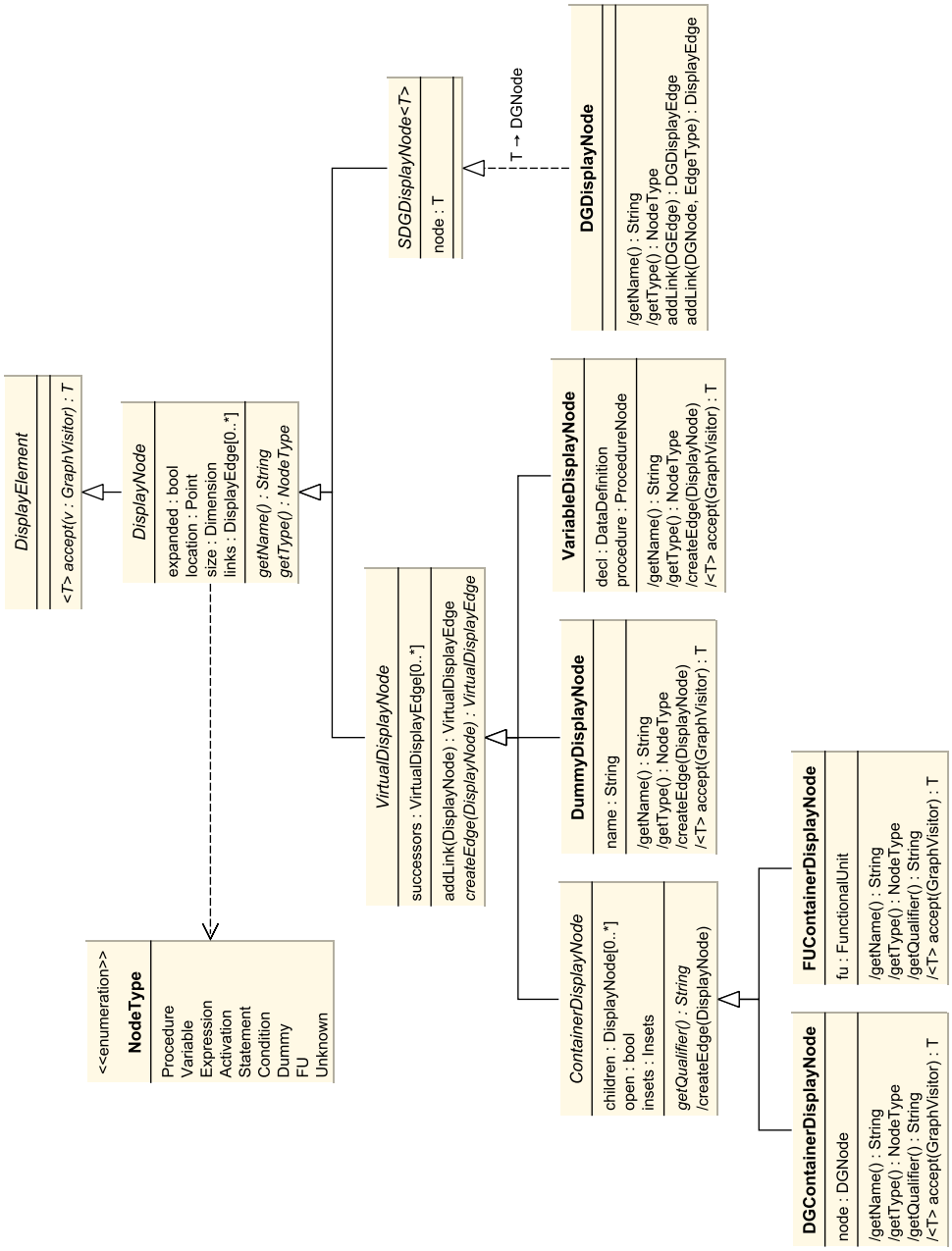

Figure 5.3: Class hierarchy of display graph nodes

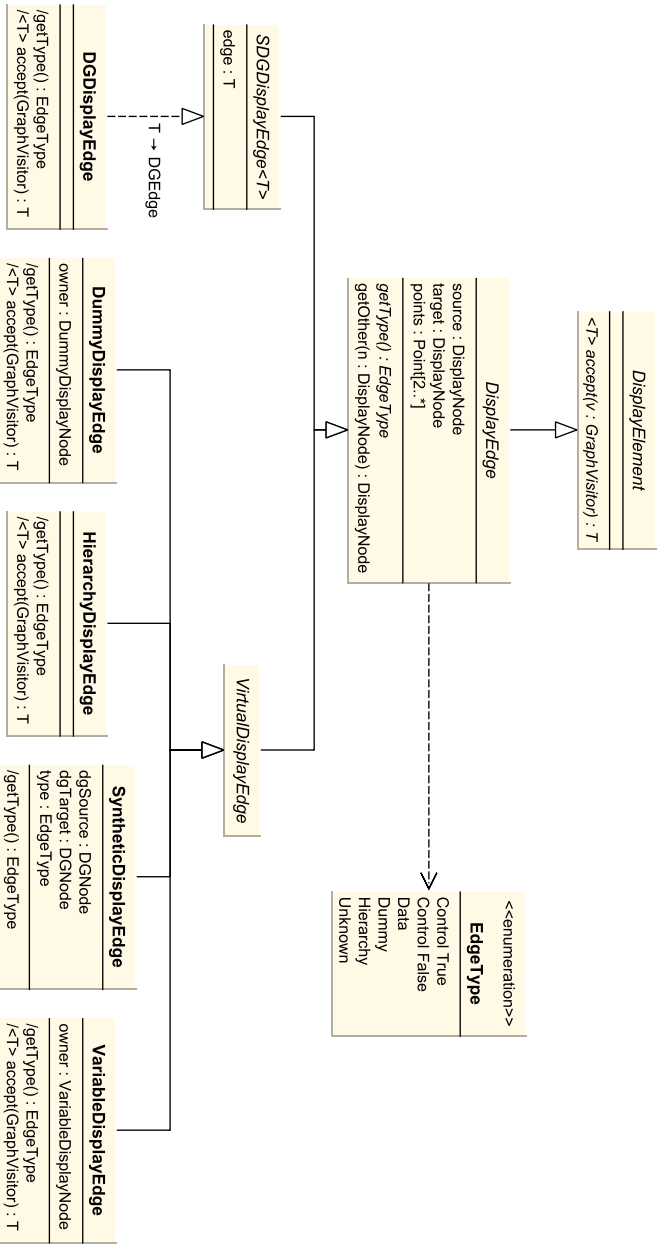

Figure 5.4: Class hierarchy of display graph edges

5.2.2 Use Cases

Use cases are implemented as concrete classes derived from `AbstractModel`. This class contains the display graph and instance tree and maintains properties which are used to store the view configuration (e.g. whether flagged nodes are hidden). The abstract method `getType()` must be implemented by concrete use cases and gets the name of the use case (i.e. the type of model). Figure 5.5 shows a diagram of the concrete model classes.

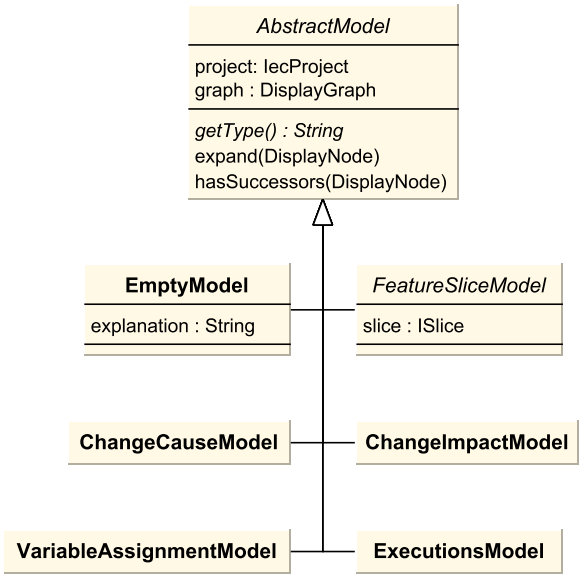

Figure 5.5: Class hierarchy of use cases

The two most important methods in the `AbstractModel` class, which implement a model's strategy for adding successors of a node to the graph when it is expanded, are `hasSuccessors(DisplayNode)` and `expand(DisplayNode)`. Their default implementations delegate to overloads for concrete display node types. Subclasses need only implement (some of) them, depending on what types of nodes are created.

The view uses `hasSuccessors` to determine whether a node can be expanded to indicate this to the user. It should follow the same logic as `expand` so the view is consistent with the model semantics. Nodes are added to the graph in the `expand` method, whenever a user clicks on a node this method is called to add successors of the clicked node to the graph.

There are two model classes which do not correspond to one of the use cases defined in Chapter 3: `EmptyModel` and `FeatureSliceModel`. The empty model is used whenever there are no instances of a selection, or if a feature slice is empty, and contains one dummy node with an explanation. Class `FeatureSliceModel` is used to display an `ISlice`, which is basically an arbitrary set of SDG nodes. The current implementations of forward and backward slice models only follow control edges.

The other four `AbstractModel` subclasses each implement one concrete use case of the SDG Browser. They each override only those methods for node types which actually occur in the graph, for example `ExecutionsModel` never creates any variable nodes so it doesn't overload `hasSuccessors` or `expand` for those nodes.

5.3 Controller

The controller is the main entry point into the SDG Browser, its interface is defined in class `DependencyBrowserController` and implemented in `BrowserControllerImpl`. A controller is created for a specified `BrowserView` instance, but could easily be adapted to support multiple views. The controller registers itself on the view to receive callbacks for events such as nodes or context menu items being selected (by implementing the `BrowserView.Callbacks` interface). Figure 5.6 illustrates that part of the architecture.

The controller in this implementation has four responsibilities:

1. maintaining a history of displayed models and navigating between them;
2. providing events for node selection and view changes;
3. creating and showing new models when requested; and
4. translating events from the view to actions on the model.

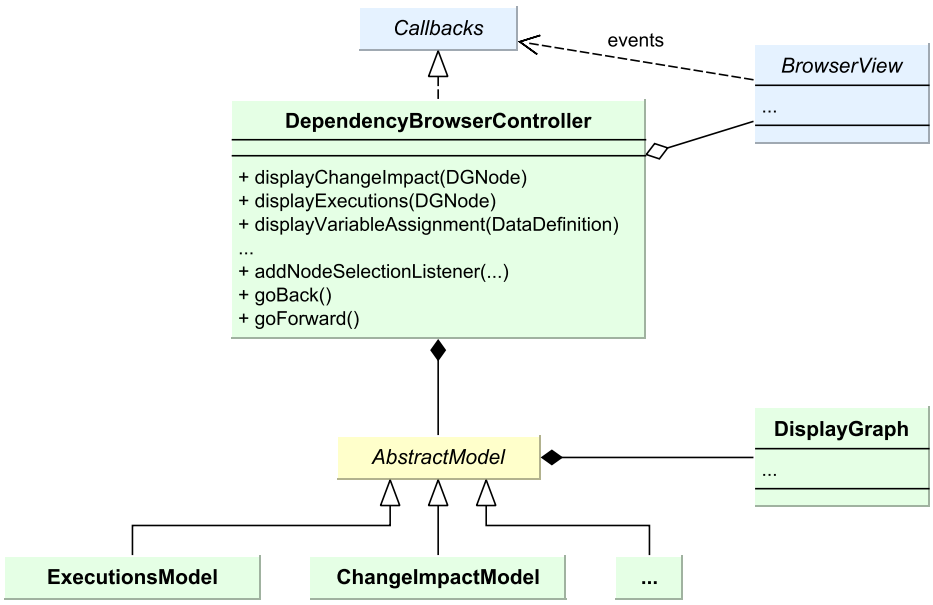

Figure 5.6: Overview of the controller architecture

Creating and showing new models is done whenever one of the `display` methods on the controller is called (there are display methods for each of the use cases, with

overloads for AST and SDG nodes). The controller checks whether the requested model is applicable to the specified node and performs preprocessing where necessary (e.g. gathering SDG nodes for specified AST nodes).

Thus, when new use cases are to be added, corresponding display methods need to be added to the controller.

Translating events from the view to actions on the model is straightforward, the only exception to the usual MVC pattern is the *model change request* callback. By using this callback the view can request from the controller that it show another model than the current one. This is currently only used when a feature slice model is displayed. The user can then select a different slice from the same feature, which results in a different model being shown.

5.4 View

The view is defined by interface `BrowserView`. The main methods of this interface are `setModel(AbstractModel)` and `update()`. It also declares methods for showing notifications and a context menu. Events generated by the view are communicated via the `BrowserView.Callbacks` interface, i.e. the view accepts an object implementing this interface to send events to (typically the controller).

A partial implementation of the `BrowserView` interface is `AbstractBrowserView`, an Eclipse `ViewPart` that provides some of the logic and plugs into the Eclipse workbench. This singleton view instantiates a controller when it is initialized and publishes that controller via the plugin class. This way the controller can be accessed from anywhere in the plugin. Figure 5.7 illustrates the view architecture.

The Eclipse view part provides a container for controls and defines a number of controls itself. Controls are provided for navigating between models (forward, backward), changing whether flagged nodes are hidden, changing whether to dim inactive nodes, and for selecting the feature slice to show. The view part also provides infrastructure for saving the state of the view (control, zoom state) when another model is displayed, using model properties for storage. Subclasses can override methods `fillLocalPullDown`, `fillLocalToolBar`, `fillLocalStatusLine`, and `addControls` to add new controls and status line items. Methods `saveState` and `restoreState` can be overridden to add to the view state.

5.4.1 Draw2D View

The class `Draw2DBrowserView` is a concrete view implementation which inherits from `AbstractBrowserView` and uses Draw2D for rendering the graph via a `GraphCanvas`.

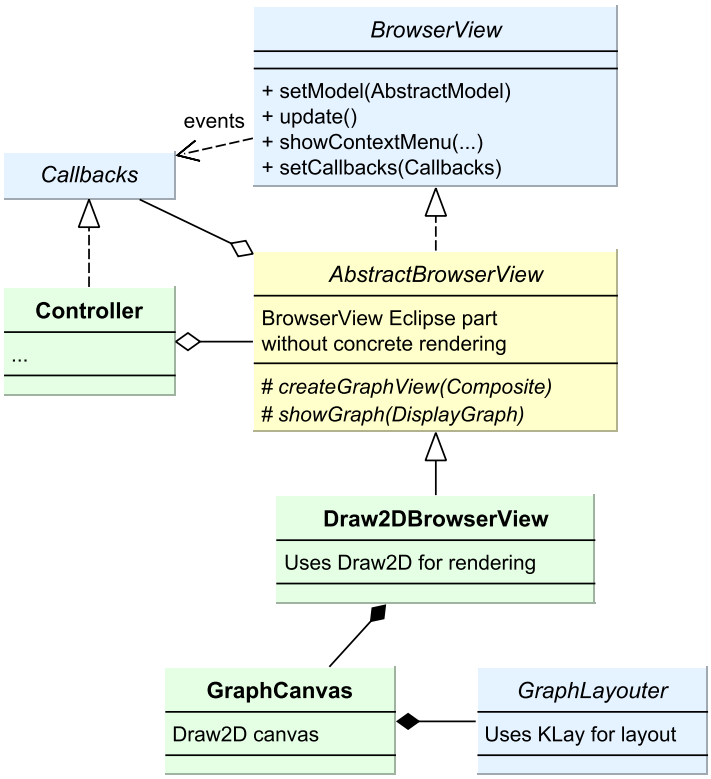

Figure 5.7: Overview of the view architecture

The `GraphCanvas` class is responsible for tracking which display graph elements are displayed in the view and maps display graph elements to their figures. It also provides hit testing (currently only used for finding an element for a tooltip when hovering over the view). When the displayed graph changes, it provides sets of *entering* and *exiting* elements (new elements not yet displayed and elements no longer in the new graph, respectively). The `GraphCanvas` also translates events on figures to calls on the view `Callbacks` via listener classes `NodeListener` and `EdgeListener` registered on the respective figures.

Layout and rendering is implemented in three phases in the `GraphCanvas` class.

0. Before the first phase, reachable graph elements are collected and the *enter* and *exit* sets calculated. The layout for the graph is calculated at this point using a `GraphLayouter` implementation (currently only KLay is supported).

1. In the *pre-layout* phase figures are created for entering elements and child elements are reassigned to entering containers if necessary.

2. At the beginning of the *layout* phase the current state of all figures is captured for the animation. Figures get assigned their new geometry (position, size or bend points) from the layout calculated earlier and their properties are updated from the corresponding display graph element. At the end of the layout phase

the new state of all figures is captured again and the transition starts.

3. In the *post-layout* phase, after the animation has finished, exiting elements are finally removed from the canvas.

Node and edge figures are represented in the view by `AbstractNodeFigure` and `AbstractEdgeFigure`, respectively. Each of them has one concrete implementation which is initialized and updated from a display graph element. The `draw2d.figures` package also contains several supporting classes such as borders and decorations.

The `draw2d.animation` package provides support for animations via classes `Animation` and `Animator`. There are several concrete implementations for `Animator` that support transitions for edge routing and node bounds. Although Draw2D supports some primitive animations itself (in fact most of the animation code is taken directly from this implementation), there is no support for asynchronous animations or for animating anything other than layout and routing.

5.5 Commands

The command classes interface the SDG Browser controller with other user interface components. The Eclipse commands inheriting from `AbstractEditorCommandHandler` work with selections in the IEC Editor, they are registered via the SDG Browser `plugin.xml` as entries in the IEC Editor context menu.

Another kind of command is represented by class `SDGBrowserCommand`. These commands work on display graph elements and are mainly used internally by the SDG Browser. There are a number of predefined commands which represent entries in the context menus for nodes and edges in the SDG Browser view. The base class and predefined commands are public, however, so they may be used externally as well, for example in response to node selection events.

5.5.1 External Command Interface

The SDG Browser also has an interface for use by external applications (e.g. Keba's IecEdit) which uses Windows named pipes. Commands received via that interface are handled by classes inheriting from `PipeCommand`. This type of command usually takes a file path and source code range (the editor selection) as its input, maps it to AST nodes, and then invokes methods on the SDG Browser controller.

The SDG Browser listens on a Windows named pipe at `\\.\pipe\\CodeProjectPipe` by default. It accepts commands as strings, according to the grammar in Listing 5.1. Method `parseCommand` in the plugin class dispatches received commands to the appropriate `PipeCommand` subclass. This is the place to add new commands.

Listing 5.1: Grammar for commands received via the Windows pipe interface

```
Message        = Command ' ' [ Instance ' ' ] SourceLocation .
Command        = ident .
Instance       = '[' ident { '.' ident } ']' .
SourceLocation = Path '(' Range ')' .
Path           = '"' filepath '"' .
Range          = integer ',' integer [ ',' integer ',' integer ] .
```

5.6 Extensions

The purpose of this chapter was not to describe every detail of the implementation, but rather to give a course overview of the general architecture and of possible places where extensions can be made. Together with the source code documentation, it should not be too difficult for new developers to get familiar with the system. This last section will briefly summarize the changes necessary for two types of extensions: adding new use cases and adding support for new SDG types.

The following changes are necessary to add a new use case.

- Create a new class derived from `AbstractModel`, or any existing use case, and implement the semantics by overriding `hasSuccessors` and `expand` as necessary. The new class will typically provide factory methods that create and initialize an instance from one or more AST or SDG nodes.
- Add `display` methods to `DependencyBrowserController` for the new use case, with overloads for AST nodes and typically also for SDG nodes.
- Add an Eclipse command handler for the new use case by inheriting from `AbstractEditorCommandHandler`, and register that handler in the SDG Browser `plugin.xml` file.

Adding support for a new SDG type is a little more involved since existing use cases and several other classes need to be updated as well. The following steps will likely be necessary (note that changes to the view are typically not required).

- Although not part of the SDG Browser itself, most likely a new project builder needs to be added to the IEC editor core plugin, and support for the new SDG type needs to be added to `IecProject`.
- Add new display graph nodes and edges for the new SDG types (typically subclasses of `SDGDisplayNode`, `SDGDisplayEdge`, and, depending on the SDG structure, of `ContainerDisplayNode`). The `DisplayGraph` class will need to be extended with methods to add nodes and edges of the new types. New types for classes like `SyntheticDisplayEdge` and `VariableDisplayNode` may be needed as well.

- Add overloads for the new types to the `GraphVisitor` class. Find all classes inheriting from `GraphVisitor` and override the new methods where necessary.
- Handle new node types in all existing use cases where necessary, and also provide factory method overloads for creating models from the new node types.
- Add overloads of `display` methods to `DependencyBrowserController` for all existing use cases which can accept a node of the new SDG type. If necessary, change existing `display` methods that accept AST nodes to create models for the new SDG type by default (i.e. map AST nodes to nodes of the new SDG instead of an old SDG).

6 Summary

This thesis presented the SDG Browser, an interactive tool for exploring system dependence graphs (SDGs). First we described the structure of the SDG and how it represents certain programming constructs, such as function calls and methods, as well as the additions and simplifications applied by the tool. The tool was created with four concrete use cases in mind, which we discussed by means of examples. The next part of the thesis was written as a user's manual for the SDG Browser and the related IEC project infrastructure. It shows the steps for creating and building a project, using the SDG Browser for concrete use cases, and explains additional features such as dead code detection. Lastly, we discussed the implementation, with a particular focus on details that may be important to someone trying to understand the SDG Browser code base and extend it.

The SDG Browser was developed in the course of our cooperation with Keba AG at the *Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems* (CDL MEVSS for short). Florian Angerer developed the SDG during his PhD work. The goal of the SDG Browser is to make it possible to interactively explore that SDG and use it for program comprehension and evolution.

The SDG is built based on the AST of an IEC program. First the PDGs are built which encode control and data dependencies within a single procedure. Then PDGs are instantiated and connected to form the object-sensitive (i.e. instance-based) SDG. The SDG contains nodes for statements, calls, procedures, and parameters; it contains edges for control and for data dependencies. The SDG Browser also introduces container nodes and dummy nodes.

There are four concrete use cases identified by Keba developers. The *Executions* use case may be used to identify how a statements may be executed, by exploring control dependencies of a statement or procedure. The *Variable Assignment* use case shows all statements which modify a particular variable, and for those statements shows their Executions. The *Change Impact* use case follows both data and control dependencies starting from any node in the SDG. It can be used to determine which statements or variables are influenced at all by the target node. Finally the *Change Cause* use case is an inverse change impact, i.e., it can find variables or statements

which influence the execution of a particular statement or the value of a variable.

The SDG Browser as an Eclipse plugin uses the build infrastructure provided by the IEC editor plugin. The SDG Browser operations can be accessed from the IEC editor context menu, navigation to the code from the SDG Browser is also easily accessible. The SDG Browser provides convenient features such as marking and hiding nodes that have been examined. Dead code detection based on a hardware configuration is possible, dead code is then highlighted in the editor and the SDG Browser.

The SDG Browser is implemented as an Eclipse plugin in a model-view-controller pattern. The model represents a use case and builds a so called display graph based on the user interactions. The display graph is an abstraction of the SDG and AST, which allows the view to be implemented independently of the underlying SDG type. An SDG Browser view only needs to implement an interface to be able to work with the controller. There is one concrete implementation for the Eclipse workbench, which uses Draw2D for rendering graph objects. The SDG Browser may be controlled from within Eclipse, but can also accept commands trough an interface for external programs, which uses named pipes for communication.

6.1 Experiences

My first approach to the SDG Browser used the D3.js framework in a browser embedded into the Eclipse IDE[1]. D3.js was chosen because of it's ability to quickly create appealing visualizations, combined with built-in support for transitions. However, we soon discovered that this would lead to significant code duplication. Some of the supporting data structures would need to be implemented both on the Eclipse side in Java and on the browser side in JavaScript. Communication between both sides also needed to happen through JSON-serialized objects, with was another unnecessary burden. We therefore decided to implement the view in pure Java using the Draw2D framework, which is native to Eclipse.

We also discovered that providing a stable layout for an incrementally built graph is rather difficult. Our layout algorithm of choice (KLay) handles this case more or less gracefully, however there are still significant layout changes at times. More research is required to provide a stable layout for changing graphs to support a nice user experience.

[1] We used JxBrowser by Ukrainian company TeamDev. The library provides a Chromium browser and works flawlessly when integrated into an SWT part. It is well documented and the developers react quickly to bug reports.

References

[1] Florian Angerer. "Configuration-Aware Program Analysis for Maintenance and Evolution in Industrial Software Product Lines." PhD Thesis. Linz, Austria: Johannes Kepler University Linz, 2017.

[2] Florian Angerer. "Variability-Aware Change Impact Analysis of Multi-Language Product Lines." In: *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden, September 15–19, 2014*. Ed. by Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher. ACM, 2014, pp. 903–906. DOI: 10.1145/2642937.2653472.

[3] Florian Angerer et al. "Configuration-Aware Change Impact Analysis." In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9–13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 385–395. DOI: 10.1109/ASE.2015.58.

[4] Florian Angerer et al. "Identifying Inactive Code in Product Lines with Configuration-Aware System Dependence Graphs." In: *18th International Software Product Line Conference, SPLC '14, Florence, Italy, September 15–19, 2014*. Ed. by Stefania Gnesi et al. ACM, 2014, pp. 52–61. DOI: 10.1145/2648511.2648517.

[5] *Architecture-Driven Modernization: Abstract Syntax Tree Metamodel (ASTM)*. Specification OMG ASTM. Version 1.0. Needham, MA, USA: Object Management Group, 2011.

[6] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Publishing Company, 1995. ISBN: 978-0-201-63361-0.

[7] Andreas Grimmer. "A Program Analysis Tool Chain for Programmable Logic Controller Programs." Master Thesis. Linz, Austria: Johannes Kepler University Linz, 2015.

[8] Daniel Hinterreiter. "A Distributed and Version-Controlled Development Approach for the Feature-Oriented Development System FORCE." Master Thesis. Linz, Austria: Johannes Kepler University Linz, 2017.

[9] Susan Horwitz, Thomas W. Reps, and David Binkley. "Interprocedural Slicing Using Dependence Graphs." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.1 (1990), pp. 26–60. DOI: 10.1145/77606.77608.

[10] Patrick Lam et al. "The Soot Framework for Java Program Analysis: A Retrospective." In: *Cetus Users and Compiler Infrastructure Workshop, CETUS 2011, Galveston Island, TX, USA, October 10–14, 2011*. ACM, 2011, pp. 35–43.

[11] Alois Mühleder. "IEC Editor in Eclipse." Bachelor Thesis. Linz, Austria: Johannes Kepler University Linz, 2017.

[12] *Programmable controllers - Part 3: Programming languages*. Standard IEC 61131-3:2003. Version 2. Geneva, Switzerland: International Electrotechnical Commission, 2003.

[13] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing Layered Graphs with Port Constraints." In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. DOI: 10.1016/j.jvlc.2013.11.005.

[14] Mark Weiser. "Program Slicing." In: *IEEE Transactions on Software Engineering* 10.4 (1984), pp. 352–357. DOI: 10.1109/TSE.1984.5010248.

Curriculum Vitae

Personal Information

Name	Peter Feichtinger
Phone	+43 (664) 8703726
Email	shippo@gmx.at
Date of Birth	14. January, 1990
Citizenship	Austria

Education

2016–2017	MSc in Computer Science Johannes Kepler University Linz
2010–2016	BSc in Electronics and Information Technology Johannes Kepler University Linz
2004–2009	Matura College of Industrial Engineering (HTL Vöcklabruck)

Employment History

2014–2017	Johannes Kepler University Linz, CDL MEVSS Software Development in Java
Jul.–Sept. 2015	Keba AG Software Development in C#
Sept. 2013	EW-BAU GmbH Construction
Aug.–Sept. 2012	EW-BAU GmbH Construction
Jul.–Sept. 2010	E. Hawle Armaturenwerke GmbH Manufacturing
2009–2010	Lebenshilfe Oberösterreich Community Service

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, am 17. September 2017

Peter Feichtinger