RWTH Aachen University
Software Engineering Group

Similarity Analysis Framework for Software Product Line Extraction

Master Thesis

presented by

Muctadir, Hossain Muhammad

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Prof. Dr. S. Kowalewski

Advisor: Christoph Schulze

The present work was submitted to the Chair of Software Engineering

Aachen, 25th October 2016

Eidesstattliche Versicherung

Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Abstract

In the context of a regular software vendor, softwares from different projects can consists of similar components. Establishing an Software Product Line (SPL) containing these components and reusing them can reduce development time and increase stability. An SPL can be created from scratch or it can be extracted from existing components. However, establishing and maintaining an SPL can have significant overhead and if not maintained, it can get obsolete quickly. An automated system capable of identifying similar components can help create and update an SPL while providing useful feedback for future projects.

In the context of FEV GmbH, there have been several researches which can find extrinsic, semantic and structural similarity among different software components. The method for calculating extrinsic similarity makes use of the PERSIST guideline, which has been developed by FEV. The results from this analysis shows the potential similar components in different projects and provides a good starting point for further structural and semantical analysis.

Structural similarity analysis method, developed at FEV, matches signals of different interfaces and uses this matching information to find degree of similarity and also a transformation cost. The matching information is also used to extract a generic core which can be extended for future development.

Semantical similarity analysis transforms provided component specific test specifications into Input/Output Extended Finite Automata (I/O-EFA) and tries to find simulation relations among test specifications of different components. This way it tries to find a similarity value for provided test specifications which is also the similarity for software components of corresponding test specifications.

From a higher level, the goal of this thesis is to develop a tool which can import data from different data sources in an enterprise environment, combine previously described methods for finding similarity among different components in a more stable and efficient way. And at the end, generate reports providing necessary information useful for maintaining SPL. In the context of FEV, an extrinsical analysis tool has been developed several months ago but has not been used for quite sometime. This tool has to be updated based on the current project status and run again to generate latest information for current context. After that this tool has to be integrated with already developed structural and semantical analysis tool which should be the initial version of the Similarity Analysis Framework. This framework should be able to generate similarity reports and these reports have to be evaluated to validate that the integration has been done properly. At this stage, any possible enhancement and improvements can be performed to the structural and semantical analysis tools. And finally re-evaluate the reports to verify that nothing has been broken because of the upgrade.

Contents

Chapter 1

Introduction

The development of software products is a resource intensive and time consuming task. Even after the development is finished, it has to be maintained and enhanced to meet the ever increasing demand. Therefore, it is clear that managing a software development life cycle (SDLC) is very complicated as well as expensive. For this reason, the enhancement, optimization and improvement of the SDLC process has been an interesting topic of research for the last couples of years. It is even more interesting for software vendors who develop and maintain multiple software products at the same time in an enterprise environment.

In recent years, the automotive domain has seen a rapid growth in the use of software products. The necessity, complexity and demand of them has been increasing while the expected time of development is getting shorter [Bro06][RSRS15]. Various sophisticated features introduced by the industry has transformed a vehicle into a smart device which puts even more emphasis on software development [PBKS07].

A software product line (SPL) is a set of software systems that share a common set of features satisfying the specific needs of a particular market segment and are developed from a common set of core assets [Nor02]. Software product line engineering (SPLE) focuses on developing software products based on these reusable core components instead of developing them from scratch [KLD02]. It has proven to be the methodology for developing a diversity of software products and software-intensive systems at lower costs, in shorter time, and with higher quality [PBvDL05].

However, establishment and maintenance of SPL is an expensive task in terms of available resources and time, specially in an enterprise environment where the vendors has to work on multiple projects simultaneously and each of them has a deadline. As a result, most software vendors do not establish an SPL or the established ones become outdated very soon due to the lack of necessary maintenance and update. Agile software product line engineering (APLE) approach allows project teams to focus on the actual implementation while at the same time maintain an SPL with minimum effort [RSRS15].

APLE is basically a collaboration between agile software development (ASD) and SPLE. ASD is a set of principles for software development where requirements and solutions evolve through the collaborative effort of self-organizing independent teams [HC01]. The ability to find similar software components in the context of APLE is very crucial. It is not only a very important step for establishing SPL but also helps deciding whether to reuse a component from the product line or create a new one from the scratch.

In [BRR14a] and [BRR$^+$14b], a set of metrics have been proposed to measure software component similarity. It is calculated on three different levels - extrinsic, syntactical and semantical. Two software components are said to be extrinsically similar if they have the same name. This first level of similarity measurement provides a primary overview of the current situation and creates a basis for further similarity analysis. Syntactical similarity is calculated among the interfaces of different software components. This concept is extended by structural similarity metric [KRS$^+$16] which matches parameters from various interfaces and calculates how similar or different they are. Finally, the semantical metric measures the behavioural similarity of the corresponding interfaces.

In the context of FEV GmbH, a prototype has been developed for calculating extrinsic similarity based on the principle explained in [BRR$^+$14b]. This Matlab based implementation reads software component information from different projects and generates a report which lists the extrinsically similar components from different project. A python based prototype calculates structural similarity based on the interface definitions which follows the methodology explained in [KRS$^+$16]. It reads the corresponding component's interface definitions in the context of FEV GmbH and analyses the structural similarity based on their attributes. The test case based semantical similarity calculation procedure is described in [RRS$^+$16]. The java based prototype translates the test cases in to Input/Output Extended Finite Automata (I/O EFA) and measures the simulation relation among them. Finally, it generates statements showing the percentages of semantical similarity for corresponding software components.

These similarity metrics and the results from the above mentioned prototypes are very useful input to the APLE process. But, the results are generated separately and correlation among them is not clearly visible. Moreover, each of the prototypes are developed with very different technologies which adds more complexity to the whole process. A framework, which is capable of combining the results from different similarity calculation prototypes and establishing a correlation among them, can increase the usefulness of this similarity information significantly. Developing such a similarity analysis framework is the main goal of this thesis.

In the context of FEV GmbH, the framework is capable of accessing data from various sources and generate similarity information based on them using the above mentioned prototypes. The similarity results are used to generate HTML based reports which includes tables and graphs to present corresponding information in a viewer friendly format. The framework not only provides an unified and convenient interface to communicate with external prototypes but also exposes various functionalities to flexibly extend the framework.

The following chapters of this thesis report will explain the framework in more details. Chapter 2 describes various theoretical concepts which will be used extensively throughout the whole report. Chapter 3 specifies the requirements which the similarity analysis framework must implement. Chapter 4 explains different algorithms used during the implementation and illustrates the whole framework from a theoretical point of view. The implementation details of the framework is laid down in Chapter 5. It also explains certain related details of previously mentioned prototypes and improvements made to them during the development of the similarity analysis framework. The quality of the code and the similarity results are evaluated in chapter 7. Finally, chapter 8 gives a short overview of the whole thesis and concludes the report.

Chapter 2

Foundation and Literature Review

2.1 AUTOSAR

AUTOSAR is an open and standardized automotive software architecture which was jointly developed by automobile manufactures, suppliers and tool developers [BHDG10]. It stands for (AUTomotiveOpen System ARchitecture). AUTOSAR was developed to become a common platform upon which future vehicle applications can be developed. It separates application software from associated hardware resulting in lower cost and high re-usability. It aims to create a foundation for collaboration on basic functions, while leaving room for competitive development of innovative new functions.

AUTOSAR divides the ECU (Engine Control Unit) software into three layers - Basic Software (BSW), Runtime Environment (RTE) and Application Layer (APSW). Various predefined modules are defined in the Basic Software layer which provides services like bus communications, memory management, IO access, system and diagnostic services. The operating system itself is part of this layer. The RTE separates BSW from APSW and provides an abstraction layer between them. The APSW contains different software components each implementing specific functionality of the ECU. RTE executes the software components from the APSW and provides data exchange service between these components and the BSW [www16].

2.2 PERSIST

PERSIST is an extension to the AUTOSAR standard which was developed with agile methodologies, long-term architecture development and efficient automated development support in mind. Although, AUTOSAR standardized the generic automotive control functions, certain aspects has not been standardized or finalized. For example, interfaces and functions for powertrain are incomplete. To address this issue and for establishing an open standard, PERSIST proposes methodologies for powertrain software development which are also compatible with AUTOSAR. PERSIST uses the component based software engineering design pattern (CBSE), where the structure of functional components corresponds to the physical system layout [RPS14].

According to PERSIST, component is a logical unit which can contain several functions. They are, in general, referred to as units, which have an interface consisting of several signals. Signals can be of different kind (IN, OUT, CAL, FIX, MP) and based on their kind they are capable to send or receive data. IN-signals support writing data, while OUT-signals can read data. CAL-signals are specialized input signals, which are usually used to set certain parameters during the testing phase.

2.3 Software product line

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [Nor02]. The concept of product line is not new in manufacturing. Companies like Ford, Boeing, Dell has been using this approach for long time [Nei09]. But, using this concept for software development is relatively new and has been proven to be very effective [FFL09]. Software product line brings the concept of strategic and planned reuse which in the past was mostly opportunistic reuse. Large companies like Hewlett-Packard, Nokia, Motorola has found that using the SPL approach has remarkable improvements in productivity and time to market [Nor02].

Software product line engineering is a highly iterative process. At the highest level there are three essential activities which blend technology and business practices [Nor02]. Figure 2.1 illustrates these essential activities and they are - Core asset development, Product development and Management.

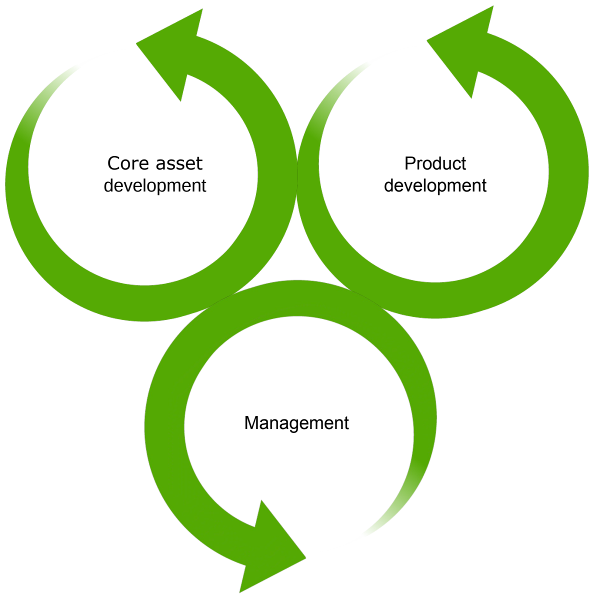


Figure 2.1: Essential software product line activities [Nor02]

Core assets are the basic building blocks of an SPL which includes architecture, reusable software components, domain models, test plans, test cases etc. These days assets are rarely developed from scratch rather some of the assets already exists and the asset base grow out of them. Each of these assets are products on its own. New products are created by taking applicable components from the common asset base and tailoring and/or adding new components according to the product requirement [McG04]. Product specific non shared features added during product development are not developed using core assets. However, creating products can generate feedbacks which might affect in core asset development, product plan even product line scope. Management at the technical level and organization level should have strong commitment to make the SPL approach a success. Technical management supervises the core asset and product development by ensuring necessary coordination among the processes and collecting necessary data to track progress. Organizational management supports the SPL by ensuring proper organizational structure and providing appropriate resources. Moreover, organizational management determines funding model and provides necessary funds [Nor02].

2.4 Agile software product line engineering (APLE)

Agile software development (ASD) describes a set of principles for software development under which requirements and solutions evolve through the collaborative effort of self-organizing cross-functional teams [HC01]. Adaptive planning, evolutionary development, early delivery, continuous improvement etc are some of the key features of ASD.

Software product line engineering (SPLE) focuses on establishing a reusable platform for a specific domain which allows to derive several customized products in an efficient manner [RSRS15]. It divides the development into two phases: domain engineering (DE) and application engineering (AE). During the DE phase a software platform in developed based on the domain specific analysis. Establishing and maintaining a common reference architecture is also done at this phase. At the AE phase specific products are developed by reusing the domain artifacts based on the platform specifications.

In the automotive domain, the demand and complexity of different software systems are growing along with the demanded quality standards. Meanwhile, expected time for development is getting shorter. Frequently changing and highly unpredictable demands have made it very difficult to make a long term plan while maintaining the reusability of software components to a certain degree. PERSIST partly solves these problems with its agile methodologies by being flexible to requirement changes and reducing the duration of development cycle. Among all these challenges, establishing and maintaining an SPL has become harder than ever. APLE is a collaboration between ASD and SPLE which allows the project teams to focus on the implementation of some specific product while maintaining the SPL with minimum effort [RSRS15]. SPLE and ASD both approaches are flexible to requirement changes and tries to reduce the time of product development. But ASD suggests short planning phase while SPLE requires intensive requirement analysis which results into slower release cycle. APLE tries to reduce the time intensive DE phase.

Figure 2.2 shows an APLE method proposed at [RSRS15]. In the figure, colored and white activities represents activities performed during DE and AE phases respectively. The proposed method is based on component-based AE-first approach where a component is the main item of an architecture and the specification of this component is specified during AE phase. Therefore, the first step of the proposed method is defining specifications for the new component. In the next step, the newly specified component is compared with the reference architecture to estimate the position of the component. If it can not be mapped, a new position in the project software architecture needs to be assigned. This step should be avoided if possible as it makes the architecture complex and difficult to maintain. Therefore, this decision is re-evaluated in another step by the DE process to make sure a

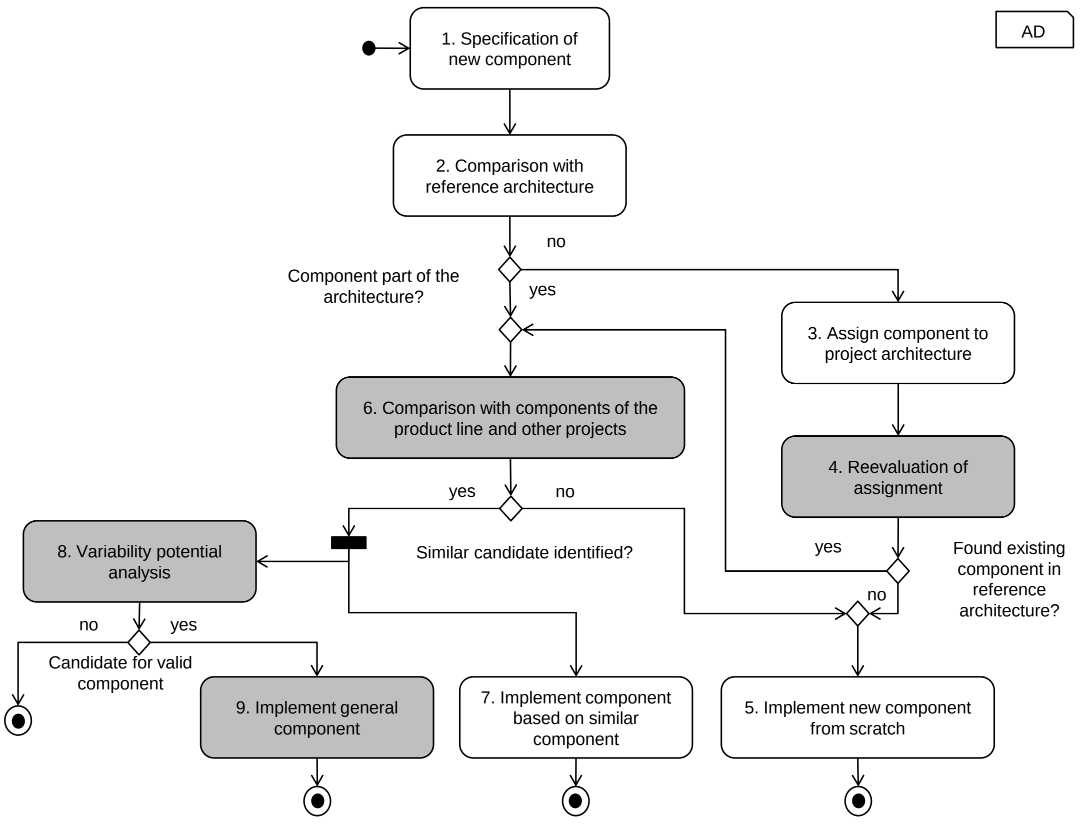

AD
1. Specification of new component
2. Comparison with reference architecture
no
Component part of the architecture?
yes
3. Assign component to project architecture
6. Comparison with components of the product line and other projects
4. Reevaluation of assignment
yes
no
Similar candidate identified?
yes
Found existing component in reference architecture?
no
8. Variability potential analysis
no
yes
Candidate for valid component
9. Implement general component
7. Implement component based on similar component
5. Implement new component from scratch

Figure 2.2: Agile software product line engineering focused on application domain. (colored = DE, white = AE) [RSRS15]

new component is really necessary. If no matching is found after this re-evaluation step, the component needs to be implemented from scratch. Although, no advantage could be gained from the product line, the effort to reach this decision is also negligible.

On the other hand, if there is an existing matching component, an extrinsic equality has been established. In the next step, the proposed component is compared with the product line and other projects. This comparison is done on structural and semantical level by the DE process. Test cases has to be available for a semantic comparison. Semantical similarity can not be evaluated, if no test cases are provided. In case of 100% semantical similarity, the whole component can be reused without any development effort. If the level of similarity is less than 100%, it can be used as a base for the new component. This speeds up the development process during AE. At the same time, possibility to extract a general item can be identified and if there is a possibility, it is extracted for future reuse. If conditions are satisfied, this general component can be used instead of developing a new component. This reduces the development time even more.

Developing a reusable generic component takes time and resources. Therefore, appropriate analysis should be done to make sure that there are potential demand for corresponding component. The more the component is reused, the more benefit can be gained from the spent effort.

2.5 Extrinsic similarity

The success of a SPLE largely depends on identifying similar software components existing in a set of software components. It is also an important step while establishing a software product line from already existing components. [BRR+14b] presents a formal definition for identifying extrinsic similarity among software components. For $n \geq 2$ software products p_i is a software product where $1 \leq i \leq n$, which can be decomposed into a set C_{p_i} of $m \geq 1$ atomic pieces. Each atomic piece c_j where $1 \leq j \leq m$, is called a software component. Two software component c_i and c_k are called extrinsically equal nonetheless to which product it belongs to iff they have the same name. This similarity information is not only useful for identifying similar interfaces from different projects but also provides a starting point for further structural and semantical similarity calculation which are discussed in following sections.

In the context of FEV GmbH, a prototype based on the described principle was developed. It has been successfully used to generate a list of components including the frequency in which they are being reused. It helps the SPLE process to identify the most frequently used components and emphasize on making them more stable and generic for possible future use.

2.6 Interface-based structural similarity

Two software components are structurally similar when their input and output parameters are similar. Base on the number of similar parameters a percentage can be derived which represents overall structural similarity between these components. As described in [RSRS15] and also discussed in section 2.4, identifying similar software components is an important aspect for successfully utilizing an SPL because this is the step upon which the decision of creating new component depends. An algorithm for calculating structural similarity is presented in [KRS+16]. Based on the proposed algorithm a prototype has been developed in the context of FEV GmbH. The algorithm focuses on finding structural similarity among software components in the context of automotive software engineering. It uses graph matching technique for similarity analysis. The algorithm is also capable of extracting generic core from a set of similar components using graph based arborescence. Graph matching and arborescence are later discussed in more details.

For a graph $G = (V, E)$, where V is the set of vertices and $E \subseteq V \times V$ is the set of edges. A graph matching is the set of edges where no two edge share a common vertex. In other words, a set of independent edges of G [Bol04]. Given two edges $(a, b), (c, d) \in E$ are independent if $(a, c), (a, d), (b, c), (b, d) \notin E$. In a maximum weighted graph matching the weight of the edges are taken into account and only those are chosen for which the overall weight is maximum. Structural similarity algorithm [KRS+16] matches different attributes of all signals from one interface to signals of the other interface which results into a weighted graph where each edge represents similarity between two signals from different interfaces and the edges' weight represents similarity percentage. At this point a maximum weighted graph matching is performed which results in multiple independent subgraphs where in each of them one signal is mapped to exactly one signal of the other interface given that there are one or more matching signals in the original graph.

An arborescence of a directed graph G is a directed spanning tree of G. A subgraph $T = (V, E)$ of G is an arborescence with respect to root r if and only if T has no cycles and for each $v \in V$ and $v \neq r$ there is exactly one edge in F that enters v. After computing the maximum weighted graph matching, extracting arborescence for each of the subgraphs provides generic core signal, if any, for each group of similar signals.

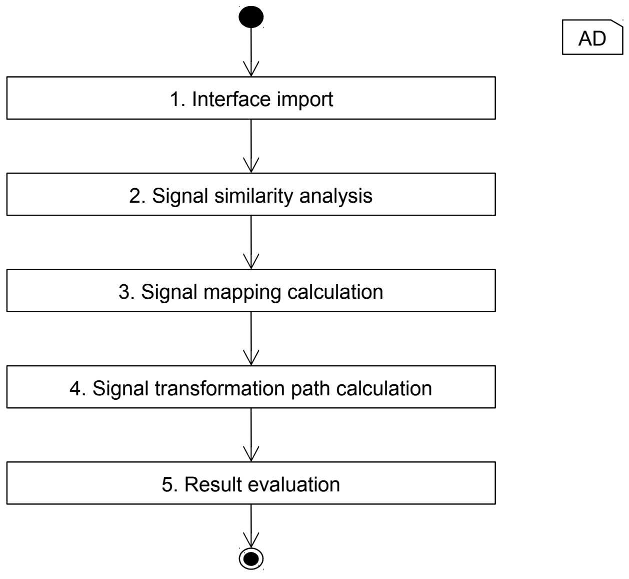

AD
1. Interface import
2. Signal similarity analysis
3. Signal mapping calculation
4. Signal transformation path calculation
5. Result evaluation

Figure 2.3: Structural similarity algorithm overview [Keh15]

The algorithm has five consecutive steps as shown in figure 2.3. First step is to import the interface definitions and store them. Data models used to store the interface information is independent of the data source making the algorithm flexible. In the next step, interfaces are compared with each other by comparing their signals. This comparison is done in pairs of interfaces. Signals from one interface is compared to the signals of another interface. Therefore, after the comparison there exists a similarity value for each pair of signals from different interfaces. The comparison of the signals is done based on their properties. The comparison results into a numeric value which represents the effort to transform one signal to another. This transformation cost is directed. Meaning, the cost for transforming signal s_1 to s_2 can be different from s_2 to s_1. Next step is generate signal mapping which results into a mapping where one signal from on interface is mapped to exactly one signal from the other interface. This mapping is done in a way so that the overall transformation cost is minimized. At this point, similarity value for two interfaces is available. However, similarity value is not sufficient when extracting a generic core. The forth step calculates a transformation path for a group of similar signals, which means it calculates a signal which all other signals can be transformed to. Finally, the results from previous steps are gathered and presented in different ways.

2.7 Test-driven semantical similarity

As described in [MTK94], semantically similar components refers to functionally similar components. In other words, semantically similar software components are components that exhibits similar behavior. In section 2.6 structural similarity has been discussed. According to [RSRS15], taking advantage from existing SPL largely depends on finding similar components and for that, semantical similarity should also be evaluated along with structural similarity. [RRS$^+$16] describes an algorithm to identify semantical similarity between software components based on test cases. These test cases are transformed into Input/Output extended finite automata (I/O EFA) and simulation relations (discussed later) are compared for automatons from different component.

According to [RSVW$^+$15] and [RRS$^+$16], an I/O EFA is a finite state machine enhanced with input and output behavior which is defined as a tuple, $A = (S, s_0, D, d_0, U, Y, E)$, where

- S is the set of states with initial state $s_0 \in S$.
- D is the set of internal variables and d_0 is the initial value.
- U and Y are respectively the set of input and output variables.
- E is the set of transitions. For $o_e, t_e \in S$ a transition from o_e to t_e is defined as $o_e \xrightarrow[y=h_e(d,u);d=f_e(d,u)]{[g_e(d,u)]} t_e$ where guard condition $g_e \subseteq D \times U$, output function $h_e : D \times U \to Y$ and data update function $f_e : D \times U \to D$. All depending on current variable $d \in D$ and input value $u \in U$.

An Input/Output Transition System (I/O TS) is an I/O EFA without internal variables and its corresponding data update function [ZK12].

According to [RSVW$^+$15], two I/O-TSs A and B are in simulation preorder means for the same input: (1) the set of all sequences of B's transition executions is a superset of all sequences of A's transition executions, and (2) both having an identical output. Formally, for I/O-TSs A and B having states S_A, S_B and transitions E_A, E_B is a binary relation $R \subseteq S_A \times S_B$, where if $(a, b) \in R$ and for input $u \in U$ the transition $a \xrightarrow{[g_a(u)]} a' \in E_A$ is enabled then states a and b produces equivalent output and there exists an u-enabled transition $b \xrightarrow{g_b(u)} b' \in E_B$ such that $(a', b') \in R$.

Test driven development (TDD) has become one of the most popular software development paradigm [MW03]. This method not only substantially reduces the amount of software bugs but also makes the test cases available at very early stage of development. In a TDD environment, components can be compared even before they are fully developed using the method from [RRS$^+$16].

Figure 2.4 illustrates the steps necessary to semantically compare test cases from two different components and derive a similarity statement. At first, structural similarity has to be ensured between comparing components and appropriate port matching information has to be available for components interfaces. In the next step, the test specifications are converted into I/O-EFAs. Each output of the test specification is transformed a separate automaton. The input and output variable of the automaton is derived from the components interface. Test specifications do not specify internal variable and therefore, the automaton has no internal variable. This makes it possible to avoid the state space explosion problem [RRS$^+$16]. The test cases usually contains several steps and for each step a transition in the automaton is created. The number of states also depends on the number of test steps. For each new transition created from a test step a new target state is created. Each transition's target state is the source state of the next transition. These automata are then transformed to I/O TSs. This is necessary to avoid the state space explosion during comparison of two automata. In the next step, the simulation algorithm is executed both ways to find similarity between both automata. The counterexample guided

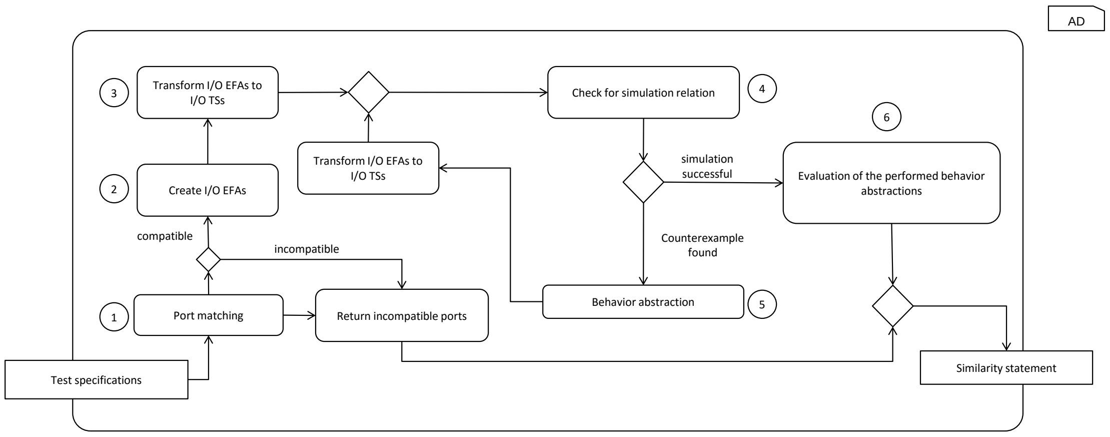

AD
Transform I/O EFAs to I/O TSs
Check for simulation relation
Transform I/O EFAs to I/O TSs
simulation successful
Evaluation of the performed behavior abstractions
Create I/O EFAs
compatible
incompatible
Counterexample found
Port matching
Return incompatible ports
Behavior abstraction
Test specifications
Similarity statement
1
2
3
4
5
6

Figure 2.4: Semantical similarity algorithm overview [RRS$^+$16]

abstraction similarity (CEGAS) metric is used to analyze the similarity of an automaton B to an automaton A by creating simulation relation between their initial state. Behaviors that can not be simulated are abstracted by provided counterexample. Step 3 to 5 (Fig. 2.4) is repeated until no further counterexample can be extracted. Finally, a similarity statement is returned based on the amount of abstractions performed.

2.8 Design patterns

In the context of software development, design patterns refers to a set of methodologies for solving commonly encountered software design problems that can be applied in different contexts. Design patterns are abstract, implementation independent and each pattern provides solution for a particular scenario. Appropriate use of design patterns results into a code that is easily testable, flexible to changes and readable [PULPT02]. This section explains some of the design patterns which has been used during this thesis. The discussions are based on object oriented software development paradigm.

2.8.1 Inversion of control and dependency injection

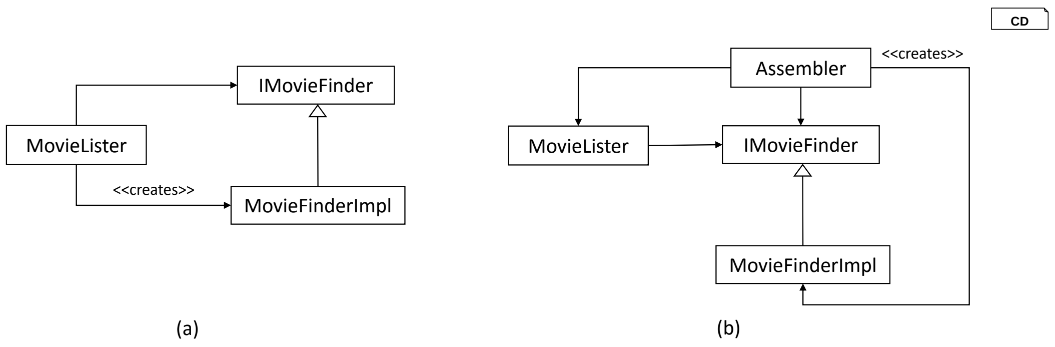


Figure 2.5: Inversion of control and dependency injection [Fow04]

In a typical software development setting, client codes calls the framework codes to take some service from them. Inversion of control (IoC) suggests to revert this conventional flow of control where the framework calls the client code [Fow04].

Looking at the figure 2.5(a) [Fow04], *MovieLister* shows a list of movies and uses the service provided by *MovieFinderImpl* to get the list. To do that, *MovieLister* creates and calls *MovieFinderImpl* directly. At this point, if there is a necessity to create another *MovieFinderImpl*2 which fetches data form a different source, the *MovieLister* has to be changed to use it which essentially makes *MovieLister* dependent on one of the finders and their interface *IMovieFinder*. This situation would not have appeared if the *MovieLister* was only dependent on the interface like it is shown in figure 2.5(b). The *Assembler* takes care of the dependency by injecting *MovieFinderImpl* while creating *MovieLister*. Now, *MovieLister* is only dependent on the interface *IMovieFinder* and *Assembler* can easily change the actual implementation.

If *IMovieFinder* was part of a framework, then the actual implementation can be provided by the client. In this scenario, the client code is being called by the framework which inverts the control from the client to the framework. This particular kind of IoC is called dependency injection (DI) [Fow04].

2.8.2 Abstract factory

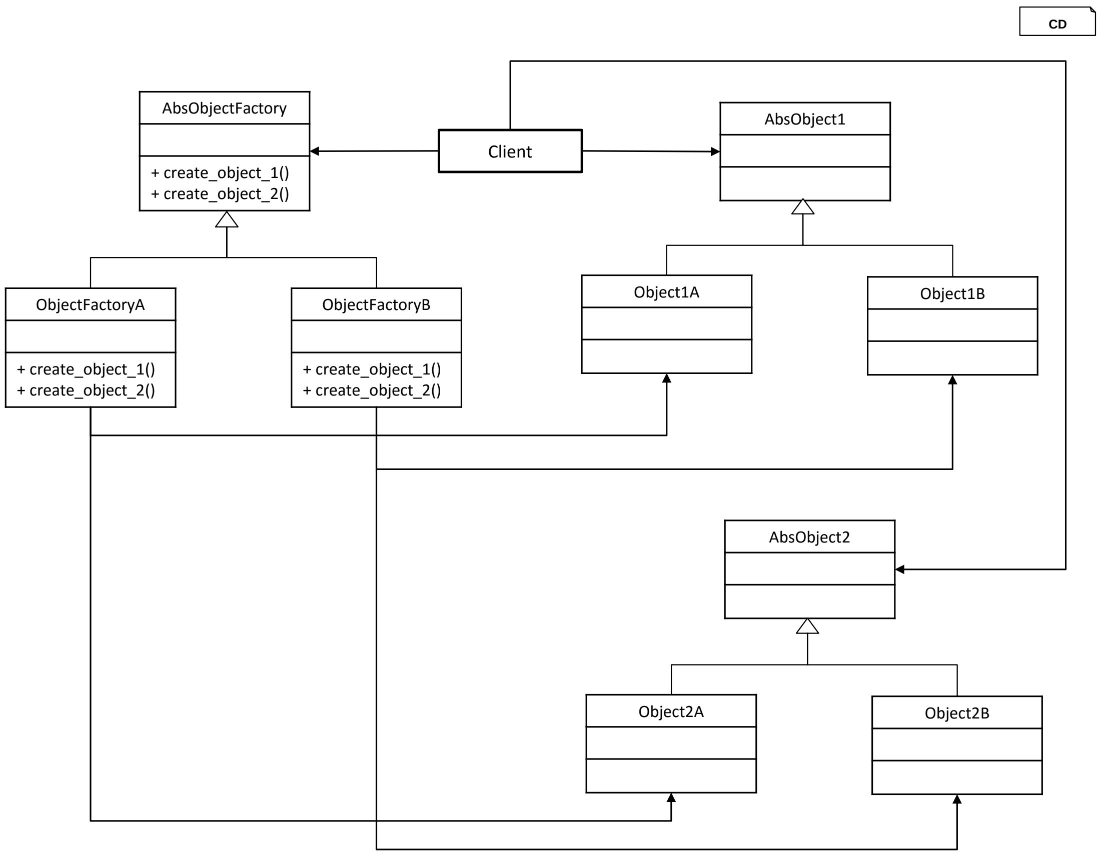


Figure 2.6: Abstract factory (inspired by [Gam95])

As explained in section 2.8.1, creating an object directly from within another object couples the objects together and makes is harder to replace or update dependencies. Abstract factory provides a layer of abstraction for creating a set of complex objects. This abstraction layer is responsible for resolving dependencies and providing object creation service to the entire framework.

The pattern works as the example shown in figure 2.6 [Gam95]. *AbsObjectFactory* object provides the abstraction for creating two different objects *AbsObject*1 and *AbsObject*2. The actual implementation to be created is decided by *ObjectFactoryA* and *ObjectFactoryB* at runtime. Therefore, the client is not aware of the actual implementation details of the objects it receives from the factories which makes it very easy to update the dependencies. In case of new dependencies, creating another factory implementation for them is enough which requires absolutely no change to the existing code.

2.8.3 Template method

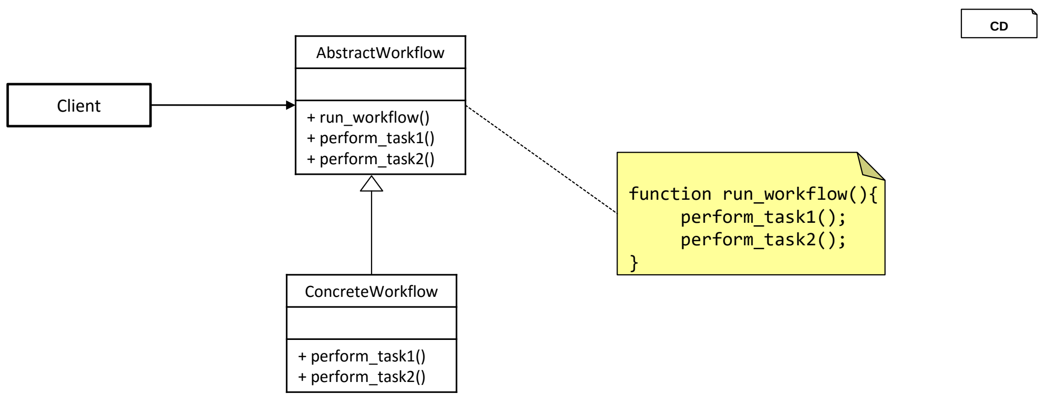


Figure 2.7: Template method (inspired by [Gam95])

A template method specifies the workflow for an algorithm without specifying the actual implementation [Gam95]. In the example shown in figure 2.7 [Gam95], a typical structure of a template method is presented. *AbstractWorkflow* defines the method *run_workflow* and this method specifies the steps necessary to perform the particular workflow. Individual steps are abstracted by *AbstractWorkflow* and implemented by *ConcreteWorkflow*. This provides the opportunity to specify different implementation for one specific workflow without thinking about the workflow itself.

2.8.4 Facade

The facade pattern hides the complexities of other interfaces and provides an unified interface to interact with them [Gam95]. A facade is basically an object which acts as a gateway and delegates method calls to multiple other related objects. In figure 2.8 [Gam95], *AbsFacade* provides an abstract interface for accessing multiple objects. *ConcreteFacade* specifies implementation for the interfaces by calling several other objects. The objects providing the actual service and complexities related to them are hidden from the client.

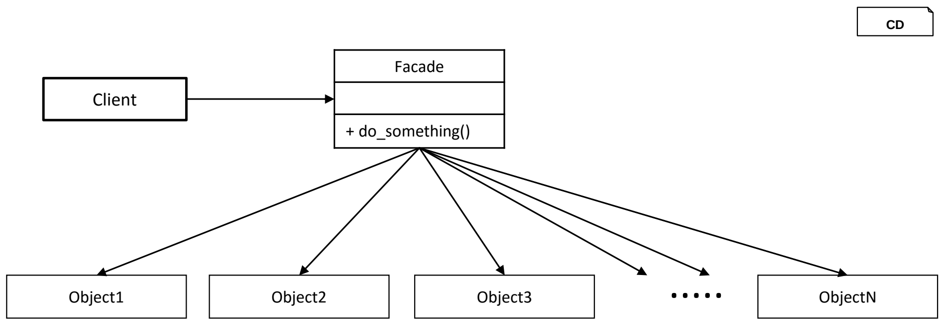


Figure 2.8: Facade design pattern (inspired by [Gam95])

2.8.5 Strategy

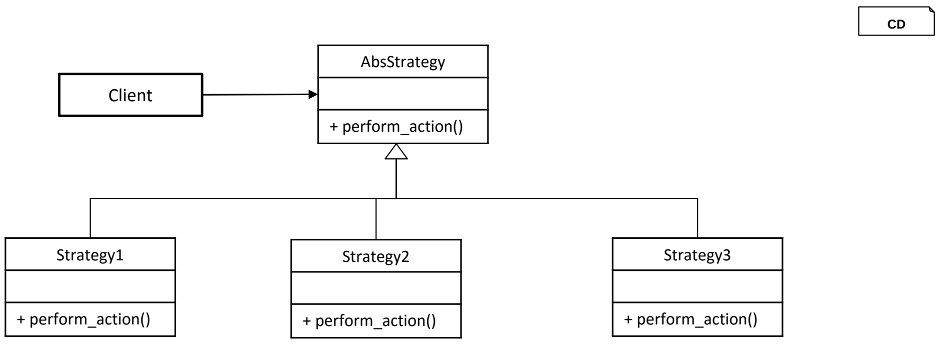


Figure 2.9: Strategy design pattern (inspired by [Gam95])

The strategy pattern provides a common interface for a family of similar algorithms [Gam95]. Each implementation can specify their own implementation of the algorithm and they are interchangeable. Figure 2.9 [Gam95] shows an example class diagram for strategy pattern. *AbsStrategy* defines the abstract method $perform\_action$ and the three different child objects can provide their own concrete implementation for this method. The *client* only refers to the abstract object and has no knowledge of the implementation which makes the implementation decoupled from its use.

Chapter 3

Similarity Analysis Framework (SimA)

In the context of a software manufacturer, establishing and maintaining an SPL (section 2.3) can improve quality of the product while significantly increasing productivity. Furthermore, the SPLE, as discussed in section 2.4, is a time and resource extensive process. The APLE (section 2.4) tries to improve the process by integrating agile methodologies to SPLE. One of the time intensive and key step of APLE is to decide whether to develop a new component from scratch or reuse and/or improve existing component from product line. This chapter proposes a similarity analysis framework for calculating similarity information among different software components which can significantly accelerate above mentioned APLE step while making it more accurate.

3.1 Framework introduction

In the previous chapter (Chapter 2), extrinsic, structural and semantical similarities has been discussed. These measurements provide similarity information among different software components on different level. Extrinsic measurements identifies similar software components from a high level signalling potential similarities on the lower level. Structural similarity evaluates similarities based on input and output parameters providing lower level similarity information. Structurally similar components are evaluated for semantical similarity by taking available test cases into account. These measurements alone can provide useful similarity data but are not enough for establishing or maintaining an SPL. For this purpose a similarity measurement is required which combines information from all three levels.

SimA is a Similarity Analysis Framework which can evaluate similarity on extrinsic, structural as well as semantical level and finally generates reports based on found similarity information. Later sections in this chapter discusses about SimA in more details and specifies different requirements which the implemented framework must satisfy.

3.2 Framework requirements

A software framework is a platform that provides generic functionalities to solve certain domain specific problem which can be specialized or overridden to a certain extend to change the default behaviour [MdL01]. Therefore, a software framework has to be static in terms of the kind of problems it solves, at the same time, flexible enough to incorporate different methods to solve these problems. Although, finding similarity among software components and generating relevant reports are said to be the fundamental functionality of SimA, the framework should implement following requirements.

Smilarity analysis - SimA should be able to perform similarity analysis on extrinsic, structural and semantical level. The framework should provide interfaces for each kind of similarity analysis. Each of these interfaces must be independent of other similarity analysis interfaces. This means that the interfaces should be flexible enough to be used as standalone as well as in conjunction to generate combined similarity statement. The framework should also be flexible enough to incorporate any other kind of similarity measurement that might emerge in future.

Report generation - SimA should provide interfaces to generate reports based on found similarity. These interfaces should only perform operations which are strictly related to the report generation. Therefore, they should be independent of any calculation/activity which might affect other part of the framework (ex. similarity calculation) in any way.

Global component list generation - A global component list lists all the components in an SPL including their identifying properties and usage information. SimA should provide an interface and possibly a default implementation for generating a global component list.

Abstract workflow definition - SimA should define the similarity calculation and report generation workflow from an abstract level. Although, it is an essential requirement for different part of the framework to be independent, defining a way for these parts to work together is also important. For example, defining in which sequence different similarities should be evaluated is critical for generating a meaningful similarity statement. This requirement is one of the defining characteristic of SimA.

Source independent I/O - In SimA reading and writing of data should be source independent. This means, the framework should always perform I/O operations from an abstract level. This abstract layer should be able to convert data from and to frameworks internal data structure when necessary. Preserving external format of the data can be of importance in certain cases and the mentioned abstract layer should be able to take care of such cases. It enables the framework to perform various calculations on this data while preserving the capability to seamlessly communicate with any external framework.

Swapable components (loose coupling) - Most of the component of SimA should be swappable by a similar component. This means, part of the framework's behaviour can be altered without affecting other parts of the framework. When two distinct software components exchange messages, they are said to be coupled [LJK+01]. A high degree of coupling is the result of a higher number of messages exchanged among components making them interdependent. This dependency makes software architecture complex and difficult to maintain. Therefore, SimA should be designed in a way so that the coupling factor remains to a minimum.

Task specific component - Each component of SimA should have one and only one specific task. In software engineering terminology, this characteristic is referred to as single responsibility principle [Mar03]. It ensures highly modular architecture making future enhancement, extension and bug fixing easier.

3.3 Platform requirements

The framework is expected to run on Windows systems. It should be highly portable so that any machine running an updated windows system can execute it with minimum or no configuration. It should be efficient enough to run on systems with limited resources. However, the reports generated by the framework has to be absolutely platform independent, portable and should be viewable without any configuration whatsoever.

Chapter 4

Concept

This chapter presents the theoretical aspects which has been used during the development of the SimA framework and the algorithms behind it. The purpose of SimA is to extract similar software components from a group of software products and present the similarity results in a report which provides useful information for software product line extraction and maintenance.

4.1 Generic similarity calculation algorithm

SimA performs similarity analysis sequentially on extrinsic, structural and semantic level. The steps necessary for similarity calculation are,

r_1) read software component definitions from all projects.

r_2) read component's interface definitions.

r_3) read component's test cases.

s_1) evaluate extrinsic similarity.

s_2) evaluate structural similarity.

s_3) calculate semantical similarity.

The sequence in which these steps has to be executed is not fixed. The framework provides a flexible interface for defining the sequence and it has to follow a certain rule. To explain the sequence rule, the notation $\prec$ is defined. For two arbitrary steps x and y, the notation $x \prec y$ means that x has to be executed before y. Let S be the set of steps necessary for similarity calculation, the steps are executed in a sequence that satisfies the condition,

$$r_n \prec r_{n+1} \wedge s_n \prec s_{n+1} \wedge r_n \prec s_n \tag{4.1}$$

where $n \in \{1, 2, 3\}$ and $r_n, s_n \in S$. For example, according to the condition, the execution sequence $(r_1, s_1, r_2, s_2, r_3, s_3)$ is valid but $(r_2, s_2, r_1, s_1, r_3, s_3)$ is not.

Evaluation of extrinsic, structural and semantical similarity are evaluated based on software component definitions, software interface definitions and test cases respectively. Moreover, extrinsic, structural and semantical similarity has to be evaluated in this respective order. The condition 4.1 ensures a orderly execution and availability of necessary

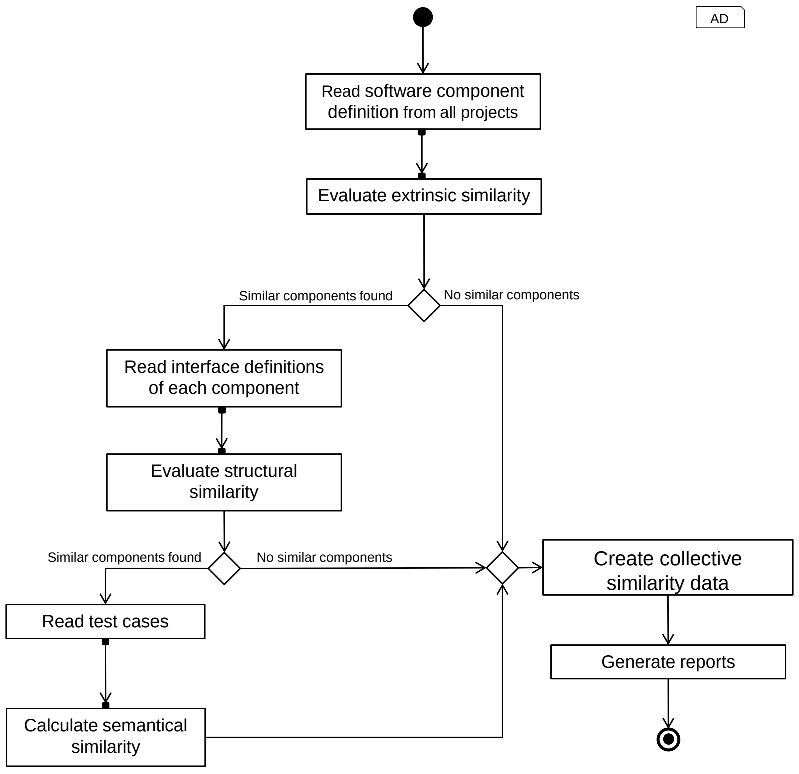

AD
Read software component definition from all projects
Evaluate extrinsic similarity
Similar components found
No similar components
Read interface definitions of each component
Evaluate structural similarity
Similar components found
No similar components
Read test cases
Calculate semantical similarity
Create collective similarity data
Generate reports

Figure 4.1: SimA lazy loading workflow

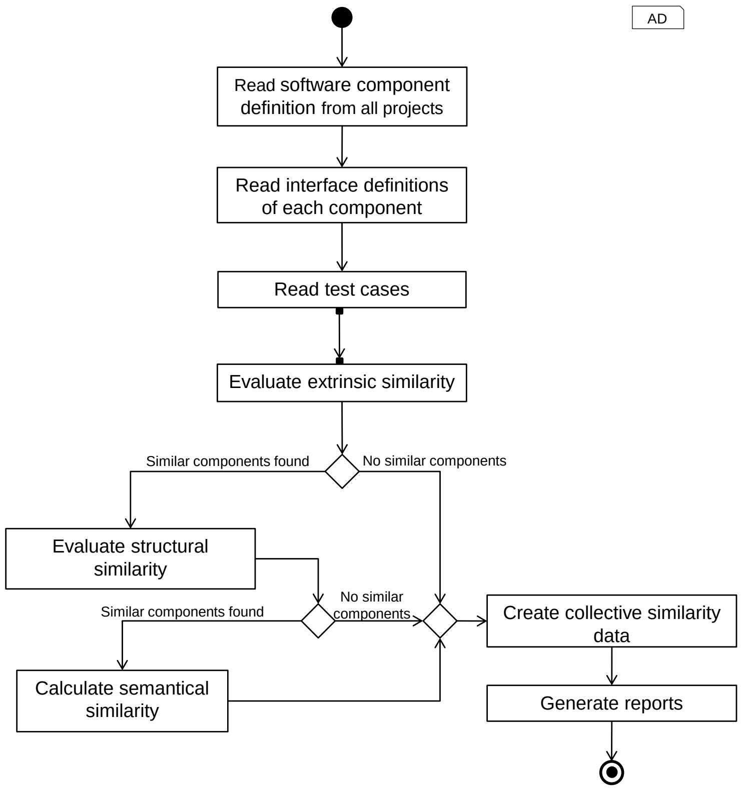

AD
Read software component definition from all projects
Read interface definitions of each component
Read test cases
Evaluate extrinsic similarity
Similar components found
No similar components
Evaluate structural similarity
No similar components
Similar components found
Calculate semantical similarity
Create collective similarity data
Generate reports

Figure 4.2: SimA pre-loading workflow

data at each similarity evaluation stage. Steps necessary for similarity calculation are later explained in greater details.

Two example workflows are shown in figure 4.1 and figure 4.2. Figure 4.1 demonstrates a lazy loading workflow. Data necessary for each type of similarity analysis is read only when necessary which reduces number of disc read operations to a minimum. At the beginning, all available software component definitions are read from all projects. These definitions are used to evaluate extrinsic similarity. The extrinsic similarity divides all software components from different projects into several smaller groups where each component belongs to exactly one group and each group contains one or more components which are extrinsically similar. Interface definitions are imported for all software components which has two or more extrinsic matches. Structural similarity is evaluated based on these interface definitions. For each group generated during the extrinsic similarity step, structural similarity identifies a port mapping between each possible pair of components of the particular group which identifies similar ports between each pair. The similarity information found at this stage and available test cases are used to evaluate semantical similarity. The semantical similarity identifies behavioural similarity between two output ports from different components. At this point, a data model holding all the calculated similarity information is generated. Finally, reports are generated based on the collective similarity data. Furthermore, at any similarity calculation step if no similarity is found, a collective similarity is created with similarities found so far and directly go to report generation.

Current implementation of SimA follows the workflow shown in figure 4.2. It preloads all necessary data before calculating any kind of similarity. Certain reports generated by the current implementation shows data availability for different projects. Therefore, it is necessary read all available information.

Furthermore, it should also be noted that SimA's flexible workflow structure and the sequence rule specified by the condition 4.1 generates identical results for similarity calculation only. Any other additional operation (ex. report generation) may need specific workflow based on the requirements.

4.1.1 Read software component definition

First step of SimA algorithm is to read component definitions from different software projects. The component definitions are read along with all other project specific information. Data models are defined to store project and their associated software component information. Component definitions are read into a list and stored as an attribute of the project data model. These models that hold read information are used for further computation. This not only makes the reading of component definitions customizable without changing any other part of the algorithm, but also makes the data model reusable. According to the requirements (Section 3.2), the read operation is source independent. Hence, projects can have heterogeneous data source or location.

4.1.2 Evaluate extrinsic similarity

Summarizing from the definition presented in section 2.5, if a software product can be decomposed into two or more atomic pieces which can function independently, then each of those atomic pieces are called components of the particular software product. Software components from different projects are matched to each other to find extrinsically similar components. As discussed in section 2.5, two software components are called extrinsically similar, regardless of which projects they belong to, if they both has the same name. This principle is used for calculating extrinsic similarity which is explained in algorithm 1.

Algorithm 1 Extrinsic similarity calculation algorithm

```
 1: $s \leftarrow \emptyset$                      ▷ set of sets to store similar components
 2: $f \leftarrow \emptyset$                      ▷ set of components for which similarity already found
 3: for $p_i \in P$ where $P$ = all software products and $i \in \{1, 2, 3.....n\}$ do
 4:     for $c_i \in C_i$ where $C_i$ = components in $p_i$ and $c_i \notin f$ do
 5:         $s_i \leftarrow \{c_i\}$
 6:         $f \leftarrow f \cup c_i$
 7:         for $p_j \in P$ where $j \in \{1, 2, 3.....n\}$ and $j \geq i$ do
 8:             for $c_j \in C_j$ where $C_j$ = components in $p_j$ do
 9:                 if $name(c_i) = name(c_j)$ then
10:                     $s_i \leftarrow s_i \cup c_j$
11:                     $f \leftarrow f \cup c_j$
12:         $s \leftarrow s \cup \{s_i\}$
13: return $s$
```

The goal of the algorithm is to divide components from all the projects into several groups where each group contains components which are extrinsically similar and one component belongs to only one group. To do that the algorithm defines a list s to hold the groups. Another list f is defined which maintains a list of components which has already been compared. Initially, s and f are both empty. At this point, the algorithm goes through each component c_i of project p_i and tries to find out a component c_j belonging to another project p_j where c_i and c_j has the same name. A list s_i, holding components having same name as c_i, is created and finally s_i is added to s. This is done for all components of all projects. At each iteration c_i is added to f and whenever a match is found c_j is added to f. Thus, f holds all the components which has been compared and possibly already belongs to a group. Therefore, the component c_i is considered for comparison only if it does not belong to f which significantly reduces the number of iteration as the algorithm proceeds. Once the algorithm has finished iterating all projects, s holds extrinsically similar components in separate groups.

4.1.3 Read interface definitions

Interface definitions of different software components are read and stored as an attribute of the corresponding component. Data model, specified by the implementation, is used to hold the interface definitions which is used for further computation. In accordance to the requirement described at section 3.2, the reading of interfaces is independent of the type or location of the source. Hence, any other part of the SimA algorithm is disassociated with this particular step.

4.1.4 Evaluate structural similarity

Interface of a software component consists of a set of signals. According to [Keh15], a signal $s_i \in S$, where S is the set of signals of interface c_i, is defined as a tuple $s_i = (N, T, R, W, D, U)$ where,

- N is a string representing the name of the signal.
- T is the direction of the signal. Direction can be either incoming or outgoing.
- $R = \{min, max\}$, where max is the upper bound and min is the lower bound.
- W is a scalar a where $a \in \{1, 2, 3.....n\}$ and $a \geq 1$
- $D \in \{UINT8, UINT16, UINT32, UINT64, INT8, INT16, INT32, INT64, FLOAT, BOOL, STRUCT, VOID\}$
- U is any basic or complex physical unit.

The structural similarity evaluation process is performed in several steps (Section 2.6). At the beginning of the comparison process, all signals from one interface are compared with all signals of another interface. The comparison of two signals is broken down to the comparison of their attributes [Keh15]. A metric is used during the comparison process which ensures a defined rule. The metric generates a cost value which represents the effort to transform one signal to another. This cost is basically a weighted average of the transformation cost for all attributes of both signals. A higher transformation cost corresponds to a lower transformation effort [Keh15] and higher level of similarity. The transformation cost is represented by a value $x \in [0, 1]$ where 0 represents infinite transformation cost and 1 represents no transformation cost. If one or more signal attributes can not be transformed, it is represented with infinite transformation cost. The transformation cost between two signals is infinite when one or more of their attribute's transformation cost is infinite [Keh15][KRS+16]. Two functions named $trans_p$ and sim_p are defined in [KRS+16]. The function $trans_p(p_1, p_2, p)$ identifies if any transformation is possible from value p_1 to value p_2 of same attribute p. The function $sim_p(p_1, p_2, p)$ calculates the transformation cost x when $trans_p(p_1, p_2, p)$ is *true*. For signal attributes, both functions are described as follows,

- **name (N)** - $trans_p(n_1, n_2, name)$ is always *true* for names n_1 and n_2. $sim_p(n_1, n_2, name)$ is calculated based on the Levenshtein distance [Lev66].
- **width (W)** - For two given widths w_1 and w_2, $trans_p(w_1, w_2, width)$ is *true* if and only if $w_1 = w_2$. Therefore, $sim_p(w_1, w_2, width) := 1$.
- **range (R)** - For two ranges $r_1 := [min_1, max_1]$ and $r_2 := [min_2, max_2]$ the function $trans_p(r_1, r_2, range)$ is *true* if and only if $r_1 \subseteq r_2$. For $R_1 := max_1 - min_1$ and $R_2 := max_2 - min_2$, transformation cost is calculated as,

$$sim_p(r_1, r_2, range) := \frac{max(R_1, R_2) - |R_1 - R_2|}{max(R_1, R_2)} \quad (4.2)$$

- **data type (D)** - Data type d_1 can be transformed to data type d_2 if and only if d_1 can be represented with d_2. For example, an 8 bit *integer* can be represented with a 16 bit *integer* and in this case $trans_p(int8, int16, datatype)$ is *true*. The transformation cost is calculated as,

$$sim_p(d_1, d_2, datatype) := \frac{min(bit(d_1), bit(d_2)}{max(bit(d_1), bit(d_2)} \tag{4.3}$$

where $bit(a)$ returns the number of bits necessary for storing data type a.

- **unit (U)** - For two units u_1 and u_2, $trans_p(u_1, u_2, unit)$ is *true* if and only if the u_1 can be converted to u_2. For example, meter can be converted to kilometer. A quantity Q is expressible with base SI units as $[Q] := 10^n \times m^\alpha \times kg^\beta \times s^\gamma \times A^\delta \times mol^\zeta \times cd^\eta$ where $-24 \leq n \leq 24$ [TT01]. Therefore, the transformation cost can be written as,

$$sim_p(u1, u2, unit) := \frac{49 - (|n_1 - n_2|)}{49} \tag{4.4}$$

Once the transformation cost for each attribute is available, the transformation cost from signal s_1 to s_2 is calculated as,

$$trans_s(s1, s2) := \frac{\sum\limits_{p \in props} weight_p \times sim_p(prop(s1, p), prop(s2, p), p)}{|props|} \tag{4.5}$$

where $trans_s(s_1, s_2)$ returns the transformation cost from s_1 to s_2, *props* is the set of signal attributes, $weight_p$ is the weight for attribute $p \in props$ and $prop(s_1, p)$ returns the value of attribute p for signal s_1. The result of this step is a directed complete bipartite graph where each signal from one interface has a directed edge to all signals from the other interface. The value of the edge represents a directed transformation cost for the vertices it connects.

In the next step, a mapping is created between signals from different interfaces where each signal from one interface is mapped to exactly one signal from the another interface. The mapping is created so that the total transformation cost from one interface to another interface is maximized [Keh15]. It is possible to have signals that do not have a mapped counterpart. Signal pairs having the an infinite transformation cost are not mapped. At the end of this step, the bipartite graph generated in the previous stage is reduced where one vertex is connected to only one another vertex with a directed edge where the value of that edge is finite.

At this point, the directed similarity between two interfaces can be calculated. Let S_1 and S_2 be set of signals for interfaces I_1 and I_2 respectively. The similarity from I_1 to I_2 can be calculated with the function $sim_i(I_1, I_2)$ which is defined as,

$$sim_i(I_1, I_2) := \frac{\sum\limits_{s_1 \in S_1, s_2 \in S_2} trans_s(s_1, s_2)}{|S_1|} \tag{4.6}$$

Therefore, the output of the structural similarity step is a signal mapping which maps signals form one interface to signals to other interface and an attribute based similar which specifies how much similarity the mapped signals are.

4.1.5 Read test cases

Test cases for each software component are read and stored as an attribute of the corresponding software component. Like any other read operations in the SimA algorithm, the reading of test cases follows requirements defined in section 3.2 and is absolutely independent of any other part of the algorithm.

4.1.6 Calculate semantical similarity

As discussed in section 2.7, semantical similarity refers to behavioural similarity. In [RSW+15], a method is presented to find behavioural compatibility between two simulink models. Unfortunately, simulink models contains internal variables. These variables must be replaced with all possible values to perform the analysis which causes a state space explosion problem [Kot03]. Therefore, semantical similarity is calculated based on test cases associated to software components [Thi15]. A given test specification includes one or more tests for exactly one component. Tests may be individual test cases or sequences of test cases. The test cases are executed in a predetermined order and each test case can influence the next test case. Test cases only contain input and output behaviour of the component without any internal variables. Hence, the state space explosion problem is avoided.

[Thi15] presents the process of semantical similarity evaluation based on test cases. The process receives two test specifications as input which, in ideal cases, should belong to different software components. These test specifications are converted to I/O EFAs (section 2.7). One I/O EFA is created for each output port [RRS+16]. These I/O EFAs has to be deterministic for further analysis.

After creation, the I/O EFAs are tested for compatibility. Here compatibility refers to compatible data type and value ranges for each input port. Ports of the I/O EFAs from one test specification needs to be mapped to ports of the I/O EFAs from other test specification. If a test specification has more ports than the other, the extra ports are mapped to constant values. These port mappings are generated during the structural similarity phase. If there are ports that are not mapped, further analysis is not possible and the whole process stops here.

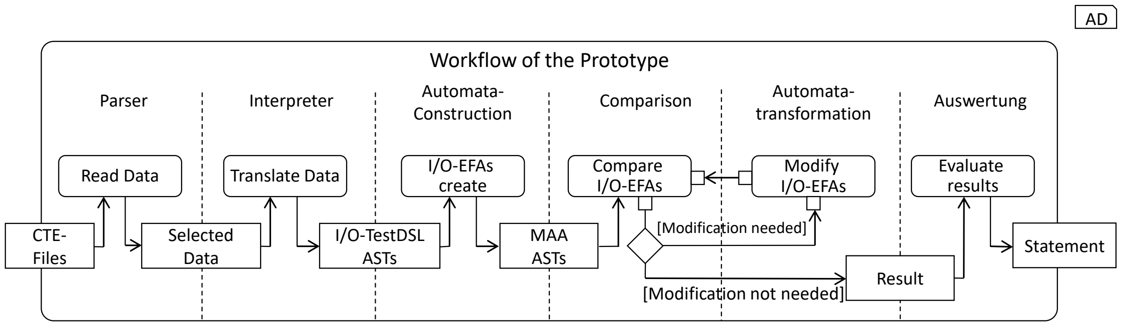


Figure 4.3: Workflow of the semantical similarity prototype [Thi15]

The semantical similarity is calculated based on the simulation relation (section 2.7) between two automatons A and B. For this purpose, I/O EFAs created in previous step needs to be transferred to I/O TSs A' and B' (section 2.7) [RSW+15]. Counterexample Guided Abstraction Similarity Metric (CEGAS) [RRS+16] is used to analyse the similarity of B' to A' by creating a simulation relation. The simulation algorithm proposed in [RSW+15] provides counterexample when B' cannot be simulated by A'. The output function of the transition in which B' defers from A' are removed iteratively based on the counterexamples and replaced with ϕ. ϕ represents an undefined behaviour and the CEGAS metric is designed to match it with any arbitrary behaviour. Thus, the behaviour of B' that does not match A' is abstracted iteratively until B' can be simulated by A'. After abstraction, automatons A' and B' are changed to A'' and B'' respectively. The degree of semantical similarity in the direction A to B represented with $G_{CEGAS}(A, B)$ [RRS+16] is calculated as,

$$G_{CEGAS}(A, B) := \frac{|\{e | e \in E_{A''} \cup E_{B''}, h_e \neq \phi\}|}{|\{e | e \in E_{A'} \cup E_{B'}\}|} \tag{4.7}$$

where, E_X is the set of transitions of any automaton X and h_e is the output function of corresponding transition. Therefore, the semantical similarity is the ratio between amount of transitions that has a defined output and total amount of transitions in unmodified automatons.

As stated earlier, each automaton represents an output port of a particular software component. Therefore, the semantical similarity between two automata from different components represents the similarity between those specific output ports.

4.1.7 Collective similarity data generation

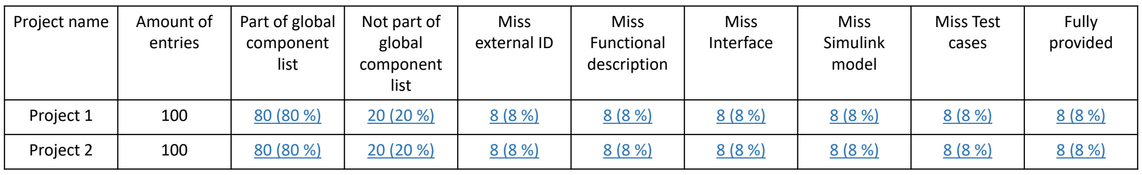
At this step, all calculated similarities are stored in a data model and is used for any further computation. While storing similarity information, the data model also calculates useful correlations among those data. For example, extrinsically similar components for a given name, structural and semantical similarity information for a group of extrinsically similar components, all structural and semantical similarity information for a given component etc. These information are useful for querying similarity information based on different criteria. The data model uses hash based data structure which results into very fast read and write operations [ML75]. Therefore, the similarities are calculated once, stored in collective similarity data model and fetched when necessary.

4.1.8 Generate reports

As discussed in 3.1, the whole purpose of the similarity calculation and the SimA framework is to help developers establish and maintain an SPL following the APLE methodology. The report generation module is flexible and can be extended to generate different kind of reports. The SimA framework by default generates two kind of reports.

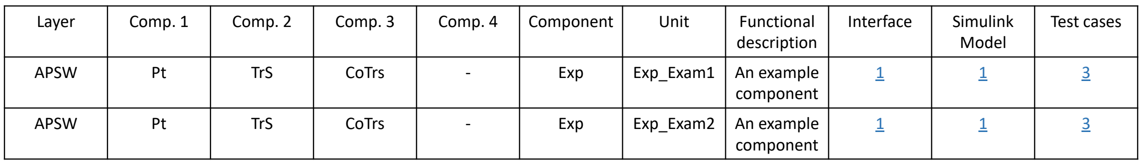
4.1.8.1 Project status

This report presents various information about different software components for each project. These information are presented in two different kind of tables. The projects status table, as an example shown in figure 4.4, shows the overview for each project. And



Project name	Amount of entries	Part of global component list	Not part of global component list	Miss external ID	Miss Functional description	Miss Interface	Miss Simulink model	Miss Test cases	Fully provided
Project 1	100	80 (80 %)	20 (20 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)
Project 2	100	80 (80 %)	20 (20 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)	8 (8 %)

Figure 4.4: Project status report table structure.



Layer	Comp. 1	Comp. 2	Comp. 3	Comp. 4	Component	Unit	Functional description	Interface	Simulink Model	Test cases
APSW	Pt	TrS	CoTrs	-	Exp	Exp_Exam1	An example component	1	1	3
APSW	Pt	TrS	CoTrs	-	Exp	Exp_Exam2	An example component	1	1	3

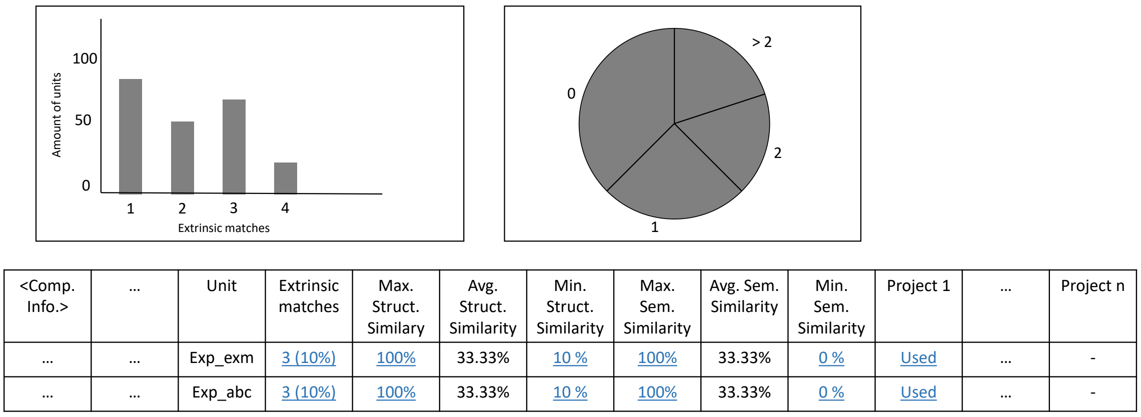
Figure 4.5: Software component list table structure.

each row of the software component list table (example in figure 4.5) presents information about one specific software component.

For the project status table, as the figure 4.4 shows, the first column specifies the project and other columns shows various information about the software components of the respective project. For example, the *"Miss Interface"* column shows the number and percentage of software components missing an interface definition. The coloured and underlined texts are links to component list table each having different data listed. For example, the links of the *"Part of global component list"* column for *Project 1* row leads to a component list table that lists software components from *Project 1* which are part of global component list.

4.1.8.2 Component usage

This report is a collection of interconnected reports which presents the computed similarities in a viewer friendly format. As shown in figure 4.6, the similarity overview report



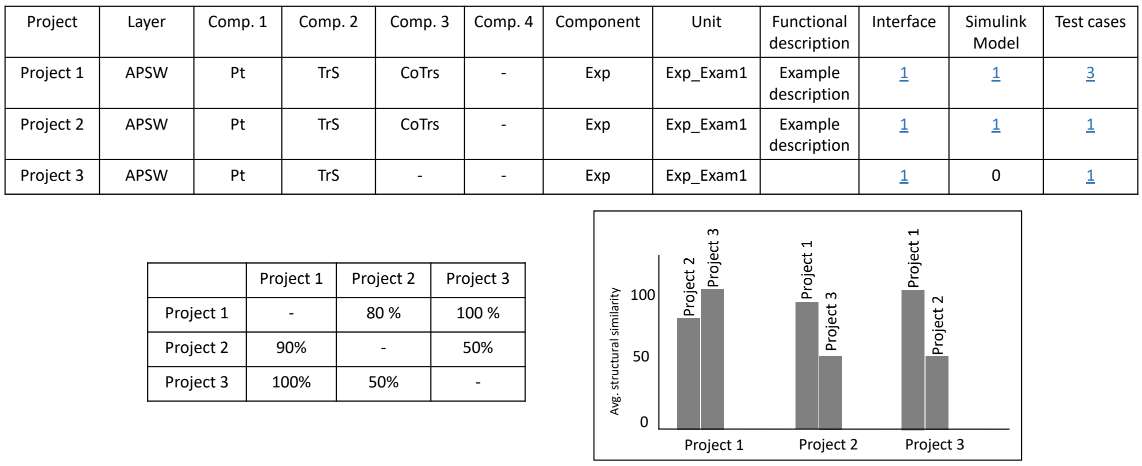
<Comp. Info.>	…	Unit	Extrinsic matches	Max. Struct. Similary	Avg. Struct. Similarity	Min. Struct. Similarity	Max. Sem. Similarity	Avg. Sem. Similarity	Min. Sem. Similarity	Project 1	…	Project n
…	…	Exp_exm	3 (10%)	100%	33.33%	10 %	100%	33.33%	0 %	Used	…	-
…	…	Exp_abc	3 (10%)	100%	33.33%	10 %	100%	33.33%	0 %	Used	…	-

Figure 4.6: Similarity overview report example.

lists all unique software components from different projects, their usage and similarity information. At the beginning, the bar diagram and pie chart shows a summery regarding the extrinsic matches.

In the table, first few columns provides information about the software component itself (like the report shown in fig. 4.5). The *"Extrinsic matches"* column shows how many extrinsic matches are available or how many projects are using this particular software component. The next few columns show maximum, average and minimum similarity information for structural and semantical analysis. These columns links to a more detailed similarity report for the information presented in them (later explained with figure 4.9). The last few columns shows which projects are using it.

However, the text in the *"Extrinsic matches"* column links to the structural similarity overview report which is shown in figure 4.7. The report starts with a table which presents



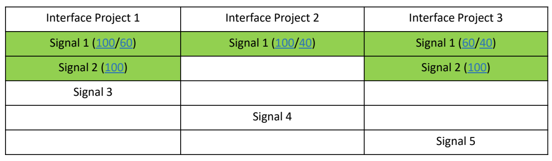
Project	Layer	Comp. 1	Comp. 2	Comp. 3	Comp. 4	Component	Unit	Functional description	Interface	Simulink Model	Test cases
Project 1	APSW	Pt	TrS	CoTrs	-	Exp	Exp_Exam1	Example description	1	1	3
Project 2	APSW	Pt	TrS	CoTrs	-	Exp	Exp_Exam1	Example description	1	1	1
Project 3	APSW	Pt	TrS	-	-	Exp	Exp_Exam1		1	0	1

	Project 1	Project 2	Project 3
Project 1	-	80 %	100 %
Project 2	90%	-	50%
Project 3	100%	50%	-

Figure 4.7: Structural similarity report example for different projects.

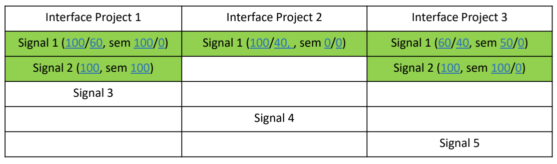
information about extrinsically matched software components from different projects. The next table and the multi-bar diagram presents the structural similarity information for the corresponding components. For example, the first row of the table shows how similar is the component from *Project 1* to the ones from *Project 2* and *Project 3*. One important thing to note here is that the structural similarity information presented here are directed.

As explained earlier in section 4.1.4, software component's interfaces are comprised of multiple signals and structural similarity information regarding them are shown in the signal's structural similarity overview report which is shown in figure 4.8. This report



Interface Project 1	Interface Project 2	Interface Project 3
Signal 1 (100/60)	Signal 1 (100/40)	Signal 1 (60/40)
Signal 2 (100)		Signal 2 (100)
Signal 3		
	Signal 4	
		Signal 5

(a) Without semantical similarity information

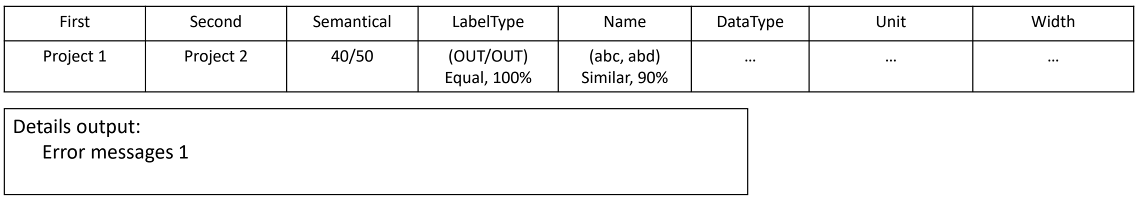


Interface Project 1	Interface Project 2	Interface Project 3
Signal 1 (100/60, sem 100/0)	Signal 1 (100/40, , sem 0/0)	Signal 1 (60/40, sem 50/0)
Signal 2 (100, sem 100)		Signal 2 (100, sem 100/0)
Signal 3		
	Signal 4	
		Signal 5

(b) With semantical similarity information

Figure 4.8: Signal similarities overview example for multiple interfaces

shows which signals are similar to which ones from different project and their similarity percentage. Semantical similarity percentage is also shown (fig. 4.8b) for signals which are eligible for semantical analysis. Each percentage number beside the signals names links to even more detailed attribute similarity report (fig. 4.9) for corresponding signals. This



First	Second	Semantical	LabelType	Name	DataType	Unit	Width
Project 1	Project 2	40/50	(OUT/OUT) Equal, 100%	(abc, abd) Similar, 90%	...	...	...

Details output:
Error messages 1

Figure 4.9: Signal attribute similarity report example

report shows different attribute values for the corresponding signals and their similarity information. Semantical similarity information is also shown for eligible signals. A section below the table shows some more details regarding the semantical similarity.

4.2 Global component list generation

Global component list is a list of all the components which are used in different projects. It provides information about which software components are currently available and are being used by different projects. These information are very useful for creating an SPL from scratch. The SimA framework is capable to generate this global component list. Figure 4.10 shows the generic workflow for generating the list.

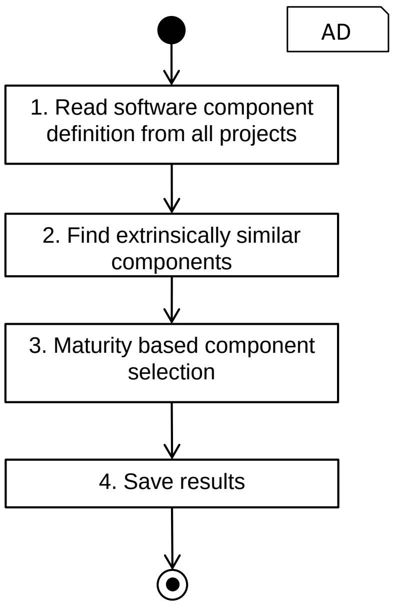

AD
1. Read software component definition from all projects
2. Find extrinsically similar components
3. Maturity based component selection
4. Save results

Figure 4.10: Calculate global component list

Step 1 and 2 from figure 4.10 are similar to the similarity analysis steps described in 4.1.1 and 4.1.2 respectively. Therefore, step 3 and 4 are explained here.

4.2.1 Maturity based component selection

As explained in section 4.1.2, extrinsic analysis generates groups of extrinsically similar components. In this step, exactly one component is chosen from each of those groups. This selection is done based on the maturity of the component and the one with highest maturity is chosen. If there are multiple components in a group with similar maturity status, one of them is chosen randomly. Since, a component can be available from multiple sources, it takes the most stable one while computing the global list.

4.2.2 Export component list

Component list generated in the previous step is exported as a file. This file contains different information about each component, it's usage in different projects and degree of reuse. In current SimA implementation, the file is exported as an excel file. Each row of the excel file holds data about one component.

Chapter 5

Implementation

Based on the requirements discussed in chapter 3 and according to the theoretical concept explained in chapter 4, the SimA framework has been developed. The development and testing of the framework was done in the context of FEV GmbH[1]. Currently, it is a part of FEV's nightly build scripts and generates valuable reports supporting the SPL activities.

This chapter thoroughly discusses the implementation details of SimA. The discussion starts with different individual components and finally, ends with explaining how they work together to calculate similarity and generate corresponding reports.

5.1 Code conventions

Python[2] 2.7 has been chosen to develop the SimA for its flexibility, object oriented nature, rich built in functionalities and a huge ecosystem of freely available open source libraries. The SimA framework follows the coding style and guideline developed for python by Google [PPJ+13]. Some of the most important ones are discussed here.

Import module - Modules or classes should be imported using the full package path as shown in listing 5.1.

```python
# this is the correct way
from similarity_framework.data.models import RepositoryItem

# this is not the right way
from models import RepositoryItem
```

Listing 5.1: Importing modules.

Exceptions - To signal some failed operations or change the flow of the code, exceptions can be used. However, it should be used with caution. Any custom exception class must extend from the build-in `Exception` class. Exceptions should be raised using the standard `"raise MyException('Error message')"` syntax. Catching the base class `Exception` in a `try/except` block will catch any exception and possibly make debugging very difficult. Therefore, catching `Exception` should be avoided unless it is reraised.

[1]http://www.fev.com/de/germany.html
[2]https://www.python.org/

Global variables - In python global variables are the ones declared on the module level. Using global variables can change module behavior during import [PPJ+13]. Therefore, use of global variable should be avoided as much as possible.

Single line expressions - Short hand single line expressions like generator expression, list comprehension and conditional expressions are very useful. But, they should only be used for simple cases. In case of complex situation, standard python loop or `if/else` syntax should be used. Generator expressions are useful for lazy loading scenarios. But for complex cases a function with `yield` statement should be used.

Default arguments - In general, default arguments provide a very useful way to assign values to method parameters if no value has been provided. It is particularly useful for functions with lots of parameters. It also eliminates the necessity to define multiple different methods with same name but varied parameters. However, default argument for mutable types (e.g. list, dict) should be avoided. As shown in listing 5.2, mutable default values can be modified by the function and therefore can result into wrong output.

```
def print_list(data=[]):
    data.append(4)
    print data

print_list()     # output [4]
print_list()     # output [4, 4]
```

Listing 5.2: Mutable default argument problem.

Classes - Each class, including nested classes, should have a parent class. If a class inherits from no other class, it should be explicitly inherited from `object`. Listing 5.3 shows correct and incorrect class definitions.

```
class MyClass(object): # this is correct
    pass

class InheritedClass(MyClass): # this is also correct
    pass

class AnotherClass: # this is NOT correct
    pass
```

Listing 5.3: Class inheritance from object.

Access control - Accessor or `get/set` functions should be avoided if the access to the variable is simple which minimizes the cost of function calls. Properties can be used if the access to the variable requires some additional functionality. Accessor functions should only be used in case of complex data access.

Indentation - Mixing tab and space for indentation must be avoided. In general, using space is encouraged and each indentation should be 4 spaces long.

Naming conventions - Table 5.1 lists the naming conventions specified by [PPJ+13]. The first column lists the type of the entity, the second column shows the naming convention for corresponding type if it is defined as public member and the third column shows the naming convention if it is internal.

Table 5.1: Naming conventions [PPJ+13]

Type	Public	Internal
Packages	lower_with_under	
Modules	lower_with_under	_lower_with_under
Classes	CapWords	_CapWords
Exceptions	CapWords	
Functions	lower_with_under()	_lower_with_under()
Global/Class Constants	CAPS_WITH_UNDER	_CAPS_WITH_UNDER
Global/Class Variables	lower_with_under	_lower_with_under
Instance Variables	lower_with_under	_lower_with_under (protected)
Method Names	lower_with_under()	_lower_with_under() (protected)
Function/Method Parameters	lower_with_under	
Local Variables	lower_with_under	

5.2 Dependencies

The SimA framework provides similarity calculation and report generation functionality in the context of FEV GmbH. This implementation is dependent on various other libraries, packages and projects. The reporting module [Kha16], structural similarity calculation module [Keh15] and the semantical similarity [Thi15] are the project dependencies. Different python packages are also required. Some of these packages are required by the above mentioned projects and some are required by SimA itself. *PIP*[3] is the python package manager used to locate and install these dependencies. The Python packages and libraries upon which SimA is dependent are listed below. All of these packages can be found at the Python package index[4].

xlrd - Read and write data to older excel files (e.g. `xls`). This package is a dependency for structural similarity analysis project [Keh15].

pint - It is used to manipulate physical units (e.g. $km, m/s^2$). The structural similarity framework [Keh15] uses it to compare signal units.

pyparsing - This library can parse simple expressions. It is used by structural similarity framework [Keh15] to check unit convertibility.

networkx - A python based graph manipulation library. It is used extensively by structural similarity framework [Keh15] for managing its internal data structures.

svn - It provides a python interface to execute SVN commands.

openpyxl - This package can read and write newer (e.g. `xlsx`) excel files. It is used to read and write global unit list files.

numpy - It is a library capable of performing complex numerical calculations. It is used during extrinsic similarity calculation.

mlab - To call Matlab functions from python, this package is used. Matlab scripts provided by FEV GmbH are used to read different data from the company repository.

[3]`https://wiki.python.org/moin/CheeseShopTutorial#Pip`
[4]`https://pypi.python.org/pypi`

enum34 - Enumerations was first introduced to at python 3.4. *enum34* package is a backport for all python versions older than 3.4. This package is used by SimA to represent different enumerations.

beautifulsoup4 - It is a pure python based HTML scrapping library. The reporting package [Kha16] uses it for HTML generation.

Jinja2 - A template engine written in pure python. It is being used by the reporting package [Kha16] to dynamically generate HTML files based on templates.

5.3 Package structure

The SimA framework and it's project dependencies are structured into several packages as shown in the figure 5.1. The diagram shows all the major packages, sub-packages and modules. Modules are the leaf nodes of the hierarchy tree which contains classes and functions that provide various functionalities. Among these packages the `similarity_framework` and `extrinsic_similarity` were developed as part of this thesis. The purpose and functionality of these packages are briefly discussed in this section.

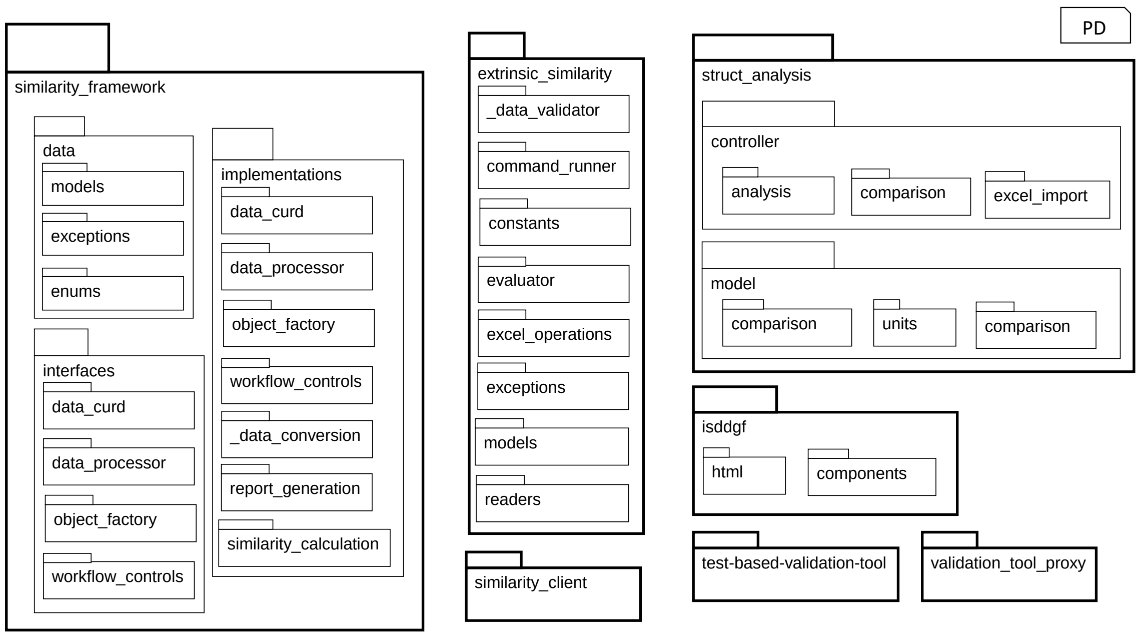


Figure 5.1: SimA package structure

Figure 5.2 shows the package dependencies. Packages marked with grey colour are library packages which are installed with different package managers (e.g. pip or maven). The `similarity_client` package is the entry point for executing the full SimA workflow.

similarity_framework - The SimA framework is implemented under this package. It is divided into three sub-packages. The `data` package contains all the data model definitions. Different python abstract classes are defined under the `interfaces` package. The SimA framework provides various functionalities by exposing related

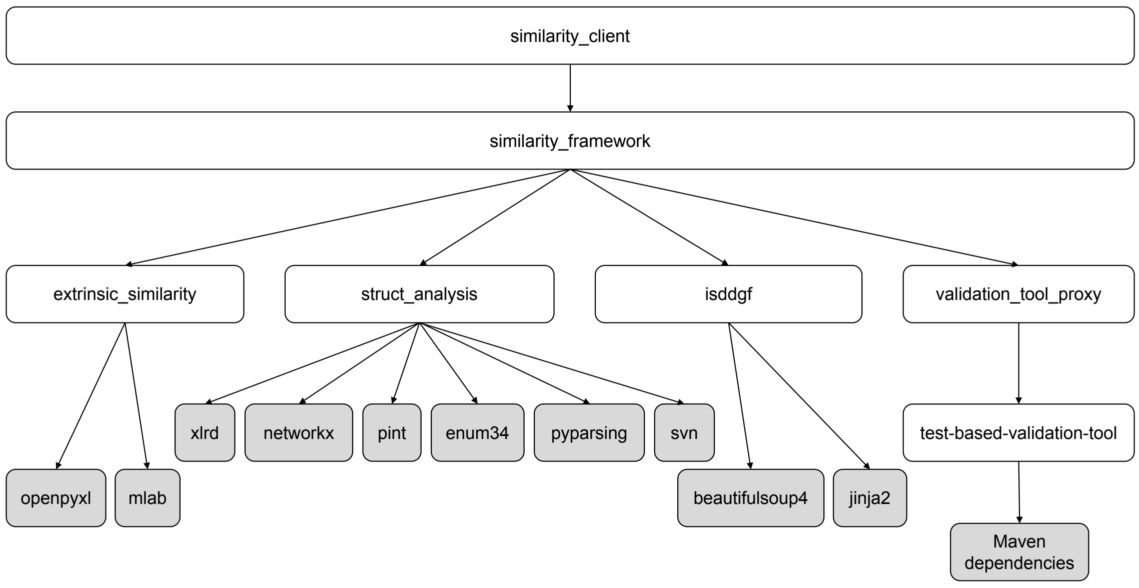


Figure 5.2: Package dependencies.

interfaces through these abstract classes. The `implementations` package contains modules which provide implementations of the interfaces in `interfaces` package.

extrinsic_similarity - This package provides the extrinsic similarity calculation and related data read-write functionalities. It is divided into several modules. The `evaluator` module contains classes and functions to calculate extrinsic similarity (sec. 4.1.2) and global component list calculation (sec. 4.2). The `readers` module provides functionalities like reading project configuration excel file, reading global component list excel file etc. The `command_runner` module which contains the functionality to run matlab functions from python interface via the `mlabwrap` library (sec. 5.2).

struct_analysis - Structural similarity and related functionalities are provided by this package. It was developed as by [Keh15] and is a dependency of the SimA framework. The package diagram shown in figure 5.1 only shows the packages and modules which are used by the SimA framework. The `model` package holds all data model definitions. Functionalities like similarity calculation and data reading are provided by the `controller` package. For example, the `excel_import` module provides functions for reading the software component definitions.

test-based-validation-tool - This package calculates semantical similarity based on the test cases as described in section 4.1.6. It is a dependency of the SimA framework and was developed by [Thi15].

validation_tool_proxy - This package provides a simple interface for communicating with the `test-based-validation-tool` package. Unlike the SimA framework, `test-based-validation-tool` package is developed with Java. Therefore direct communication is not possible from python. `validation_tool_proxy` provides functionality which can be called from python interface and returns the results as json sting which can be read and converted into python objects.

isddgf - The implementation of reporting module of SimA framework is dependent on this package. It was developed by [Kha16] and provides python based interfaces to create html based reports.

similarity_client - This package works as an entry point to the SimA framework. The framework provides interfaces which are context independent. But, the implementations to those interfaces are dependent to the context of FEV GmbH and requires certain inputs (e.g. path to the SVN repository) which are specific for the company. The `similarity_client` provides these values to the SimA framework and initiates the similarity calculation and report generation workflow.

5.4 The `extrinsic_similarity` package

The `extrinsic_similarity` package is an standalone implementation which, in the context of FEV GmbH, can read data from different projects, find extrinsically similar software components from them and finally generate an excel based global unit list. The extrinsic similarity calculation is based on the algorithm explained in section 4.1.2.

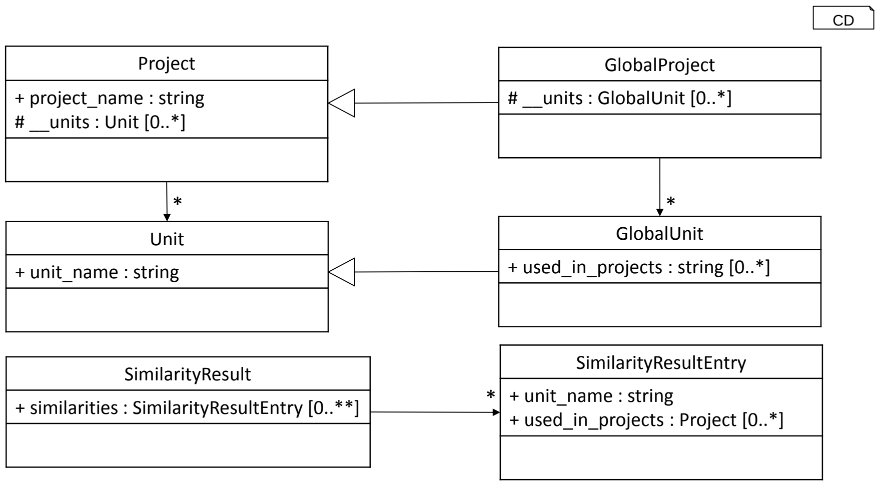


Figure 5.3: `extrinsic_similarity` data models

5.4.1 Data models

The `extrinsic_similarity` package has its own set of data models. The important ones are shown in figure 5.3.

The `Project` model represents a project in the context of FEV GmbH. It contains information like corresponding project's SVN repository location, component list file path etc. It holds a list of software components and each of these components are represented by `Unit`. Each `Unit` is primarily identified by its name. It holds various other attributes like the interface definition file location, maturity status of the unit etc.

`GlobalProject` and `GlobalUnit` extends from `Project` and `Unit` respectively. The `GlobalUnit` represents a software component which has been used in one or more software projects. Therefore, it has an additional attribute called `used_in_projects`. This attribute is the list of those project names which uses the corresponding software component. A `GlobalProject` is a dummy project which holds a list of `GlobalUnit`. This list represents the global software component list and also contains usage information for each of those components. In the context of FEV GmbH, a global component list is stored in an excel file where each row contains information for one component and its usage. The `GlobalProject` is a representation of this global list.

The `SimilarityResult` model represents the calculated extrinsic similarity. It is basically a collection of `SimilarityResultEntry`. Each `SimilarityResultEntry` contains a name of an `Unit` and list of `Project` which are using it.

5.4.2 Data import and similarity calculation

Besides the extrinsic similarity calculation, the SimA framework utilizes some of the data read/write functionalities implemented by the `extrinsic_similarity` package. Figure 5.4 shows the key entities which are used by the SimA framework.

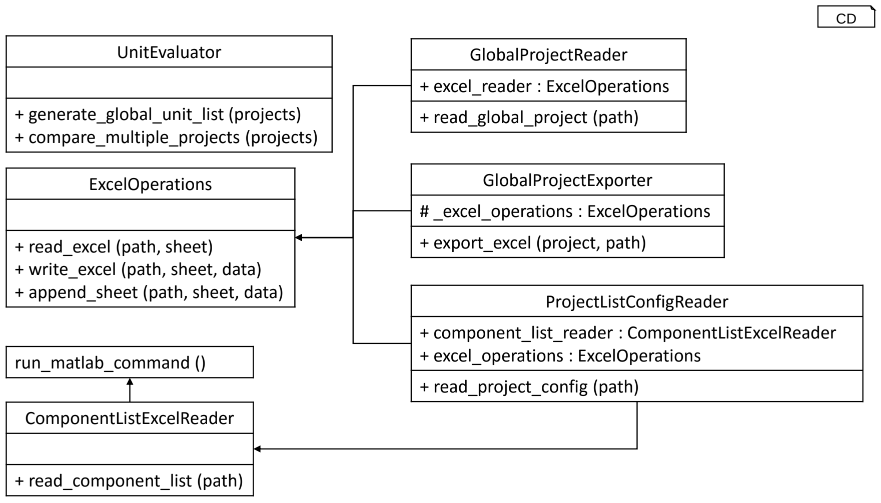

CD
UnitEvaluator
+ generate_global_unit_list (projects)
+ compare_multiple_projects (projects)
ExcelOperations
+ read_excel (path, sheet)
+ write_excel (path, sheet, data)
+ append_sheet (path, sheet, data)
run_matlab_command ()
ComponentListExcelReader
+ read_component_list (path)
GlobalProjectReader
+ excel_reader : ExcelOperations
+ read_global_project (path)
GlobalProjectExporter
# _excel_operations : ExcelOperations
+ export_excel (project, path)
ProjectListConfigReader
+ component_list_reader : ComponentListExcelReader
+ excel_operations : ExcelOperations
+ read_project_config (path)

Figure 5.4: Key classes of `extrinsic_similarity` package

ExcelOperations - This class provides necessary functions to read and write to excel files and belongs to `excel_operations` module. It uses the *openpyxl* (sec. 5.2) package to perform the actual read/write. The `read_excel` function reads specified sheet of an existing excel file. The `write_excel` function creates a new excel file and writes data to specific sheet of that file. It is also possible to add a new sheet to an existing excel file with the `append_sheet` function.

GlobalProjectReader - In the context of FEV GmbH, the global list of software components are stored as an excel file and this list belongs to a dummy project called global project. Each entry of this list provides information about the software component and the names of the projects which are using this particular component. The `GlobalProjectReader` reads this component list excel file and returns it as a `GlobalProject`. The `ExcelOperations` is used for reading this excel file.

GlobalProjectExporter - This class exports a `GlobalProject` to an excel file. This excel file, as described earlier, lists all software components from all projects and their usage information. `ExcelOperations` is used for writing to an excel file.

ComponentListExcelReader - Software component definitions from different projects are read with `ComponentListExcelReader`. In the context of FEV GmbH, each project has an excel file based software component list. This list contains software component information like it's name, composition levels, interface information and various other data.

FEV GmbH provides Matlab scripts to read these component list files. The matlab function `getProjectUnits`, which is part of FEV's toolchain system, reads the excel files and returns the data as matlab `struct`. Another matlab function `exportUnitsToJson`, developed along `extrinsic_similarity` package, which takes path to the component list files and calls the FEV scripts with this path and converts the returned results to json string and writes it to a temporary file. With the help of *mlabwrap* (sec. 5.2) package, `ComponentListExcelReader` calls the `exportUnitsToJson` function and reads the exported json data from the temporary file. Finally, this data is converted to a list of `Unit` objects.

ProjectListConfigReader - In the context of FEV GmbH, basic information about all software projects are stored in an excel file. Each row of this file contains information like name of the project, project type, project's SVN repository path etc. `ProjectListConfigReader` reads the configuration file and returns a list of `Project` objects. `ExcelOperations` is used for reading the excel file.

UnitEvaluator - This class can perform various analysis based on the data available from different projects. The function `compare_multiple_projects` calculates extrinsic similarities among provided list of `Project` and return the results as `SimilarityResult` object. This calculation is implemented based on the algorithm explained in section 4.1.2.

The `UnitEvaluator` class can also generate global component list with function `generate_global_unit_list`. It takes a list of `Project` and return a `GlobalProject` representing the global component list.

5.5 The `struct_analysis` package

The `struct_analysis` package provides an implementation for performing structural comparison among interfaces of different software components [Keh15]. In the context of FEV GmbH, interface of a software component is stored as a excel file which lists all *in* and *out* signals related to corresponding interface. The `struct_analysis` tool can read these interface sheets and perform an attribute value based structural similarity analysis on them (algorithm explained in section 4.1.4). It can also generate various other similarity information based on the analysis results.

5.5.1 Data models

In figure 5.5 a class diagram has been presented showing data models which are important in the context of SimA framework. Descriptions, based on [Keh15], regarding these data models are provided below.

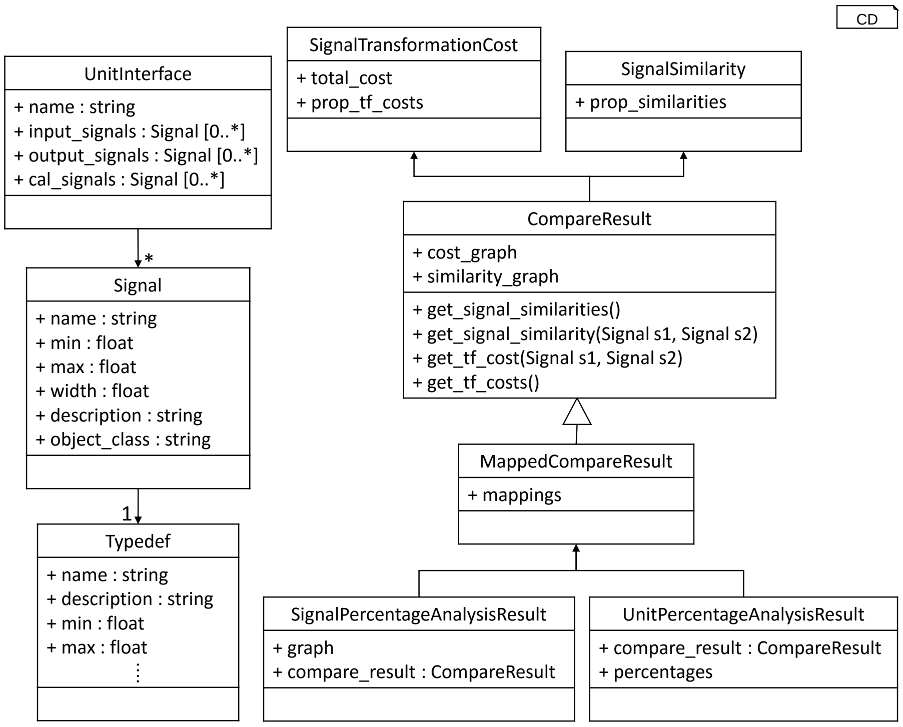


Figure 5.5: Data model package structure [Keh15]

UnitInterface - Represents an interface of a software component. It contains necessary information according to the PERSIST architecture and is also used for similarity calculation.

Signal - Each `UnitInterface` usually contains a list of `Signal` each representing an *in* or *out* port and all necessary information regarding corresponding port. An `out` signal can send data to other interface and `in` signal is used to receive data. In the context of FEV GmbH, there are five kind of `Signal` which are *in*, *out*, *mp*, *cal* and *fix*. Each `Signal` has a property of type `TypeDef` representing the type of data corresponding port can handle.

SignalSimilarity - It represents an undirected similarity between two `Signal`. It encapsulates the similarity values for each property of both signals.

SignalTransformationCost - The transformation cost from one `Signal` to another is represented with this model. Since, transformation costs are direction dependent, values represented with this data model is also direction dependent. Therefore, transformation cost from signal s_1 to signal s_2 might be different if it is calculated in other direction.

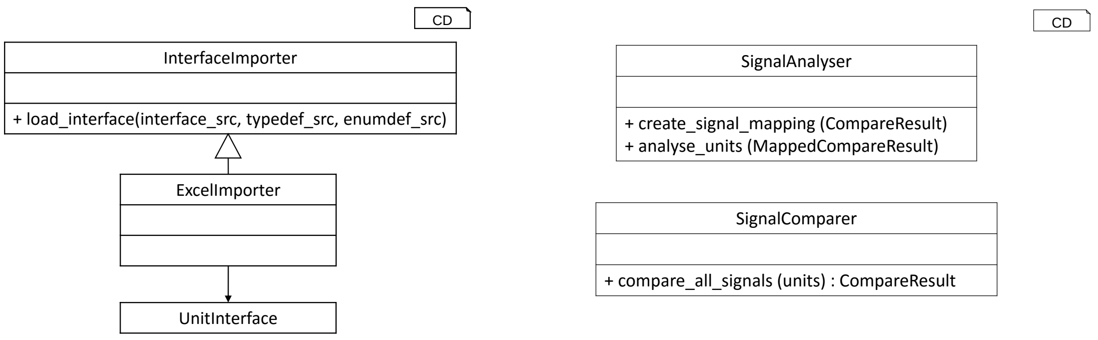
CompareResult - In section 4.1.4, the process of structural similarity calculation has been described. For two interfaces I_1 and I_2, each signal from I_1 is compared with all the signals of I_2 and the comparison results are stored with `CompareResult`. Both directed and undirected similarities are stored. Undirected similarity values are stored with `SignalSimilarity` and `SignalTransformationCost` represents the directed similarity values.

MappedCompareResult - In the mapping step, as explained in section 4.1.4, a mapping between two interfaces is calculated where one signal is mapped to only one signal of another interface. Such a mapping is represented with `MappedCompareResult`. It extends from the `CompareResult`.

UnitPercentageAnalysisResult - For each pair of `Unit`, this model stores the percentage of `Signal` which are equal, similar, weakly similar or different. The stored similarity values are directed. Therefore, for two units u_1 and u_2, the similarity value for u_1 to u_2 might be different from u_2 to u_1.

5.5.2 Interface import and similarity calculation

Figure 5.6 presents classes which are used by SimA for interface data import and structural similarity calculation. They are explained below.


CD
InterfaceImporter
+ load_interface(interface_src, typedef_src, enumdef_src)
ExcelImporter
UnitInterface
CD
SignalAnalyser
+ create_signal_mapping (CompareResult)
+ analyse_units (MappedCompareResult)
SignalComparer
+ compare_all_signals (units) : CompareResult

(a) Interface data import (b) Structural similarity calculation

Figure 5.6: Classes related to interface data import and similarity calculation [Keh15]

ExcelImporter - This class reads the excel files containing interface information. The `struct_analysis` package provides the interface `InterfaceImporter` with a `load_interface` function. This function is responsible for reading interface information. `ExcelImporter` provides implementation for reading excel file based interface definition and returns an `UnitInterface`.

SignalComparer - The comparison between two or more signals are taken care of by `SignalComparer`. It defines a metrics and uses it to calculate the similarity level between two signals. As defined in section 4.1.4, all signals from multiple interfaces are compared with each other and the similarity values are stored for future calculation. `SignalComparer` performs this comparison with `compare_all_signals` method and returns a `CompareResult`.

SignalAnalyser - This class provides several functions to perform various analysis on `CompareResult`. The structural similarity calculation algorithm (section 4.1.4) computes a mapping between signals of two interfaces. This calculation is performed with `create_signal_mapping` function which returns a `MappedCompareResult`. The `SignalAnalyser` also computes the percentage of similar signals for a pair of interfaces with the function `analyse_units`.

5.5.3 Enhancements

During the development of SimA, several enhancement has been performed to the package `struct_analysis`. Some of these enhancements optimizes the performance and some of them are bug fixes. As explained in section 5.1, classes which do not have a parent class should explicitly extend from `object`. Certain model classes of `struct_analysis` package was written without a parent class and they have been changed to inherit from `object`. This change makes it possible to take the advantage of various functions provided by `object` class. For example, to use any object as a `dictionary` key, it should have a `__hash__` function which returns a unique hash value for each different object. The previous implementation of the `Signal` model provided a custom hashing function which generated a hash value based on the attributes. Therefore, two different signal objects having same attribute values will generate same hash value. Using them both in the same `dictionary` as key can overwrite existing data which can result into wrong output. On the other hand, `Signal` can inherit the hashing function from `object` which provides a unique hash value regardless of the attribute values. This change not only follows the standard python convention (sec. 5.1) but also solves bugs which are quite difficult to track. Therefore, several model classes have been changed to inherit from the `object`.

Several enhancement has been performed on the `ExcelImporter` class. Firstly, it was refactored to extract several reusable methods and they are called multiple times instead of having same code at several places. Several long methods has also been eliminated which are hard to read and debug. Several variables, having names which are similar to python build-in modules or methods, has been changed to avoid name conflicts. The *fix* sheet of the interface definition excel file had been skipped while reading the definitions. These *fix* signals has been re-included into the interface definitions.

Error handling has also been improved. In certain circumstances, excel files can have invalid data or the data might not be properly formatted. This can cause error in computation if not handled properly. Proper handling of these situations has been provided for cases where they were not being handled.

5.6 The `test-based-validation-tool` package

This package is responsible for calculating test case based semantical similarity. The calculation process has been described in section 4.1.6. It takes a pair of test cases from two different interfaces, converts them to two sets of I/O EFAs and calculates simulation relation between related automatons.

The `test-based-validation-tool` package is developed in java. Therefore, it follows a slightly different naming convention than the ones described in section 5.1. All aspects of the package itself is not covered by the scope of this chapter. Things that are related and used by SimA framework is discussed here.

5.6.1 Data models

The `test-based-validation-tool` converts the test cases to I/O EFAs. The data model class `ConstructionConfiguration` defines the configurations necessary for creating these automatons. The similarity analysis is performed according to the configuration specified by the class `AnalysisConfiguration`. It contains path to both test files, type of the test cases, port mapping information etc. In the context of SimA, the port mapping information is extracted from the structural similarity analysis and specifies similar ports from both interfaces. The `AnalysisConfiguration` also contains two `ConstructionConfiguration` each for the corresponding test file.

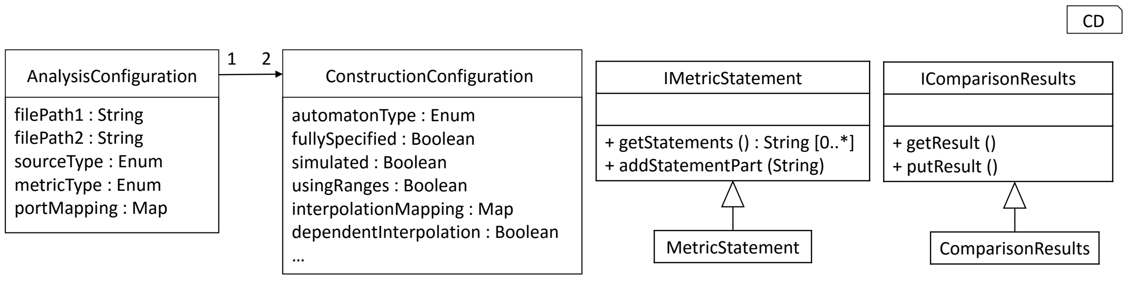


Figure 5.7: Data models

During the semantical analysis phase, the results are stored as a `ComparisonResults`. It has `getResult` and `putResult` functions for getting and adding new results to the object. After finishing the analysis phase, the list of `ComparisonResult` are converted to `MetricStatement`. It is basically a list of `String` which is a readable representation of the corresponding `ComparisonResult`.

5.6.2 Semantical analysis and evaluation

The `TestDrivenCompatibilityAnalysis` class extends from `AAnalysis` and performs the semantical analysis. The `AAnalysis` class implements the `analyze` method and defines the workflow for the semantical analysis as shown in the figure 5.8. First, the automatons are created form the test files and are checked for validity. After that, the comparison step is executed which is follow by the evaluation step. The simulation relation between two automatons is evaluated during the comparison step. It generates a list of `IComparisonResult`. This results are translated to `IMetricStatement` in the evaluation step.

The `AAnalysis` defines the workflow from an abstract level with a template method and `TestDrivenCompatibilityAnalysis` provides implementation for them. The `compare` function returns a `ComparisonResults` object. In the evaluation phase, `CEGAREvaluator` is used to evaluate the result and create readable strings represented with `MetricStatement`. The `AnalysisException` is thrown if anything unexpected occurs at any stage of the whole analysis process.

The `TestDrivenCompatibilityAnalysis` compares the test files as a whole. As a result, the total analysis procedure is halted if an error occurs during analysing one port and analysis for rest of the ports are not executed. Therefore, some potential matches can not be found and the similarity information remains incomplete. To make the analysis more robust, the `TestDrivenCompatibilityAnalysisErrorAsStatement` has been developed. It extends from `TestDrivenCompatibilityAnalysis` and overrides certain methods so that the workflow can continue regardless of any error. The errors are stored and later, during the evaluation step, added to `MetricStatement`. For this purpose, a special evaluator `CEGAREvaluatorWithProblemStatements` is used. This

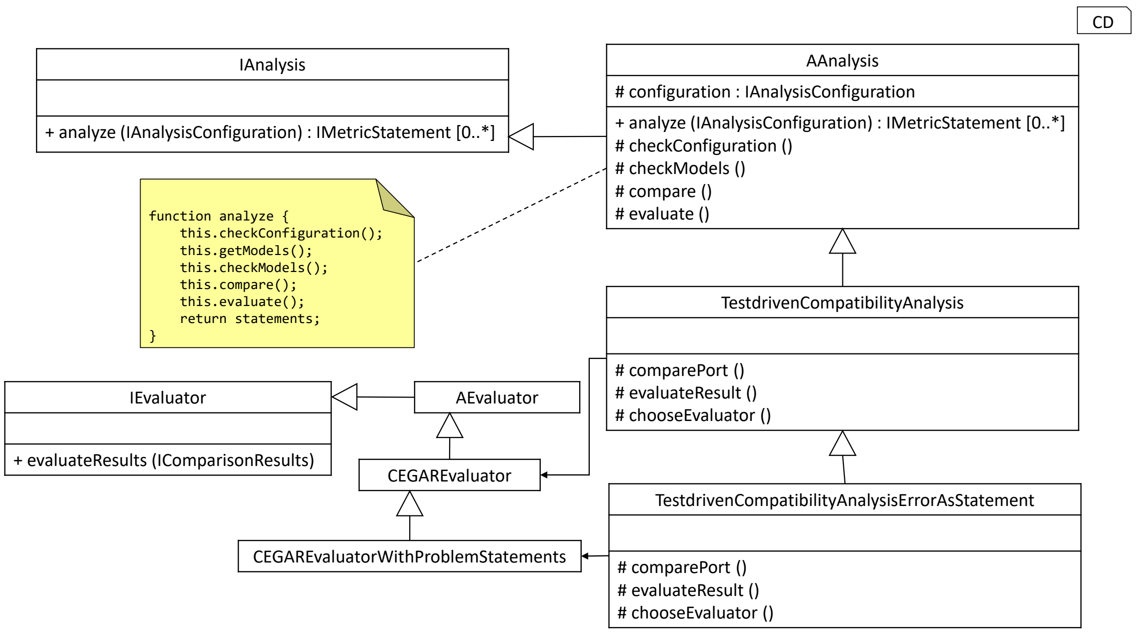

CD
IAnalysis
+ analyze (IAnalysisConfiguration) : IMetricStatement [0..*]
AAnalysis
# configuration : IAnalysisConfiguration
+ analyze (IAnalysisConfiguration) : IMetricStatement [0..*]
# checkConfiguration ()
# checkModels ()
# compare ()
# evaluate ()
function analyze {
this.checkConfiguration();
this.getModels();
this.checkModels();
this.compare();
this.evaluate();
return statements;
}
TestdrivenCompatibilityAnalysis
# comparePort ()
# evaluateResult ()
# chooseEvaluator ()
IEvaluator
+ evaluateResults (IComparisonResults)
AEvaluator
CEGAREvaluator
CEGAREvaluatorWithProblemStatements
TestdrivenCompatibilityAnalysisErrorAsStatement
# comparePort ()
# evaluateResult ()
# chooseEvaluator ()

Figure 5.8: Classes related to semantical evaluation and calculation

enhancement provides SimA framework with all necessary insight of the semantical analysis while preserving the original workflow implementation.

The similarity statements generated by `CEGAREvaluatorWithProblemStatements` are of format,

    <automata name> simulates <number> % of <automata name>

And they can be later parsed to extract the similarity values for signal ports corresponding to the mentioned automations.

5.7 The **`isddgf`** package

To make the results of analysed similarities available in an understandable and reader friendly format, SimA generates various reports (concept in sec. 4.1.8). The `isddgf` package, developed by [Kha16], provides Python api for generating javascript enabled interactive HTML documents. With the help of this package, SimA generates HTML reports to present the similarity results. Creating HTML table with sort and search support, constructing nvd3[5] based charts, bootstrap[6] based layout management are some of the very convenient and frequently used features of `isddgf` package.

5.7.1 HTML components

The `isddgf` package has built-in support for generating various HTML elements. Each of these elements can be used separately and some of them can be used as a container to create compound element (e.g. `div`). Figure 5.9 presents a class diagram showing classes

[5]http://nvd3.org/
[6]http://getbootstrap.com/

each of which represents important HTML entities. Each of the classes representing HTML elements extends from `IComponent` which contains very basic methods for HTML code generation. `BasicComponent` is a subclass of `IComponent` and acts as a parent class for container type HTML elements (e.g. `Div`). `AbstractHtmlComponent` is the parent class for all standalone HTML elements (e.g. `Hyperlink`).

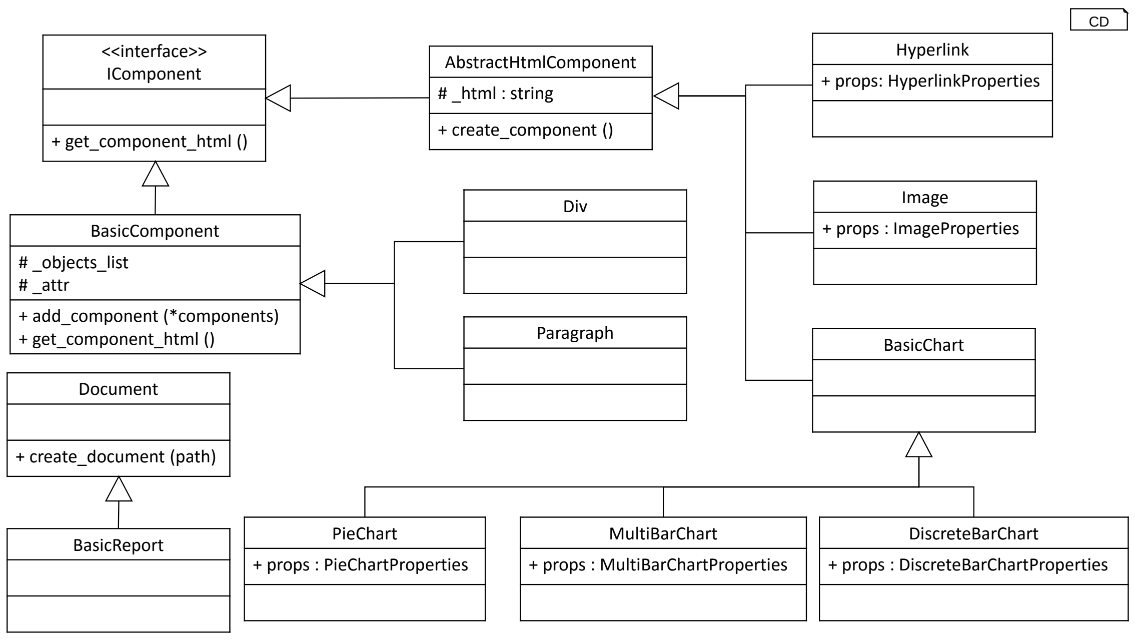


Figure 5.9: `isddgf` HTML components

Containers - As explained before, a container extends from `BasicComponent`. Figure 5.9 shows two of such classes `Div` and `Paragraph`. A container's `_objects_list` attribute contains a list of HTML elements and the `add_component` function is used to add elements to this list. An element can be anything of type `IComponent`. The `_attr` attribute maintains a list of HTML attribute for the container.

Elements - Classes which represents a single and standalone HTML element extends from `AbstractHtmlComponent`. It has functions to generate corresponding HTML code. The child classes maintain a `props` attribute (e.g. `HyperlinkProperties` for `Hyperlink`) which contains different HTML property values for the corresponding element.

Charts - The `BasicChart` is another HTML component. Like other component objects, it extends form `AbstractHtmlComponent` and works as base class for different specific chart object. The class diagram in figure 5.9 shows three different chart implementations - `PieChart`, `MultiBarChart` and `DiscreteBarChart`. The section 4.1.8 explains how these charts can be used to represent related information. Visual representation of these charts, in the context of SimA implementation, will be presented later in chapter 7.

Document - The `Document` class represents a full HTML file by defining a skeleton document. It provides functions to design an HTML document by inserting different

HTML elements to different part of the document. Finally, the document can be written to a file using `create_document` function. This class also takes care of related resources (javascript, styling and images).

5.7.2 HTML table generation

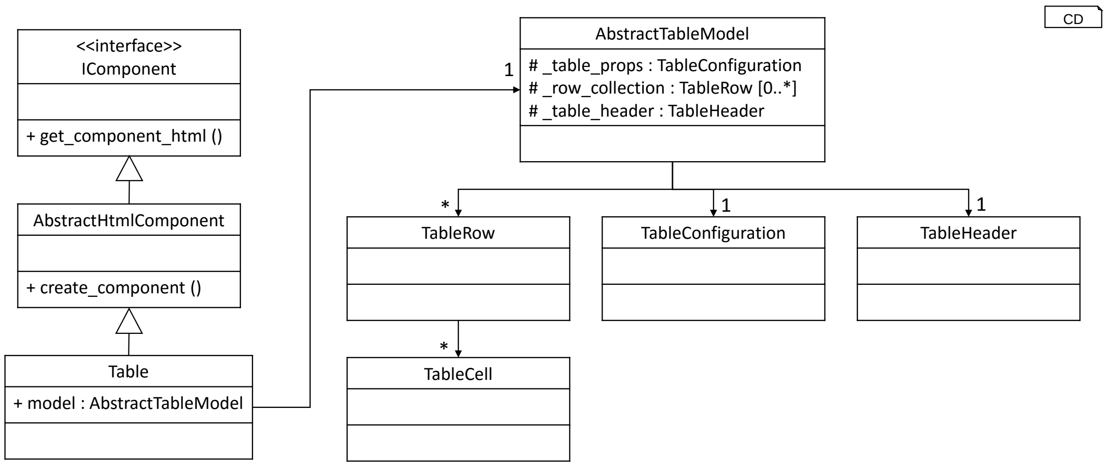


Figure 5.10: `isddgf` table and related data model

In the context of SimA, table generation is the most frequently used feature of `isddgf` package. The class diagram in figure 5.10 shows the basic structure of `Table` and `AbstractTableModel` class. The `AbstractTableModel` contains data and all necessary configuration information for generating an HTML table. The `_table_header` attribute contains styling information and data for table header. `AbstractTableModel` contains multiple `TableRow` and each of them has multiple `TableCell`. `TableCell` contains the actual data which can be any kind of text or another HTML element (e.g. hyperlink). The `_table_props` contains configuration information regarding sorting, searching, pagination etc.

5.8 The `similarity_analysis` package

This package contains the implementation of the whole SimA framework and was developed as a part of this thesis. SimA's data models, interfaces and their implementations are all contained in this package. This section explains all these parts and their purposes. Initially, the data models are explained in detail and later different interfaces and their implementations are discussed. The discussion regarding the implementations also explains how previously discussed packages are utilized in the context of SimA.

5.8.1 SimA data models

Data models define the basic data structure of an application. They are developed to imitate the real life scenario that they represent. Each model groups logically related data and stores them. Data models also define how one model is related to another and indirectly defines the data flow throughout the application. This section discusses the different data models which were defined, developed and used during the implementation of the SimA framework. Figure 5.11 presents the package structure of the data models developed for the SimA framework.

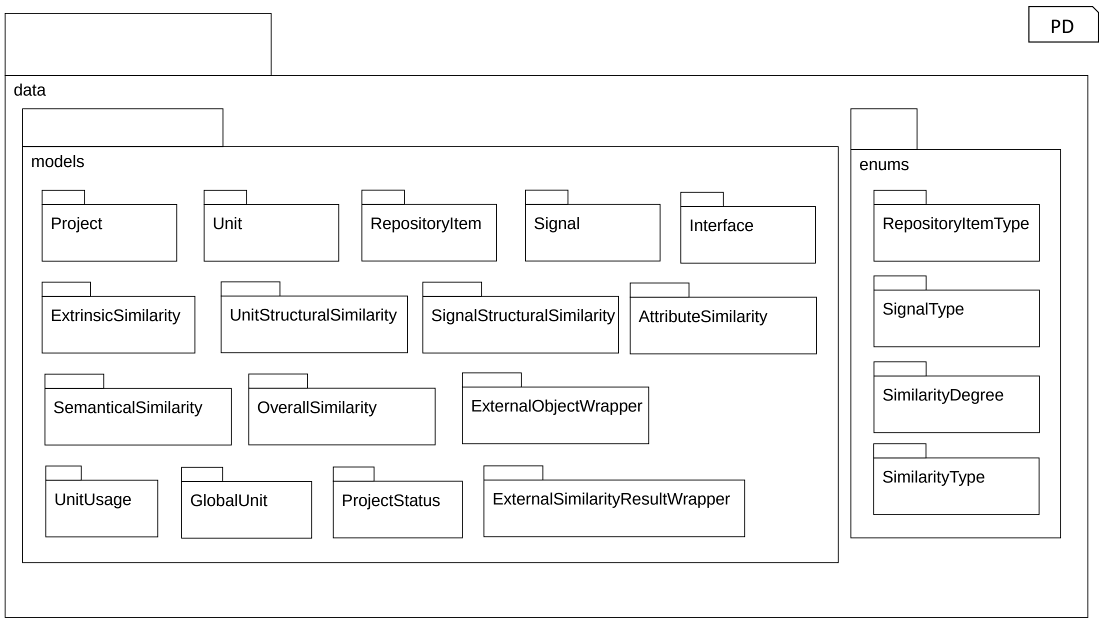


Figure 5.11: Data model package structure

5.8.1.1 Project data models

As already discussed in chapter 4, the purpose of the SimA framework is to find similar software components from different projects. Therefore, data models representing project and its related properties has to be defined. This section discusses the project model and all the other models necessary to define the project structure. Relations among these projects are also discussed here. The project structure is developed based on the PERSIST (sec. 2.2) architecture. Figure 5.12 presents a class diagram showing the overview of the project data models.

Project - The `Project` model is a part of `data.models` module. This model holds data related to a project. Since, a project can have multiple software components, each `Project` holds a list of `Unit` (explained later). The `data_source` attribute of the model holds information about the location of the project data which can be any `object`. However, in the context of FEV GmbH, project information are preserved with a Microsoft office excel file. Therefore, in current implementation it is a path `string` pointing to the specific file.

Unit - A software component is represented with the `Unit` model that is a part of `data.models` module. The `project` attribute of the `Unit` model points to the

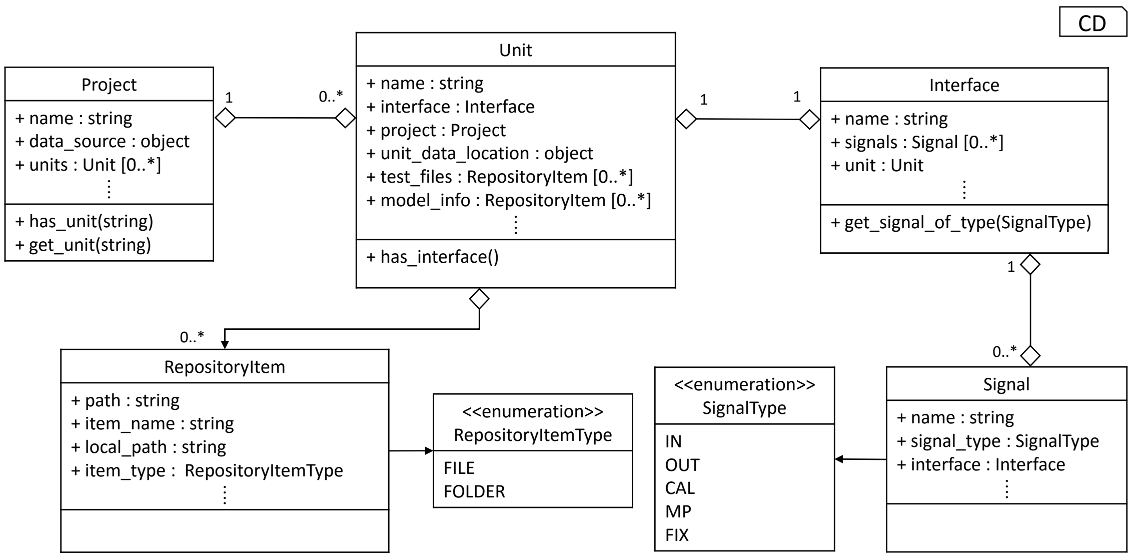


Figure 5.12: Project data models

`Project` it belongs to. Therefore, the `Project` and `Unit` models has a one-to-many relation. `Unit` also holds information regarding the available test specifications (`test_files`), simulink models (`model_info`) and interface (`interface`) of the software component. All of these are discussed later.

Interface - The `Interface` model holds information regarding the interface of the corresponding software component. An interface of a software component is a defined entry point through which multiple software components can communicate. An `Interface` has multiple different kind of ports for communicating called `Signal` (explained later). The `unit` attribute of an `Interface` points to an `Unit` model specifying the software component it belongs to.

Signal - Interfaces from different software components communicate through a well-defined port which is represented with the `Signal` model. Section 4.1.4 presents a detailed definition of a signal. The type of the `Signal` is specified with the `signal_type` attribute which can be one of the five as shown in figure 5.12.

RepositoryItem - The `RepositoryItem` model represents a remote file or folder. In the context of FEV GmbH, it represents an item hosted in an SVN repository. The `item_type` attribute specifies if it is a file or folder. The `path` attribute is the path to the remote item and the `local_path` attribute holds the path to the corresponding local item if it is locally available (e.g. checked out from SVN).

5.8.1.2 Similarity data models

As discussed in chapter 4, the SimA framework calculates similarity on extrinsic, structural and semantical level. After calculating these similarities, the results along with related information are stored with different data models. This section discusses each of these similarity result data models. In figure 5.13, a class diagram is presented which shows these similarity data models and the relation among them.

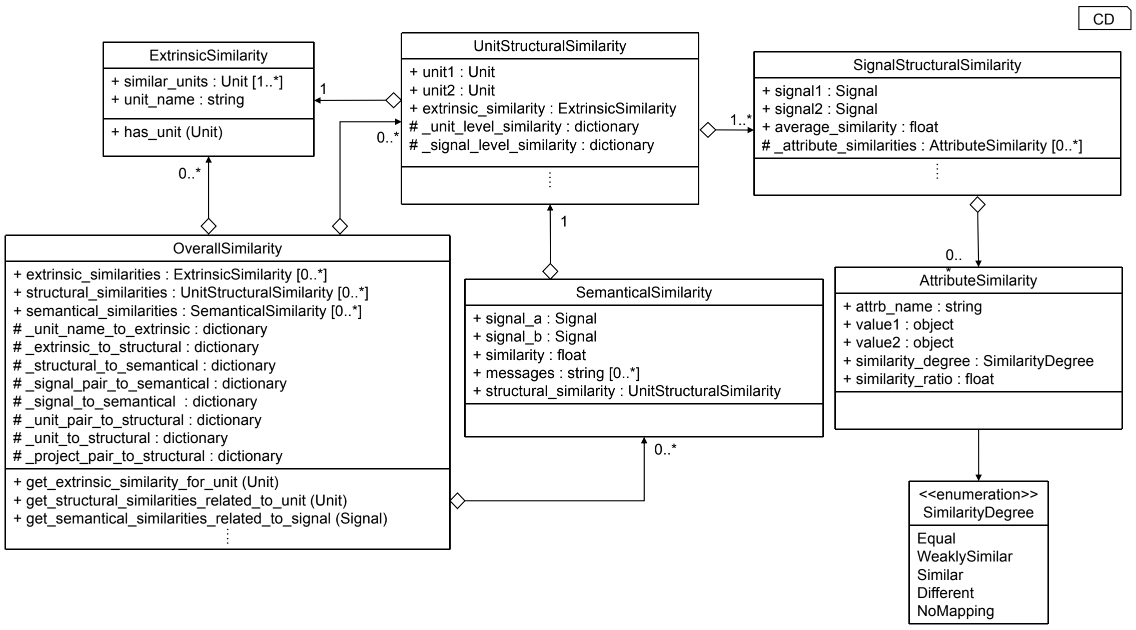

CD
ExtrinsicSimilarity
+ similar_units : Unit [1..*]
+ unit_name : string
+ has_unit (Unit)
UnitStructuralSimilarity
+ unit1 : Unit
+ unit2 : Unit
+ extrinsic_similarity : ExtrinsicSimilarity
# _unit_level_similarity : dictionary
# _signal_level_similarity : dictionary
SignalStructuralSimilarity
+ signal1 : Signal
+ signal2 : Signal
+ average_similarity : float
# _attribute_similarities : AttributeSimilarity [0..*]
OverallSimilarity
+ extrinsic_similarities : ExtrinsicSimilarity [0..*]
+ structural_similarities : UnitStructuralSimilarity [0..*]
+ semantical_similarities : SemanticalSimilarity [0..*]
# _unit_name_to_extrinsic : dictionary
# _extrinsic_to_structural : dictionary
# _structural_to_semantical : dictionary
# _signal_pair_to_semantical : dictionary
# _signal_to_semantical : dictionary
# _unit_pair_to_structural : dictionary
# _unit_to_structural : dictionary
# _project_pair_to_structural : dictionary
+ get_extrinsic_similarity_for_unit (Unit)
+ get_structural_similarities_related_to_unit (Unit)
+ get_semantical_similarities_related_to_signal (Signal)
SemanticalSimilarity
+ signal_a : Signal
+ signal_b : Signal
+ similarity : float
+ messages : string [0..*]
+ structural_similarity : UnitStructuralSimilarity
AttributeSimilarity
+ attrb_name : string
+ value1 : object
+ value2 : object
+ similarity_degree : SimilarityDegree
+ similarity_ratio : float
<<enumeration>>
SimilarityDegree
Equal
WeaklySimilar
Similar
Different
NoMapping
1
0..*
0..*
1..*
1
0..*
0..*
0..*

Figure 5.13: Data models for similarity results

ExtrinsicSimilarity - As defined in section 4.1.2, two or more Units are extrinsically similar when they have the same name. The similar_units property of the ExtrinsicSimilarity model holds an extrinsically similar list of Units. The name property of the data model holds the name of these Units.

UnitStructuralSimilarity - This model holds structural similarity information for two Units. As described in 4.1.4, the structural similarity basically refers to attribute based directed signal similarity. Therefore, UnitStructuralSimilarity model contains directed Signal similarity information (e.g. from unit1 to unit2 and vice versa) for all similar pairs of Signals from both Units. And this similarity information is stored as a dictionary represented by the attribute _signal_level_similarity. The _unit_level_similarity attribute contains directed numeric similarity value for corresponding Units. The attribute extrinsic_similarity holds the extrinsic similarity information from which current structural similarity was calculated.

SignalStructuralSimilarity - Structural similarity between two Signals are represented with this model. The _attribute_similarities is a list containing attribute based similarity information for each attribute of both Signals. average_similarity is the mean value of all the attribute based similarity values.

AttributeSimilarity - It represents structural similarity of one specific attribute for a pair of Signals. The name of the attribute is defined with attrb_name. The similarity_ratio is the numeric similarity value where 0 means not similar and 1 represents absolutely equal attribute values.

SemanticalSimilarity - The semantical similarity is calculated between two OUT type Signals. SemanticalSimilarity contains directed semantical similarity information from signal_a to signal_b. The similarity attribute is the numerical percentage of the semantical similarity. An unsuccessful similarity calculation is represented by `None` value for the `similarity` attribute and the error message are stored in `messages` attribute. The `structural_similarity` attribute holds structural similarity information from which current semantical similarity was calculated.

OverallSimilarity - This model maintains the relations among different similarities. According to the workflow proposed in section 4.1, different kind of similarities are calculated sequentially and results from one similarity calculation are used in the next step. For example, `Signal` mapping found in the structural similarity calculation is used for finding semantical similarity. Therefore, these two similarity results are related. `OverallSimilarity` keeps track of these relations. It also establishes relationship among different project models (sec. 5.8.1.1) and similarity models. For example, the function `get_semantical_similarities_related_to_signal` returns all `SemanticalSimilarity` which is related to provided `Signal`.

5.8.1.3 External object encapsulation

The SimA framework calculates similarities between software components from different level. From an implementation point of view, some of these similarity calculation is provided by external frameworks. For example, the structural similarity is calculated with the framework developed in [Keh15] and the semantical similarity calculation is provided by [Thi15]. The implementation details regarding the similarity calculation is discussed in section 5.8.3. In the context FEV GmbH, some of SimA's data read/write operations are based on the scripts provided by company itself. Therefore, SimA has to work with data generated by different frameworks and scripts. At the same time, external frameworks can only work with data models defined by themselves. Directly manipulating these data makes SimA dependent on that specific code which is a direct violation of the requirement specified in section 3.2.

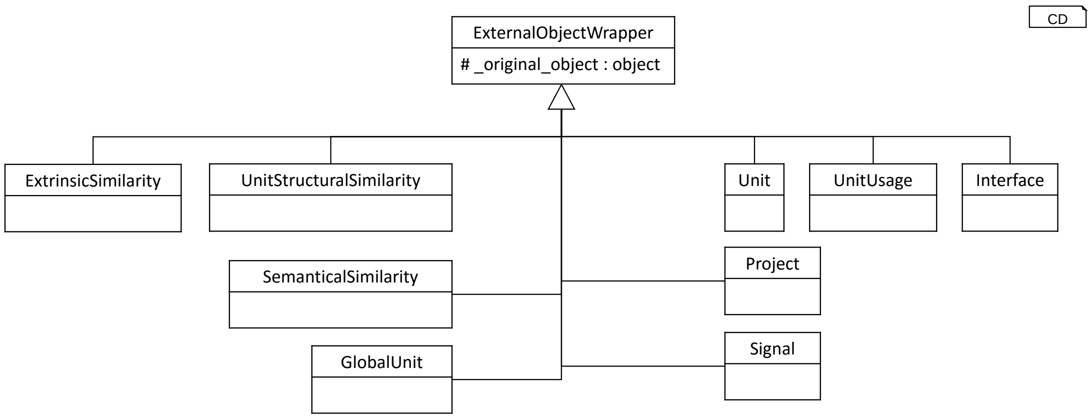


Figure 5.14: `ExternalObjectWrapper` inheritance.

For a seamless communication with external frameworks and keeping SimA's internal data manipulation consistent, `ExternalObjectWrapper` abstract model has been developed. The `_original_object` attribute of this model holds information read or received from the external frameworks. As shown in figure 5.14, `ExternalObjectWrapper` is extended by different data models. These data are read from or generated by different external frameworks. The child models has their own attributes which are used for SimA's internal computation. The `_original_object` is used during communication with the external frameworks.

5.8.1.4 Data models for reports

Section 4.1.8 discusses about different kind of reports the current implementation of SimA framework can generate. Besides similarity analysis, SimA can performs other miscellaneous operations, for example global component list generation (sec. 4.2), which helps producing more meaningful and information rich reports. This section discusses about the data models which are generated by these operations and are illustrated in figure 5.15.

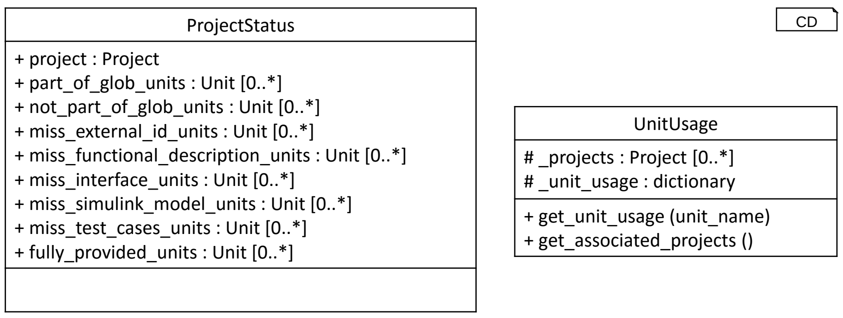

ProjectStatus
+ project : Project
+ part_of_glob_units : Unit [0..*]
+ not_part_of_glob_units : Unit [0..*]
+ miss_external_id_units : Unit [0..*]
+ miss_functional_description_units : Unit [0..*]
+ miss_interface_units : Unit [0..*]
+ miss_simulink_model_units : Unit [0..*]
+ miss_test_cases_units : Unit [0..*]
+ fully_provided_units : Unit [0..*]
CD
UnitUsage
# _projects : Project [0..*]
# _unit_usage : dictionary
+ get_unit_usage (unit_name)
+ get_associated_projects ()

Figure 5.15: Reporting specific models.

UnitUsage - This model holds information about different `Projects` which uses the same `Unit`. The `_unit_usage` attribute is a `dictionary` that maps an `Unit` to a list of `Project` which the `Unit` is part of. The access to this attribute is restricted and information from it is available via the function `get_unit_usage`. It takes a `Unit's` name and return the `Projects` using it.

ProjectStatus - Different software components in various projects are often missing information. `ProjectStatus` represents the quality of a project in terms of information availability by listing which `Units` are missing which information. For example, the `miss_interface_units` attribute lists those `Units` which are missing interface information. The `fully_provided_units` lists `Units` which contains all necessary information. For the corresponding project, it also holds information regarding which units are part of the global component list and which are not.

5.8.2 SimA data read and write

SimA reads different kind of data from various sources before performing similarity or other computations. Computed results are sometimes exported as files. SimA implements some of the read/write functionalities. Some of these functionalities are provided by external frameworks and SimA directly uses them. SimA defines interfaces as abstract classes for reading and writing data. Different implementations of these interfaces provide different functionalities. The abstract classes not only specify a defined structure to read/write data but also creates a layer of abstraction which hides the implementation details from usage. This section discusses the data read/write interfaces and their implementations which has been used in different stages of the SimA workflow.

5.8.2.1 Remote file reader

A remote file reader reads information about files or folders which are hosted in a remote repository (e.g. SVN). In the context of FEV GmbH, information necessary for similarity calculation are often not locally available and has to be checked out from remote official SVN repositories before performing any further operation.

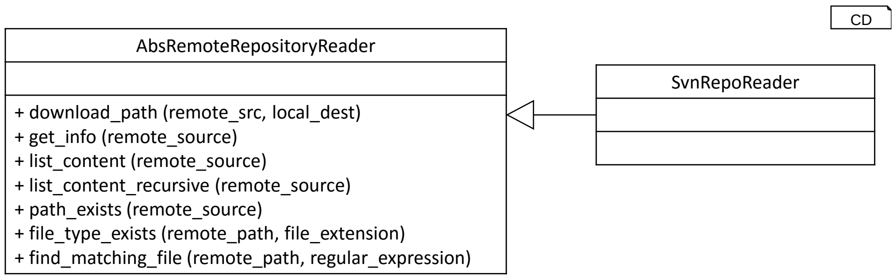

CD
AbsRemoteRepositoryReader
+ download_path (remote_src, local_dest)
+ get_info (remote_source)
+ list_content (remote_source)
+ list_content_recursive (remote_source)
+ path_exists (remote_source)
+ file_type_exists (remote_path, file_extension)
+ find_matching_file (remote_path, regular_expression)
SvnRepoReader

Figure 5.16: Remote file reader interface and implementation.

`AbsRemoteRepositoryReader` (fig. 5.16) defines an interface for performing readonly operations on a remote file or folder. The interface provides various functions which can perform different tasks. For example, `download_path` downloads a remote folder to a specified local path, `path_exists` checks the availability of a remote file or folder, `find_matching_file` searches for files having specified name pattern, `list_content` returns a list containing information about the contents of a remote folder and so on. Functions returning file or folder information return them in the form of `RepositoryItem` (sec. 5.8.1.1) object.

`SvnRepoReader` implements the `AbsRemoteRepositoryReader` interface and provides previously mentioned functionalities for an SVN repository. For executing all SVN commands it is dependent on the *svn* python library specified in section 5.2.

5.8.2.2 File readers

SimA defines generic interfaces for reading data regardless of source or format. Files which are locally available or checked out from a remote repository are read with different file readers. And these file readers are implemented based on defined data reader interfaces. This section discusses about the data reader interfaces and their file based implementations.

Data reader interfaces - SimA defines two abstract classes namely `AbsDataReader` and `AbsProjectDataReader`. These classes create a layer of abstraction for reading data. `AbsDataReader` is designed to be the base class for all data reader classes. The `read_data` method reads the data and `convert_data` method is used for data conversion if necessary.

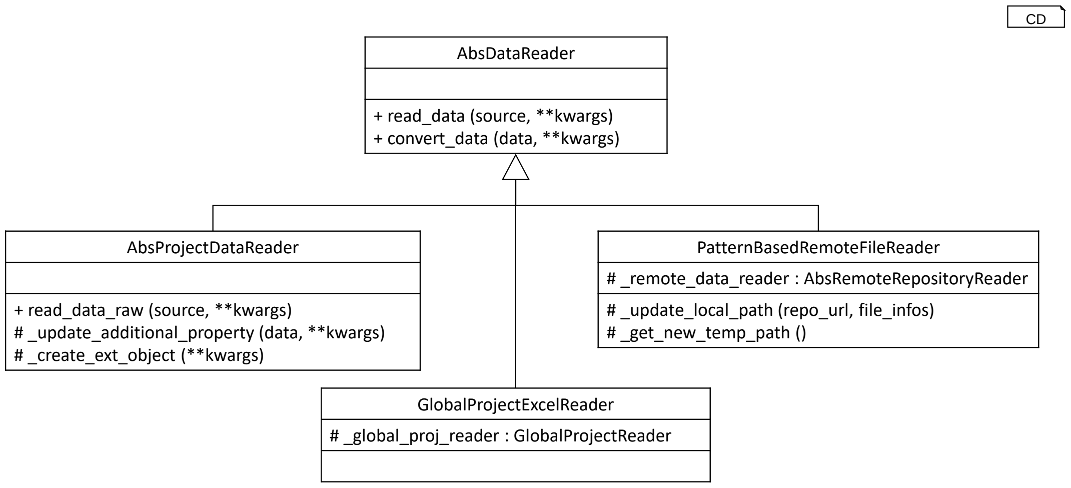

CD
AbsDataReader
+ read_data (source, **kwargs)
+ convert_data (data, **kwargs)
AbsProjectDataReader
+ read_data_raw (source, **kwargs)
# _update_additional_property (data, **kwargs)
# _create_ext_object (**kwargs)
PatternBasedRemoteFileReader
# _remote_data_reader : AbsRemoteRepositoryReader
# _update_local_path (repo_url, file_infos)
# _get_new_temp_path ()
GlobalProjectExcelReader
# _global_proj_reader : GlobalProjectReader

Figure 5.17: Data reader interfaces and implementations.

As the class diagram in figure 5.17 shows, the `AbsProjectDataReader` extends from `AbsDataReader` and defines another abstraction for reading `Project` and its related attributes. It provides an implementation for `read_data` and defines three more functions. The newly defined `read_data_raw` reads specific kind of data and returns it without any change. Now, it might be the case that different properties of the already read data are available at different sources. For example, in the context of FEV GmbH, an `Unit` and it's related `Interface` is defined separately. To take care of this kind of situations, `_update_additional_property` method is used which, based on the implementation, reads and updates properties of read data. The function `_create_ext_object` is used for very special cases. Section 5.8.1.3 discuses about how data from external frameworks are preserved in their original form and used when interacting with corresponding frameworks. In certain cases when the external objects can not be preserved in this way, the function `_create_ext_object` creates a dummy external object based on the original data. This way the communication with external frameworks are kept consistent. The implementation for `read_data` is a template method (sec. 2.8.3). It reads the raw data and, based on the use case, performs conversion and/or additional property update. Finally, it returns an object which is part of SimA data model (sec. 5.8.1).

Data reader implementations - Data reader interfaces, described in previous section, are implemented by different classes where each class reads one particular kind of data. These implementations are discussed in this section.

PatternBasedRemoteFileReader - This class searches files based on regular expression and retrieves information about those files from a remote repository. It extends

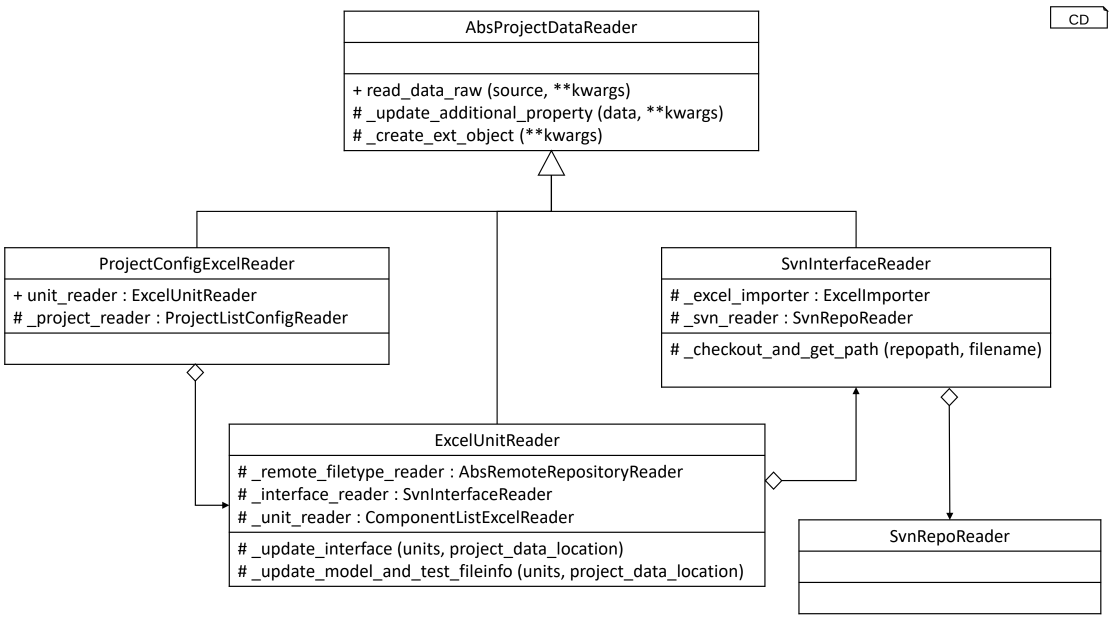

CD
AbsProjectDataReader
+ read_data_raw (source, **kwargs)
# _update_additional_property (data, **kwargs)
# _create_ext_object (**kwargs)
ProjectConfigExcelReader
+ unit_reader : ExcelUnitReader
# _project_reader : ProjectListConfigReader
SvnInterfaceReader
# _excel_importer : ExcelImporter
# _svn_reader : SvnRepoReader
# _checkout_and_get_path (repopath, filename)
ExcelUnitReader
# _remote_filetype_reader : AbsRemoteRepositoryReader
# _interface_reader : SvnInterfaceReader
# _unit_reader : ComponentListExcelReader
# _update_interface (units, project_data_location)
# _update_model_and_test_fileinfo (units, project_data_location)
SvnRepoReader

Figure 5.18: Project related data readers.

from `AbsDataReader` and uses any class of type `AbsRemoteRepositoryReader` to retrieve information from remote repository. The `read_data` function takes the data source path and a list of file name patterns. It matches file names with provided pattern and keeps track of the matched files. For each matched files `similarity_analysis.models.RepositoryItem` object is created. At this point the matched files are downloaded to a cache directory and the local path of those files are used to update the `local_path` attribute of the corresponding `RepositoryItem`. Finally, it returns a dictionary where keys are the pattern strings and the values are list of `RepositoryItem` which matches the corresponding pattern string.

ProjectConfigExcelReader - This class extends the `AbsProjectDataReader` and returns primary information for different software projects. In the context of FEV GmbH, these information are listed in an excel file. Each row contains information like - name, project type, SVN repository location etc for one specific project. The `ProjectConfigExcelReader` reads this file form the provided local path and returns a list of `similarity_analysis.data.models.Project`.

The `ProjectListConfigReader` class (sec. 5.4.2), which is provided by the `extrinsic_similarity` package, is used for reading this data. It reads this excel file and returns a list of `extrinsic_similarity.models.Project`. Each item of this list is converted into `similar_analysis.data.models.Project` object. While converting, the original object is stored as an attribute of the converted object (sec. 5.8.1.3) which is later used to communicate with the `extrinsic_similarity` package. At this point, for each project `ExcelUnitReader` is called which returns a list of `similarity_analysis.data.models.Unit` for the corresponding project. This list is assigned to the project's `units` attribute and finally, a list containing information about all the listed projects are returned.

ExcelUnitReader - It extends from `AbsProjectDataReader` and reads the software component definitions from provided location. In the context of FEV GmbH, for each project these definitions are stored in an excel file. Each row of the excel file represents one software component and contains information like - name, composition levels, implementation location, interface definition information etc. The `extrinsic_similarity` package already implements the reading of this data with `ComponentListExcelReader` class (sec. 5.4.2) and is used to read the actual data. It returns a list of `extrinsic_similarity.models.Unit` which is converted to a list of `similarity_analysis.data.models.Unit`. During the conversion, the original objects are preserved as explained in section 5.8.1.3.

An `Unit` defines an interface and in ideal cases, contains one or more test cases and simulink models. In the context of FEV GmbH, these data are stored in a project specific SVN repository. At this point the `ExcelUnitReader` calls other classes to get these data and updates corresponding attribute of the `Unit` object with them. The `SvnInterfaceReader` is used to read interface definitions from the SVN repository and the `PatternBasedRemoteFileReader` for getting information about the test cases and simulink models. After updating related attributes regarding these data, the list of `Unit` is returned.

SvnInterfaceReader - This class extends from `AbsProjectDataReader` and reads interface files from remote SVN repository. In the context of FEV GmbH, interface definitions are stored as an excel file. Ideally, each of these files has five sheets - *in, out, cal, mp* and *fix*. The name of the sheet identifies the type of signals it lists. For example, an `in` sheet contains all `in` signals and so on. These files are stored in the SVN repository of the project it belongs to.

The `SvnInterfaceReader` checks out the interface file, reads it and returns a `similarity_analysis.data.models.Interface` object. The SVN related operations are performed with `SvnRepoReader` (sec. 5.8.2.1). `ExcelImporter` class provided by `struct_analysis` package is used for reading the checked out interface files. Returned `struct_analysis.model.units.UnitInterface` object is converted to `similarity_analysis.data.models.Interface`. The containing list of `struct_analysis.model.units.Signal` are also converted to a list of `similarity_analysis.data.models.Signal` and assigned to the `signals` attribute of the interface object. The original signal and interface objects are preserved for further communication (sec. 5.8.1.3) with the `struct_analysis` package.

GlobalProjectExcelReader - The idea behind global component list generation is explained in the section 4.2. In the context of FEV GmbH, this list is preserved in the form of an excel file. `GlobalProjectExcelReader` reads this excel file and returns a list of `similarity_analysis.data.models.GlobalUnit` (sec. 5.8.1.4). It extends `AbsDataReader` and uses the `GlobalProjectReader` class (sec. 5.4.2), provided by `extrinsic_similarity` package, for the data read functionality. The data returned by `GlobalProjectReader` is converted to `GlobalUnit` and returned.

Data exporters - SimA defines `AbsExporter` abstract class for exporting data to some external destination. It is the base class for all data exporting implementations of

SimA framework. The `export_data` function exports the provided data to the specified destination.

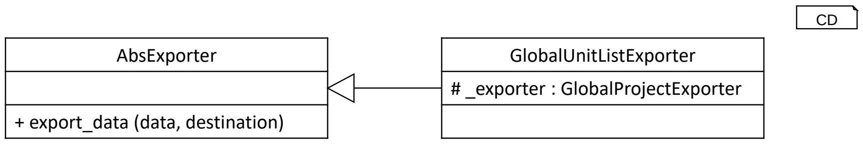

CD
AbsExporter
+ export_data (data, destination)
GlobalUnitListExporter
# _exporter : GlobalProjectExporter

Figure 5.19: Data exporters.

The `GlobalUnitListExporter` extends the `AbsExporter` and exports a global unit list to an excel file. It takes a list of `similarity_analysis.data.models.Projects`. Each of these `Project` object has an `_original_object` (sec. 5.8.1.3) attribute which is basically a `extrinsic_similarity.models.Project` object. A list from all the `_original_object` is created which is used to create a global unit list using the class `UnitEvaluator` (sec. 5.4.2) provided by `extrinsic_similarity` package. At this point, the `GlobalProjectExporter` (sec. 5.4.2) class of the `extrinsic_similarity` package is used to export the global unit list to the destination excel file.

5.8.3 Similarity calculation

The SimA framework computes extrinsic, structural and semantical similarity among software components from different projects (sec. 4.1). The `extrinsic_similarity` (sec. 5.4), `struct_analysis` (sec. 5.5) and `test-based-validation-tool` (sec. 5.6) packages respectively provides functionality to calculate these similarities. The SimA framework makes use of these functionalities provided by different packages and puts them together (algorithm in sec. 4.1) to generate a `OverallSimilarity` which relates the separately generated similarity results from all the three different levels.

The SimA framework interacts with these different packages with facades. A facade, as described in section 2.8.4, is a class that receives services from different external entities and exposes a simple unified interface to interact with them. SimA defines a facade abstract class `AbsSimilarityFacade` which is the base class for all similarity calculation classes (fig. 5.20). The `find_all_similarities` is a template function (design pattern explained in sec. 2.8.3) which defines the workflow for similarity calculation.

Initially, the actual similarity is calculated with the `_find_similarities_impl` which is an abstract function and, much like the strategy pattern explained in 2.8.5, different implementation determines the type of similarity to be calculated. This function takes a list of objects to calculate similarity for, performs calculation for each pair from the list and returns a list of `ExternalSimilarityResultWrapper`. This object encapsulates the external similarity calculation result along with any other related information which is necessary for converting the result to SimA's internal similarity representations (related data models explained in sec. 5.8.1.2). At this point, each of the similarity results are converted with `convert_similarity_result` which is another abstract function and the conversion algorithm is determined by the implementation. In the following sections three different implementations of the `AbsSimilarityFacade` are explained.

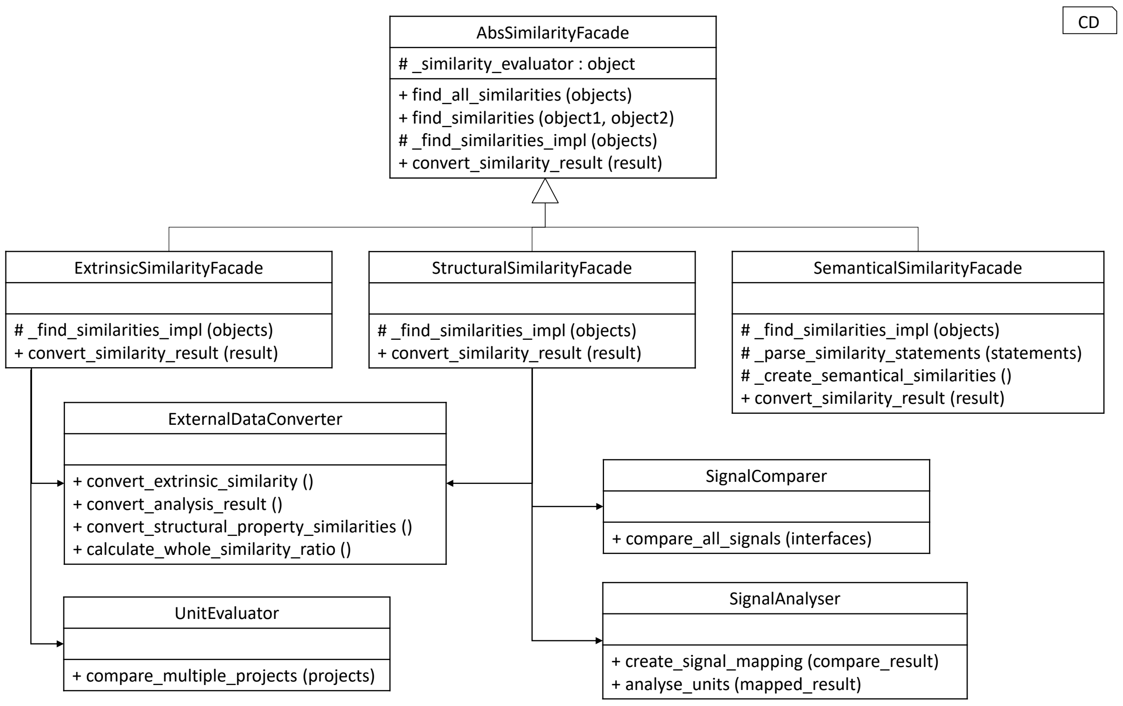

CD
AbsSimilarityFacade
# _similarity_evaluator : object
+ find_all_similarities (objects)
+ find_similarities (object1, object2)
# _find_similarities_impl (objects)
+ convert_similarity_result (result)
ExtrinsicSimilarityFacade
# _find_similarities_impl (objects)
+ convert_similarity_result (result)
StructuralSimilarityFacade
# _find_similarities_impl (objects)
+ convert_similarity_result (result)
SemanticalSimilarityFacade
# _find_similarities_impl (objects)
# _parse_similarity_statements (statements)
# _create_semantical_similarities ()
+ convert_similarity_result (result)
ExternalDataConverter
+ convert_extrinsic_similarity ()
+ convert_analysis_result ()
+ convert_structural_property_similarities ()
+ calculate_whole_similarity_ratio ()
SignalComparer
+ compare_all_signals (interfaces)
UnitEvaluator
+ compare_multiple_projects (projects)
SignalAnalyser
+ create_signal_mapping (compare_result)
+ analyse_units (mapped_result)

Figure 5.20: Similarity calculation facades.

5.8.3.1 ExtrinsicSimilarityFacade

According to the algorithm discussed in section 4.1.2, the extrinsic similarity is calculated among software components from different projects. `ExtrinsicSimilarityFacade` is the implementation of `AbsSimilarityFacade` which is responsible to calculate extrinsical similarity (fig. 5.20). Under the hood, the `UnitEvaluator` class which is a part of the package `extrinsic_similarity` is used to calculate the extrinsic similarity.

The `ExtrinsicSimilarityFacade` gets a list of `Project` and uses `UnitEvaluator` to perform the extrinsic similarity. Since the class `UnitEvaluator` is part of the `extrinsic_similarity` package, it can only work with data models which are defined in the corresponding package (sec. 5.4.1). In this particular case, the `Project` object defined in `extrinsic_similarity` package. And this object is preserved as the `_original_object` attribute of SimA's `Project` object (external object handling explained in sec. 5.8.1.3). The `_original_object` attribute was updated when the projects were read (sec. 5.8.2.2). Now, a list is created out of all `_original_object` and then it is passed to `UnitEvaluator` for extrinsic similarity calculation.

The function `compare_multiple_projects` (details in sec. 5.4.2) of `UnitEvaluator` class takes a list of projects as parameter and finds units which are extrinsically similar. It generates a `SimilarityResult` (sec. 5.4.1) and returns it. This similarity result object is not part of the SimA data model. Therefore, it is passed to the `convert_similarity_result` function of `ExtrinsicSimilarityFacade` which converts it to a list of `ExtrinsicSimilarity` object.

The `convert_extrinsic_similarity` function of `ExternalDataConverter` class is used to convert `SimilarityResult`. Each `SimilarityResultEntry` (sec. 5.4.1) contained in SimilarityResult represents the usage of one unit in different projects. The convert function processes only those SimilarityResultEntry which has multiple usages and creates an ExtrinsicSimilarity object from it. And finally, a list of ExtrinsicSimilarity is returned.

5.8.3.2 StructuralSimilarityFacade

The structural similarity calculation functionality is provided by the struct_analysis package. The StructuralSimilarityFacade, a subclass of AbsSimilarityFacade, works as a facade over this package and uses it to calculate structural similarity. It receives a list of Unit, calculates structural similarity for each pair of units from the list and, finally, a list of UnitStructuralSimilarity is returned which represents the calculated similarity results.

The structural similarity is calculated based on the unit's interface definition. This definition is represented with the Interface data model (sec. 5.8.1.1) and is contained in interface attribute of the corresponding Unit. However, the struct_analysis package can only work with its internal representation of interface definition. And this data is contained in the _original_object (sec. 5.8.1.3) attribute of the Interface object. Therefore, a list out of the _original_object attribute of all Interface objects is created and used for further calculation.

As discussed in 4.1.4, the structural similarity calculation works in several steps. In the first step, all signals from both interfaces are matched using the compare_all_signals function of SignalComparer class (sec. 5.5.2). It returns a CompareResult which contains a similarity value for all possible pair of signals from both interfaces. At this point, the result is analysed to create a signal mapping using the create_signal_mapping function of SignalAnalyser class which returns a MappedCompareResult. This result lists the similar signals from both interfaces, their directed overall similarity value and directed similarity value for each attributes. The overall similarity between units themselves are calculated with the analyse_units function of SignalAnalyser class. It returns the percentage of similar or non-similar signals between both units and an overall percentage is calculated based on it. At this point, the MappedCompareResult and overall unit similarity percentages are sent to the convert_similarity_result function which returns a UnitStructuralSimilarity object for each pair of Unit.

The convert_analysis_result function of ExternalDataConverter class is used to convert the structural analysis results. First, a UnitStructuralSimilarity object is created to represent the similarity between two Unit and the overall unit based similarity percentage is added to it. At this point, a SignalStructuralSimilarity is created for each similar pair of signals which represents the similarity between them. Each SignalStructuralSimilarity contains multiple AttributeSimilarity objects which represents the similarity of the attributes for corresponding signal pair. This AttributeSimilarity objects are created based on the MappedCompareResult. Finally, the UnitStructuralSimilarity object is returned after all the signal and attribute similarities have been converted and added.

5.8.3.3 SemanticalSimilarityFacade

The semantical similarity is calculated with the `test-based-validation-tool` package. The `SemanticalSimilarityFacade` extends from `AbsSimilarityFacade` and works as a facade (sec. 2.8.4) for the mentioned package. As explained in section 5.6, this package is developed in java and section 4.1.6 explains the semantical similarity calculation procedure. Therefore, a direct communication from python is not possible. Another package `validation_tool_proxy` has been developed in java which receives data from python as a json string, converts it to appropriate java object and calls the `test-based-validation-tool` with them. The results are again converted to json string and returned to python interface. In figure 5.21, the classes in the dotted box belong to the `validation_tool_proxy` package. It shows how the communication between `SemanticalSimilarityFacade` and `test-based-validation-tool` package is maintained.

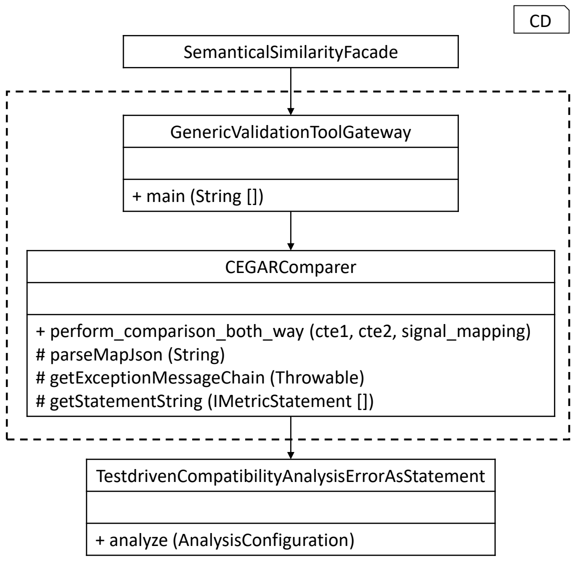

CD
SemanticalSimilarityFacade
GenericValidationToolGateway
+ main (String [])
CEGARComparer
+ perform_comparison_both_way (cte1, cte2, signal_mapping)
# parseMapJson (String)
# getExceptionMessageChain (Throwable)
# getStatementString (IMetricStatement [])
TestdrivenCompatibilityAnalysisErrorAsStatement
+ analyze (AnalysisConfiguration)

Figure 5.21: The `validation_tool_proxy` package

An executable jar, including all dependencies, is created from `validation_tool_proxy` to execute it. As shown in the figure 5.21, the `GenericValidaitonToolGateway` contains a `main` method and it is the entry point of the created jar. When the jar is executed, this `main` method is called first. Informations like path to the test files, signal mapping etc are provided as command line argument while executing the jar. The `main` method checks if appropriate number of arguments have been provided and calls `perform_calculation_both_way` of `CEGARComparer` class with all the received parameters. At this point, the `TestDrivenCompatibilityAnalysisErrorAsStatement` (sec. 5.6) is called for analysing the semantical similarity. It performs necessary computations and returns a list of `IMetricStatement` (sec. 5.6). This statement is converted to list of string and returned. All codes in`CEGARComparer` class is carefully monitored for exceptions. In case of any thrown exception, it is caught, messages are extracted and added to the result list of string.

Now, the main method receives a list of string which contains all necessary information regarding the executed semantical analysis. This list is converted to json string and returned to SemanticalSimilarityFacade. The results are converted with the convert_similarity_result function which parses them and extracts semantical similarity percentage for matched pair of signals. Signal pairs having no similarity information available are considered to have None similarity. A SemanticalSimilarity (sec. 5.8.1.2) is created for each mapped signal pair with found similarity information and any available error messages. Finally, a list of SemanticalSimilarity is returned.

5.8.4 Report generation

SimA generates viewer friendly reports to present the analysed similarity information. In the section 4.1.8, the concept behind generation of reports and their format has been discussed. Based on this concept the report generation has been implemented.

The reports are generated as HTML documents which is highly portable, capable to generate interactive document and flexible. The isddgf package (sec. 5.7) has been chosen for generating these reports which provides python api for HTML document generation.

5.8.4.1 Extending **isddgf**

Certain part of the isddgf package has been extended by SimA to match its necessity and easier implementation. Two classes in particular has been extended as shown in figure 5.22. They are BasicReport and AbstractTableModel (more details about them in sec. 5.7).

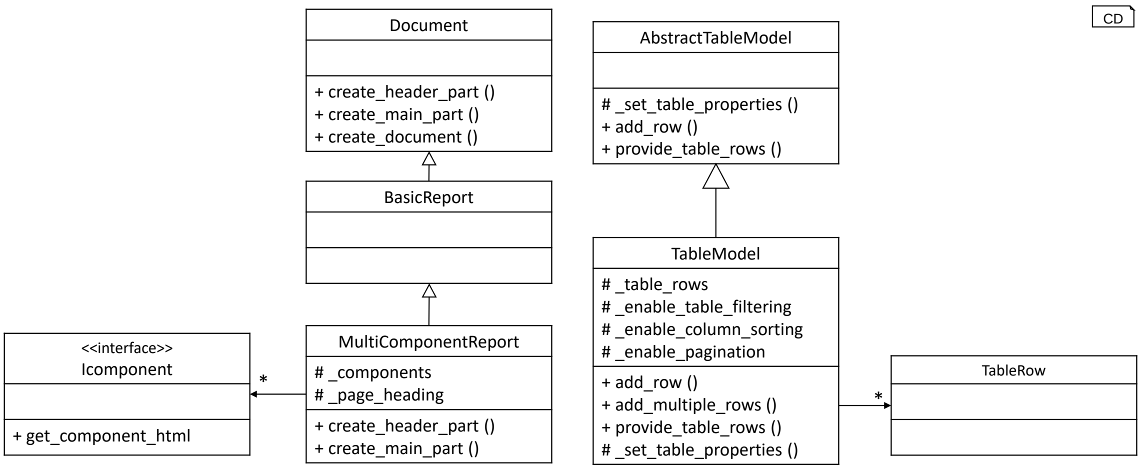


Figure 5.22: Extended classes from isddgf

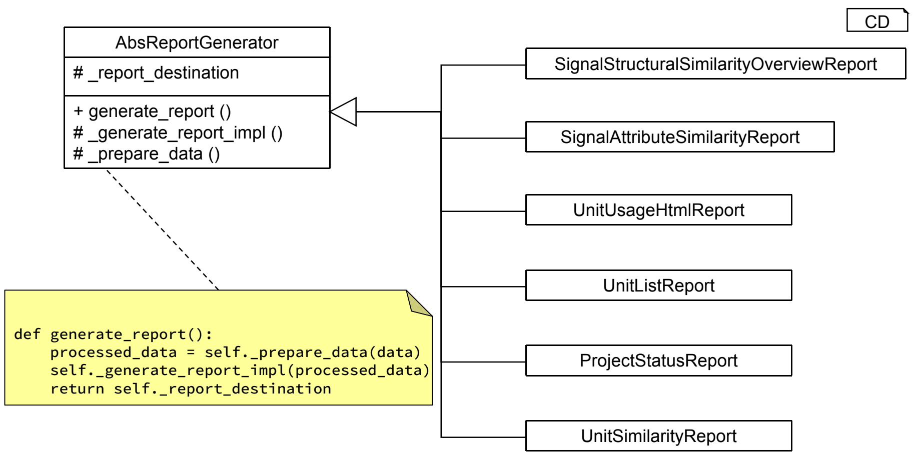
The MultiComponentReport extends BasicReport and provides a base layout for all reports generated by SimA. It holds a list of IComponent and the create_main_part function puts these components into the document. The components are placed vertically one after another in the same sequence as they are in the list. The function create_header_part creates the top part of the HTML document with the FEV and software engineering chair's[7] logo.

[7] www.se-rwth.de

The TableModel extends from AbstractTableModel and defines a generic configuration for all tables. It holds a list of TableRow and the add_multiple_rows function is added to the class for adding multiple rows easily. The TableModel class holds boolean values for enabling or disabling sorting, filtering and pagination feature for the corresponding instance.

5.8.4.2 Report generation abstraction

SimA defines AbsReportGenerator abstract class which is the base class for all classes that generate reports. The figure 5.23 shows the functions provided by this class and other classes extending it. The generate_report is a template function (design pattern explained in sec. 2.8.3) defining the workflow for report generation from an abstract level. First, it processes the data and then generates the reports based on the processed data. The functionalities regarding the processing of data and report generation is defined by the implementations.



```
def generate_report():
    processed_data = self._prepare_data(data)
    self._generate_report_impl(processed_data)
    return self._report_destination
```

Figure 5.23: Report generation base and child classes.

5.8.4.3 Report generation implementations

In section 4.1.8, the concept behind generation of two different kind of reports have been discussed and the implementation follows this concept. The ProjectStatusReport class generates a status report for different projects which is explained in section 4.1.8.1. The report presenting similarity results, discussed in section 4.1.8.2, is generated by UnitUsageHtmlReport class.

ProjectStatusReport receives a list of ProjectStatus (data model discusses in sec. 5.8.1.4) and creates HTML table report based on them. In the ProjectStatus object units of a project are grouped into different lists based on certain criteria. For example, units which are part of global component list are put into the part_of_glob_units list. During the report generation a row is created for each ProjectStatus object. Each cell of the row shows the count and percentage of units for a particular criteria. Each of these rows also contains a hyperlink to a unit list table which lists the units satisfying the corresponding criteria. The UnitListReport is used to create this unit list table. It returns the path where the report was created and this path is used to create the hyperlink.

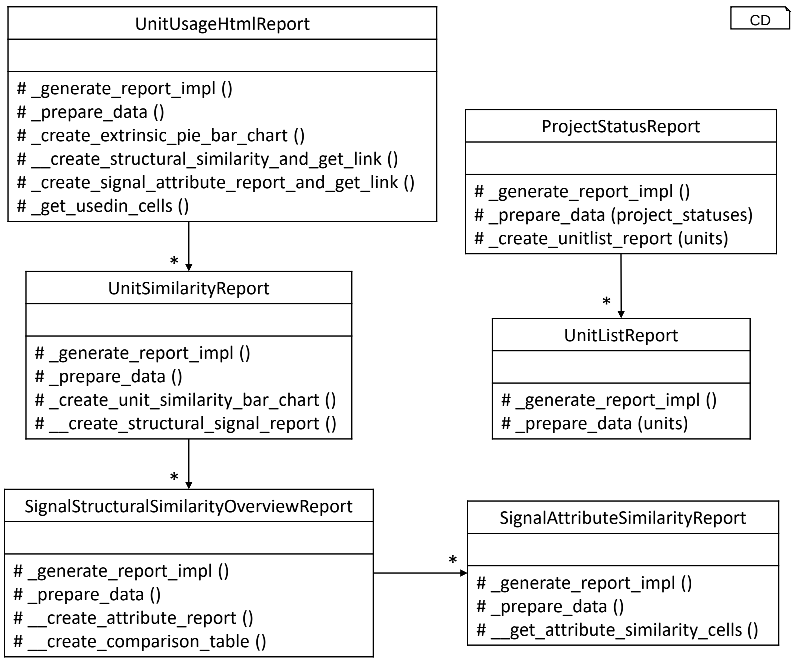

CD
UnitUsageHtmlReport
# _generate_report_impl ()
# _prepare_data ()
# _create_extrinsic_pie_bar_chart ()
# __create_structural_similarity_and_get_link ()
# _create_signal_attribute_report_and_get_link ()
# _get_usedin_cells ()
ProjectStatusReport
# _generate_report_impl ()
# _prepare_data (project_statuses)
# _create_unitlist_report (units)
UnitSimilarityReport
# _generate_report_impl ()
# _prepare_data ()
# _create_unit_similarity_bar_chart ()
# __create_structural_signal_report ()
UnitListReport
# _generate_report_impl ()
# _prepare_data (units)
SignalStructuralSimilarityOverviewReport
# _generate_report_impl ()
# _prepare_data ()
# __create_attribute_report ()
# __create_comparison_table ()
SignalAttributeSimilarityReport
# _generate_report_impl ()
# _prepare_data ()
# __get_attribute_similarity_cells ()
*
*
*
*

Figure 5.24: Classes generating reports and their dependencies.

The UnitUsageHtmlReport generates an overview similarity report which is explained in section 4.1.8.2. It receives an UnitUsage and OverallSimilarity object to generate the report. The parts of the table holding information about the units and project usage is created based on the data contained in UnitUsage. The extrinsic, structural and semantical similarity information in the table is created based on the data contained in OverallSimilarity object.

As explained in section 4.1.8.2, the extrinsic similarity column also holds hyperlink to a structural similarity overview report for the corresponding extrinsically similar units. This report is created with UnitSimilarityReport class. It receives the corresponding ExtrinsicSimilarity object and the OverallSimilarity object to generate this report. A path to the generated report is returned which is used to create the hyperlink.

The structural similarity report generated by UnitSimilarityReport holds a hyperlink to the signal based structural similarity report which is generated with the class SignalStructuralSimilarityOverviewReport. It creates a separate table for each type of signals (structure in fig. 4.8) and the similarity information in the report is generated based on corresponding UnitStructuralSimilarity object. For each pair of similar signals links are created to attribute based structural similarity report. These attribute similarity reports are created with SignalAttributeSimilarityReport class. The AttributeSimilarity list contained in corresponding SignalSimilarity is used to create this report. The OverallSimilarity object is used to find any associated semantical similarity information.

5.8.5 SimA workflow controls

Similarity calculation and report generation are two of the main activity of the SimA framework. Each of these activities consists of a very defined set of steps which has to be performed sequentially. To ensure this sequential execution for current or future implementation, SimA defines two workflow control abstract classes (fig. 5.25). These classes defines the workflow with a template method (sec. 2.8.3) and rest of their behaviour is abstracted. Here both of them and their implementation in current context has been discussed.

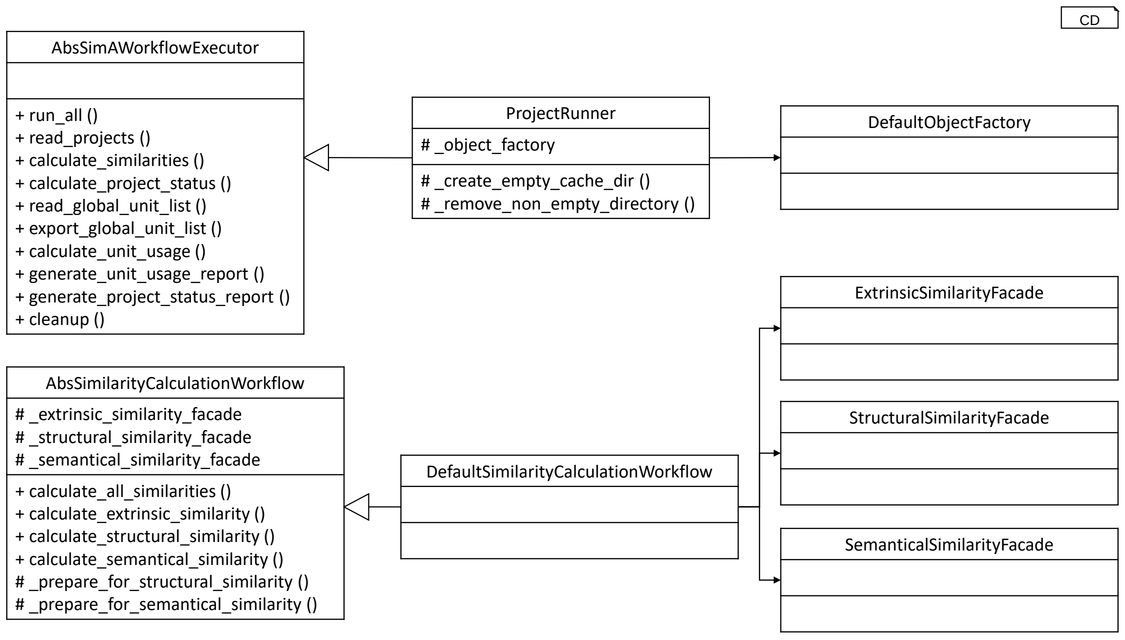

CD
AbsSimAWorkflowExecutor
+ run_all ()
+ read_projects ()
+ calculate_similarities ()
+ calculate_project_status ()
+ read_global_unit_list ()
+ export_global_unit_list ()
+ calculate_unit_usage ()
+ generate_unit_usage_report ()
+ generate_project_status_report ()
+ cleanup ()
ProjectRunner
# _object_factory
# _create_empty_cache_dir ()
# _remove_non_empty_directory ()
DefaultObjectFactory
AbsSimilarityCalculationWorkflow
# _extrinsic_similarity_facade
# _structural_similarity_facade
# _semantical_similarity_facade
+ calculate_all_similarities ()
+ calculate_extrinsic_similarity ()
+ calculate_structural_similarity ()
+ calculate_semantical_similarity ()
# _prepare_for_structural_similarity ()
# _prepare_for_semantical_similarity ()
DefaultSimilarityCalculationWorkflow
ExtrinsicSimilarityFacade
StructuralSimilarityFacade
SemanticalSimilarityFacade

Figure 5.25: Workflow controller classes.

Similarity calculation - The abstract class `AbsSimilarityCalculationWorkflow` defines this workflow. It has the `calculate_all_similarities` template function which controls the sequential execution of necessary steps from an abstract level. The constructor function takes three arguments and each of them are different implementations of the `AbsSimilarityFacade`.

The `DefaultSimilarityCalculationWorkflow` class provides concrete definitions for these abstract steps in the context of current SimA implementation. This class holds different instances of similarity calculation facades (described in sec. 5.8.3) and uses them for similarity calculation. The workflow receives a list of `Project` and using `ExtrinsicSimilarityFacade` performs extrinsical similarity analysis among them. `StructuralSimilarityFacade` is used to calculate structural similarity among the extrinsically similar units found in the previous step. In the next step `SemanticalSimilarityFacade` is used to calculate the semantical similarity. The signal mapping found in structural similarity calculation step is used here. Each calculation step is followed by a preparation step which provides window for any kind of change that might be necessary for the next step.

Finally, an `OverallSimilarity` result is created based on the results from all three similarity analysis steps. During the creation of this object, the correlation among different similarity results are calculated and preserved which is later used to understand how one similarity step lead to the next one.

Full workflow and report generation - The `AbsSimAWorkflowExecutor` controls the whole workflow starting from project related data read to report generation. It defines abstract methods for each of the steps. `ProjectRunner` extends from it and provides the implementation in the context of SimA. `ProjectRunner` has a constructor function that takes an `AbsObjectFactory` object as parameter and later uses it to get all necessary object dependencies.

The `run_all` function define and executes the whole workflow. Current implementation closely follows the workflow showed in figure 4.2 and discussed in section 4.1. First, it reads all project related data with the `read_projects` function. The following steps include similarity calculation and various report generation. The similarity calculation step basically executes the similarity calculation workflow discussed earlier. At this point, the global unit list is calculated and exported as an excel file. Finally, a cleanup step is executed with removes any temporary data (e.g. SVN cache) that has been generated during the workflow executing.

5.8.6 SimA object factory

The creation of various objects is abstracted with the `AbsObjectFactory` class. It provides methods for creating and getting different kind of objects. This technique is inspired by the abstract factory design pattern (sec. 2.8.2).

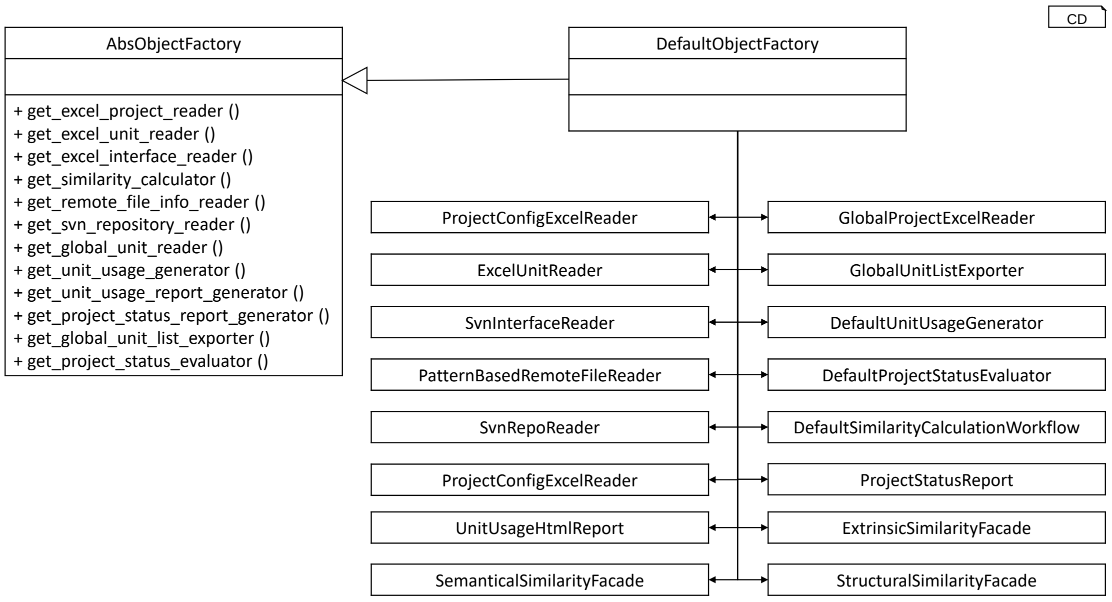

CD
AbsObjectFactory
+ get_excel_project_reader ()
+ get_excel_unit_reader ()
+ get_excel_interface_reader ()
+ get_similarity_calculator ()
+ get_remote_file_info_reader ()
+ get_svn_repository_reader ()
+ get_global_unit_reader ()
+ get_unit_usage_generator ()
+ get_unit_usage_report_generator ()
+ get_project_status_report_generator ()
+ get_global_unit_list_exporter ()
+ get_project_status_evaluator ()
DefaultObjectFactory
ProjectConfigExcelReader
GlobalProjectExcelReader
ExcelUnitReader
GlobalUnitListExporter
SvnInterfaceReader
DefaultUnitUsageGenerator
PatternBasedRemoteFileReader
DefaultProjectStatusEvaluator
SvnRepoReader
DefaultSimilarityCalculationWorkflow
ProjectConfigExcelReader
ProjectStatusReport
UnitUsageHtmlReport
ExtrinsicSimilarityFacade
SemanticalSimilarityFacade
StructuralSimilarityFacade

Figure 5.26: Object factory implementation.

`DefaultObjectFactory` extends from `AbsObjectFactory` and provides object creation service in the context of current SimA implementation. As SimA highly relies on dependency injection (sec. 2.8.1) pattern, various classes resolves dependency with their constructor function. `DefaultObjectFactory` creates these dependency objects and injects them to corresponding dependent classes in order to instantiate them. The constructor function of this factory class has a long list of parameters which are used to instantiate classes that need external parameter. The class diagram in the figure 5.26 shows different methods provided by the factory class and its dependency to various classes.

5.9 The `similarity_client` package

As explained before, the SimA framework was developed in the context of FEV GmbH. After developing the framework, it was added to the company's nightly build chain. As a result, new reports are generated with each nightly build providing updated reports based on the latest data from SVN repository. The `similarity_client` package works as an entry point for executing the whole workflow stating from reading data to report generation.

This package contains only two modules - `Main` and `constants`. The `constants` module holds various static values which are specific for FEV and necessary to create different objects. These values also include path to other packages (e.g. `extrinsic_similarity`) upon which the SimA implementation is dependent.

The `Main` module starts the whole workflow when it is executed as a python script. It receives two command line arguments - the local path to the PERSIST folder and path to the directory where generated reports will be saved. In the context of FEV GmbH, PERSIST is the name of an SVN repository which holds files and scripts necessary for SimA. Some of these information include the project list excel file, unit list for each individual project etc.

Once `Main` is executed, it gets dependent package paths from the `constants` module and adds them to the python path so that modules from other packages can be imported and used. Then, it instantiates `DefaultObjectFactory`. Constructor parameters needed to instantiate `DefaultObjectFactory` are taken from `constants`. Now, this object is used to instantiate `ProjectRunner` and, finally, call the `run_all` method to execute the whole workflow.

Chapter 6

Tutorial

The SimA framework exposes different interfaces for performing various tasks. It also provides implementation for those interfaces which have been implemented in the context of FEV GmbH. Chapter 5, discusses about these interfaces and implementations. This architecture makes the framework flexible and decoupled enough so that different component of the it can be switched with a similar one.

This chapter discusses how different part of the framework can be extended, changed or even replaced without affecting any other part. Executing current workflow implementation, installing dependencies are also in the scope of this chapter.

6.1 Resolve dependencies

As already discussed in chapter 5, the SimA framework is developed in Python 2.7. Therefore, it is the fist requirement that has to be resolved to run the framework. The Python 2.7 installer is available form the official website[1]. It must be downloaded and installed.

In the context of FEV GmbH, the company provides Matlab scripts to read certain kind of data (discussed in section 5.4.2). Current implementation of SimA has been tested with Matlab R2012b 32bit. Therefore, it has to be installed as well. Matlab is run as an automation server which has to be activated with the command shown in listing 6.1.

```
matlab -regserver
```

Listing 6.1: Activate matlab automation server.

A package dependency of SimA is `test-based-validation-tool` (details in sec. 5.3) which is developed in Java. Therefore, it has to be installed for running the semantical similarity calculation. Current implementation has been tested with Java 1.7. The corresponding installer is available from the official website[2].

The `SvnInterfaceReader` explained in section 5.8.2.2, checks out interface definition files from SVN repository of FEV. The `svn` python library is used for this purpose. But,

[1]`https://www.python.org/downloads/`

[2]`http://www.oracle.com/technetwork/java/javase/overview/index.html`

it is just a wrapper and SVN has to be installed in the system for this library to work properly. The graphical SVN tool TortoiseSVN is available at the official website[3].

The section 5.3 illustrates the packages and libraries the current implementation of SimA framework is dependent on. To be able to successfully run SimA, these dependencies has to be resolved. Packages discussed in section 5.3 are already bundles with the framework. Some packages implemented in other language (e.g. `test-based-validation-tool`) has to be properly treated (e.g. create `jar`) to make them executable. And, of course, different library dependencies has to be installed with related package manager (e.g. maven).

6.1.1 Installing Python dependencies

Python dependencies for current implementation of SimA are listed in section 5.2. These dependencies are installed with the python package manager pip (version 8.1.2)[4]. But, first pip itself has to be installed into the existing Python installation. The installation script `get-pip.py` is available from the official website[5]. Running it (listing 6.2) will install pip into the current python installation.

```
python get-pip.py
```

Listing 6.2: Install pip with `get-pip.py`.

The `dependencies.req` file, which can be found under the `codes` directory of this thesis's SVN repository, lists the python library dependencies in a format readable by pip. They can be installed simply by running the command shown in 6.3. This not only installs the listed libraries, but also their dependencies.

```
pip install -r dependencies.req
```

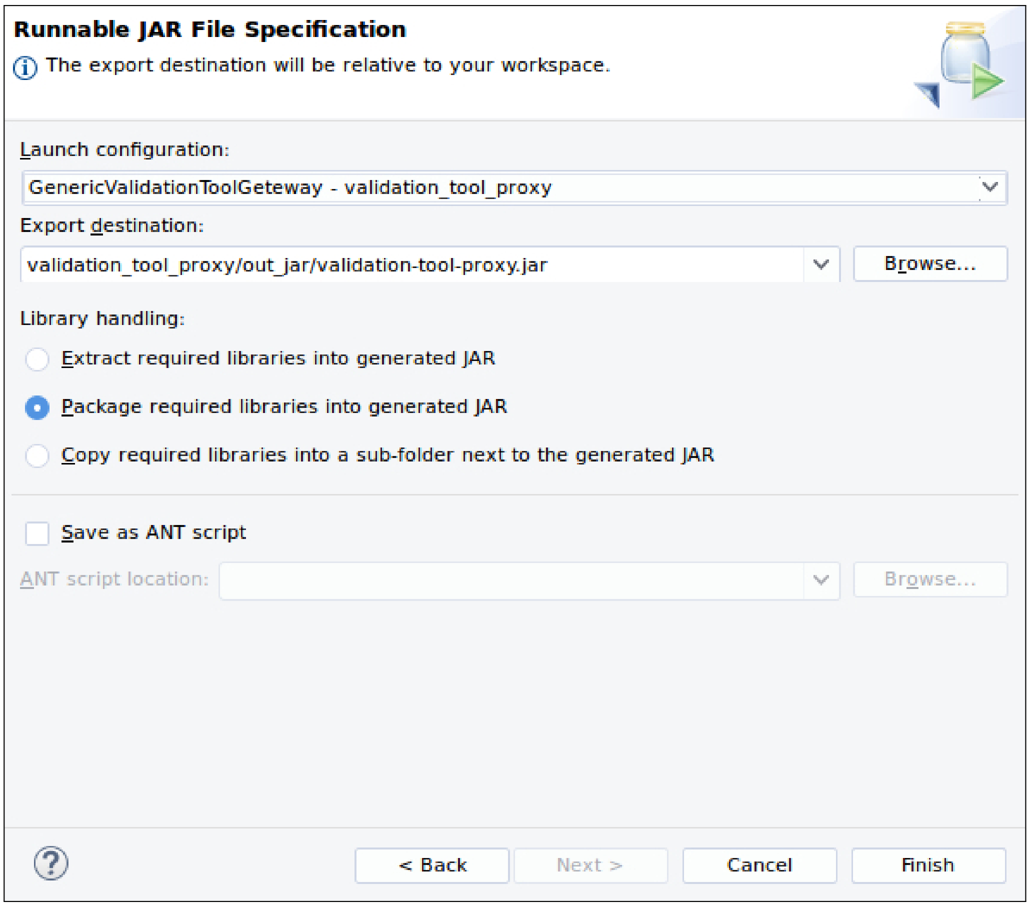
Listing 6.3: Install Python library dependencies.

6.1.2 Create Jar for `test-based-validation-tool`

Unlike SimA and other packages, the `test-based-validation-tool` is developed in Java. And as described in section 5.6, the communication between SimA and this package is mediated by another Java package `validation_tool_proxy` which works as a data conversion layer. However, to execute these projects they have to be compiled into executable file, particularly `jar` in this case. The export jar functionality of Eclipse[6] was used for this purpose.

The `test-based-validation-tool` defines all its dependencies in `pom.xml` file which is the standard practice for maven projects. These dependencies has to be installed using maven. The `validation_tool_proxy` is an Eclipse project which identifies the `test-based-validation-tool` as a build dependency. At this point, the `jar` file created with Eclipse bundles all the dependencies together creating one single executable file. In the figure 6.1, the Eclipse dialog for exporting `jar` has been shown. One important

[3]https://tortoisesvn.net/downloads.html
[4]https://pip.pypa.io/en/stable/
[5]https://pip.pypa.io/en/stable/installing/#installing-with-get-pip-py
[6]http://www.eclipse.org/downloads/packages/eclipse-ide-java-developers/mars2


Runnable JAR File Specification

The export destination will be relative to your workspace.

Launch configuration: GenericValidationToolGeteway - validation_tool_proxy

Export destination: validation_tool_proxy/out_jar/validation-tool-proxy.jar

Library handling:

- ( ) Extract required libraries into generated JAR
- (●) Package required libraries into generated JAR
- ( ) Copy required libraries into a sub-folder next to the generated JAR

[ ] Save as ANT script

ANT script location:

Figure 6.1: Export executable jar with Eclipse.

point should be noted that the `GenericValidationToolProxy` containing the `main` method (sec. 5.6) is chosen as the entry point. Therefore, this main method is executed when the `jar` is run.

This jar file is created at `validation_tool_proxy/out_jar` directory which contains necessary SMT-solver[7] executables used for semantical similarity calculation.

6.1.3 Checkout FEV's `PERSIST` repository

The current implementation of SimA is based on the data provided by FEV GmbH. Certain scripts from the FEV toolchain are also used by the framework. Necessary data and scripts for running SimA are available in the FEV's `PERSIST` SVN repository. This repository has to be checked out and locally available.

6.2 Execute the full workflow

Once the dependencies are resolved, the whole SimA workflow (described in sec. 5.8.5) can be executed. At this point, running the `Main.py` script of the `similarity_client`

[7] `http://smtlib.cs.uiowa.edu/`

package with necessary argument, as shown in listing 6.4, will execute the whole workflow.

```
python Main.py <path_to_persist> <report_generation_directory>
```

Listing 6.4: Execute SimA workflow.

The script receives two command line arguments. The first argument specifies path to the directory where the `PERSIST` SVN repository is checked out. And second argument specifies the path where the HTML reports will be generated.

6.3 Extend SimA

To perform various tasks SimA exposes interfaces and provides implementations to them. The framework is designed in a way so that one or more of the provided implementations can be switched with a similar implementation. This section discusses about how SimA can be extended by defining new implementation for various interfaces and how these implementations can be plugged into the system. The discussion is done based on examples and demo code samples.

Extend data reader - `AbsDataReader` and `AbsProjectDataReader` are two data reader interfaces exposed by SimA (sec. 5.8.2.2). As an example for adopting the implementation in a context other than FEV, a different scenario can be imagined. Lets suppose, instead of checking out interface definitions from SVN for reading, they are locally available. And they are stored in a different file format for which there is already a reader available. Also a data converter has already been written which converts the read interface data to `Interface` object.

For including this functionality into SimA, a new class `LocalSvnInterfaceReader` can be defined which is shown in listing 6.5. One important thing should be noted that, the conversion function preserves the external data with the `_original_object` attribute. This data can be later used to communicate with the corresponding external package or framework (more explanation in sec. 5.8.1.3).

```python
class LocalSvnInterfaceReader(AbsProjectDataReader):

    def read_data_raw(self, file_path, **kwargs):
        return self.interface_reader.read_interface(file_path)

    def convert_data(self, data, **kwargs):
        interface = self.converter.convert_interface_data(data)
        interface._original_object = data
        return interface

    def _update_additional_property(self, data, **kwargs):
        pass
```

Listing 6.5: Interface definition reader implementation.

Now, this interface data reader can be used instead of the default `SvnInterfaceReader`. Just overriding the factory function which creates interface reader to return an object of `LocalSvnInterfaceReader` is sufficient and no other change is required.

Extend similarity calculator - The AbsSimilarityFacade is the exposed interface for any similarity calculation. To continue with the scenario constructed in the previous section, lets suppose, a new package for calculating structural similarity is available and it has to be integrated into the current system.

```
class AlternateStructuralComparisonFacade(AbsSimilarityFacade):

    def _find_similarities_impl(self, objects, **kwargs):
        orig_objects = [obj._original_object for obj in objects]
        results = self._similarity_evaluator.
            analyse_structural_similarity(orig_objects)
        return [ExternalSimilarityResultWrapper(res)
                for res in results]

    def convert_similarity_result(self, similarity_obj, **kwargs):
        return self.converter.convert_structural_result(
            similarity_obj)
```

Listing 6.6: Define a new structural similarity facade.

A new similarity calculation facade, like the one shown in listing 6.6, can be defined which calculates and converts the results using the mentioned similarity calculation package. The external object, preserved during reading interfaces with _original_object attribute, has been used to communicate with the corresponding package. Finally, all the results are converted to UnitStructuralSimilarity and returned.

Adapt object factory - In previous two sections a new data reader and structural comparison facade has been defined. The next task is to use them instead of the ones that are already provided by SimA. The recommended way to do that is to override the DefaultObjectFactory.

```
class ModifiedObjectFactory(DefaultObjectFactory):

    def get_excel_interface_reader(self, *args):
        return LocalSvnInterfaceReader()

    def get_similarity_calculator(self):
        return DefaultSimilarityCalculationWorkflow(
                ExtrinsicSimilarityFacade(),
                AlternateStructuralComparisonFacade(),
                SemanticalComparisonFacade(self.
                    test_based_similarity_jar_location))
```

Listing 6.7: Override default factory implementation.

As explained in section 5.8.6, the factory class provide methods for creating objects of different kind and the DefaultObjectFactory returns all default implementations. Since, only two types of new class has been defines, overriding the corresponding methods from DefaultObjectFactory is sufficient. An example is presented in listing 6.7 which shows the overriding of necessary methods.

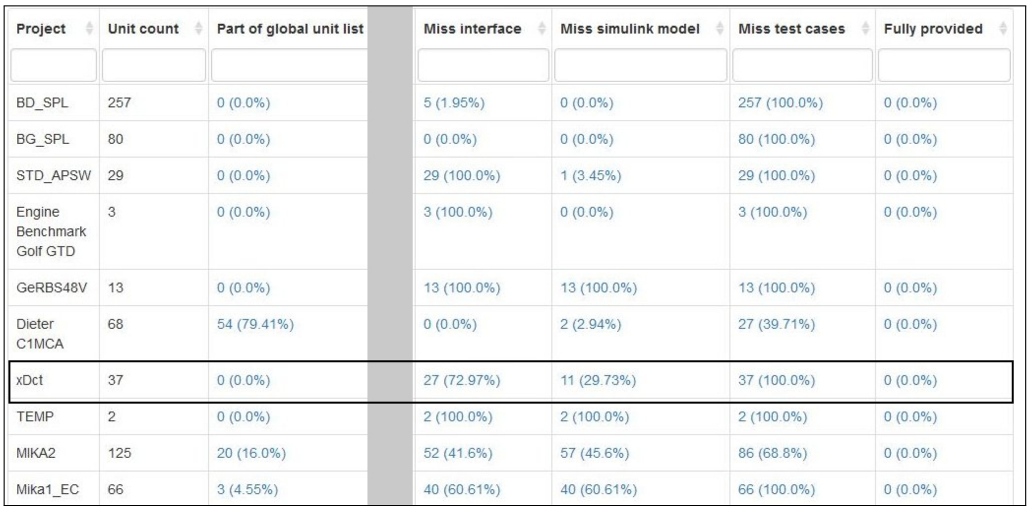
Chapter 7

Evaluation

The primary goal of SimA is to support the APLE process by establishing a SPL and providing useful information to maintain it. To achieve this goal, SimA performs similarity analysis based on the data read from various project sources and generates reports from the results. These generated reports contains not only the similarity results but also information about different projects. The framework has been developed and various reports were generated in the context of FEV GmbH. In this chapter different steps leading to report generation are evaluated based on the corresponding reports.

7.1 Access Data

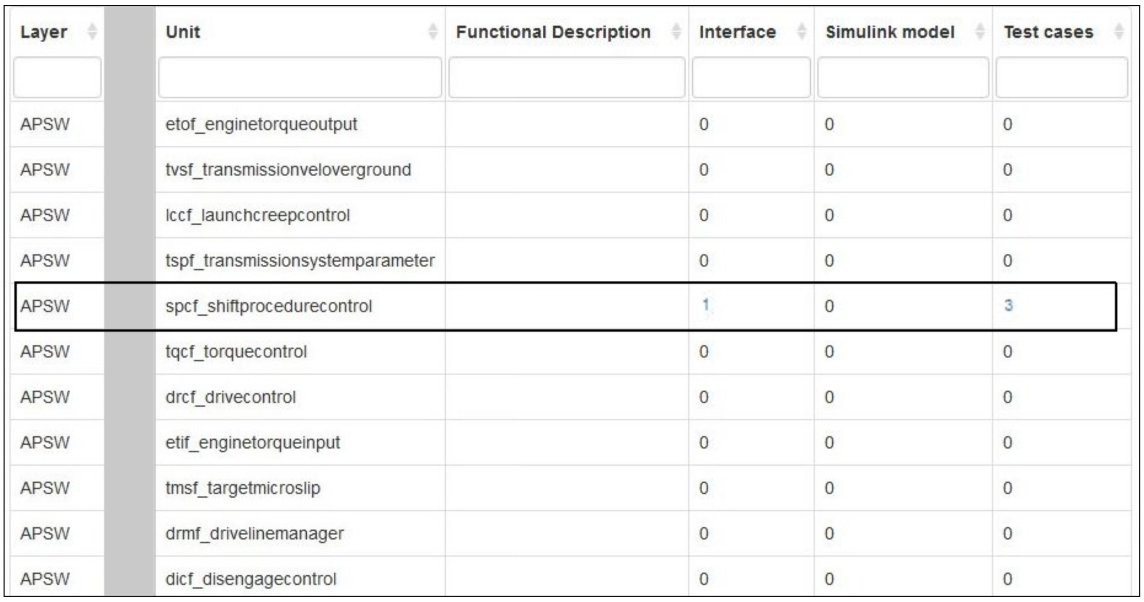
The SimA framework provides implementation to read data from different sources in the context of FEV GmbH. These information is later used for similarity calculation and report generation. Therefore, reading data and interpreting them correctly is one of the initial but most important steps.



Project	Unit count	Part of global unit list	Miss interface	Miss simulink model	Miss test cases	Fully provided
BD_SPL	257	0 (0.0%)	5 (1.95%)	0 (0.0%)	257 (100.0%)	0 (0.0%)
BG_SPL	80	0 (0.0%)	0 (0.0%)	0 (0.0%)	80 (100.0%)	0 (0.0%)
STD_APSW	29	0 (0.0%)	29 (100.0%)	1 (3.45%)	29 (100.0%)	0 (0.0%)
Engine Benchmark Golf GTD	3	0 (0.0%)	3 (100.0%)	0 (0.0%)	3 (100.0%)	0 (0.0%)
GeRBS48V	13	0 (0.0%)	13 (100.0%)	13 (100.0%)	13 (100.0%)	0 (0.0%)
Dieter C1MCA	68	54 (79.41%)	0 (0.0%)	2 (2.94%)	27 (39.71%)	0 (0.0%)
xDct	37	0 (0.0%)	27 (72.97%)	11 (29.73%)	37 (100.0%)	0 (0.0%)
TEMP	2	0 (0.0%)	2 (100.0%)	2 (100.0%)	2 (100.0%)	0 (0.0%)
MIKA2	125	20 (16.0%)	52 (41.6%)	57 (45.6%)	86 (68.8%)	0 (0.0%)
Mika1_EC	66	3 (4.55%)	40 (60.61%)	40 (60.61%)	66 (100.0%)	0 (0.0%)

Figure 7.1: Project status report. (For maintaining a readable size the image is trimmed at the grayed out part. Full report is in appendix A.1)

The project data is read by different classes (sec. 5.8.2) and stored under the `Project` object as property or nested properties. The project status report (sec. 5.8.4) presents these imported data as an HTML table. Figure 7.1, shows part of a project status report table which was generated in the context of FEV GmbH. It shows number of software components for each project and information related to them like how many of them are part of the global unit list. For example, no component from the project *xDct* is part of the global unit list and 11 of them are missing simulink models.



Layer		Unit	Functional Description	Interface	Simulink model	Test cases
APSW		etof_enginetorqueoutput		0	0	0
APSW		tvsf_transmissionveloverground		0	0	0
APSW		lccf_launchcreepcontrol		0	0	0
APSW		tspf_transmissionsystemparameter		0	0	0
APSW		spcf_shiftprocedurecontrol		1	0	3
APSW		tqcf_torquecontrol		0	0	0
APSW		drcf_drivecontrol		0	0	0
APSW		etif_enginetorqueinput		0	0	0
APSW		tmsf_targetmicroslip		0	0	0
APSW		drmf_drivelinemanager		0	0	0
APSW		dicf_disengagecontrol		0	0	0

Figure 7.2: Units missing simulink models from project *xDct*. (For maintaining a readable size the image is trimmed at the grayed out part. Full report is in appendix A.2)

The texts, that presents information about the components, are also link to other reports which shows more details about the corresponding software components. For example, the project *xDct* has 11 units which do not have any simulink model. Further information about these units are presented with another report which is reachable by clicking on the corresponding text of the project status table. Figure 7.2 shows the unit list. This type of table has been used in multiple cases for presenting a list of units (e.g. show units which are part of global unit list). It shows not only basic information about the units but also the number of available interface definition, simulink model and test cases of the corresponding unit. For example, the boxed entry in figure 7.2 represents an unit which has 1 interface definition and 3 test cases available. These texts are hyperlinks to corresponding file or folder in the SVN repository.

7.2 Similarity calculation

The project data read from various sources are used for finding similar software components. As explained in section 4.1, the similarity calculation process is divided into three consecutive stages - extrinsic (sec. 4.1.2), structural (sec. 4.1.4) and semantical (sec. 4.1.6). Each similarity calculation step is dependent on project data and results from previous similarity calculation steps.

The similarity calculation results are used to generate reports which are connected with hyperlinks. The reports are generated hierarchically. Starting from an overview report, each level presents more and more similarity details. At the beginning of the similarity

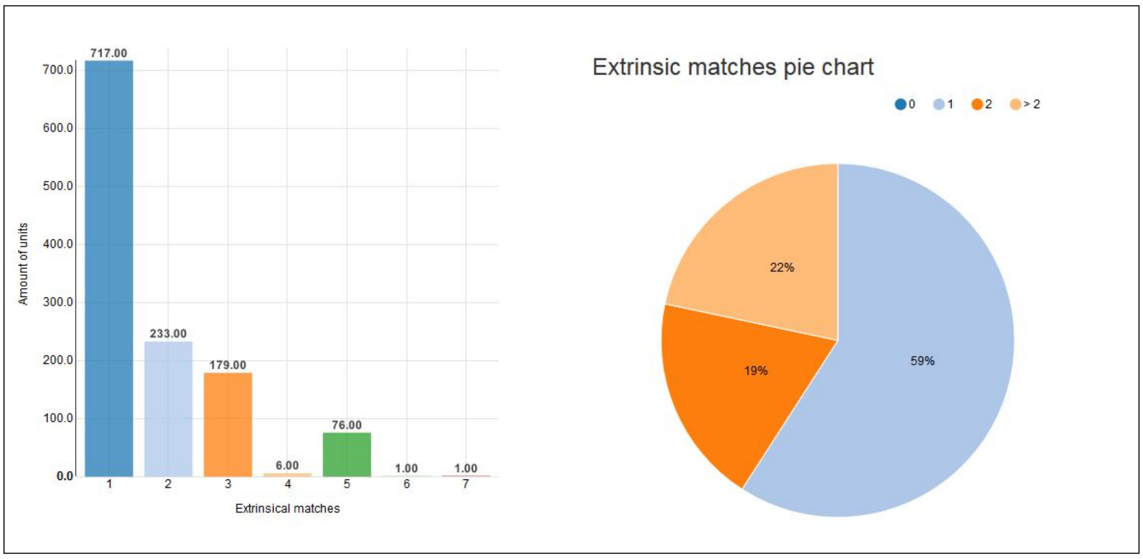


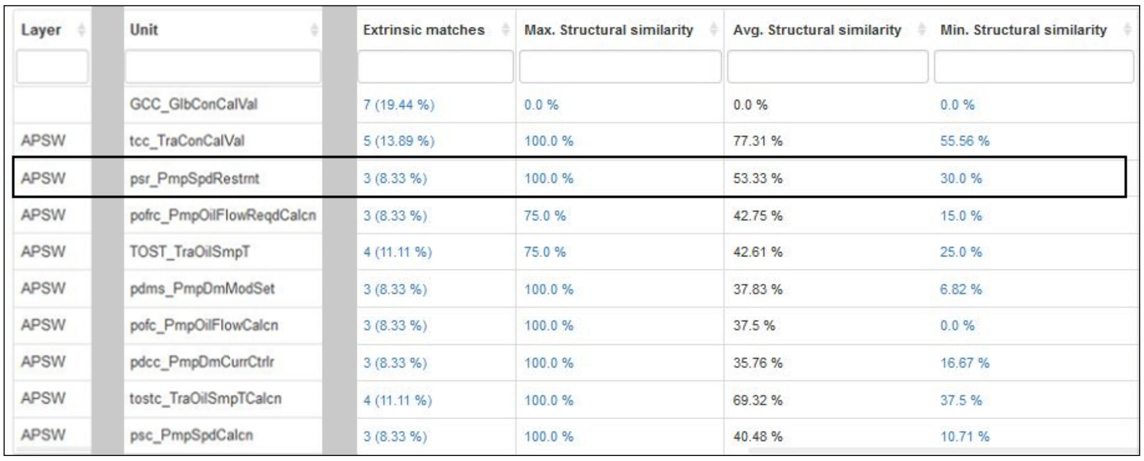
Figure 7.3: Charts showing extrinsic similarity outline.

overview report, there is a bar chart and a pie chart showing an outline of the extrinsic similarity results. An example is show in figure 7.3 which was generated in the context of FEV GmbH. The bar chart plots extrinsic similarity count in the X axis against corresponding number of units in the Y axis. According to this graph, 717 units has no corresponding extrinsically similar component and they are the only ones of their kind. The second bar represents that 233 units has one extrinsically matching component from other project and the 2 in the X axis means there are 2 units for each particular kind.

There is a table just below the chats which contains every unique units from all projects, their similarity overview and usage information among different projects. Figure 7.4 shows part of the overview report which contains similarity information. The sub-figure 7.4a presents extrinsic and structural overview part. The extrinsic similarity column shows how many extrinsically similar units are available for the corresponding unit. In other words, how many projects are using this particular unit. For example, in figure 7.4a the boxed entry shows that the unit `psr_PmpSpdRestrnt` has been used by three different projects. The boxed entry in figure 7.4b shows part of the report that shows which projects are using this specific unit.

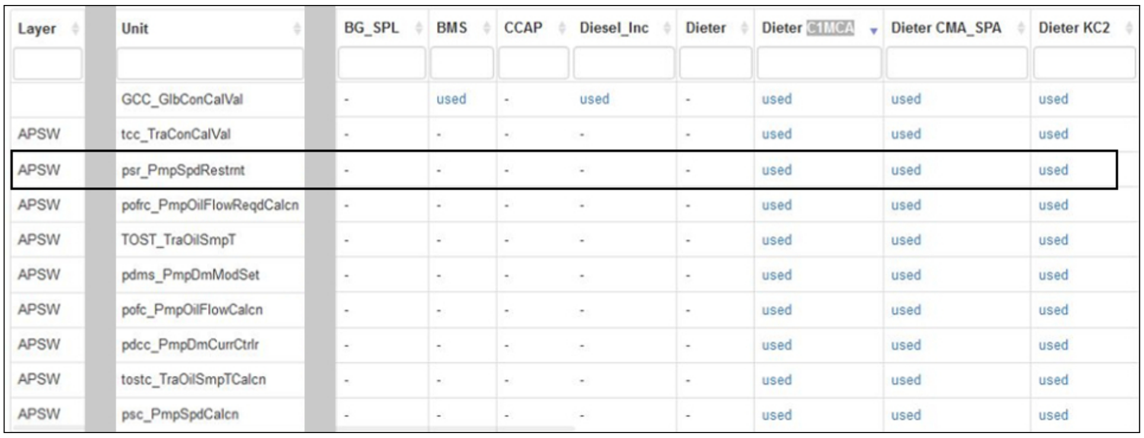
The next three columns in figure 7.4a present maximum, average and minimum structural similarity information. As explained in section 4.1.4, at the fundamental level structural similarity is calculated among signals from different units. Therefore, the maximum and minimum structural similarity are basically structural similarity value of two signals where the corresponding units of them are extrinsically similar. The average column holds the mean of all signal based structural similarity values. For example, in figure 7.4a the boxed entry shows that the corresponding units from different projects has at least one signal pair with 100% structural similarity.

Figure 7.5, shows the maximum, average and minimum semantical similarity values for the



Layer	Unit	Extrinsic matches	Max. Structural similarity	Avg. Structural similarity	Min. Structural similarity
	GCC_GlbConCalVal	7 (19.44 %)	0.0 %	0.0 %	0.0 %
APSW	tcc_TraConCalVal	5 (13.89 %)	100.0 %	77.31 %	55.56 %
APSW	psr_PmpSpdRestrnt	3 (8.33 %)	100.0 %	53.33 %	30.0 %
APSW	pofrc_PmpOilFlowReqdCalcn	3 (8.33 %)	75.0 %	42.75 %	15.0 %
APSW	TOST_TraOilSmpT	4 (11.11 %)	75.0 %	42.61 %	25.0 %
APSW	pdms_PmpDmModSet	3 (8.33 %)	100.0 %	37.83 %	6.82 %
APSW	pofc_PmpOilFlowCalcn	3 (8.33 %)	100.0 %	37.5 %	0.0 %
APSW	pdcc_PmpDmCurrCtrlr	3 (8.33 %)	100.0 %	35.76 %	16.67 %
APSW	tostc_TraOilSmpTCalcn	4 (11.11 %)	100.0 %	69.32 %	37.5 %
APSW	psc_PmpSpdCalcn	3 (8.33 %)	100.0 %	40.48 %	10.71 %

(a) Extrinsic and structural similarity overview.



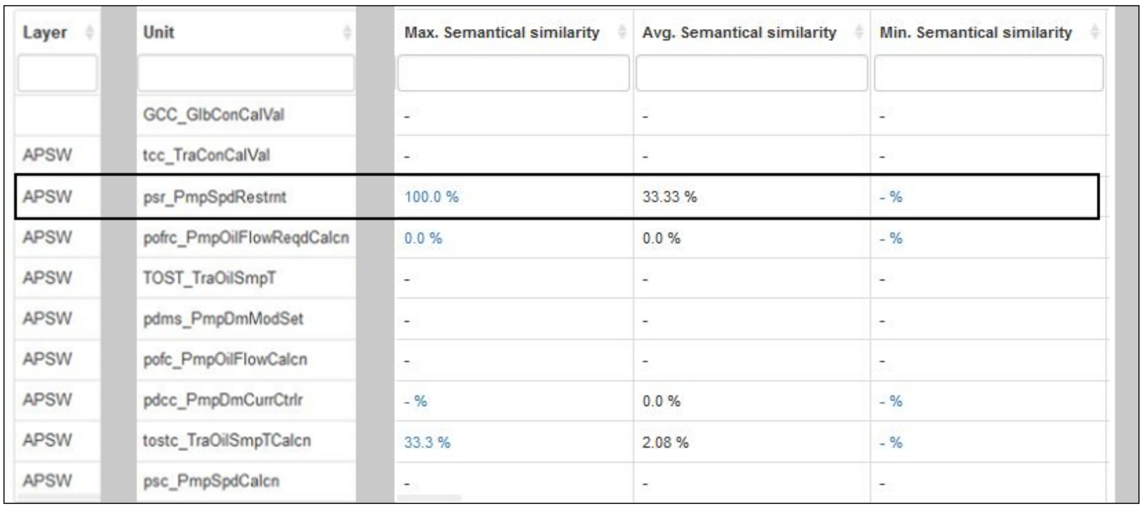
Layer	Unit	BG_SPL	BMS	CCAP	Diesel_Inc	Dieter	Dieter C1MCA	Dieter CMA_SPA	Dieter KC2
	GCC_GlbConCalVal	-	used	-	used	-	used	used	used
APSW	tcc_TraConCalVal	-	-	-	-	-	used	used	used
APSW	psr_PmpSpdRestrnt	-	-	-	-	-	used	used	used
APSW	pofrc_PmpOilFlowReqdCalcn	-	-	-	-	-	used	used	used
APSW	TOST_TraOilSmpT	-	-	-	-	-	used	used	used
APSW	pdms_PmpDmModSet	-	-	-	-	-	used	used	used
APSW	pofc_PmpOilFlowCalcn	-	-	-	-	-	used	used	used
APSW	pdcc_PmpDmCurrCtrlr	-	-	-	-	-	used	used	used
APSW	tostc_TraOilSmpTCalcn	-	-	-	-	-	used	used	used
APSW	psc_PmpSpdCalcn	-	-	-	-	-	used	used	used

(b) Unit usages among different projects.

Figure 7.4: Extrinsic and structural part of the whole overview report. (The whole image including skipped grayed out parts is in appendix A.3).

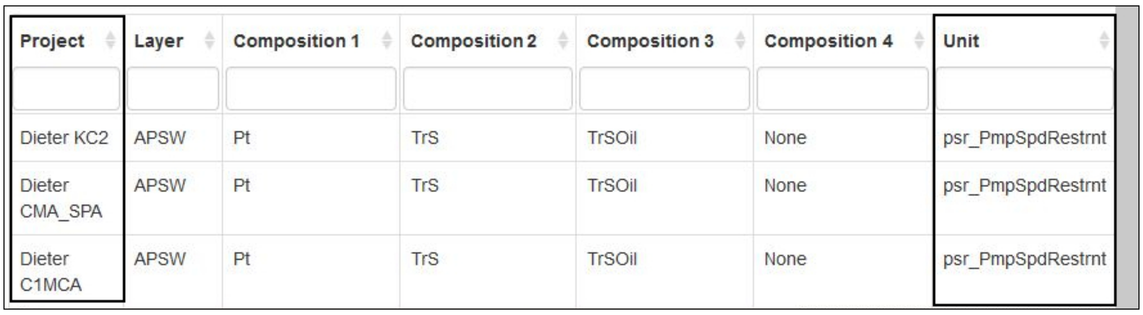
corresponding unit. In this figure, the unit related columns from figure 7.4a are repeated for better understandability. Semantical similarity is calculated between outgoing signals from different units. Therefore, like the structural similarity columns, the semantical similarity values presented in the table are also based on signal's semantical similarity. For example, the marked box in figure 7.5 shows that the unit with name `psr_PmpSpdRestrnt` from different projects has signals with maximum semantical similarity of 100%.

The overview similarity report is linked to reports that shows more detailed similarity information. The extrinsic similarity column contains hyperlink to structural similarity overview report. This report is basically divided into several parts. The first part is a table listing corresponding extrinsically similar units. After that a table is presented which shows the directed average structural similarity values for each pair of units. For example, the boxed entry in the overview report (fig. 7.4a) shows that there are 3 extrinsically similar units with name `psr_PmpSpdRestrnt`. It is linked to the structural overview report. The table shown in figure 7.7 is the first part of this report. It shows detailed information about the extrinsically similar units. A close inspection of this table reveals



Layer	Unit	Max. Semantical similarity	Avg. Semantical similarity	Min. Semantical similarity
	GCC_GlbConCalVal	-	-	-
APSW	tcc_TraConCalVal	-	-	-
APSW	psr_PmpSpdRestrnt	100.0 %	33.33 %	- %
APSW	pofrc_PmpOilFlowReqdCalcn	0.0 %	0.0 %	- %
APSW	TOST_TraOilSmpT	-	-	-
APSW	pdms_PmpDmModSet	-	-	-
APSW	pofc_PmpOilFlowCalcn	-	-	-
APSW	pdcc_PmpDmCurrCtrlr	- %	0.0 %	- %
APSW	tostc_TraOilSmpTCalcn	33.3 %	2.08 %	- %
APSW	psc_PmpSpdCalcn	-	-	-

Figure 7.5: Semantical similarity from the overview report. (The whole image including skipped grayed out parts is in appendix A.3)



Project	Layer	Composition 1	Composition 2	Composition 3	Composition 4	Unit
Dieter KC2	APSW	Pt	TrS	TrSOil	None	psr_PmpSpdRestrnt
Dieter CMA_SPA	APSW	Pt	TrS	TrSOil	None	psr_PmpSpdRestrnt
Dieter C1MCA	APSW	Pt	TrS	TrSOil	None	psr_PmpSpdRestrnt

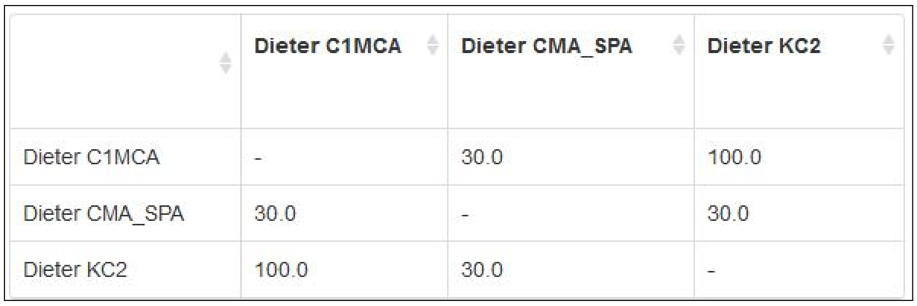
Figure 7.6: Extrinsically matched units.

that the units are from different projects but they have the same name (boxed entries in fig. 7.6). The second part of this structural overview report is shown in figure 7.7. It shows the structural similarity percentage for units from different projects. For example, the unit that comes from *Dieter C1MCA* is 30% similar to the one coming from *Dieter CMA_SPA* and 100% similar to *Dieter KC2*.

The rest of the structural similarity overview report contains signal based directed similarity information. This part is divided into several tables where each of the table is dedicated to one particular kind of signal. An example is shown in figure 7.8 which shows the structural similarity percentages of *CAL* signals for the extrinsically similar unit `psr_PmpSpdRestrnt` from different projects. Other tables dedicated to other kind of signals are similar to this table except the table for *OUT* signals.

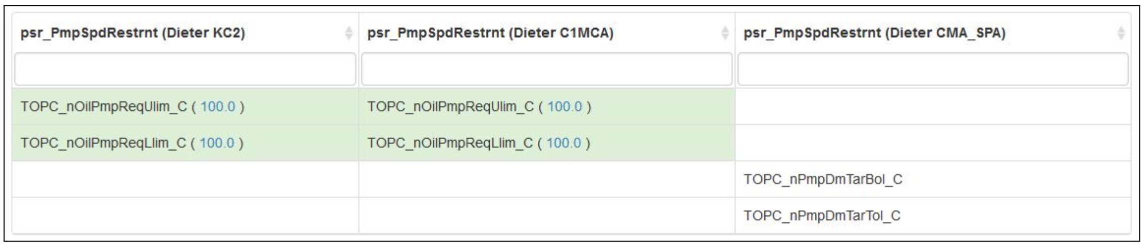
Besides structural similarity information the table for *OUT* signal also show corresponding semantical similarity information. As the figure 7.9 presents, there are two groups of percentage beside every signal name. The first group shows the structural similarity and the second one shows the semantical information. A dash (-) instead of a numeric value means that the semantical similarity calculation process did not complete.

Similarity percentages beside the signal names are also link to a more detailed structural



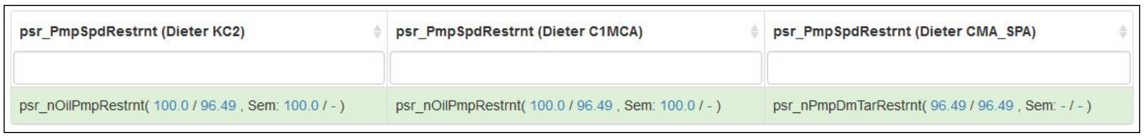
	Dieter C1MCA	Dieter CMA_SPA	Dieter KC2
Dieter C1MCA	-	30.0	100.0
Dieter CMA_SPA	30.0	-	30.0
Dieter KC2	100.0	30.0	-

Figure 7.7: Structural similarity overview.



psr_PmpSpdRestrnt (Dieter KC2)	psr_PmpSpdRestrnt (Dieter C1MCA)	psr_PmpSpdRestrnt (Dieter CMA_SPA)
TOPC_nOilPmpReqUlim_C (100.0)	TOPC_nOilPmpReqUlim_C (100.0)	
TOPC_nOilPmpReqLlim_C (100.0)	TOPC_nOilPmpReqLlim_C (100.0)	
		TOPC_nPmpDmTarBol_C
		TOPC_nPmpDmTarTol_C

Figure 7.8: Structural similarity overview of CAL signals.



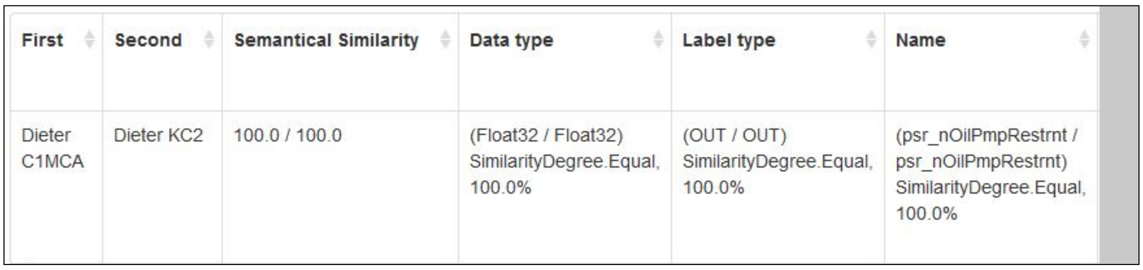
psr_PmpSpdRestrnt (Dieter KC2)	psr_PmpSpdRestrnt (Dieter C1MCA)	psr_PmpSpdRestrnt (Dieter CMA_SPA)
psr_nOilPmpRestrnt(100.0 / 96.49 , Sem: 100.0 / -)	psr_nOilPmpRestrnt(100.0 / 96.49 , Sem: 100.0 / -)	psr_nPmpDmTarRestrnt(96.49 / 96.49 , Sem: - / -)

Figure 7.9: Structural similarity overview of OUT signals.

and, in case of OUT signals, semantical similarity report. This report breaks down the signal based structural similarity to individual attributes. In case of OUT signals a directed semantical similarity information is also shown. An example is presented in figure 7.10, showing the attribute similarity information including the attribute values. The first two columns shows the signal's source project and the attribute similarity values are directed from the first project to the second project.

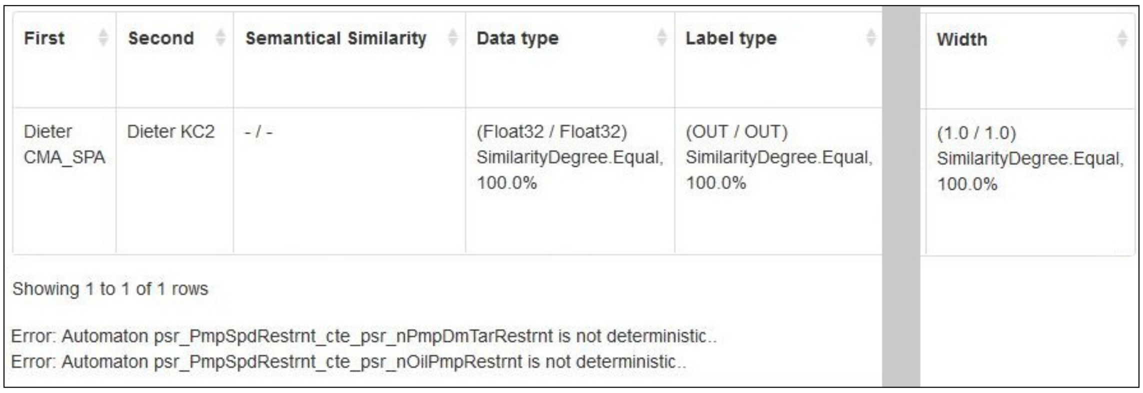
In case of OUT signal an additional column showing directed semantical similarity is also added. For example, the figure 7.10 shows that the corresponding OUT signals are semantically 100% similar to each other. Further semantical similarity information is shown after the table like it is shown in figure 7.11. In this case, the semantical similarity was not calculated due to the non-deterministic nature of the corresponding automatons.

The similarity reports shown in figure 7.11 are also linked from the similarity overview report (fig. 7.4). As explained earlier, this reports shows maximum and minimum values of structural and semantical similarity for the corresponding units. And these cells not only contains text but also hyperlinks which leads to the corresponding attribute based similarity report.



First	Second	Semantical Similarity	Data type	Label type	Name
Dieter C1MCA	Dieter KC2	100.0 / 100.0	(Float32 / Float32) SimilarityDegree.Equal, 100.0%	(OUT / OUT) SimilarityDegree.Equal, 100.0%	(psr_nOilPmpRestrnt / psr_nOilPmpRestrnt) SimilarityDegree.Equal, 100.0%

Figure 7.10: Attribute based structural similarity and semantical similarity (The whole image including skipped grayed out parts is in appendix A.4)



First	Second	Semantical Similarity	Data type	Label type	Width
Dieter CMA_SPA	Dieter KC2	- / -	(Float32 / Float32) SimilarityDegree.Equal, 100.0%	(OUT / OUT) SimilarityDegree.Equal, 100.0%	(1.0 / 1.0) SimilarityDegree.Equal, 100.0%

Showing 1 to 1 of 1 rows

Error: Automaton psr_PmpSpdRestrnt_cte_psr_nPmpDmTarRestrnt is not deterministic..
Error: Automaton psr_PmpSpdRestrnt_cte_psr_nOilPmpRestrnt is not deterministic..

Figure 7.11: Semantical similarity messages. (The whole image including skipped grayed out parts is in appendix A.5)

7.3 Code quality

For evaluating the code quality of the SimA framework, the evaluation criteria or metrics defined in [RH97] are used. This metric has been developed for object oriented languages and therefore, suitable for the SimA framework. The metrics defined in [RH97] involves complex mathematical calculation to generate score values. Based on these results the quality is measured. However, for the sake of simplicity, in this section the metrics are used from a hypothetical point of view. Which means, the SimA framework is examined based on the idea behind each of the metric skipping the complex calculation.

Method size - Ease of understandability or readability depends on the size of a method. Larger methods are harder to understand and read. The size of a method can be calculated based on the line of codes [RH97]. In the context of SimA framework, it has been tried to keep the method size less than 10 lines. Sometimes, long lines has been broken to multiple likes which can be counted as a single line. However, a few long methods can be found in the `report_generation` and `similarity_calculation` module. In these cases, the methods has been given very descriptive name and the task performed by these methods has been kept very specific to avoid any ambiguity. For example, the `create_units_table_data` method from `report_generation` module. It generates a `TableData` object containing information about a list of `Unit`. The name clearly identifies the intension.

Methods per class - The number of methods per class is an indication of the class's complexity. More methods results into a complex and large class. This measure is often used with the *Method size* metric to predict effort and maintenance difficulty of a class [RH97]. In the context of SimA, classes are kept simple and performs very specific task. Hence, they are not very long and most of the classes contain less than 5 methods. For example, the `ExcelUnitReader` class has 5 methods and each of them are shorter than 10 lines. This class was developed for reading `Unit` from excel files and that is what it does. On the other hand, there are classes like `UnitUsageHtmlReport` which has 14 methods and some of them are as long as 15 lines. It generates the similarity overview report which requires some complex data processing. This process has been broken down to smaller tasks which resulted into higher count of methods. Some of them could have been extracted to create a data processing class. But, these operations are very specific to the `UnitUsageHtmlReport` class and creating a separate class would simply increase the development time without any significant gain in return.

Lack of cohesion of methods - Methods in a class are in high level of cohesion if they are processing same kind of data [RH97]. This high level of cohesion implies that the methods of that class are focused to perform specific task. Lack of cohesion among the methods of a class refers to the possibility of sub-dividing the methods to two or more disjoint classes and increases the complexity of the corresponding class. The development of SimA has followed the single responsibility principle [Mar03] where one class has only one defined task. Which has resulted into smaller classes and high level of cohesion among methods of same class.

Coupling between object classes - Two objects or classes are considered as coupled when they are directly communicating with each other or using methods and attributes from the other class. Inheriting from another class couples the subclass with the superclass [RH97]. Non-inheritance coupling is considered to be harmful for modular design and makes classes less reusable. SimA solves this problem with dependency injection (explained in sec. 2.8.1). With this technique, class dependencies are always passed as constructor parameter. The expected type of the parameter object is an interface which can be replaces with any implementation. Therefore, two classes communicate with each other through an interface and never directly. This keeps the coupling factor to a bare minimum.

Depth of inheritance - In an inheritance hierarchy the maximum depth starting from the root is the depth of inheritance [RH97]. A higher depth means that the child class inherits more methods from the parent classes which makes it harder to predict the exact behavior and significantly increases the class complexity. The SimA framework does not create any unnecessary hierarchy and tries to keep the depth as short as possible. Most classes in SimA has a inheritance depth of 3 considering the `object` root class. The highest depth of inheritance is 4 (e.g. `ExcelUnitReader`).

Chapter 8

Conclusion

The goal of this thesis was to design and develop a tool that can analyse similarities among different software components for providing valuable feedback to the APLE process and help maintain an active SPL. The similarity analysis framework (SimA) was developed to achieve this goal. It performs similarity analysis from different levels and generates reports based on the results. Moreover, SimA is not only a tool that can perform above mentioned tasks, but also is a framework that exposes interfaces to extend and alter its default behaviour to a certain extent.

SimA's current implementation of various interfaces is implemented in the context of FEV GmbH. The framework reads various project and software component data from the company's SVN repository and performs extrinsic, structural and semantical similarity analysis on them. Finally, HTML based reports are generated which presents the similarity results and useful information about different software projects.

The similarity calculation functionalities are provided by external tools and SimA provides simple unified interfaces to communicate with them. The extrinsic similarity calculation is provided by the `extrinsic_similarity` package which follows the algorithm explained in [BRR+14b]. The `struct_analysis` package is responsible for structural similarity analysis which is implemented based on the methodologies presented in [KRS+16]. The project `test-based-validation-tool` is implemented in Java and calculates test case based semantical similarity according to the algorithm presented in [RRS+16]. These projects and packages follow their own data representations and generate results in different formats. SimA converts them into its own standard data representation and processes the results to establish correlation among them.

The calculated similarity result is used for generating reports. SimA defines separate interface for report generation and controls it from an abstract level. On the concrete implementation level the `isddgf` package is used for report generation. This package was developed according to the methodologies presented in [Kha16] and provides Python interface for generating HTML based documents.

A workflow definition specified by SimA governs the whole process string from data import to report generation. The workflow also includes generation of excel based global list of software components which includes information about all available components from different projects and their usage data. This list is automatically updated every time the workflow is executed keeping the global list up to date.

SimA has been developed to be robust and work under erroneous data situations. In the context of FEV GmbH, it has been tested and improved several times which has contributed to its robustness. Currently, it is part of FEV's nightly build process and the generated reports are being used to maintain a more effective yet cheaper software development life cycle.

Bibliography

[BHDG10] Huang Bo, Dong Hui, Wang Dafang, and Zhao Guifan. Basic concepts on autosar development. In *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, volume 1, pages 871–873. IEEE, 2010.

[Bol04] Béla Bollobás. *Extremal graph theory*. Courier Corporation, 2004.

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software engineering*, pages 33–42. ACM, 2006.

[BRR14a] Christian Berger, Holger Rendel, and Bernhard Rumpe. Measuring the ability to form a product line from existing products. *arXiv preprint arXiv:1409.6583*, 2014.

[BRR$^+$14b] Christian Berger, Holger Rendel, Bernhard Rumpe, Carsten Busse, Thorsten Jablonski, and Fabian Wolf. Product line metrics for legacy software in practice. *arXiv preprint arXiv:1409.6581*, 2014.

[FFL09] Fabrizio Fabbrini, Mario Fusani, and Giuseppe Lami. One decade of software process assessments in automotive: a retrospective analysis. In *Computing in the Global Information Technology, 2009. ICCGI'09. Fourth International Multi-Conference on*, pages 92–97. IEEE, 2009.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. 2004.

[Gam95] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[HC01] Jim Highsmith and Alistair Cockburn. Agile software development: The business of innovation. *Computer*, 34(9):120–127, 2001.

[Keh15] Philipp Kehrbusch. Structural compatibility analysis of software components, 2015.

[Kha16] Safdar Dabeer Khan. Interactive software design document generation framework for the automotive industry, 2016.

[KLD02] Kyo C Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *IEEE software*, 19(4):58, 2002.

[Kot03] Martin Kot. The state explosion problem. 2003.

[KRS+16] Phillipp Kehrbusch, Bernhard Rumpe, Christoph Schulze, Johannes Richenhagen, and Axel Schloßer. Interface based similarity analysis of software components for the automotive industry. 2016.

[Lev66] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet physics doklady*, volume 10, page 707, 1966.

[LJK+01] Jong Kook Lee, Seung Jae Jung, Soo Dong Kim, Woo Hyun Jang, and Dong Han Ham. Component identification method with coupling and cohesion. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 79–86. IEEE, 2001.

[Mar03] Robert Cecil Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR, 2003.

[McG04] John D McGregor. Software product lines. 2004.

[MdL01] Marcus Eduardo Markiewicz and Carlos JP de Lucena. Object oriented framework development. *Crossroads*, 7(4):3–9, 2001.

[ML75] Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[MTK94] Dieter Merkl, A Min Tjoa, and Gerti Kappel. Learning the semantic similarity of reusable software components. In *Software Reuse: Advances in Software Reusability, 1994. Proceedings., Third International Conference on*, pages 33–41. IEEE, 1994.

[MW03] E Michael Maximilien and Laurie Williams. Assessing test-driven development at ibm. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 564–569. IEEE, 2003.

[Nei09] Danuza Ferreira Santana Neiva. *Riple-re: A requirements engineering process for software product lines.* PhD thesis, M. Sc. Dissertation, Universidade Federal de Pernambuco, Brazil, 2009.

[Nor02] Linda M Northrop. Sei's software product line tenets. *IEEE software*, 19(4):32, 2002.

[PBKS07] Alexander Pretschner, Manfred Broy, Ingolf H Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *2007 Future of Software Engineering*, pages 55–71. IEEE Computer Society, 2007.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques.* Springer Science & Business Media, 2005.

[PPJ+13] Amit Patel, Antoine Picard, Eugene Jhong, Jeremy Hylton, Matt Smart, and Mike Shields. Google python style guide. *available in HTML form. URL: http://google-styleguide. googlecode. com/svn/trunk/pyguide. html*, 12, 2013.

[PULPT02] Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter F Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, 2002.

[RH97] Linda H Rosenberg and Lawrence E Hyatt. Software quality metrics for object-oriented environments. *Crosstalk Journal, April*, 10(4):1–6, 1997.

[RPS14] Johannes Richenhagen, Stefan Pischinger, and Axel Schloßer. Persist-a flexible and automatically verifiable software architecture for the automotive powertrain. *Journal of Electrical Engineering*, 2(3):108–115, 2014.

[RRS^{+}16] Johannes Richenhagen, Bernhard Rumpe, Axel Schloßer, Christoph Schulze, Kevin Thissen, and Michael von Wenckstern. Test-driven semantical similarity analysis for software product line extraction. 2016.

[RSRS15] Bernhard Rumpe, Christoph Schulze, Johannes Richenhagen, and Axel Schloßer. Agile synchronization between a software product line and its products. In *GI-Jahrestagung*, pages 1687–1698, 2015.

[RSVW^{+}15] Bernhard Rumpe, Christoph Schulze, Michael Von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral compatibility of simulink models for product line maintenance and evolution. In *Proceedings of the 19th International Conference on Software Product Line*, pages 141–150. ACM, 2015.

[RSW^{+}15] Bernhard Rumpe, Christoph Schulze, Michael von Wenckstern, Jan Oliver Ringert, and Peter Manhart. Behavioral Compatibility of Simulink Models for Product Line Maintenance and Evolution. In *SPLC 2015*, 2015.

[Thi15] Kevin Thissen. Testbasierte kompatibilitätsanalysen von funktionskomponenten, 2015.

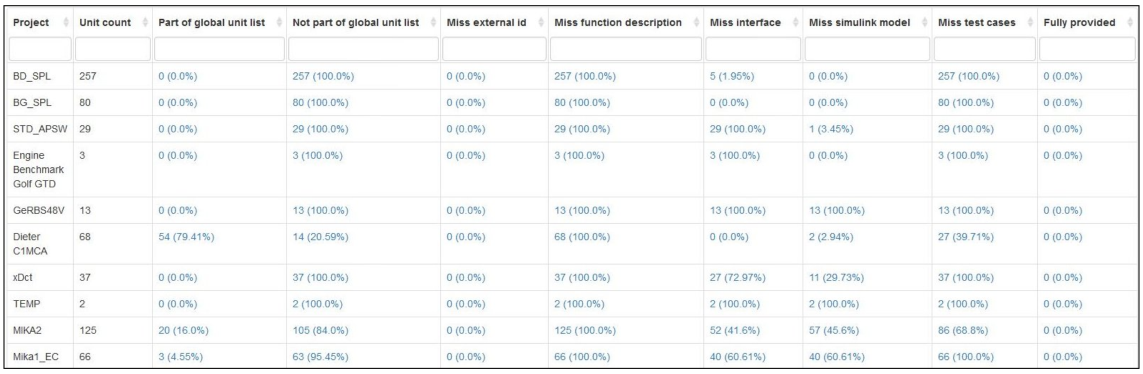
[TT01] Barry N Taylor and Ambler Thompson. The international system of units (si). 2001.

[www16] Introduction to AUTOSAR `https://elearning.vector.com/vl_autosar_introduction_en.html`, July 2016. Accessed: July 25, 2016.

[ZK12] Changyan Zhou and Ratnesh Kumar. Semantic translation of simulink diagrams to input/output extended finite automata. *Discrete Event Dynamic Systems*, 22(2):223–247, 2012.

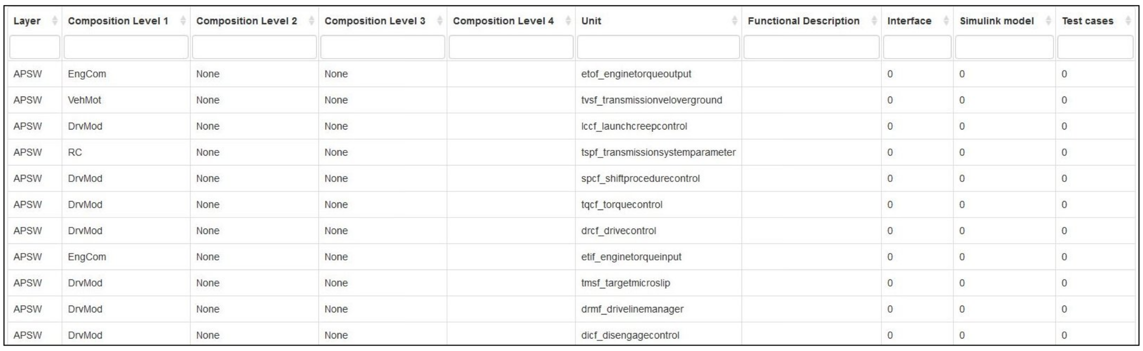
Appendix A

Appendix



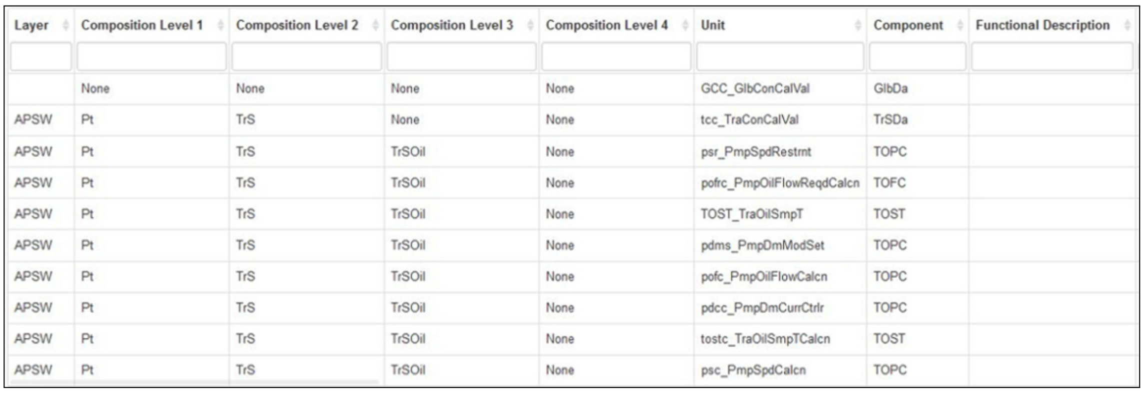
Project	Unit count	Part of global unit list	Not part of global unit list	Miss external id	Miss function description	Miss interface	Miss simulink model	Miss test cases	Fully provided
BD_SPL	257	0 (0.0%)	257 (100.0%)	0 (0.0%)	257 (100.0%)	5 (1.95%)	0 (0.0%)	257 (100.0%)	0 (0.0%)
BG_SPL	80	0 (0.0%)	80 (100.0%)	0 (0.0%)	80 (100.0%)	0 (0.0%)	0 (0.0%)	80 (100.0%)	0 (0.0%)
STD_APSW	29	0 (0.0%)	29 (100.0%)	0 (0.0%)	29 (100.0%)	29 (100.0%)	1 (3.45%)	29 (100.0%)	0 (0.0%)
Engine Benchmark Golf GTD	3	0 (0.0%)	3 (100.0%)	0 (0.0%)	3 (100.0%)	3 (100.0%)	0 (0.0%)	3 (100.0%)	0 (0.0%)
GeRBS48V	13	0 (0.0%)	13 (100.0%)	0 (0.0%)	13 (100.0%)	13 (100.0%)	13 (100.0%)	13 (100.0%)	0 (0.0%)
Dieter C1MCA	68	54 (79.41%)	14 (20.59%)	0 (0.0%)	68 (100.0%)	0 (0.0%)	2 (2.94%)	27 (39.71%)	0 (0.0%)
xDct	37	0 (0.0%)	37 (100.0%)	0 (0.0%)	37 (100.0%)	27 (72.97%)	11 (29.73%)	37 (100.0%)	0 (0.0%)
TEMP	2	0 (0.0%)	2 (100.0%)	0 (0.0%)	2 (100.0%)	2 (100.0%)	2 (100.0%)	2 (100.0%)	0 (0.0%)
MIKA2	125	20 (16.0%)	105 (84.0%)	0 (0.0%)	125 (100.0%)	52 (41.6%)	57 (45.6%)	86 (68.8%)	0 (0.0%)
Mika1_EC	66	3 (4.55%)	63 (95.45%)	0 (0.0%)	66 (100.0%)	40 (60.61%)	40 (60.61%)	66 (100.0%)	0 (0.0%)

Figure A.1: Project status report



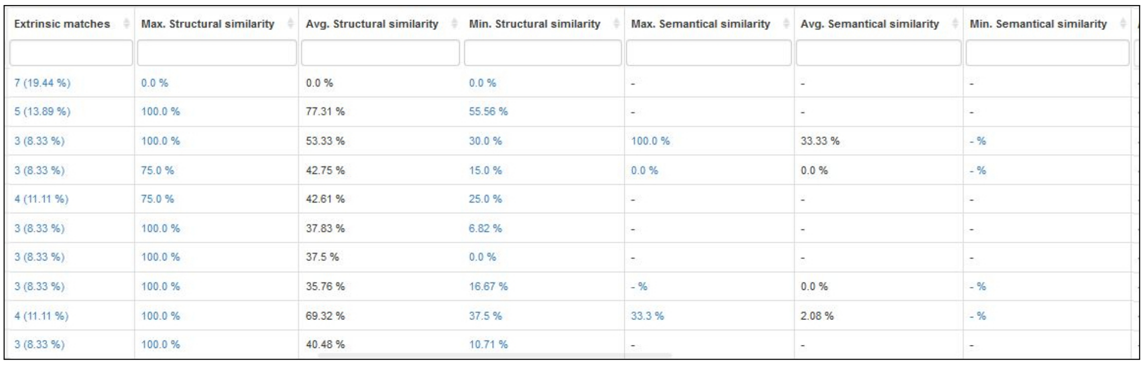
Layer	Composition Level 1	Composition Level 2	Composition Level 3	Composition Level 4	Unit	Functional Description	Interface	Simulink model	Test cases
APSW	EngCom	None	None		etof_enginetorqueoutput		0	0	0
APSW	VehMot	None	None		tvsf_transmissionveloverground		0	0	0
APSW	DrvMod	None	None		lccf_launchcreepcontrol		0	0	0
APSW	RC	None	None		tspf_transmissionsystemparameter		0	0	0
APSW	DrvMod	None	None		spcf_shiftprocedurecontrol		0	0	0
APSW	DrvMod	None	None		tqcf_torquecontrol		0	0	0
APSW	DrvMod	None	None		drcf_drivecontrol		0	0	0
APSW	EngCom	None	None		etif_enginetorqueinput		0	0	0
APSW	DrvMod	None	None		tmsf_targetmicroslip		0	0	0
APSW	DrvMod	None	None		drmf_drivelinemanager		0	0	0
APSW	DrvMod	None	None		dicf_disengagecontrol		0	0	0

Figure A.2: Units missing simulink models from project *xDct*.



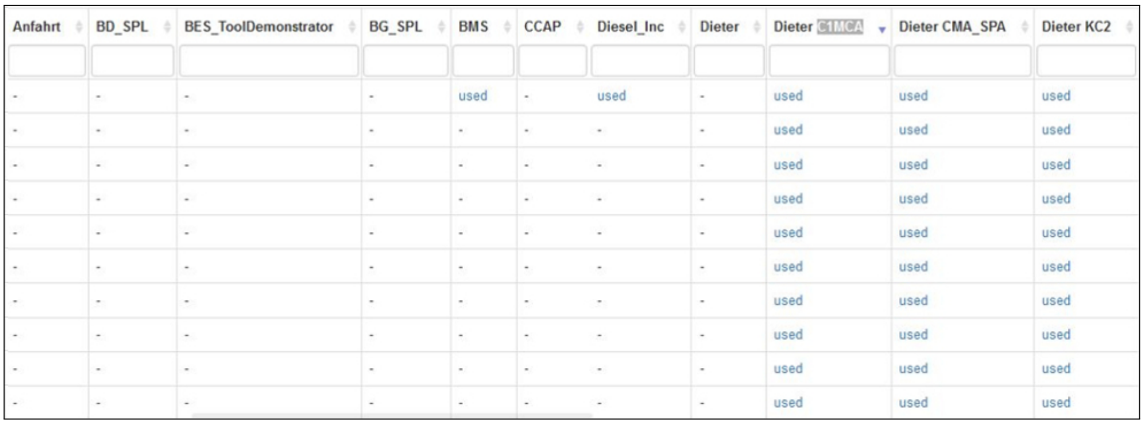
Layer	Composition Level 1	Composition Level 2	Composition Level 3	Composition Level 4	Unit	Component	Functional Description
	None	None	None	None	GCC_GlbConCalVal	GlbDa	
APSW	Pt	TrS	None	None	tcc_TraConCalVal	TrSDa	
APSW	Pt	TrS	TrSOil	None	psr_PmpSpdRestrnt	TOPC	
APSW	Pt	TrS	TrSOil	None	pofrc_PmpOilFlowReqdCalcn	TOFC	
APSW	Pt	TrS	TrSOil	None	TOST_TraOilSmpT	TOST	
APSW	Pt	TrS	TrSOil	None	pdms_PmpDmModSet	TOPC	
APSW	Pt	TrS	TrSOil	None	pofc_PmpOilFlowCalcn	TOPC	
APSW	Pt	TrS	TrSOil	None	pdcc_PmpDmCurrCtrlr	TOPC	
APSW	Pt	TrS	TrSOil	None	tostc_TraOilSmpTCalcn	TOST	
APSW	Pt	TrS	TrSOil	None	psc_PmpSpdCalcn	TOPC	

(a) Part 1



Extrinsic matches	Max. Structural similarity	Avg. Structural similarity	Min. Structural similarity	Max. Semantical similarity	Avg. Semantical similarity	Min. Semantical similarity
7 (19.44 %)	0.0 %	0.0 %	0.0 %	-	-	-
5 (13.89 %)	100.0 %	77.31 %	55.56 %	-	-	-
3 (8.33 %)	100.0 %	53.33 %	30.0 %	100.0 %	33.33 %	- %
3 (8.33 %)	75.0 %	42.75 %	15.0 %	0.0 %	0.0 %	- %
4 (11.11 %)	75.0 %	42.61 %	25.0 %	-	-	-
3 (8.33 %)	100.0 %	37.83 %	6.82 %	-	-	-
3 (8.33 %)	100.0 %	37.5 %	0.0 %	-	-	-
3 (8.33 %)	100.0 %	35.76 %	16.67 %	- %	0.0 %	- %
4 (11.11 %)	100.0 %	69.32 %	37.5 %	33.3 %	2.08 %	- %
3 (8.33 %)	100.0 %	40.48 %	10.71 %	-	-	-

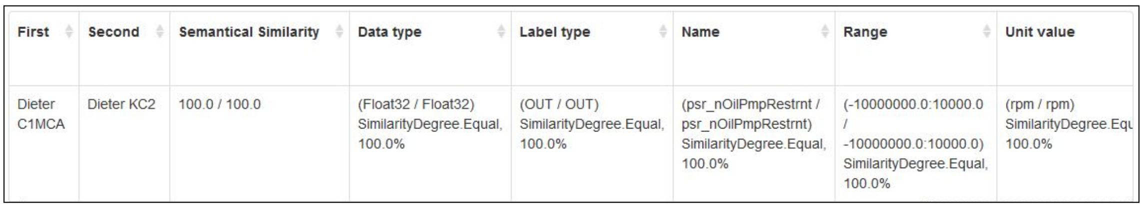
(b) Part 2



Anfahrt	BD_SPL	BES_ToolDemonstrator	BG_SPL	BMS	CCAP	Diesel_Inc	Dieter	Dieter C1MCA	Dieter CMA_SPA	Dieter KC2
-	-	-	-	used	-	used	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used
-	-	-	-	-	-	-	-	used	used	used

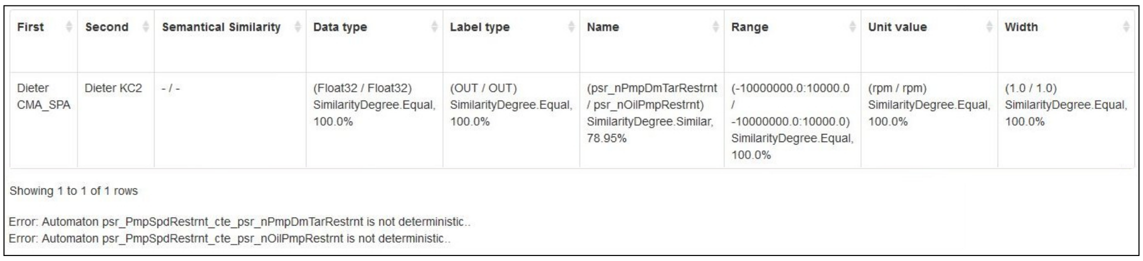
(c) Part 3

Figure A.3: Similarity overview report sliced into parts.



First	Second	Semantical Similarity	Data type	Label type	Name	Range	Unit value
Dieter C1MCA	Dieter KC2	100.0 / 100.0	(Float32 / Float32) SimilarityDegree.Equal, 100.0%	(OUT / OUT) SimilarityDegree.Equal, 100.0%	(psr_nOilPmpRestrnt / psr_nOilPmpRestrnt) SimilarityDegree.Equal, 100.0%	(-10000000.0:10000.0 / -10000000.0:10000.0) SimilarityDegree.Equal, 100.0%	(rpm / rpm) SimilarityDegree.Equ 100.0%

Figure A.4: Attribute based structural similarity and semantical similarity full image.



First	Second	Semantical Similarity	Data type	Label type	Name	Range	Unit value	Width
Dieter CMA_SPA	Dieter KC2	- / -	(Float32 / Float32) SimilarityDegree.Equal, 100.0%	(OUT / OUT) SimilarityDegree.Equal, 100.0%	(psr_nPmpDmTarRestrnt / psr_nOilPmpRestrnt) SimilarityDegree.Similar, 78.95%	(-10000000.0:10000.0 / -10000000.0:10000.0) SimilarityDegree.Equal, 100.0%	(rpm / rpm) SimilarityDegree.Equal, 100.0%	(1.0 / 1.0) SimilarityDegree.Equal, 100.0%

```
Showing 1 to 1 of 1 rows

Error: Automaton psr_PmpSpdRestrnt_cte_psr_nPmpDmTarRestrnt is not deterministic..
Error: Automaton psr_PmpSpdRestrnt_cte_psr_nOilPmpRestrnt is not deterministic..
```

Figure A.5: Semantical similarity messages full image.