Development of Tool Support for Compositional Verification of Railway Interlocking Systems

Master Thesis

Cebrail Erdogan

Kongens Lyngby 2017

DTU Compute
Department of Applied Mathematics and Computer Science

Master of Science in Engineering
Computer Science and Engineering

Development of Tool Support for Compositional Verification of Railway Interlocking Systems
Master Thesis

Cebrail Erdogan
s113414@student.dtu.dk

Technical University of Denmark
Department of Applied Mathematics and Computer Science

DTU Compute
Richard Petersens Plads
Building 303b
2800 Kgs. Lyngby
DENMARK

Tel. +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

This thesis describes the entire development of a decomposition tool, supporting the RobustRailS' compositional verification of railway interlocking systems.

Running a verification process with big railway networks can be a problem due to the state space explosion problem. A decomposition of railway systems can significantly reduce the verification time and increase the success rate of execution. With the tool developed in this project, the verification tool is supported for large scale industrial use by letting users easily decompose networks into smaller chunks.

A divide strategy called *border cut* is presented along with different approaches on how to apply this cut. The tool is designed to easily be extended with new types of cuts. Therefore, other divide strategies are also introduced as potential future extensions.

The RAISE Specification Language (RSL) is used to assist in the specification and design of the software. Starting from abstract specifications and developing them into concrete specifications, remarkably contributes to the overall quality of the product. Furthermore, development processes such as test-driven development (TDD) has been used to encourage simple design and confidence in the product.

The end product is a command line interface tool that accepts networks defined in XML and generates new XML files containing the sub-networks. Multiple experiments with self-created and real-world networks show that the tool can handle networks of different sizes, providing a very smooth user experience as well.

Sammenfatning

Denne afhandling beskriver hele udviklingen af et dekomponeringsværktøj, der understøtter RobustRailS' kompositionelle verifikation af jernbane-sikringsanlæg.

At køre en verifikationsproces for store jernbanenet kan være problematisk på grund af kombinatorisk eksplosion i tilstandsrummet. En dekomponering af jernbanenet kan reducere verifikationstiden betydeligt og øge antallet af succeskørsler af verifikationsværktøjet. Med dekomponeringsværktøjet udviklet i dette projekt støttes verifikationsværktøjet til industriel brug i stor skala ved at lade brugere nemt dele et netværk i mindre stykker.

En skæringsstrategi kaldet *border cut* præsenteres sammen med forskellige tilgange til hvordan man kan anvende denne strategi. Værktøjet er designet til nemt at kunne udvides med nye skæringtyper, derfor introduceres der også andre skæringsstyper som potentielle udvidelser.

RAISE Specification Language (RSL) er brugt til at formulere specifikationen og designet af softwaren. At udvikle specifikationer fra abstrakt- til konkret niveau har bidraget meget til den overordnede kvalitet af produktet. Desuden er udviklingsprocesser som *test-driven development* (TDD) blevet benyttet til at opnå simpelt design og tillid til produktet.

Slutproduktet er et kommandolinjeværktøj, der accepterer netværk defineret i XML. Værktøjet genererer nye XML-filer, som indeholder delnetværker. Eksperimenter med fiktive og ikke-fiktive netværker viser, at værktøjet kan håndtere netværker af forskellige størrelser, hvilket også giver en meget jævn brugeroplevelse.

Preface

This thesis was prepared at the department of Applied Mathematics and Computer Science at the Technical University of Denmark under supervision of Associate Professor Anne E. Haxthausen, in fulfillment of the requirements for acquiring an M.Sc. in Computer Science and Engineering.

The thesis deals with the development of a decomposition tool supporting the RobustRailS' verification of railway interlocking systems.

The project period for the thesis was from January 2nd, 2017 to August 4th, 2017.

Kongens Lyngby, August 4, 2017

Cebrail Erdogan

Acknowledgements

I would first like to thank my thesis advisor Professor Anne E. Haxthausen of the Department of Applied Mathematics and Computer Science at DTU. The door to Prof. Haxthausen office was always open whenever I ran into a trouble spot or had a question about my software development or writing. She consistently allowed this paper to be my own work but steered me in the right the direction whenever she thought I needed it.

I would also like to acknowledge Linh Hong Vu and Hugo Daniel at DTU with their assistance on using the verification tool and providing materials.

Finally, I must express my very profound gratitude to my family for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of writing this thesis. This accomplishment would not have been possible without them. Thank you.

Contents

Summary i

Sammenfatning iii

Preface v

Acknowledgements vii

Contents ix

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Goal . . . 1
- 1.3 Thesis Structure . . . 2

2 Background 3
- 2.1 RobustRailS . . . 3
- 2.2 Railway Interlocking System . . . 3
- 2.3 Existing Tools . . . 6
- 2.4 GUI Tool . . . 7
- 2.5 Raise Specification Language . . . 8

3 Analysis 9
- 3.1 Intended use of Tool . . . 9
- 3.2 Cuts and Decomposition Methods . . . 9
- 3.3 Border Cut . . . 10
- 3.4 Border Cut Conditions . . . 11
- 3.5 Soundness Conditions . . . 12
- 3.6 Decomposition Methods . . . 13
- 3.7 Other cut types . . . 17

4 Requirements 21
- 4.1 Functional Requirements . . . 21
- 4.2 Non-Functional Requirements . . . 21
- 4.3 RSL Requirements . . . 22

5 Design 29
- 5.1 RSL Modules Overview . . . 29

5.2 Well-formedness of Cuts 30
5.3 Soundness of Cuts 33
5.4 Decomposition Specifications 35
5.5 XML schema 42
5.6 Class Diagrams 44
5.7 Sequence Diagrams 47
5.8 Adding a New Cut Type 53

6 Implementation & Tests 55

6.1 The C++ Project Structure 55
6.2 Parser 56
6.3 XML Writer 56
6.4 Cloning of networks 57
6.5 Tests 58

7 Experiments 61

7.1 Goal of the Experiments 61
7.2 Experimental Approach 61
7.3 Mini Extended 61
7.4 EDL 63
7.5 Roskilde station 64

8 Discussion 67

8.1 Results 67
8.2 Limitations 67
8.3 Directions for Future Work 68

9 Conclusion 69

A Installing the tool 71

A.1 Prerequisites 71
A.2 Building 72

B Using the tool 73

B.1 Prerequisites 73
B.2 Usage 73

C Networks and Cuts in XML 75

C.1 Original network 75
C.2 Cut Specifications 76
C.3 Sub-networks 78

D Tests 87

D.1 Decomposition_TEST 87
D.2 RSL Results 92
D.3 C++ Results 93

E RSL Specifications **95**
E.1 Decomposition_DESIGN 95
E.2 Decomposition_COMMON 97

F C++ Code **107**
F.1 Cut Types 107
F.2 Main decomposition files 110

Bibliography **115**

CHAPTER 1
Introduction

The overall goal of this project is to provide a command line tool, that can be used for decomposition of future Danish interlocking systems. In this chapter, we will motivate for this, explain the goal of the project, and at last give an overview of report structure.

1.1 Motivation

The Danish Railway system is moving towards a new digital interlocking system [10], along with the rest of the Europe. With that said, it is only expected that new tools are introduced for a more integrated system in the future compatible with the standardized **E**uropean **T**rain **C**ontrol **S**ystem (ETCS) Level 2 [15].

An interlocking system controls the dynamic allocation of routes for trains and thereby creating a safe system where trains can safely travel through the network without colliding and derailing. The overall purpose of the new interlocking system is to make the interlocking safer and smarter.

The Danish Council for Strategic Research has funded the project *RobustRailS*[12]. One of its goals is to provide methods and tools for formal verifications of railway interlocking systems. DTU has an active part of this project and amongst its contributions, it does also develop software.

A verification tool for interlocking systems is already in existence in the form of a command line interface (CLI), developed as part of the RobustRailS project in DTU [13]. Although the tools exist, the verification can be a problem when processing large networks as it may fail to give results due to the state space explosion problem, often seen in model-checkers [6]. A temporary workaround for this has been to decompose networks defined in XML files manually. This process is not user-friendly and takes too much time.

1.2 Goal

The goal of this project is to tackle the state space explosion problem by providing a decomposition tool which converts big networks into smaller chunks. It will provide a user-friendly experience that will make it possible to generate sub-networks easily from a given cut specification.

The decomposition tool must preserve the network safety properties when applying cuts. This ensures that the sub-networks are valid representations that can be used

with the verification tool. The developed tool should be extendible, such that a developer may with ease add new cut types.

A tool that achieves these goals will support the compositional verification by providing a decomposition solution.

1.3 Thesis Structure

Chapter 1: Introduction This chapter introduces the reader to the motivation behind the project along with the goal it accomplishes.

Chapter 2: Background This chapter provides background information about the RobustRailS project and the domain of interlocking systems. The already existing tools are also presented here.

Chapter 3: Analysis This chapter contains an analysis of cut types and decomposition methods. A cut type called *border cut* is analyzed to give the reader a complete understanding of how it can be applied using different decomposition methods.

Chapter 4: Requirements This chapter covers all requirements, from abstract functional and non-functional requirements to concrete RSL requirements.

Chapter 5: Design In this chapter, the design of the solution is presented with focus on satisfying the established requirements.

Chapter 6: Implementation & Tests The implementation process and the technical solutions of the design are described. Some examples of tests and their results are also described in this chapter.

Chapter 7: Experiments This chapter describes experiments of the decomposition tool.

Chapter 8: Discussion This chapter contains a discussion of the obtained results from the experiments. Furthermore, the limitations and potential improvements are discussed.

Chapter 9: Conclusion This chapter concludes the paper.

CHAPTER 2

Background

2.1 RobustRailS

Robustness in **Rail**way Operation**S** is a large interdisciplinary project which tries to discover if we can get the trains to run on time [2]. The project aims to develop systems that provide robustness in the planning of, and during, railway operations. RobustRailS can have a enormous impact on people traveling by train, as it may potentially save them precious time. Funded by a large grant from the Danish Council for Strategic Research, four departments here at DTU are carrying out RobustRailS.

RobustRailS has as one of its goals to develop a holistic method, which in turn will support the verification of the interlocking systems based on the safety properties. This new signaling system, which will be deployed in the period 2009-2021 complies with the standardized European Train Control System (ETCS) Level 2 [15].

2.2 Railway Interlocking System

Interlocking systems of the new signaling systems will ensure that all dynamically allocated train routes are to be fulfilled without violating any safety properties. It creates a safe system where trains can safely travel through a network without colliding and derailing.

A domain specific language (DSL) has already been developed as part of the holistic method, that is used to verify the interlocking systems [15]. Its primary function is to increase the productivity and significantly reduce errors in the specifications of railway interlocking systems. A DSL is used to describe the interlocking system and its operational environment.

A specification is the top most element in the DSL. It consists of two elements, a network layout, and an interlocking table. A combination of the two provides all the necessary data to specify an interlocking system. A network layout focuses on the physical alignment of the elements, and the interlocking table contains the supporting data of the routes.

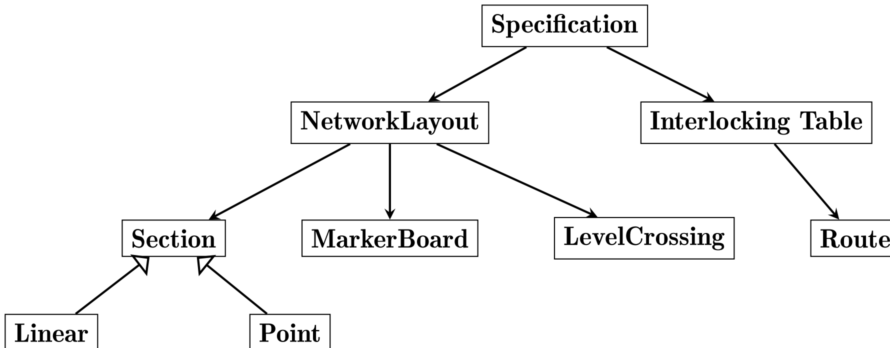


Figure 2.1: Domain specific language of the interlocking system. Solid arrows points to sub-elements and hollow arrows points to base elements.

The specification elements in this domain specific language are described in the subsequent subsections. For simplicity, we do not consider *level crossing* since it has no relevance in this dissertation.

2.2.1 Network Layout

A network layout revolves around the placement of elements and their properties, e.g. neighboring. A network layout specification contains *linears*, *points* and *marker boards*. A network layout has two directions, *up* and *down*. The *up* direction is represented as the right direction in figures and *down* in the opposite direction. See the network layout *mini* in Figure 2.2 as a simple example of a network layout. The lines seen in the figure represents bi-directional track sections which can only be occupied by one train at a time. The short vertical lines with gaps in between represent connections/neighboring.

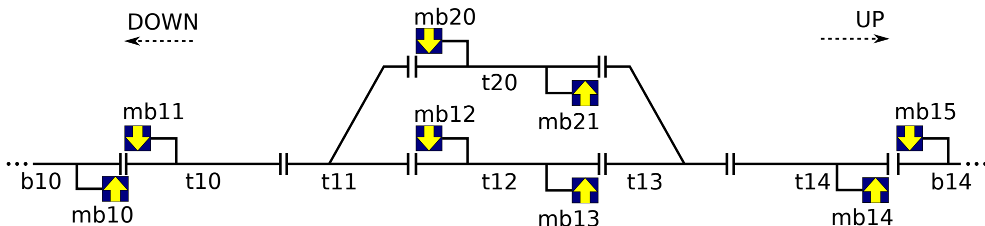


Figure 2.2: An example of network layout called *Mini* [7].

A *linear* is a type of section and contains at most two neighbors, one in every end. The linear sections are represented as straight lines. If a linear only contains one neighbor, then it is called a *boundary*, which can be identified by the dashed lines on its disconnected side. A linear has only two properties, its neighbors, and its length. In *Mini* layout (Figure 2.2), the linear sections are *b10*, *t10* *t20*, *t12*, *t14* and *b14*,

where *b10* and *b14* are boundaries.

A *point* is also a type of section and contains three ends connected to different sections. The three ends are known as *stem*, *plus* and *minus*. The stem is the lone end in the opposite direction of plus and minus. The plus end is parallel to the stem. The plus- and stem end can be seen as a linear section as they represent the straight sub section of a point. The minus end of a point diverts from away this linear section and creates a branch in the network. A point also has two properties, its neighbors, and its length. The points in *Mini* layout are *t11* and *t13*.

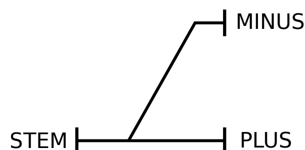


Figure 2.3: A point with its three ends. A point can face any direction.

It is possible for two different points to have their minus ends connected to each other. An example is shown in Figure 2.4.

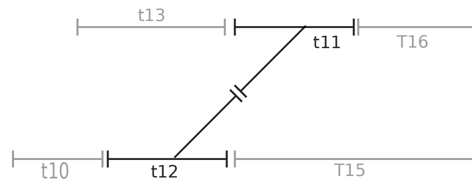


Figure 2.4: Two points connected to each other. An excerpt of Figure 3.19.

A *marker board* also called a *signal*, is mounted in one of the ends of a linear section. It can only be seen in one direction. If its direction is *up*, then it can only be seen by a train moving the same direction and vice versa. A marker board can not exist on the disconnected side of a boundary. Additionally, it has a distance to the end it is facing. So, in total, a marker board has three properties: A reference to a linear section, a direction, and a distance.

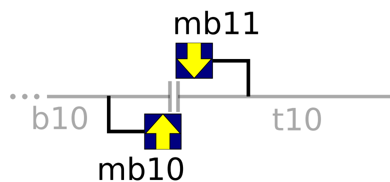


Figure 2.5: Marker boards with different directions. The figure is a excerpt from *Mini* layout.

2.2.2 Interlocking Table

An interlocking table contains information about the routes. The routes are listed as rows with their values on the columns. A route is a collection of trackside elements contained within a specific network. It has a path going from a source marker board to a destination marker board. If no marker boards with the same direction are located between the source marker board and the destination marker board, then we are dealing with an elementary route. The routes in an interlocking table are all of this type. As stated in [6], a route *r* currently consists of:

src(r) The source signal of r, given as a marker board.

dst(r) The destination signal of r, given as a marker board. Both the source and destination must face the same direction, i.e. UP or DOWN.

path(r) The list of sections along the path from src(r) to dst(r).

overlap(r) A list of sections in r's overlap. An overlap is a buffer space after dst(r) that is used in case a train overshoots its path.

points(r) The points used by r and their required positions (MINUS or PLUS).

markerboards(r) A set of protecting marker boards used for flank or front protection for the route.

conflicts(r) A set of conflicting routes which must not be set while r is set.

id	src	dst	path	points	marker boards	conflicts
1a	mb10	mb13	t10;t11;t12	t11:p;t13:m	mb11; mb12; mb20	1b; 2a; 2b; 3; 4; 5a; 5b; 6b; 7
...	...	...	...	...	...	...
7	mb20	mb11	t11; t10	t11:m	mb10;mb12	1a; 1b; 2a; 2b; 3; 5b; 6a

Table 2.1: An excerpt of an interlocking table for the network layout in Figure 2.2 [6]. The overlap column is ommited, since no overlaps exists in this table.

2.3 Existing Tools

The RobustRailS project has resulted in multiple products. In this section, the existing tools that are described.

2.3.1 RobustRailS Verification Tool

Currently, RobustRailS has a verification tool that can verify any given interlocking system. A toolchain is used to accomplish this task which can be seen in Figure 2.6 [14].

The verification tool, designed as a command line tool, is developed by Linh Vu [13]. The tool accepts configurations in XML, where a configuration consists of a network and an interlocking table.

As it can be seen in Figure 2.6, the tool chain has several steps. In the first box (yellow), the given network and the generated interlocking table are statically checked. This verifies if the input is well formed or not, if not, then the correctness of the network is not checked. It is a requirement that the given specification must comply with the static semantic rules defined for the DSL. Note that the generation of an interlocking table is optional.

In case of a successful static check, a model instance will be generated from a generic model which describes the dynamic behavior of the Danish interlocking system.

Concurrently, safety properties are generated from given specification, which runs through the defined safety-properties.

Finally, the generated model instance is verified against the safety properties using a combination of bounded model checking (BMC) and inductive reasoning. If the model instance does not satisfy the properties, counterexamples will be generated.

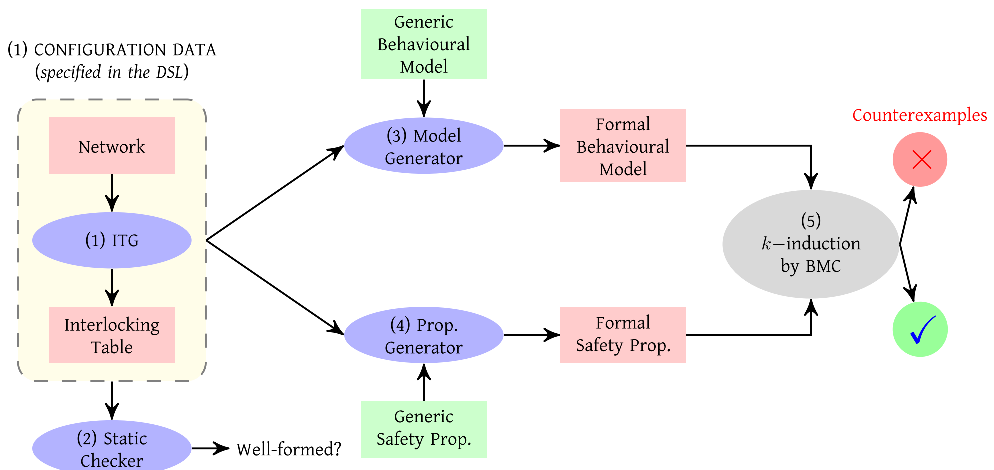


Figure 2.6: The tool-chain of the verification system.

2.4 GUI Tool

The GUI tool is developed by Andreas Foldager [5] using the Eclipse platform. It provides a visual environment for both creating networks, generating route tables and running the verification tool. The networks are created using a visual editor, and the verification tool can be directly triggered from the properties panel which will show the result in the same panel.

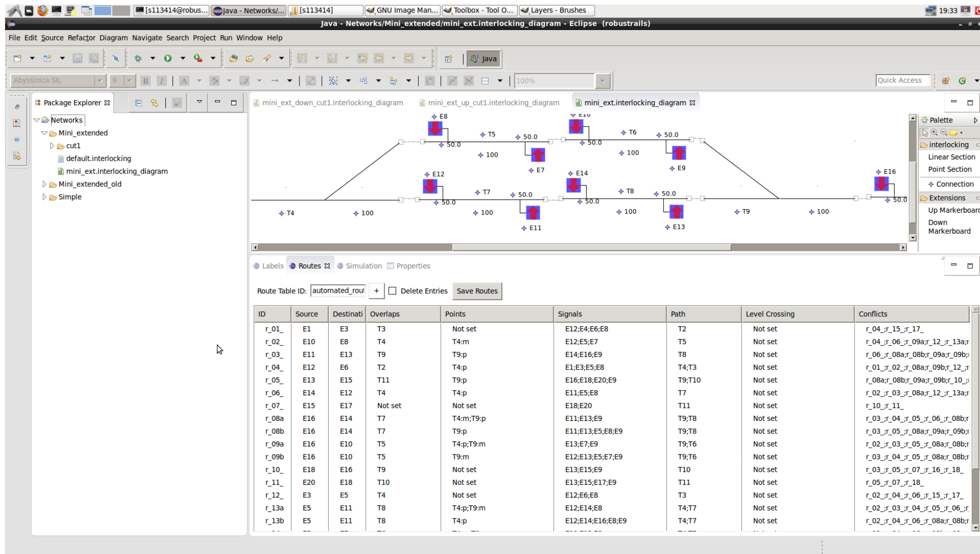


Figure 2.7: A screenshot of the Eclipse GUI tool.

2.5 Raise Specification Language

The specification of the developed tool in this project uses the RAISE formal method which consists of RAISE development method and RSL.

RAISE (Rigorous Approach to Industrial Software Engineering) was developed for the use on the LaCoS project, including other industrial and research projects [1].

The Raise Specification Language (RSL) is a wide-spectrum specification language. It is capable of expressing high-level, abstract specifications, as well as low-level designs. RSL specifications can be considered as mathematical models of software systems.

The advantage of using RAISE is that it has a huge contribution to the design of the product. It gives the developers an early insight of the design and encourages simpler implementation in later stages. Knowing the notation of RSL alongside the implementation language is all that it takes.

CHAPTER 3

Analysis

In this chapter, the role of the decomposition tool is analyzed in the context of how it can fit the existing toolchain and what it should be capable of. Furthermore, cut types and decomposition methods are analyzed.

3.1 Intended use of Tool

The decomposition tool can seamlessly be used with the existing tool chain. By creating the tool as a standalone, it will have flexibility due to separation from the verification tool, moreover, easy testing during development, since the outcome of the decomposition tool can be tested with the already existing verification tool-chain. A more in-depth integration with the tool-chain is possible but is more reasonable after the decomposition tool is tested on the field for some time.

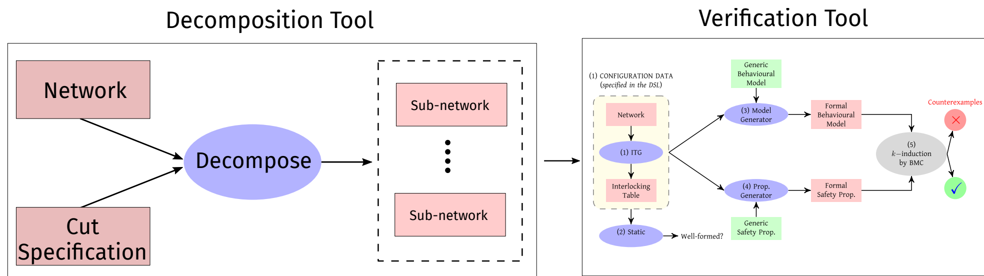


Figure 3.1: Integration with existing tool-chain.

The decomposition tool accepts network layout and a cut specification in XML-format. The network and cut specification will be parsed, and the network will be decomposed. This might result in two or more sub-networks, which must all be verified with the verification tool.

3.2 Cuts and Decomposition Methods

Before introducing the *cuts* and *decomposition methods*, a clear distinction between these two is needed. A cut also referred as a *divide strategy*, is a set of operations that are applied at a given place in a network with the purpose of disconnecting without losing the safety properties. A cut specification contains information about

the properties of the cut. The described cuts in this paper are *border cut*, *linear cut*, and *horizontal cut*. Only *border cut* is implemented of the listed.

A decomposition method, on the other hand, takes a cut, or a set of cuts as an input. It applies the cuts with a specified algorithm that is suited for a specific scenario. The described decomposition methods in this paper are: *single cut (decomposition)*, *cluster cut (decomposition) and multi cut (decomposition)*.

The description here focuses on the application of the cuts and methods, and not their type specifications, where the differences between them are not as clear.

3.3 Border Cut

A *border cut* cuts between two linear sections [7]. Therefore, it can simply be specified by two sections, *section up* and *section down*. When a border cut happens, both sections are preserved. However, two new boundaries are created, one belonging to the sub-network *down* and one to the sub-network *up*. In the down network, the *section up* is converted to a boundary, and in the up network, the *section down* is converted. If the new boundaries contain invalid marker boards, then they are removed too. Figure 3.2 shows a border cut on a minimal network possible. In this example, the *section down* is `t2` and the *section up* is `t3`.

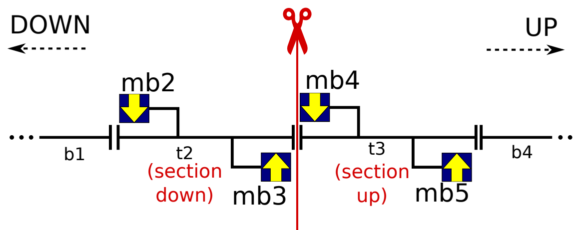


Figure 3.2: A valid border cut.

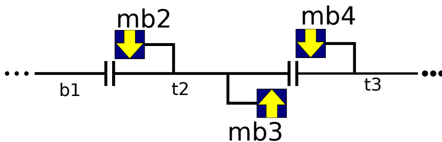


Figure 3.3: *Down* sub-network.

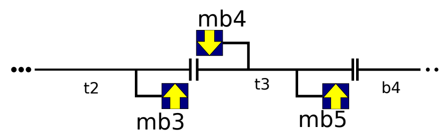


Figure 3.4: *Up* sub-network.

Notice that the marker board *mb5* is removed in *down* sub-network and *mb2* is removed in *up* sub-network since they are no longer valid. The two given linear sections (*t2* and *t3*) are preserved in both sub-networks and converted to boundaries respectively to the sub-network they are placed.

3.4 Border Cut Conditions

For a border cut to be well-formed, the border cut conditions must be satisfied. Here well-formedness is equivalent to syntactically correct. Four pre-conditions have been defined for this particular type of cut. These conditions make sure that the cuts are valid and that they will result in valid networks.

BCC 1

The two cut sections (*down* and *up*) must be neighbors. If the two given sections are not neighbors, then the input is invalid since there is no way of telling where to perform the cut.

BCC 2

The up- and down sections must not already be boundaries.

BCC 3

The two sections (*down* and *up*) must not be in the same elementary route. Therefore, the down section must contain an up marker board, and the up section must contain a down marker board. Where up and down are the directions they face.

BCC 4

The sections up and down must not be reachable to each other if disconnected. In that case, a cut will not result in two networks. Figure 3.5 shows a cut representing this case, here, a cut will disconnect `t4` and `t5`, but the other branch will make the all sections still reachable - making the cut invalid (not well-formed).

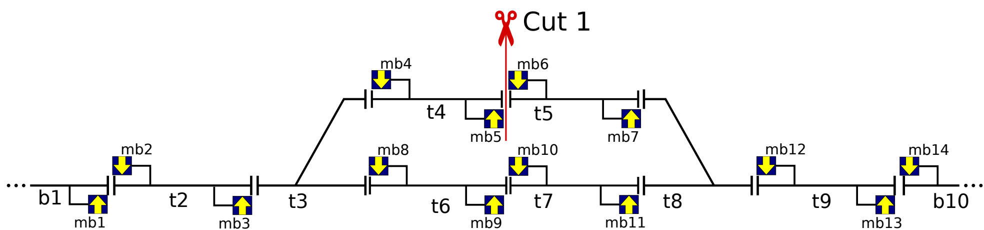


Figure 3.5: An invalid cut. The result will be two not well-formed sub-networks since all sections are still reachable.

3.5 Soundness Conditions

Soundness conditions are checked after the sub-networks are generated from a decomposition. The used cut type or decomposition method does not matter. The purpose of the soundness checks is to check whether sub-networks still satisfy the safety properties regarding flank and front protection. These safety properties in the networks prevent head-to-head and flank collisions.

If the sub-networks do not satisfy the safety properties, the sub-networks will reveal dependencies on each other. These dependencies can be found by generating interlocking tables for the sub-networks.

If the soundness conditions are satisfied, then the sub-networks are sound and do not contain violation of flank/front protection. If soundness conditions are not satisfied on the sub-networks, then it can for sure be said that either the network or the cut was invalid.

Four soundness rules exist, and they can only be checked after a cut has been performed as they require an interlocking table to be generated from the sub-networks. A specification for interlocking generation from a network layout is already supplied by Vu [13]. It would have been preferable to be able to check for these conditions as pre-conditions, but the need for interlocking tables for sub-networks makes that impossible. The procedure for checking the SCs (Soundness Conditions) are very similar. The network is initially decomposed, and the individual tables are generated, after that, the routes are checked for dependencies on other tables or sub-networks.

SC 1

This condition checks for overlap requirements. The overlap buffer must not be less than the minimum safety distance. If any overlap section exists, it must reside in the generated interlocking table or the sub-network it was based on.

SC 2

This condition checks if the points in the routes only reside in the same interlocking table or the sub-network it was based on.

SC 3

This condition checks if the marker boards in the routes only resides in the generated interlocking table or the sub-network it was based on.

SC 4

This condition checks if the conflicts in routes are independent. All conflicting routes must reside in the same interlocking table.

3.6 Decomposition Methods

In this section, different methods to decompose using border cuts are presented. The intention is to easily be able to decompose networks in different scenarios. The scenarios to consider are:

Scenario 1 A network which needs to be decomposed between two stations, where the stations are connected by single linear section.

Scenario 2 A network which needs to be decomposed between two stations, e.g. in a network where the stations are connected by two or more consecutive linear sections. In this scenario, all sections are in the same branch.

Scenario 3 A network that needs to be decomposed in different levels of branches resulting in two sub-networks, e.g. in a network two stations are connected by linear sections side-by side running in parallel.

Scenario 4 An arbitrary large network that needs multiple decompositions in one execution resulting in more than two sub-networks.

Scenario 5 A network that needs to be decomposed at the *minus* end of two connected points.

The rest of the section goes through decomposition methods that can solve the problems in these scenarios.

3.6.1 Single Cut Decomposition

A single cut decomposition is the simplest of all the decomposition methods. It is an execution of a single border cut, hence the name *single*. The single cut is applicable for a scenario where there are enough consecutive linear sections on the same branch. This makes it applicable in simple networks and networks mentioned in **scenario 2**.

When applying a single cut on a network, two networks are always produced. The Figure 3.6 below shows a simple example of a single cut. Notice that marker boards are omitted in the next few figures for simplicity. In this example, the single cut is defined by `sc:{t2,t3}`.

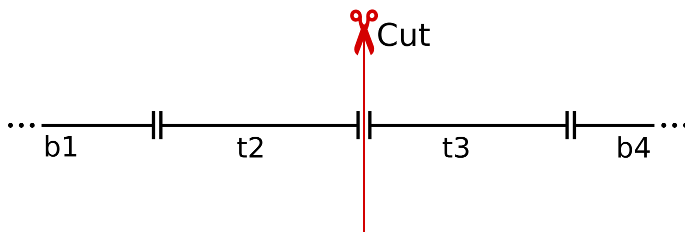


Figure 3.6: Single cut on a simple network.

The single cut decomposition method generates the sub-networks *down* and *up* which can be seen in Figure 3.7.

(a) *Down* network. **(b)** *Up* network.

Figure 3.7: Sub-networks of a single cut decomposition on a simple network.

As mentioned, the single cut decomposition method is applicable for scenario 2. An example of this scenario can be seen Figure 3.8. It is very common to see stations to be connected in this way. Decomposing between stations can be a very practical since every sub-network can be interpreted as an independent network that is not difficult to examine.

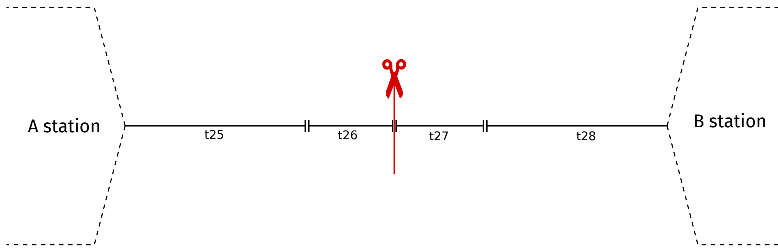
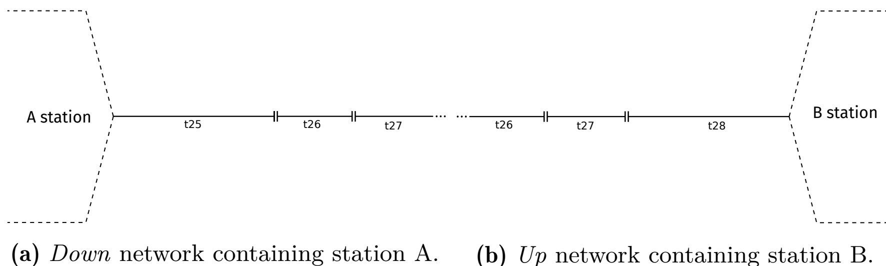


Figure 3.8: Network containing both stations.



(a) *Down* network containing station A. **(b)** *Up* network containing station B.

Figure 3.9: Decomposition between two stations.

A single cut may, later on, be extended with new cut types other than border cut. This can enable the user to use this decomposition method in different scenarios opposed to only being restricted to scenario 2. This topic is further analyzed in Section 3.7.

3.6.2 Cluster Cut Decomposition

A single cut can, however, be exploited such that it can be used in other scenarios. This is what cluster cut decomposition does. It contains multiple single cuts which it executes to produce two sub-networks. It only produces two well-formed networks if all of the cuts are executed. A partial execution of cluster cut will not result in a well-formed network. Cluster cut makes it possible to apply cuts on different levels of branches to achieve two sub-networks. See Figure 3.10 for an example of this case, where the cluster cut is defined by `cc:{sc:{t4,t5};sc:{t6,t7}}`.

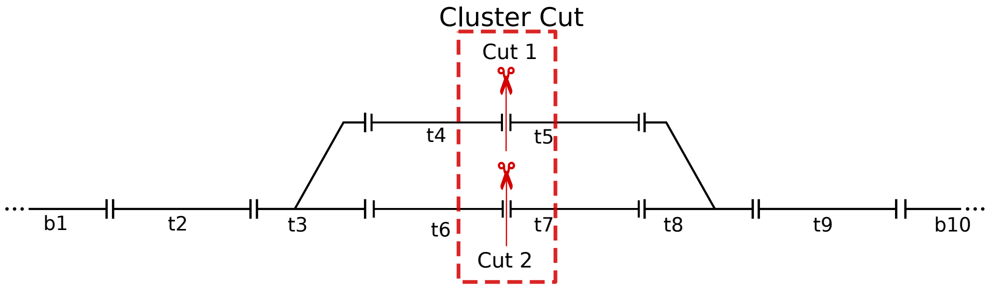
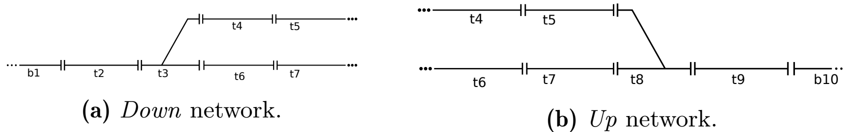


Figure 3.10: Cluster cut on the *mini* network layout.

The cluster cut decomposition method generates the sub-networks *down* and *up* which can be seen in Figure 3.11.



(a) *Down* network.

(b) *Up* network.

Figure 3.11: Decomposition of the *mini* network layout.

As it can be seen the cluster cut is applicable for the **scenario 3** and just like single cut, it can also be applied between stations. Figure 3.12 shows an example of this, station *A* and station *B* is connected by two parallel linears sections that are running side-by-side.

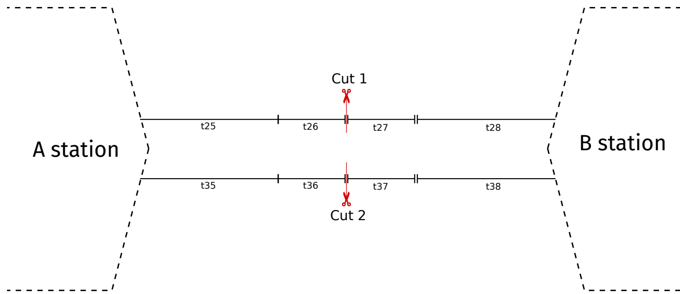
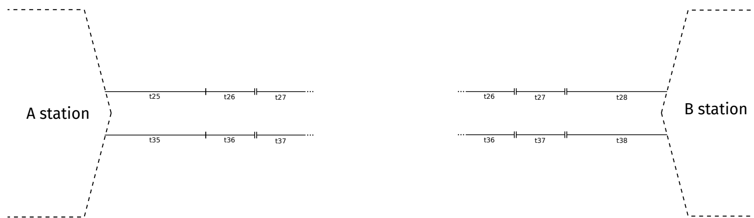


Figure 3.12: Network containing both stations.



(a) *Down* network containing station A. **(b)** *Down* network containing station B.

Figure 3.13: Decomposition between two stations.

3.6.3 Multi cut Decomposition

A general decomposition method can be designed that can take a set of cuts, containing a mixture of both single- and cluster cuts. This allows us to define more complex cut specifications that will serve as input for the decomposition function. The result of running a decomposition of such input can produce more than two networks. The number of outputs depends on the given number of cuts. A decomposition method of this kind is applicable for all the scenarios that single- and cluster cut covers. Besides, it does also cover **scenario 4**, making it easy to decompose large networks in one execution. See Figure 3.14 for an example of a multi cut, which is defined by `{sc:{t2,t3},cc:{sc:{t4,t5};sc:{t6,t7}}}`.

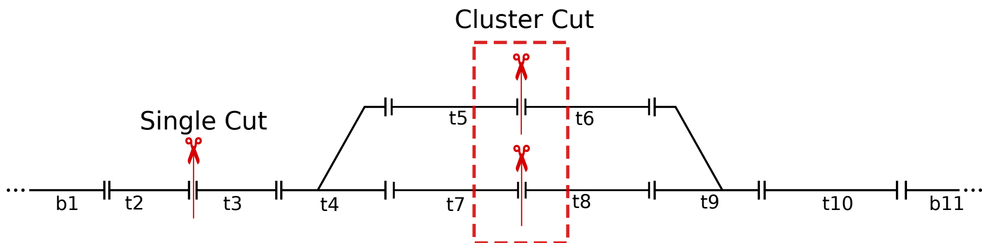
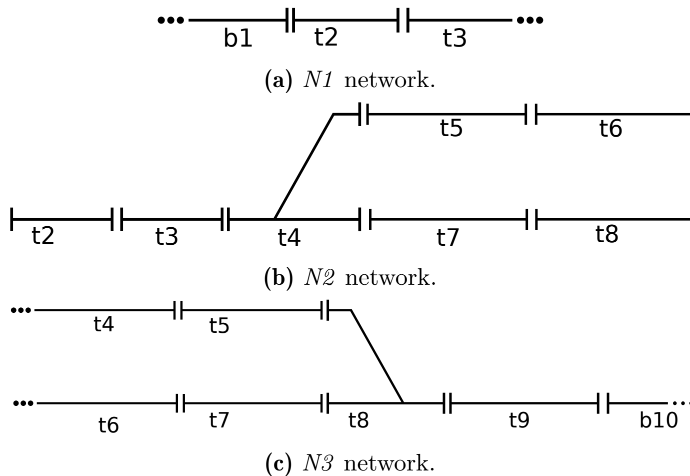


Figure 3.14: A multi cut on a extended version of *mini*.

This particular multi cut decomposition generates three sub-networks. These sub-networks are *N1*, *N2*, and *N3* which can be seen in Figure 3.15.



(a) *N1* network.

(b) *N2* network.

(c) *N3* network.

Figure 3.15: The three sub-networks resulted by a multi cut decomposition method.

3.7 Other cut types

Even though decomposition methods with border cut covers must of the scenarios listed, you may have noted that **scenario 1** and **scenario 5** were not applicable. For this reason, new cut types are under development that targets these scenarios. These new cuts are described in the next subsections.

Linear Cut

Linear cut is introduced in the paper [8] and is similar to border cut. It is used when the routes of the two sub-models partially overlap. The linear cut needs only one linear section in common for the sub-networks *down* and *up*, while a border cut needs two. Sometimes there are simply not enough linear sections between stations to apply a border cut. Linear cut is a good solution for these cases, e.g. **scenario 1**. Figure 3.16 shows a case where a linear cut is applicable. `T2` can be specified as the cut specification. A cut here will add new boundaries to the `T2` section. Marker boards will also be added to ensure sane boundaries.

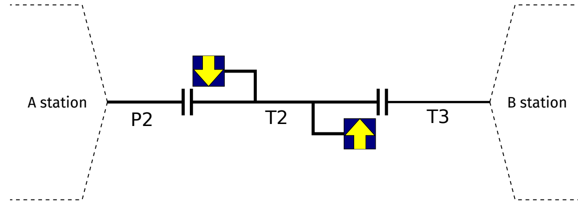


Figure 3.16: The linear section `T2` is connected to two stations through P2 and T3.

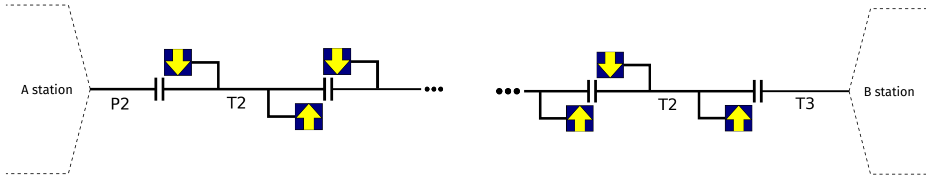


Figure 3.17: Resulting *Down* network

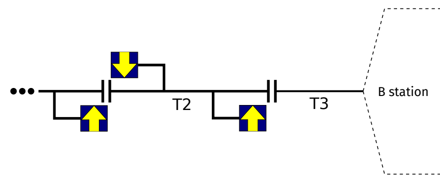


Figure 3.18: Resulting *Up* network

Horizontal cut

Horizontal cut is another cut that is under development and is introduced in the paper [4]. It will be capable of cutting between two minus ends of points, thus targeting **scenario 5**. How it does this is best explained with an example. Figure 3.19 is based on a figure from the paper regarding this particular cut. A horizontal cut is performed on the shown red line, crossing the link between `t9` and `t10`. To each of the two sub-networks, above the cut and below it, two consecutive linear section is added on the other side of the cut forming a *stub*. This abstracts the whole other sub-network. The new sections have marker boards to satisfy the network conditions. The two resulting sub-networks are shown in Figures 3.20 and 3.21. Notice that the *high* and *low* names are used because *down* and *up* are reserved for network directions.

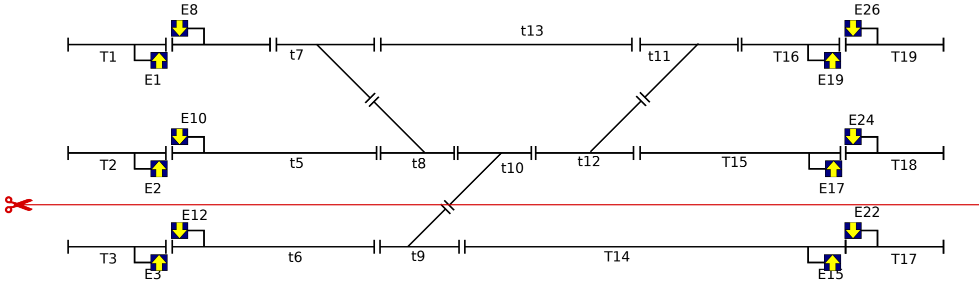


Figure 3.19: The total network and the cut.

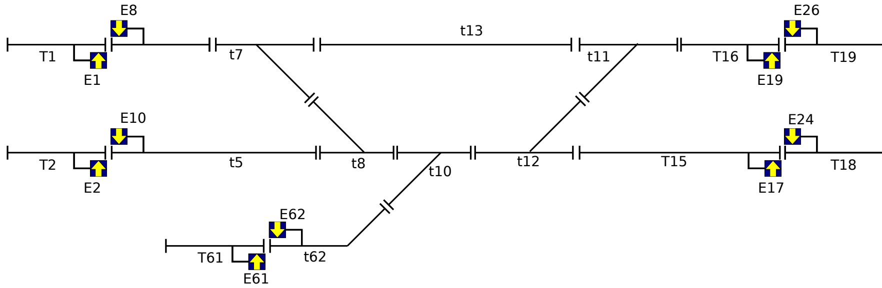


Figure 3.20: The *high* sub-network.

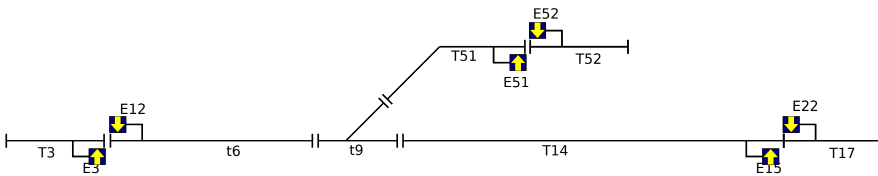


Figure 3.21: The *low* sub-network.

CHAPTER 4
Requirements

This chapter covers all requirements, from abstract functional and non-functional requirements to concrete RSL requirements.

4.1 Functional Requirements

FR01: Create Specifications This requirement is going to affect the approach to how we design the product. RSL will be used for this requirement.

FR02: CLI Tool Create an exclusive command line interface tool that takes a network XML file and creates new XML files from decomposition results. This tool must support *border cut* all of the decomposition methods introduced in Chapter 3.

FR03: XML Support The tool must be able to parse networks, interlocking tables and cut specifications in XML. An XML writer for the same objects must also be supported.

FR04: Extendibility New cut types must easily be implementable by design.

4.2 Non-Functional Requirements

QR01: Linux OS Support The tools must be compatible mainly with OS system Linux. Compatibility with other operating systems will be considered as an advantage but not a requirement.

QR02: Reliable The decomposition tool must be reliable and able to handle networks in all sizes.

4.3 RSL Requirements

In this section, the type definitions are initially described using RSL. After that, decomposition method requirements are listed as well. The requirements here are going to be presented as pre- and post-conditions. The conditions are wrapped in predicate functions for simplicity.

4.3.1 Types Specification

The interlocking types are already defined by Vu in [13] as part of his project. The cut types are defined in this project. The full RSL code of the cut types can be seen in Appendix E.2.

Interlocking Types

The basic elements, *sections*, *marker boards* and *routes* are all given a unique id, therefore an identifier is supplied for all of them.

```
Id = Text,
SecId = Id,
MbId = Id,
RouteId = Id
```

Network Layout

A network layout is represented as maps of *linears*, *points* and *marker boards*. Each map contains identifiers as key values mapping to its element.

```
NetworkLayout ::
    linears : SecId -m-> Linear
    points : SecId -m-> Point
    marker_boards : MbId -m-> MarkerBoard
```

A *linear* is a record containing its neighbors and length.

```
Linear ::
    neighbors : LinearEnd -m-> SecId
    length : Distance
```

Where `LinearEnd` contains the direction of the neighbor and the `Distance` contains the distance of the linear as a natural number.

```
LinearEnd = Direction,
Distance = Nat,
Direction == UP | DOWN
```

A *point* is also a record containing its neighbors and length. However its neighbors are defined with `PointEnd`.

```
Point ::
    neighbors : PointEnd -m-> SecId
    length : Distance

PointEnd == NB_STEM | NB_PLUS | NB_MINUS,
```

Lastly, a *marker board* element is defined by which linear section it is mounted on, which direction it is facing and its distance to the end of linear it is facing.

```
MarkerBoard ::
    section : SecId
    dir : LinearEnd
    distance : Distance
```

Interlocking table

An interlocking table specifies the elementary routes for a specific network. See Section 2.2.2 for more details about it. A *route* is specified as a short record definition with all of the columns included.

```
Route ::
    source : MbId
    dest : MbId
    path : SecId-list
    overlap : SecId-list
    points : SecId -m-> PointPos
    signals : MbId-set
    conflicts : RouteId-set,
InterlockingTable = RouteId -m-> Route
```

A record containing both the network layout and the interlocking table is simply called an `Interlocking`.

```
Interlocking ::
    track_layout : L.NetworkLayout
    interlocking_table : InterlockingTable
```

Cut Types

The border cut, presented in Chapter 3, can be defined given two sections.

```
BorderCut  ::
    section_down:SecId
    section_up:SecId
```

A single cut specification can be defined by the cut it executes. If more cuts have to be included, then single cut has to be extended. Section 5.8 shows an example of this.

```
SingleCut = BorderCut
```

A cluster cut is a set of single cut types.

```
ClusterCut = SingleCut-set,
```

A multi cut specification is a set of cut type specifications.

```
MultiCut = Cut-set,
```

Where a cut element in the set is either a cluster cut or a single cut.

```
Cut ==  Cut_from_SingleCut(cut_to_singlecut: SingleCut) |
        Cut_from_ClusterCut(cut_to_cluster: ClusterCut)
```

4.3.2 Decomposition Requirements

The requirements for different decomposition methods are presented here as RSL specification. The main focus here is to determine the state of a network before and after a decomposition. The well-formedness check functions that appear in the listings, such as the `cut_wf`, are defined later in Section 5.2. The overloading feature of RSL is used to define the function `decompose` multiple times for different decomposition methods.

Single Cut Decomposition Requirements

The single cut decomposition can be defined to have the inputs and outputs:

```
decompose:  C.ITG.I.L.NetworkLayout >< C.SingleCut -~->
            C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
decompose(n,sc) as (n_down,n_up)
```

It has the pre-conditions:

1. The network and the cut specification must be well formed.

```
pre  C.ITG.I.L.is_wf(n) /\ C.cut_wf(n,sc),
```

The post requirements are defined in the predicate function:

```
post post_single_cut_decomposition(sc,n,n_down,n_up)
```

Which contains the following requirements:

1. The union of n_{down} (sub-network *down*) and n_{up} (sub-network *up*) elements must be the same elements from n (Original network).

```
dom n_down_linears      union dom n_up_linears = dom n_linears /\
dom n_down_points       union dom n_up_points    = dom n_points   /\
dom n_down_markerboards union dom n_up_markerboards = dom n_markerboards  /\
```

Where `n_down_linears`, `n_down_markerboards` and `n_down_points` are the respective elements from n_{down}. The same naming principle holds for n_{up} and n. These local names are assigned as seen in RSL snippet below.

```
n_linears =  C.ITG.I.L.linears(n),
n_points =  C.ITG.I.L.points(n),
n_markerboards =  C.ITG.I.L.marker_boards(n),

n_down_linears =  C.ITG.I.L.linears(n_down),
n_down_points =  C.ITG.I.L.points(n_down),
n_down_markerboards =  C.ITG.I.L.marker_boards(n_down),

n_up_linears =  C.ITG.I.L.linears(n_up),
n_up_points =  C.ITG.I.L.points(n_up),
n_up_markerboards =  C.ITG.I.L.marker_boards(n_up)
```

The prefixes in the listings such as `C.ITG.I.L` are used to access modules and are described in Chapter 5.

2. The intersection of n_{down} and n_{up} elements must yield the linears from the cut specification and the marker boards placed on them. The marker boards facing the boundary sides are not included because they are removed in the decomposition process. The intersection of points must not result in any elements.

```
dom n_down_linears inter dom n_up_linears = {l_down, l_up} /\
dom n_down_markerboards  inter dom n_up_markerboards  =
    {C.ITG.I.L.signals(l_down,n)(UP), C.ITG.I.L.signals(l_up,n)(DOWN)} /\
dom n_down_points inter dom n_up_points  = {} /\
```

3. All elements in n_{down} must be the very same elements in n, except `l_up` which is the new boundary.

```
(all l:SecId :- l isin n_down_linears \ {l_up} => n_down_linears(l) = n_linears(l)) /\
(all p:SecId :- p isin C.ITG.I.L.points(n_down) =>
    C.ITG.I.L.points(n_down)(p) = C.ITG.I.L.points(n)(p)) /\
(all mb:SecId :- mb isin C.ITG.I.L.marker_boards(n_down) =>
    C.ITG.I.L.marker_boards(n_down)(mb) = C.ITG.I.L.marker_boards(n)(mb)) /\
```

4. For the new boundary `l_up` at n_{down}, the following must hold.

```
UP ~isin C.ITG.I.L.signals(l_up,n_down) /\
UP ~isin C.ITG.I.L.neighbors(n_down_linears(l_up)) /\
C.ITG.I.L.neighbors(n_down_linears(l_up))(DOWN) =
     C.ITG.I.L.neighbors(n_linears(l_up))(DOWN)  /\
C.ITG.I.L.length(n_down_linears(l_up)) =  C.ITG.I.L.length(n_linears(l_up))  /\
```

The *up* neighbor and the marker board mounted at `UP` must no longer exist. The length and neighboring properties of `l_up` in *n_down* must match the properties of `l_up` in n.

5. All elements in n_{up} must be the very same elements in n, except `l_down` which is the new boundary.

```
(all l:SecId :- l isin n_up_linears \ {l_down} => n_up_linears(l) = n_linears(l)) /\
(all p:SecId :- p isin C.ITG.I.L.points(n_up) =>
    C.ITG.I.L.points(n_up)(p) = C.ITG.I.L.points(n)(p)) /\
(all mb:SecId :- mb isin C.ITG.I.L.marker_boards(n_up) =>
    C.ITG.I.L.marker_boards(n_up)(mb) = C.ITG.I.L.marker_boards(n)(mb)) /\
```

6. For the new boundary `l_down` at n_{up}, the following must hold.

```
DOWN ~isin C.ITG.I.L.signals(l_down,n_up)  /\
DOWN ~isin C.ITG.I.L.neighbors(n_up_linears(l_down)) /\
C.ITG.I.L.neighbors(n_up_linears(l_down))(UP) =
   C.ITG.I.L.neighbors(n_linears(l_down))(UP)  /\
C.ITG.I.L.length(n_up_linears(l_down)) =  C.ITG.I.L.length(n_linears(l_down))  /\
```

The *down* neighbor and the marker board mounted at `DOWN` must no longer exists. The length and neighboring properties of `l_down` must match the properties of `l_down` in n.

7. Finally, the sub-networks must be well-formed.

```
D.ITG.I.L.is_wf(n_down) /\ D.ITG.I.L.is_wf(n_up)
```

Cluster Cut Decomposition Requirements

The cluster cut decomposition can be defined to have the inputs and outputs:

```
decompose:  C.ITG.I.L.NetworkLayout >< C.ClusterCut --->
            C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
decompose(n,cc) as (n_down,n_up)
```

It has the pre-conditions:

1. The network and the cut specification must be well formed.

```
pre  D.ITG.I.L.is_wf(n) /\ D.cut_wf(n,sc),
```

The post-conditions are defined in the predicate function:

```
post post_cluster_cut_decomposition(cc,n,n_down,n_up)
```

The cluster cut post-conditions are basically the same as single cut requirements, the difference lies in the amount of single cuts that has to be checked. The requirements are the following:

1. The union of n_{down} and n_{up} must have the same elements as n.

```
dom n_down_linears      union dom n_up_linears = dom n_linears /\
dom n_down_points       union dom n_up_points    = dom n_points   /\
dom n_down_markerboards union dom n_up_markerboards = dom n_markerboards   /\
```

They are defined the same way as in single cut requirement.

2. The intersection of n_{down} and n_{up} elements must yield the linears from the cut specifications and the marker boards placed on them. Once again, the marker boards facing the boundary side are not included because they are removed in the decomposition process. The intersection of points does again result in an empty set.

```
dom n_down_linears inter dom n_up_linears = l_downs  union l_ups /\
dom n_down_markerboards inter dom n_up_markerboards  =
  {C.ITG.I.L.signals(l_down,n)(UP) | l_down:SecId  :- l_down isin l_downs} union
  {C.ITG.I.L.signals(l_up,n)(DOWN) | l_up:SecId  :- l_up isin l_ups} /\
dom n_down_points  inter dom n_up_points  = {} /\
```

3. The elements in n_{down} must be the very same elements in n as for the single cut, except `l_ups` which is the set of new boundaries.

```
(all l:SecId :- l isin n_down_linears \ l_ups =>
    n_down_linears(l) = n_linears(l)) /\
(all p:SecId :- p isin C.ITG.I.L.points(n_down) =>
    C.ITG.I.L.points(n_down)(p) = C.ITG.I.L.points(n)(p)) /\
(all mb:SecId :- mb isin C.ITG.I.L.marker_boards(n_down) =>
    C.ITG.I.L.marker_boards(n_down)(mb) = C.ITG.I.L.marker_boards(n)(mb)) /\
```

4. For the new boundaries `l_ups` at n_{down}, the same requirements from single cut must hold.

```
(all l_up: SecId :- l_up isin l_ups =>
    UP ~isin C.ITG.I.L.signals(l_up,n_down)) /\
(all l_up: SecId :- l_up isin l_ups =>
    UP ~isin C.ITG.I.L.neighbors(n_down_linears(l_up))) /\
(all l_up: SecId :- l_up isin l_ups =>
    C.ITG.I.L.neighbors(n_down_linears(l_up))(DOWN) =
         C.ITG.I.L.neighbors(n_linears(l_up))(DOWN)) /\
(all l_up: SecId :- l_up isin l_ups =>
    C.ITG.I.L.length(n_down_linears(l_up)) =  C.ITG.I.L.length(n_linears(l_up))) /\
```

5. The elements in n_{up} must be the very same elements in n, except `l_downs` which is the set of new boundaries.

```
(all l:SecId :- l isin n_up_linears \ l_downs =>
    n_up_linears(l) = n_linears(l)) /\
(all p:SecId :- p isin C.ITG.I.L.points(n_up) =>
    C.ITG.I.L.points(n_up)(p) = C.ITG.I.L.points(n)(p)) /\
(all mb:SecId :- mb isin C.ITG.I.L.marker_boards(n_up) =>
    C.ITG.I.L.marker_boards(n_up)(mb) = C.ITG.I.L.marker_boards(n)(mb)) /\
```

6. For the new boundaries `l_ups` at n_{down}, the same requirements from single cut must hold.

```
(all l_down: SecId :- l_down isin l_downs =>
    DOWN ~isin C.ITG.I.L.signals(l_down,n_up)) /\
(all l_down: SecId :- l_down isin l_downs =>
    DOWN ~isin C.ITG.I.L.neighbors(n_up_linears(l_down))) /\
(all l_down: SecId :- l_down isin l_downs =>
    C.ITG.I.L.neighbors(n_up_linears(l_down))(UP) =
       C.ITG.I.L.neighbors(n_linears(l_down))(UP)) /\
(all l_down: SecId :- l_down isin l_downs =>
    C.ITG.I.L.length(n_up_linears(l_down)) =  C.ITG.I.L.length(n_linears(l_down))) /\
```

7. Finally, the networks must be well-formed.

```
C.ITG.I.L.is_wf(n_down) /\ C.ITG.I.L.is_wf(n_up)
```

Multi Cut Decomposition Requirements

The multi cut decomposition can be defined to have the inputs and outputs:

```
decompose:  C.ITG.I.L.NetworkLayout >< C.MultiCut  -~->
            C.ITG.I.L.NetworkLayout-set
decompose(n,mc) as ns
```

It has the following pre-conditions:

1. The network and the cut specifications in the set must all be well-formed.

   ```
   pre C.ITG.I.L.is_wf(n) /\ C.cuts_wf(n, mc)
   ```

The post-conditions are defined in the predicate function:

```
post post_multi_cut_decomposition(mc,n,ns)
```

The function inherits the same requirements as single- and cluster cuts. It simply uses the relevant post function depending on the cut type.

1. The post requirements of both types must be satisfied.

   ```
   (all c: C.Cut:- c isin mc =>
                case c of
                   C.Cut_from_SingleCut(sc) ->
                       let (n_down,n_up) = decompose(n,sc)
                       in post_single_cut_decomposition(sc,n,n_down,n_up) end,
                   C.Cut_from_ClusterCut(cc) ->
                       let (n_down,n_up) = decompose(n,cc)
                       in post_cluster_cut_decomposition(cc,n,n_down,n_up) end
                end) /\
   ```

2. Finally all of the networks must be well formed.

   ```
   (all n : C.ITG.I.L.NetworkLayout :- n isin ns => C.ITG.I.L.is_wf(n))
   ```

CHAPTER 5

Design

This chapter covers the specification and the design of the command line tool. The design choices made to satisfy the requirements from Chapter 4 are described with RSL specifications and UML diagrams.

5.1 RSL Modules Overview

The RSL modules have a nested structure. A scheme and object can be defined within a class expression. A module that creates an object of a class is represented by an arrow, where the arrow points to the inherited class. In Section 5.1, an overview of the modules can be seen. The Decomposition module has access to all the other modules by creating an object of Interlocking Table Generator module, which inherits other modules.

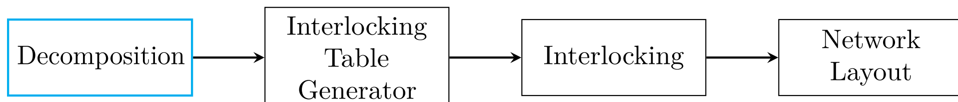


Figure 5.1: The RSL modules overview

The Decomposition module is developed in this project as an extension to the other modules developed by Linh Vu [13]. The Decomposition module showed in the figure above is simplified, it contains the submodules in Figure 5.2. By decoupling the specifications into different sub-modules, the development and maintenance are simplified. Below is a list describing each sub-module.

Decomposition_TEST Contains tests of decomposition- and auxiliary functions.

Decomposition_DESIGN Contains the main decomposition functions.

Decomposition_COMMON Contains commonly used functions from different modules.

Decomposition_REQ Contains the requirements of decompositions methods.

Decomposition_THEOREM Contains theorems of decomposition methods.

The assigned prefix names associated with each module can be seen below, they appear frequently in the RSL code listings when accessing types in objects.

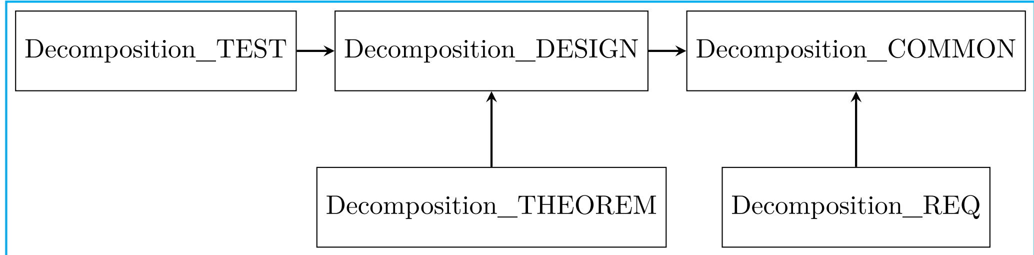


Figure 5.2: Sub-modules of the Decomposition module

(D) Decomposition_DESIGN

(C) Decomposition_COMMON

(ITG) InterlockingTableGenerator

(I) Interlocking

(L) NetworkLayout

5.2 Well-formedness of Cuts

In this section, the RSL specifications for well-formedness checks are showed. The network layout and interlocking table are checked using RSL specifications provided by Vu [13]. The cut checks, on the other hand, are defined in this project. The well-formedness of a cut is determined before any decomposition operation. The only inputs needed are a network layout and a cut specification. In Chapter 3 the BCC pre-conditions were introduced. In this section, the reader will see how they are specified in RSL to check for well-formedness.

5.2.1 Well-formed Single Cut

When checking if a single cut is well-formed the followings must hold:

C-01 Linear section *up* from the cut specification exists in the network.

C-02 Linear section *down* from the cut specification exists in the network.

C-03 Border cut conditions are satisfied.

```
cut_wf: ITG.I.L.NetworkLayout >< SingleCut -> Bool
cut_wf(n,sc) is
let l_up = section_up(sc),
    l_down = section_down(sc)
in
    /* Does linear section up exists in network? */
    ITG.I.L.l_exists(l_up,n) /\
    /* Does linear section down exists in network? */
    ITG.I.L.l_exists(l_down,n) /\
    /* Is BCC1 satisfied? */
    BCC1(n,sc)
end,
```

Below is a short recap of the border cut conditions described in Chapter 3.

BCC1 Check if single cut sections are neighbors

BCC2 Check if single cut sections contain one up- and one down signal

BCC3 Check if single cut sections are already boundaries

BCC4 Check if single cut sections are still reachable after disconnecting

The RSL specification of the condition checks are as follows below.

```
BCC: ITG.I.L.NetworkLayout >< SingleCut -> Bool
BCC(n, sc) is
let l_up = section_up(sc),
    l_down = section_down(sc)
in
    /* Are the sections neighbours? */
    (ITG.I.L.are_neighbors(l_down,l_up, n) /\
    /* Is down-linear a boundary? */
    ~ITG.I.L.is_boundary(l_down,n) /\
    /* Is up-linear a boundary? */
    ~ITG.I.L.is_boundary(l_up,n) /\
    /* Do we have a down- and up signal from linears in cut specification? */
    DOWN isin dom ITG.I.L.signals(l_up,n) /\ UP isin dom ITG.I.L.signals(l_down,n)) /\
    /* Check if linears are is still reachable when disconnected */
    let disconnected_network = get_disconnectedNetwork(n,l_down,l_up) in
        l_up ~isin get_reachableNetworkSet(disconnected_network,l_down) /\
        l_down ~isin get_reachableNetworkSet(disconnected_network,l_up)
    end
 end
```

The last check disconnects the cut section such that they are no longer neighbors, then `l_up` is checked if it is reachable in the *down* network and vice versa. In a decomposition, the cut sections are disconnected with the sections *around* them and not between them as it is done here (see Section 3.3). This does however not affect the reachability check because this approach is only one single linear section away from where the actual disconnection happens.

5.2.2 Well-formed Cluster Cut

The initial intuition for this specification was to reuse the well-formedness for single cuts by saying:

```
cut_wf: ITG.I.L.NetworkLayout >< ClusterCut -> Bool
cut_wf(n,cc) is
        (all sc: SingleCut :- sc isin cc => cut_wf(n, sc)),
```

This, however, is not possible due to the last check is done by the BCC function. The reachability check to be precise. The problem is that the sub-networks should only be unreachable *after* all of the single cuts in a cluster cut are disconnected. To solve this problem a new *BCC* function in RSL is explicitly designed for cluster cut. Here the network is disconnected in all given single cut locations, then the reachability is checked. It could be argued that the reachability could be checked only for the last single cut, but that could potentially exclude some not well-formed cuts. The RSL specification can be seen below.

```
cut_wf: ITG.I.L.NetworkLayout >< ClusterCut -> Bool
cut_wf(n,cc) is
        (all sc: SingleCut :- sc isin cc =>
             /* Does linear section up exists in network? */
             ITG.I.L.l_exists(section_up(sc),n) /\
             /* Does linear section down exists in network? */
             ITG.I.L.l_exists(section_down(sc),n)
        ) /\
        /* Is BCC rules satisfied? */
        BCC(n,cc)


BCC: ITG.I.L.NetworkLayout >< ClusterCut -> Bool
BCC(n, cc) is
        (all sc: SingleCut :- sc isin cc =>
            let l_up = section_up(sc),
                l_down = section_down(sc)
            in
                /* Are the sections neighbours? */
                (ITG.I.L.are_neighbors(l_down,l_up, n) /\
                /* Is down-linear a boundary? */
                ~ITG.I.L.is_boundary(l_down,n) /\
                /* Is up-linear a boundary? */
                ~ITG.I.L.is_boundary(l_up,n) /\
                /* Do we have a down- and up signal from linears in cut specification?
                 ↪  */
                DOWN isin dom ITG.I.L.signals(l_up,n) /\ UP isin dom
                 ↪  ITG.I.L.signals(l_down,n)) /\
                /* Check if linears are is still reachable when disconnected --
                 ↪  Disconnects all cuts every time */
                let disconnected_network = get_disconnectedNetwork(n,cc) in
                    l_up ~isin get_reachableNetworkSet(disconnected_network,l_down) /\
                    l_down ~isin get_reachableNetworkSet(disconnected_network,l_up)
                end
            end
        )
```

5.2.3 Well-formed Multi Cut

A multi cut can consist of a mixture of cluster cuts and single cuts. The already defined well-formed check functions can be reused. In this function, every cut type must be checked, so the correct function is applied. A case matching is used for this purpose. The RSL specification can be seen below.

```
cuts_wf: ITG.I.L.NetworkLayout >< MultiCut -> Bool
cuts_wf(n , mc) is
        (all c: Cut:- c isin mc =>
             case c of
                  Cut_from_SingleCut(sc)  -> cut_wf(n, sc),
                  Cut_from_ClusterCut(cc) -> cut_wf(n, cc)
             end)
```

5.3 Soundness of Cuts

In this section, the RSL code for soundness checks is shown. The soundness checks are defined in `Decomposition_COMMON` but are used as theorems (axioms) in the `Decomposition_THEOREM` module. A theorem gives the opportunity to define the consequence of a function. It is different from a post-condition, where a post-condition defines the state of the actual output. The first specification shows the soundness check for a single cut. It initially checks if the sub-networks are well-formed, followed by soundness checks.

```
[Single_decompose_SC] in Decomposition_DESIGN |-
    all (n1,n2): C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout,
         n: C.ITG.I.L.NetworkLayout,
         c: C.SingleCut:-
            (n1,n2) = decompose(n,c) =>
                        C.ITG.I.L.is_wf(n1) /\ C.ITG.I.L.is_wf(n2) /\
                        C.SC2(n,n1,n2) /\ C.SC3(n,n1,n2) /\
                        C.SC4(n,n1,n2) /\ C.SC5(n,n1,n2),
```

The same checks are done for cluster cut.

```
[ClusterCut_decompose_SC] in Decomposition_DESIGN |-
    all (n1,n2): C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout,
         n: C.ITG.I.L.NetworkLayout,
         c: C.ClusterCut:-
            (n1,n2) = decompose(n,c) =>
                        C.ITG.I.L.is_wf(n1) /\ C.ITG.I.L.is_wf(n2) /\
                        C.SC2(n,n1,n2) /\ C.SC3(n,n1,n2) /\
                        C.SC4(n,n1,n2) /\ C.SC5(n,n1,n2),
```

The multi cut theorem does the same checks, but it uses pattern matching to distinguish between the cuts.

```
[MultiCut_decompose_post] in Decomposition_DESIGN |-
    all  n: C.ITG.I.L.NetworkLayout,
         mc: C.MultiCut:-
                (all c:C.Cut :- c isin mc =>
                    case c of
                      C.Cut_from_SingleCut(sc) ->
                       (all (n1,n2): C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
                        ↪   :- (n1,n2) = decompose(n,sc) =>
                                   C.ITG.I.L.is_wf(n1) /\ C.ITG.I.L.is_wf(n2) /\
                                   C.SC2(n,n1,n2) /\ C.SC3(n,n1,n2) /\
                                   C.SC4(n,n1,n2) /\ C.SC5(n,n1,n2)),
                      C.Cut_from_ClusterCut(cc)->
                       (all (n1,n2): C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
                        ↪   :- (n1,n2) = decompose(n,cc) =>
                                   C.ITG.I.L.is_wf(n1) /\ C.ITG.I.L.is_wf(n2) /\
                                   C.SC2(n,n1,n2) /\ C.SC3(n,n1,n2) /\
                                   C.SC4(n,n1,n2) /\ C.SC5(n,n1,n2))
                    end
                )
          end
```

Before showing the actual RSL specification for the soundness checks, a recap of the conditions are listed.

SC1 If any overlap section exists in the route, it must reside in the generated interlocking table or the sub-network it was based on.

SC2 Any point in the route must only reside in the same interlocking table or the sub-network it was based on.

SC3 Any marker boards in the route must only reside in the generated interlocking table or the sub-network it was based on.

SC4 All conflicting routes must reside in the same interlocking table.

The specification for SC1 can be seen on the next page. The function takes the original network and the sub-networks as input. An interlocking table is generated from the original network and this table is divided into two sub-tables. One table with routes belonging to *down* and one table with routes belonging to *up*. The function iterates all of the routes in *down* table and checks if all found overlaps are part of the sub-network it is based on. The same is done for the *down* table.

```
SC1: ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout -> Bool
SC1(n,n_down,n_up) is
        let
          t = ITG.mk_table(n),
          down_routes = get_sub_routes(n_down,t),
          up_routes = get_sub_routes(n_up,t)
        in
          (all r : ITG.I.Route :- r isin rng down_routes =>
               (all s: SecId :- s isin ITG.I.overlap(r) =>
                    s isin ITG.I.L.sections(n_down)))
           /\
          (all r : ITG.I.Route :- r isin rng up_routes =>
              (all s: SecId :- s isin ITG.I.overlap(r) =>
                   s isin ITG.I.L.sections(n_up)))
        end,
```

The same exact procedure is applied for other soundness checks too, they are therefore not shown here, except for SCC2. The SCC2 is listed for comparison, so the reader can see how similar the check is performed. The rest of the soundness condition definitions can be seen in Appendix E.

```
SC2: ITG.I.L.NetworkLayout ><  ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout -> Bool
SC2(n, n_down,n_up) is
        let
          t = ITG.mk_table(n),
          down_routes = get_sub_routes(n_down,t),
          up_routes = get_sub_routes(n_up,t)
        in
          (all r : ITG.I.Route :- r isin rng down_routes =>
              (all s_id : SecId :- s_id isin dom ITG.I.points(r) =>
                   s_id isin ITG.I.L.sections(n_down)))
           /\
          (all r : ITG.I.Route :- r isin rng up_routes =>
              (all s_id : SecId :- s_id isin dom ITG.I.points(r) =>
                   s_id isin ITG.I.L.sections(n_up)))
        end,
```

5.4 Decomposition Specifications

In this section, the decomposition method specifications are described along with their auxiliary functions. The main decomposition methods are placed in the `Decomposition_DESIGN` module while the auxiliary functions are placed in the RSL module `Decomposition_COMMON`.

5.4.1 Single cut Decomposition

A single cut decomposition execution divides a network into two sub-networks. The cut specification tells the `decompose` function where to perform the cut. If the cut specification is well-formed, then it will output two networks, the sub-network *down* and the sub-network *up*. The function is defined as shown in the listing below.

It receives the cut specification and then calls the `apply_bc` function followed by `get_reachableNetwork`.

The `apply_bc` function is called twice for each sub-network to be generated. It receives the direction (`DOWN/UP`) for targeted sub-networks (*down/up*) and the cut specification as input. `apply_bc` has subroutines which perform the specific operations defined for a border cut. After it has executed, both sub-networks are in a not well-formed state, that is because they still contain both sides of the cut.

The `get_reachableNetwork` function is used to extract only the targeted sub-network after the cut operations are performed by `apply_bc`. It requires an input of a section that resides in the targeted sub-network. The extracted sub-network will be well-formed if the cut is applied properly.

```
decompose:  C.ITG.I.L.NetworkLayout >< C.SingleCut -~->
            C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
decompose(n,sc) is
let
    l_down = C.section_down(sc),
    l_up = C.section_up(sc),

    cut_applied_down = C.apply_bc(n,sc, DOWN),
    cut_applied_up = C.apply_bc(n,sc, UP),

    n_down = C.get_reachableNetwork(cut_applied_down,l_down),
    n_up = C.get_reachableNetwork(cut_applied_up,l_up)
in
    (n_down,n_up)
end
pre C.ITG.I.L.is_wf(n) /\ C.cut_wf(n,sc),
```

5.4.2 Cluster cut Decomposition

A cluster cut decomposition is an execution of a set of single cuts, which all combined produces two networks. The specification of a cluster cut decomposition is defined as seen in the listing below.

```
decompose:  C.ITG.I.L.NetworkLayout >< C.ClusterCut -~->
            C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
decompose(n,cc) is
  let
    l_ups =   { C.section_up(c)   | c:C.SingleCut :- c isin cc },
    l_downs = { C.section_down(c) | c:C.SingleCut :- c isin cc },

    cut_applied_down = C.apply_bc(n,cc, DOWN),
    cut_applied_up = C.apply_bc(n,cc, UP),

    n_down = C.get_reachableNetwork(cut_applied_down, hd l_downs),
    n_up = C.get_reachableNetwork(cut_applied_up, hd l_ups)
  in
    (n_down, n_up)
  end
  pre C.ITG.I.L.is_wf(n) /\ C.cut_wf(n, cc),
```

The decompose specification of cluster cut resembles the single cut specification, but, there are some differences. The apply_bc function receives a cluster cut as input which applies cut operations on all of the single cuts. Furthermore, the get_reachableNetwork function receives arbitrary sections from the sets *l_ups* and *l_downs*. As a side note, it does not matter which of the cut sections are used in get_reachableNetwork, since they appear in both sub-networks, but for consistency, the lower sub-network receives l_down and upper sub-network l_up.

5.4.3 Multi cut Decomposition

The multi-cut decomposition is the execution of multiple single cuts and cluster cuts. The idea of the algorithm used in this function is to reuse the already defined single cut and cluster cut decomposition functions. All of the cuts will be executed recursively. The cuts are executed one by one in arbitrary order and the algorithm is finished when no more applicable cuts remain.

A pre-condition with cuts_wf is only possible using a boolean flag to tell if the input is the original network or not. All of the cuts are only expected to be well-formed for the original network. The reason for not using the pre-condition for all instances is due to the non-applicable cuts being carried over to next instances. To filter them out, the get_applicableCuts function is used to return the subset of all well-formed cuts for the particular targeted network. The RSL specification for the multi cut can be seen in the listing below.

```
decompose:  C.ITG.I.L.NetworkLayout >< C.MultiCut >< Bool -~->
            C.ITG.I.L.NetworkLayout-set
decompose(n,mc, isOriginalNetwork) is
  let
    applicableCuts = C.get_applicableCuts(n,mc)
  in
    if applicableCuts = {}
    then {n}
    else let cut= hd applicableCuts in
       let
            (n1,n2) = case cut of
                        C.Cut_from_SingleCut(sc)  ->  decompose(n,sc),
                        C.Cut_from_ClusterCut(cc) ->  decompose(n,cc)
                      end,
            ns1 = decompose(n1, mc \ {cut}, false),
            ns2 = decompose(n2, mc \ {cut}, false)
       in
            ns1 union ns2
       end
    end
    end
  end
  pre C.ITG.I.L.is_wf(n) /\ if isOriginalNetwork then C.cuts_wf(n, mc) else true end
```

Every time a cut is applicable for a network, the correct decompose function is called, creating n1 and n2. The created sub-networks are then executed with the rest of the remaining cuts. When no applicable cuts remain, the current network is returned. The returned networks are assigned as ns1 and ns2. These networks are

accumulated using the union operator. The union of all the returned sub-networks will eventually be returned. This process can be visualized as a binary tree of networks where only the leaves are included in the result, see Figure 5.3. It can also be seen that a multi cut with n cuts will produce $n+1$ sub-networks.

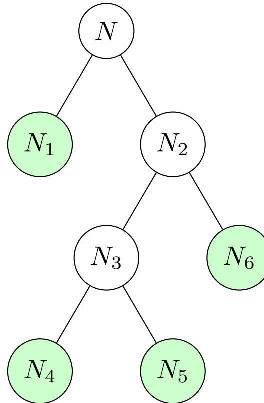


Figure 5.3: Every child node in the tree represents a sub-network. The green leafs in the binary tree identifies the final networks that are part of the multi-cut decomposition result.

Applying a Border Cut

The specification of `apply_bc` for a single cut can be seen below. This function does two operations. Initially, it obtains the sections to be disconnected and then disconnects them by removing the connecting neighbors from both sides. The obtained sections in question lie *around* the cut sections.

Then the function removes the extra marker board placed at the new boundary (if there exists any). How this function operates depends on the given direction `d`, which identifies the targeted sub-network.

```
apply_cut : ITG.I.L.NetworkLayout >< SingleCut >< Direction -~-> ITG.I.L.NetworkLayout
apply_cut(n,sc,d) is
    let
       l_down_id = section_down(sc),
       l_up_id = section_up(sc),
       l_down = ITG.I.L.get_linear(l_down_id,n),
       l_up = ITG.I.L.get_linear(l_up_id,n),

       disconnect_down = if d = DOWN then l_up_id else ITG.I.L.down(l_down) end,
       disconnect_up = if d = DOWN then ITG.I.L.up(l_up) else l_down_id end,
       disconnected_network = get_disconnectedNetwork(n, disconnect_down, disconnect_up),

       -- Remove invalid markerboard at the interested side of the disconnection
       extra_mb_removed_n =
           if d = DOWN then remove_extra_mb(disconnected_network,l_up_id,d)
           else remove_extra_mb(disconnected_network,l_down_id,d)
```

```
        end
    in
       extra_mb_removed_n
    end,
```

The specification of `apply_bc` for cluster cut differs by recursively applying single cut operations on the same network until no more cuts are left.

```
apply_cut : ITG.I.L.NetworkLayout >< ClusterCut >< Direction -~-> ITG.I.L.NetworkLayout
apply_cut(n,cc,d) is
    if cc = {} then n else
      let
        sc = hd cc,
        sc_subnet = apply_cut(n,sc,d)
      in
        apply_cut(sc_subnet,cc \ {sc},d)
      end
    end,
```

Reachable Network

The `get_reachableNetwork` is a function that discovers a network from a given section. It returns the discovered network as a `NetworkLayout`. This function is useful and is applied in different scenarios. The function is specified as follow:

```
get_reachableNetwork : ITG.I.L.NetworkLayout >< SecId -~-> ITG.I.L.NetworkLayout
get_reachableNetwork(n,s) is
 let
    -- Get all sections
    sections = get_all_sections(n,{s},{}),
    -- Get all linears from given sections
    linears = get_all_linears(n,sections),
    -- Get all points from given sections
    points  = get_all_points(n, sections),
    -- Get all signals from given sections (signals only exists in linears)
    signals = get_all_signals(n, sections)
 in
    -- Instantiate a new network layout
    ITG.I.L.mk_NetworkLayout(linears,points,signals)
 end
 pre ITG.I.L.s_exists(s,n),
```

The interesting function in this specification is `get_all_sections`. This function traverses a network and outputs all the sections it finds. Once the sections are known, the linears, points, and signals can easily be extracted using map comprehension to form the new sub-network. The specification of the `get_all_sections` function can be seen below. To see the other functions (`get_all_linears`, `get_all_points` and `get_all_signals`) see Appendix E.2.

```
get_all_sections: ITG.I.L.NetworkLayout >< SecId-set >< SecId-set -~-> SecId-set
get_all_sections(n,tv,v) is
  if tv = {} then v
  else
```

```
    let
      current = hd tv,
      visited = {current} union v,

      toVisit = (tv union ITG.I.L.get_neighbors(current,n)) \ visited
      in
        get_all_sections(n,toVisit,visited)
    end
  end,
```

For this function, a general traversal algorithm is used which is based on Depth First Search (DFS) and Breadth First Search (BFS) [3]. Every section is discovered by looking at the neighbors of the current one. The algorithm traverses in all directions, it has the inputs (n, tv, v), where n is the network, tv is the set of nodes to be visited and v is the set of visited sections. Both tv and v are of type `SecId-set`. Using sets means that the next node to visit is arbitrary, so the algorithm does not exactly follow DFS or BFS. That does however not matter since all of the nodes must be visited anyway, in which order it happens, affect neither the result nor the performance. The algorithm is finished when no more unvisited nodes exist (when tv is empty), in this case, v is returned.

The start section given to the algorithm does not matter as long as it resides in the correct sub-network to be discovered. A benefit of this algorithm is that we can be sure all nodes are only visited once.

Disconnecting a Network

This specification function takes two neighbor sections and disconnects them. The disconnections happen on both sections, meaning that both sections will no longer have the other one as a neighbor. Since there exist two types of sections (linears and points), the function must be able to cover all combinations. If one of the sections is a point, then the correct point-end must be determined to disconnect properly. The specification of `get_disconnectedNetwork` can be seen in the listing below.

```
get_disconnectedNetwork : ITG.I.L.NetworkLayout >< SecId >< SecId -~->
↪  ITG.I.L.NetworkLayout
get_disconnectedNetwork(n,secIdDown, secIdUp ) is
 let
    points = ITG.I.L.points(n),
    linears = ITG.I.L.linears(n),

    linearsToDisconnect =
            -- if down section is a linear
            [ l +> remove_nb(ITG.I.L.get_linear(l,n), UP) | l:SecId :- l isin
             ↪  ITG.I.L.linears(n) /\ l = secIdDown] !!
            -- if up section is a linear
            [ l +> remove_nb(ITG.I.L.get_linear(l,n), DOWN) | l:SecId :- l isin
             ↪  ITG.I.L.linears(n) /\ l = secIdUp],

    pointsToDisconnect =
            -- if down section is a point
            -- point end to disconnect:  ITG.I.L.get_p_end_by_nb_id(s,s,n)
```

```
            [ p +> remove_nb(ITG.I.L.get_point(secIdDown,n),
            ↪  ITG.I.L.get_p_end_by_nb_id(secIdDown, secIdUp, n)) | p:SecId :- p isin
            ↪  ITG.I.L.points(n) /\ p = secIdDown] !!
            -- if up section is a linear
            [ p +> remove_nb(ITG.I.L.get_point(secIdUp,n),
            ↪  ITG.I.L.get_p_end_by_nb_id(secIdUp, secIdDown, n)) | p:SecId :- p isin
            ↪  ITG.I.L.points(n) /\ p = secIdUp]
 in
    -- Update the network by adding the new boundaries
    ITG.I.L.mk_NetworkLayout(linears !! linearsToDisconnect, points !!
     ↪  pointsToDisconnect, ITG.I.L.marker_boards(n))
 end
 pre ITG.I.L.are_neighbors(secIdDown, secIdUp, n),
```

Another version explicitly for cluster cut is defined that reuses the already defined `get_disconnectedNetwork` for a single cut. This version visits all single cuts and disconnects between the cut section one by one. The returned network will probably be disconnected multiple places.

```
get_disconnectedNetwork : ITG.I.L.NetworkLayout >< ClusterCut -~->
 ↪  ITG.I.L.NetworkLayout
get_disconnectedNetwork(n,cc) is
    if cc = {} then n
    else  let
           sc = hd cc,
           sc_disconnected = get_disconnectedNetwork(n, section_down(sc),
            ↪  section_up(sc))
          in
           get_disconnectedNetwork(sc_disconnected,cc \ {sc})
          end
    end
 pre (all sc : SingleCut :- sc isin cc => ITG.I.L.are_neighbors(section_down(sc),
  ↪  section_up(sc), n)),
```

Removing invalid marker boards

When new boundaries are created as a product of disconnection, it must be ensured that these new boundaries do not contain invalid marker boards. Invalid marker boards are removed with the `remove_extra_mb` function, previously seen used in the `apply_bc` specification. The specification for this functions can be seen below.

```
remove_extra_mb :  ITG.I.L.NetworkLayout >< SecId >< Direction -> ITG.I.L.NetworkLayout
remove_extra_mb(n,l,d) is
    let
    -- Get the extra marker boards to be removed (if exists)
    extra_mbs = { ITG.I.L.signals(l,n)(-d)}
    in
    -- Update the network by deleting the extra marker board
    ITG.I.L.mk_NetworkLayout(ITG.I.L.linears(n),
                             ITG.I.L.points(n),
                             ITG.I.L.marker_boards(n) \ extra_mbs)
 end,
```

5.5 XML schema

The XML schemas for cuts are presented in this section. The XML schemas created in this project are based on the previously defined schemas in Vu's project [13], e.g. the network layout XML schema.

A single cut using border cut can be defined using the `borderCut` tag. Inside this tag, two sections must be defined using the `trackSection` tag. The following XML listing shows how a border cut may be defined.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <borderCut id="miniBorderCut">
            <trackSection id="b10" side="down" type="linear"/>
            <trackSection id="t10" side="up" type="linear"/>
    </borderCut>
</xmi:XMI>
```

The `id` attribute is the section identifier that already exists in the network layout. The `side` attribute is used to define `l_down` and `l_up` from the specification. The type of the sections are also included, even though only linears sections are currently allowed, it may be beneficial for new cut types in the future to have this attribute.

A cluster cut can be defined using the `clusterCut` tag. Inside the cluster cut tag, there can be any arbitrary number of border cut definitions. The identifiers of border cuts are used to distinguish between them.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <clusterCut id="miniClusterCut">
        <borderCut id="miniBorderCut">
            <trackSection id="b10" side="down" type="linear"/>
            <trackSection id="t10" side="up" type="linear"/>
        </borderCut>
    </clusterCut>
</xmi:XMI>
```

A multi cut is defined with the `multiCut` tag. Inside the tag, there can be any number of single and cluster cuts. The following listing shows an example.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <multiCut id="miniMultiCut">
        <borderCut id="BorderCut1">
            <trackSection id="t2" side="down" type="linear"/>
            <trackSection id="t3" side="up" type="linear"/>
        </borderCut>
        <clusterCut id="ClusterCut1">
            <borderCut id="CC_BorderCut1">
                <trackSection id="t9" side="down" type="linear"/>
                <trackSection id="t10" side="up" type="linear"/>
```

```
            </borderCut>
            <borderCut id="CC_BorderCut2">
                <trackSection id="t20" side="down" type="linear"/>
                <trackSection id="t21" side="up" type="linear"/>
            </borderCut>
        </clusterCut>
    </multiCut>
</xmi:XMI>
```

In the given example, the multi cut contains one border cut (single cut) and one cluster cut. The cluster cut contains two additional border cuts. Both cuts (`BorderCut1` and `ClusterCut1`) produces each two sub-networks, so we can expect to produce 3 sub-networks in total when decomposing consecutively.

5.6 Class Diagrams

In the rest of the sections, the design choices made for the command line tool are described. In this section, the class diagrams of the decomposition tool are presented to give the reader an overview of the software structure design.

5.6.1 Cut Classes

The cut types are structured as it can be seen in Figure 5.4. All cuts are inherited from the base `Cut` class. A new cut class can be added by further extending the `SingleCut` class. Extending the single cut class by introducing a new cut type, alike border cut, automatically puts the new cut type in a good place to be used with cluster- and multi cut. The multiplicities show that cluster cut and multi cut can have any number of contents in them. The uni-directional associations show that only the higher level cuts are aware of the relationship.

A variant type could have been used to accomplish a more similar structure to the RSL specification. Unfortunately, the variant type `std::variant` was not yet supported by the compiler used in this project, which was below C++17 [11].

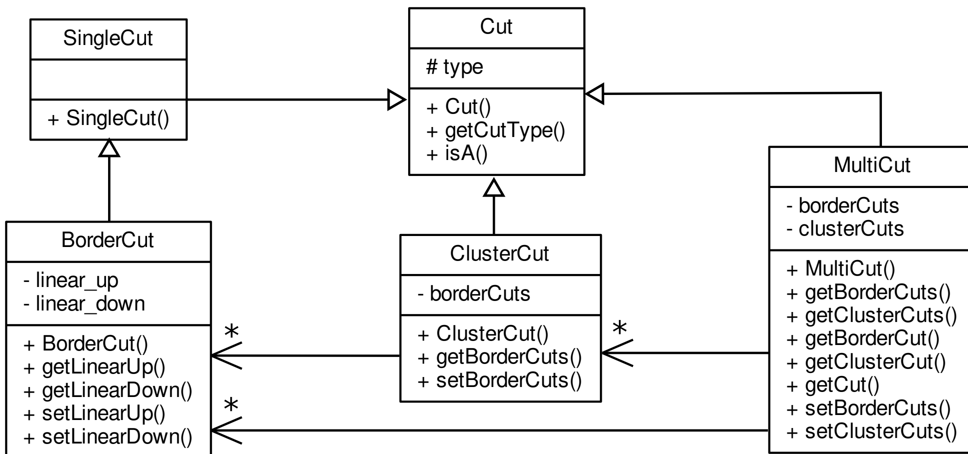


Figure 5.4: Class diagram of the cut types.

5.6.2 Parser Classes

The parsers do not have any inheritance involved, but their multiplicities correspond to the cut classes. The class diagram in Figure 5.5 does also include the parser classes along with their relations to the cut classes. The parsers aggregate cut types by instantiating a private local value. Every parser class is responsible for the type it aggregates.

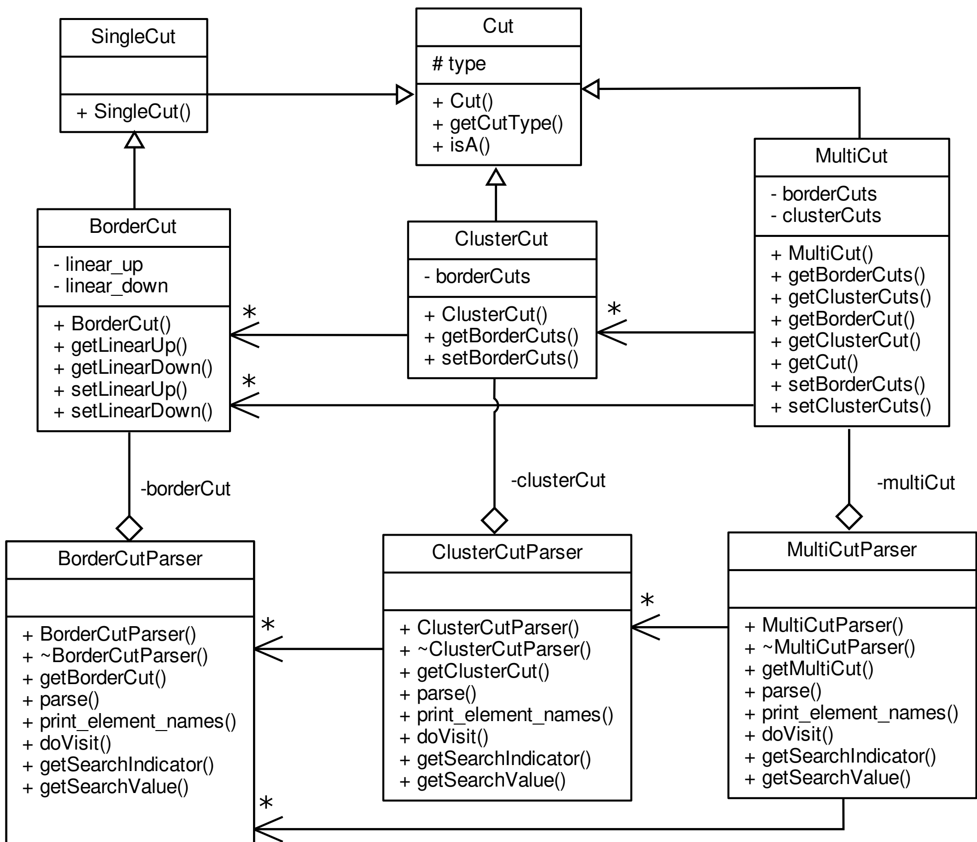


Figure 5.5: Class diagram of the parsers.

5.6.3 Domain

Finally, all of the classes are shown in Figure 5.6. This figure can be interpreted as the domain model. The relationship between the main decomposition classes and the other classes are introduced. The already showed relationships in previous diagrams are not included this time for simplicity.

The class *Decomposition* is the main class used to trigger the decomposition process and the class *DecompositionCommon* contains commonly used functions by other classes, which is why it is associated with them all.

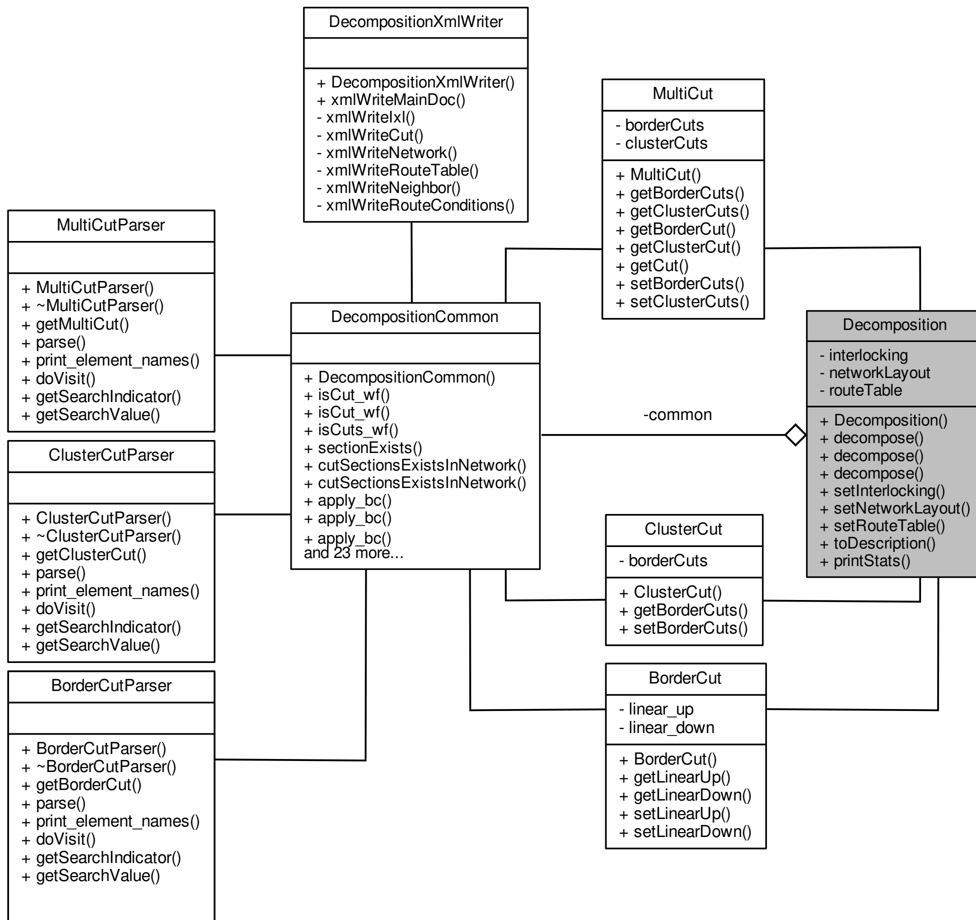


Figure 5.6: Domain diagram.

5.7 Sequence Diagrams

In this section, the sequence diagrams of the tool are presented. The sequence diagrams created show the high-level method calls during executions. Low-level operations such as assignments are omitted, so are irrelevant and obvious calls. A method call that has been shown before might sometimes be simplified in another sequence diagram by showing only its return value, instead of it inner calls again.

5.7.1 Parser Sequences

The parser sequence diagrams show the sequence flow of the parsing of different decomposition methods. The diagrams clearly show how the parsers in some cases reuse lower level parser methods, which is the case with cluster and multi cut.

The parser in Figure 5.7 shows the sequence diagram of the border cut. The `parser` instance starts by calling the `cutparser` with the cut file path as the argument. The `cutparser` then calls the `borderCutParser` with the XML node identified from the `borderCut` tag. The `borderCutParser` will then create a new border cut object that it will populate with its properties. The result will then be returned all the way back to the `parser` object.

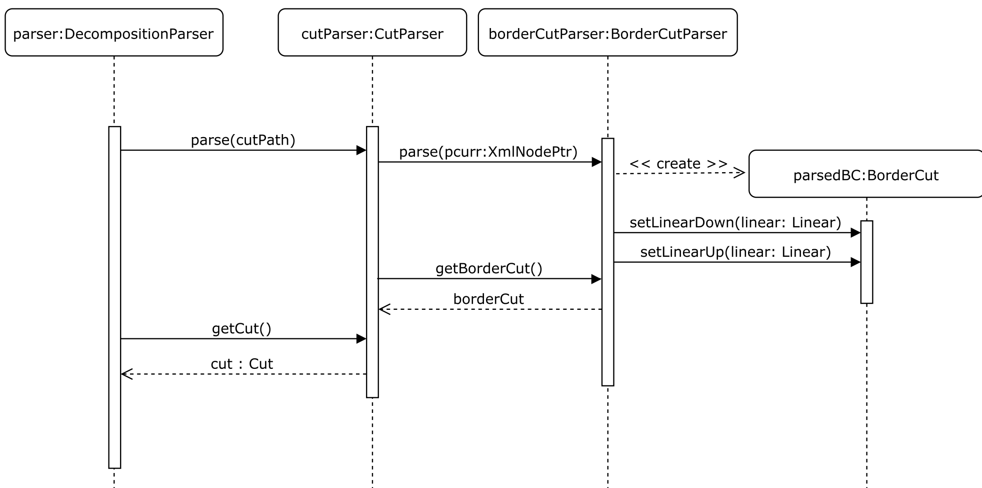


Figure 5.7: Sequence diagram of the border cut parser.

The `DecompositionParser` in the sequence diagram does also call the parser for network layout, which is how it differentiates from `CutParser`. The parsing of the network layout is omitted since it is not developed in this project. Notice that the cut name and its type is generalized in the last return value. This is possible because of the inheritance scheme used for the cuts (see Figure 5.4), a derived cut type can be cast to its base class and vice versa. Casting to another cut type is however not possible.

The Figure 5.8 shows a sequence diagram of the cluster cut parsing. The initial steps are the same until `clusterCutParser` is called. The `clusterCutParser` instance contains a loop that visits all XML nodes starting from the given `pcurr:XmlNodePtr`. If a border tag occurs, then a `BorderCutParser` is created and called with `parse` method. It can be seen from the previous sequence diagram that a bordercut object will be returned from the `BorderCutParser`. The loop stop until no more XML nodes exist. Finally, a `ClusterCut` object is created which is populated with found border cuts. The cluster cut object is then returned all the way back to the `parser` object.

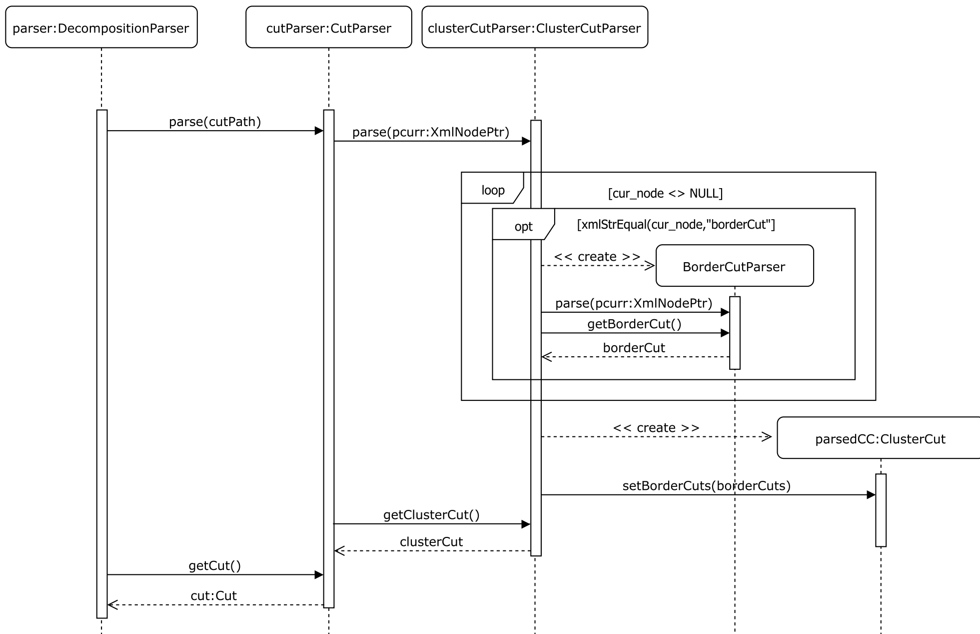


Figure 5.8: Sequence diagram of the cluster cut parser.

The last parser sequence to examine is the multi cut parser which can be seen in Figure 5.9. The multi cut resembles the cluster cut parser sequence but is slightly more complex. The sequence shows how the multi cut parser can exploit the already defined parsers for border cut and cluster cut. The `multiCutParser` contains a big for loop block that checks for border cuts and cluster cuts in the given XML file. Any occurrence creates the respective parser for the found cut type. The received objects from the parsers are saved into a new `MultiCut` object, which is the returned object to the initial caller.

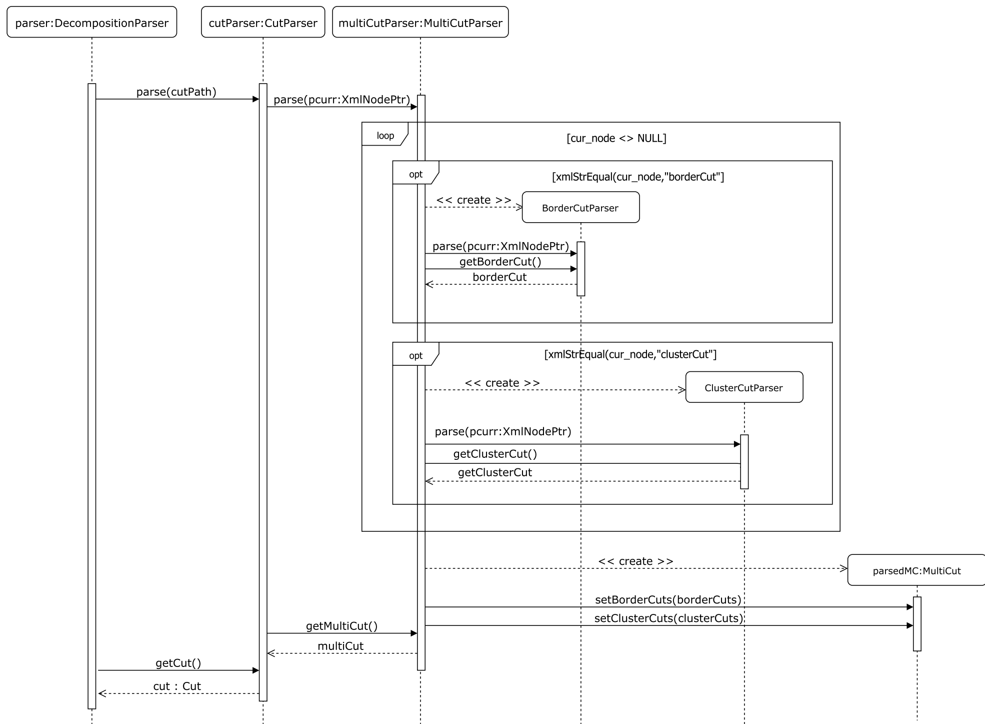


Figure 5.9: Sequence diagram of the multi cut parser.

5.7.2 Decomposition Sequences

The decomposition sequence diagrams show the most crucial part of the tool that must be designed properly from the start. The decomposition of the networks begins after the parsing is done, it is therefore not included, nor is the XML generations. Just like the sequence diagrams for the parsers, repetitive sequences will occur. The specification created for decomposition methods has a huge impact on the design shown in this section, this will be obvious for the reader immediately when examining the diagrams.

Initially, the single cut decomposition is introduced in Figure 5.10. As it been mentioned before, the single cut decomposition applies a border cut operation on a given cut specification and network. The singleton object *main* represents the lifeline

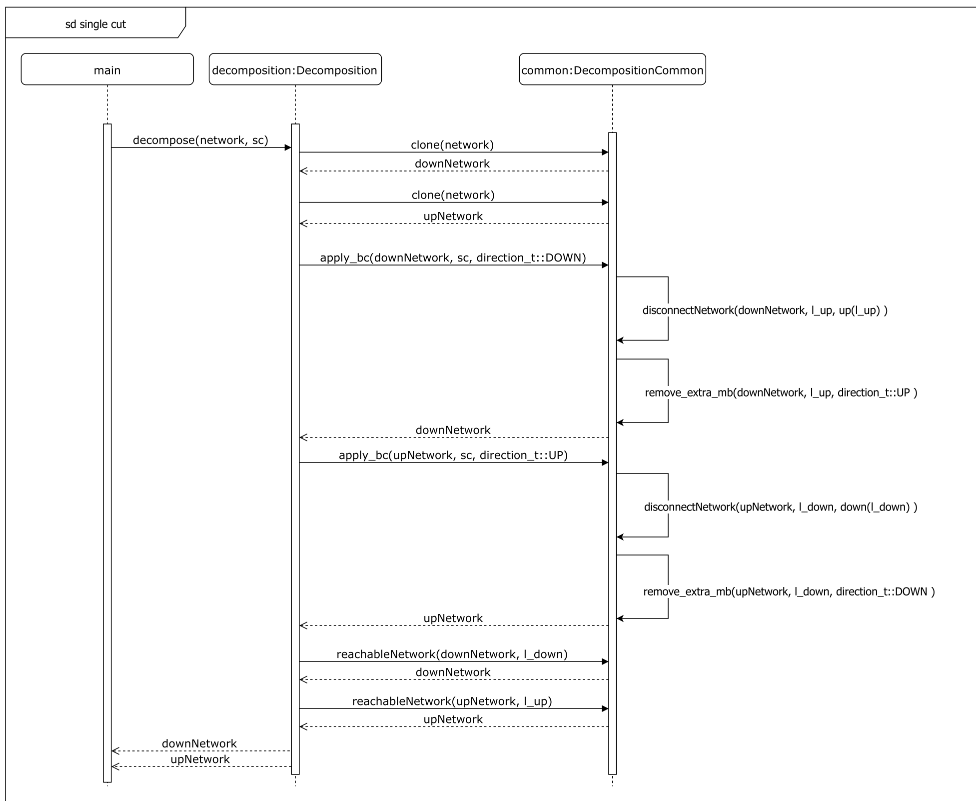


Figure 5.10: Sequence diagram of single cut decomposition.

of the main class which will be executable. It starts by calling the `decomposition` instance with a network and a single cut as arguments. The `decomposition` instance will create two new instances of the network (`downNetwork` and `upNetwork`) using the `clone` function defined in common (see Section 6.4). Thereafter will both net-

works be modified by the `apply_bc` method that will disconnect sections and remove unnecessary marker boards at a cut position. The networks are then respectively rediscovered using `reachableNetwork` before returning the results back to the `main` object.

The next sequence diagram in Figure 5.11 shows the cluster cut decomposition. It resembles very much single cut decomposition, but the `apply_bc` method in this cases triggers a loop that will call the `apply_bc` for all the single cuts. The `downNetwork` and `upNetwork` will be altered multiple times before called back to the `main` object. Notice that `apply_bc` is an overloaded function that differs by its input arguments.

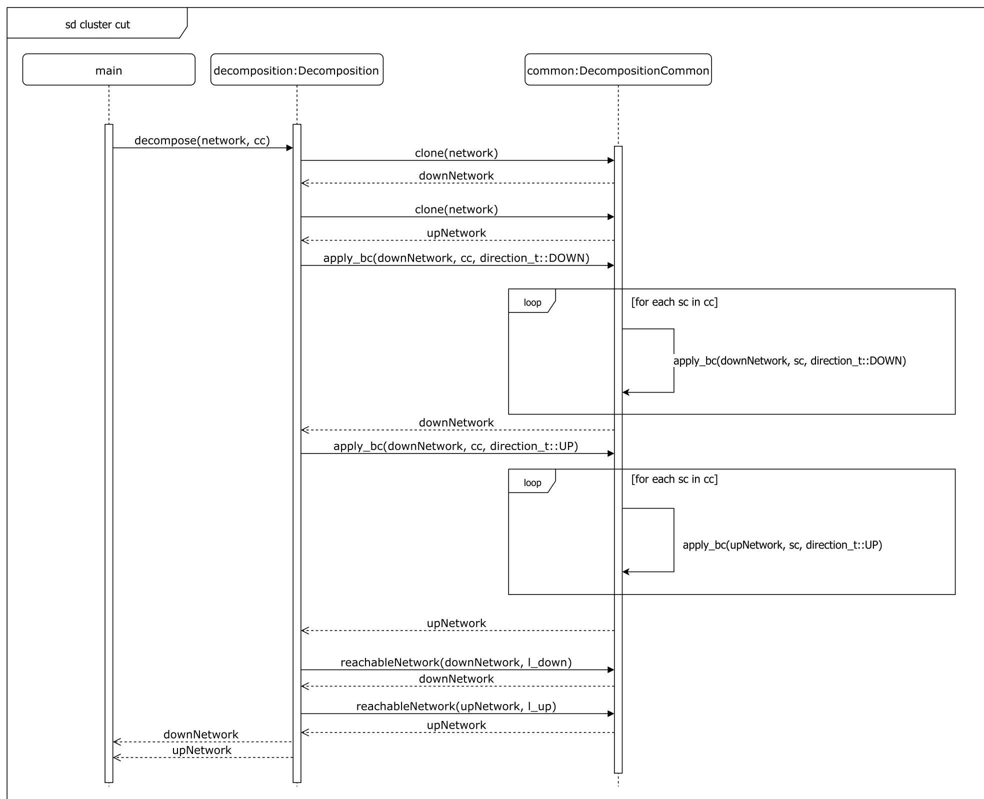


Figure 5.11: Sequence diagram of cluster cut decomposition.

Finally, the sequence diagram for a multi cut is shown in Figure 5.12. This diagram is simplified by referencing the previous sequences with the frames `sd single cut` and `sd cluster cut`. Notice that the referenced frames can call objects not shown in this figure, such as the `DecompositionCommon`. Even though the sequence is simplified, it still has a complex flow. This can be observed by the recursive calls. The algorithm here is based on the specification in Section 5.4.3. The `decompose` function in `decomposition` starts by finding applicable cuts. If no applicable cuts exist, then none of the code in `opt` frame is executed. If applicable cuts exist, then a decomposition sequence is executed depending on the cut type. The same `decompose` function is called again for the created sub-networks `ns.down` and `ns.up`. Eventually, when no more cuts exist, all of the networks are returned as a result to the `main` object.

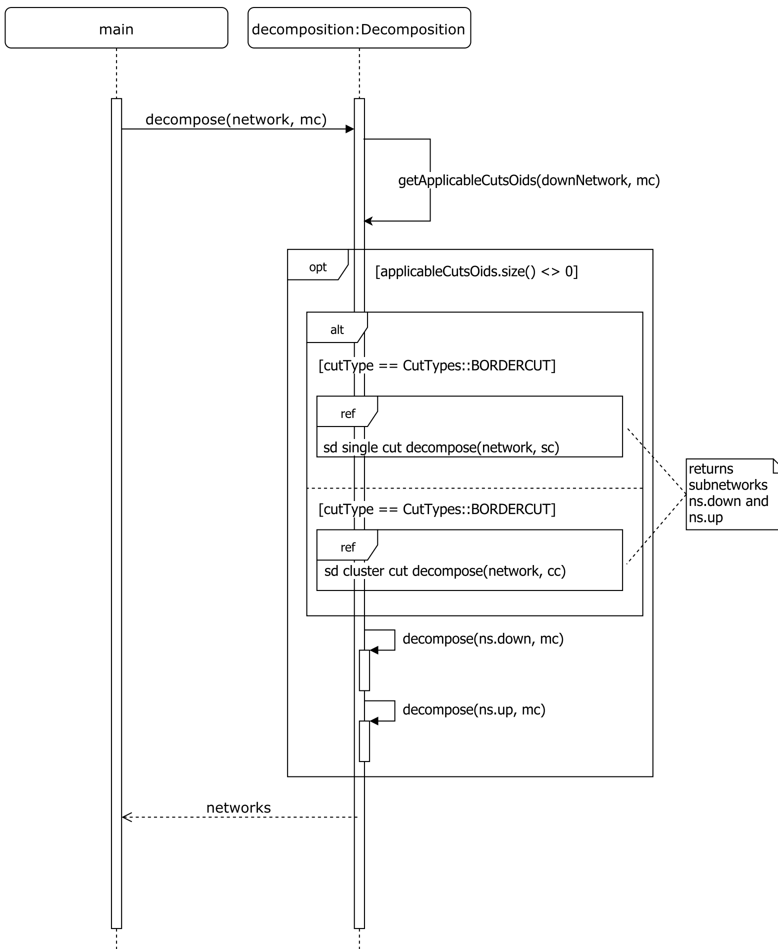


Figure 5.12: Sequence diagram of multi cut decomposition.

5.8 Adding a New Cut Type

The ability to extend the decomposition tool with new cut types is one of the requirements that has been determined. This requirement has an influence on the structure of the software and its design. The extendibility of the software has been touched upon a couple of times, but to really understand, one must know how to add a new cut. To add a new cut, the list below shall be done. As an example, the addition of *linear cut* is portrayed.

5.8.1 Specifications

1. Specify the requirements of *linear cut* in RSL.
2. Create RSL specification for cut.
   a) Introduce the new cut:
   ```
   LinearCut  ::
       section:SecId,
   ```
   b) Extend single cut to include the new cut.
   ```
   SingleCut ==  SingleCut_from_BorderCut(singlecut_to_bordercut: BorderCut) |
                 SingleCut_from_LinearCut(singlecut_to_linearcut: LinearCut),
   ```
   c) Create a specification for cut operations, call it `apply_lc`.
3. Use constructors to avoid ambiguousness.
   ```
   linearcut = singlecut_to_linearcut(sc)
   ```

5.8.2 C++ tool

1. Add the new cut *LinearCut* such that `SingleCut` is its base class (See Figure 5.4).
2. Add a new XML definition and parser for *LinearCut* based on the existing parsers. Since all the cuts specifications refer to section identifiers, adding a new one should be straightforward.
3. Add the new function that specifies the cut operations, call it `apply_lc`.
4. Use casting to avoid ambiguousness.

The given direction to add a new cut assumes that the nature of the cut does not deviate extremely from the already defined *border cut*. In that case, additional development of the tool might be necessary. The advantage of the decomposition methods single-, cluster- and multi cut, is that they can still be applicable for different cut types. A cluster cut or a multi cut consisting of different cut types will provide a huge flexibility.

CHAPTER 6
Implementation & Tests

This chapter makes a run-through of the implementation of the decomposition tool. Unlike previous chapter (Chapter 5), this chapter focuses not on the overall structure but technical challenges unique to this project.

6.1 The C++ Project Structure

In the early phases of the implementation, it was decided to reuse the already existing parser for interlocking systems from Vu's project [13] and implement the other parsers for cuts based on that. This decision had its upsides and downsides. The overall time of parser implementation was shortened. However, the code was part of a large project with many dependencies. This meant that some files had to be included even though they are not directly used in this project, and the build process became a little more complex. Attempts have been made to separate the needed code to an independent project to overcome this issue but were not possible. However, this issue has been reduced by modifying the `CMake` configuration file `CMakeList.txt` only to build relevant files. Another attempt for simplification is made possible by separating the decomposition tool with rest of the included code by keeping all its code in a single folder. The folder has the following path:

```
[rr-project]/decomposers/networkDecomposition/
```

Where `[rr-project]` is the main project folder of RobustRailS tools. The main decomposition classes such as *main.cpp* and `Decomposition.cpp` are placed in this folder, including the executables. The rest of the files are in the sub-folders:

- `cut`. Contains cut types.
- `parser`. Contains cut parser.
- `xmlWriter`. Contains XML writer for networks.
- `common`. Contins the `DecompositionCommon` class.
- `test`. Contains the test classes.

The directly used code from Vu's project is in the paths:

```
[rr-project]/dkixl/
[rr-project]/parsers/ixlparser/
```

The `dkixl` folder contains the interlocking classes and the `ixlparser` folder contains the interlocking parser classes. The other included folders and codes will not be listed since they are irrelevant.

6.2 Parser

The **libxml2** library is used in the implementation of the cut parser. This new parser extension is based on the already existing parser for the network layouts from verification tool [13]. The basic idea behind implementing the parser is to visit every XML node and save the appropriate attributes in found occurrences. The nodes represent the tags in the XML tree created by the **libxml2** library. The challenge in this part of the project was to understand the already defined parsers and extend accordingly.

6.3 XML Writer

The `xmlwriter` is also part of the `libxml` library and is used to write XML files in the system. The files written contains the sub networks generated by the tool. The implementation of the XML writer requires that every element in the networks to be visited. This can unfortunately not be done automatically with objects as arguments, since the XML writer does not know data structure of the network layout and which data to include or not. New methods had to be implemented for this purpose.

In a given interlocking XML file, the elements are not grouped, but are listed in the order they are connected. However, all of the elements such as the *points*, *linears* and the *marker boards* are coupled together in map data types during the parsing process. This means that the information regarding the order they are listed in the XML file is lost.

A possible solution to this would be rediscovering (network traverse) the connections in the sub-networks to know which order to list them in the new XML files, however, this resulted in another problem. Rediscovering sub-networks before writing them in files may filter out errors in the program because only one side of wrongly disconnected networks will show up. This made debugging troublesome. Therefore it was chosen not to write by rediscovering their connections. Instead, the elements are written in groups of *sections*, *points* and *marker boards*. This is simply done by traversing the map data types containing the elements. This choice does not result in any functional misbehaviors, in fact, it can be beneficial to have the elements grouped in some cases.

6.4 Cloning of networks

In the process of a decomposing, it is natural to create multiple instances of networks representing the sub-networks. In the implementation process, it became apparent that this was a problem, the default copy constructors supplied by the C++ compiler did not create new instances of all the network elements, mainly of elements contained in maps. The consequence was that the new instances were referencing to the very same objects they were supposed to copy. This meant that modifying a single instance resulted in all of them being modified, a solution to this had to be implemented.

It was not possible to modify or add new copy constructors to the network objects, because that required the whole project folder to be compiled, the lack of some confidential dependent folders meant that this was not possible (see Section 6.1). Instead, new functions were created in the `DecompositionCommon` class for this purpose. This function would take any network element and return a clone (copy) of it. When a network has to be cloned, new elements such as *linears*, *points*, and *marker boards* are instantiated without any *relation* information. A relation information comprises neighboring of the sections and mount placements of marker boards. When all elements are instantiated, then their relations are set. The reason for this ordering is that the relations are represented by pointer references, and to reference other elements, they must first exist. Figure 6.1 shows an overview of relation setter functions used in the cloning process.

The implementation of this function took extensive time. Many sub-functions had to be implemented and tested thoroughly. If a single element is not cloned properly, many problems can arise later on, without a direct hint of what the root cause is.

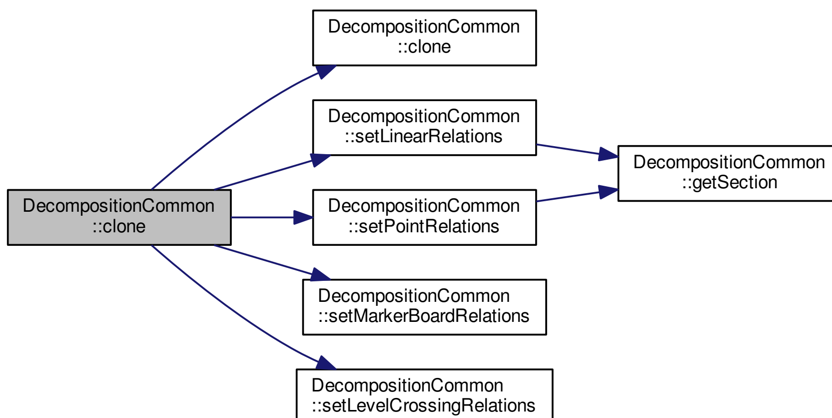


Figure 6.1: The clone function and its sub-functions.

6.5 Tests

Test driven development (TDD) methodology has been used in this project during the development of the tool. This methodology requires the test cases to be established before the code is written. Testing new functions in small cycles ensure that the written specification and the C++ code works as expected. The test cases are written in the form of unit tests. Every test contains assertions that must be true for successful a test case. Another advantage that the tests bring is the confidence in the written code. After a change in the code, one can simply run all the tests to see if all the cases are still covered.

6.5.1 RSL Tests

The RSL specifications have been tested by converting them into sml code. The sml code is thereafter executed in its own compiler. The RSL test module is mentioned before in Section 5.1. This test module starts by instantiating a network layout that can be used for testing. The network layout *mini-e* is used for this purpose. See Figure 7.1 in next chapter for a representation of the network layout. All variations of cut types are also instantiated, including invalid ones to test for negative results. Some examples of RSL tests definitions along with the results will be shown here. The test result will be at the bottom of each listing. For all of the RSL tests, please look at Appendix D. Initially, a test for section disconnections is shown below. One of the given sections is a point (t4), so the function must be able to disconnect at the correct point end.

```
[get_disconnectNetwork_with_points]
let n = D.C.get_disconnectedNetwork(mini_ext, "t3", "t4") in
    D.C.ITG.I.L.are_neighbors("t3", "t4", mini_ext) /\
    D.C.ITG.I.L.are_neighbors("t3", "t4", n) = false
end

-- sml result
[get_disconnectNetwork_with_points] true
```

The next test relies on the result of the first shown test and extends it by performing a `get_reachableNetwork` function on the disconnected network.

```
[get_reachableNetwork]
let n = D.C.get_disconnectedNetwork(mini_ext, D.C.section_down(sc1),
 ↪  D.C.section_up(sc1)) in
    D.C.get_reachableNetwork(mini_ext, "b1") ~=
    D.C.get_reachableNetwork(n, "b1")
end,

-- sml result
[get_reachableNetwork] true
```

The test below is an example of a test with an expected value of *false*. A set of non-applicable cuts are given to the function `get_applicableCuts` which will therefore return an empty set.

```
[get_applicableCuts_False]
let cuts = D.C.get_applicableCuts(mini_ext, mc2) in
    (cuts ~= {}) = false
end,

-- sml result
[get_applicableCuts_False] true
```

The RSL listing below contains a test for the multi cut decomposition method. A valid multi cut is given to the `decompose` function which generates well-formed sub-networks.

```
[decomposed_mc1_wf]
let ns = D.decompose(mini_ext, mc1, true) in
    (all n : D.C.ITG.I.L.NetworkLayout :- n isin ns => D.C.ITG.I.L.is_wf(n))
end

-- sml result
[decomposed_mc1_wf] true
```

The conversion to sml makes it possible to see the results in other constructions than *true/false* assertions. The test below prints out the sub-networks after a multi cut decomposition.

```
[decompose_mc1]
D.C.decomposed_sec_repr(D.decompose(mini_ext, mc1, true), {}),

-- sml result
[decompose_mc1] {{"t2","t3","b1"},{"t2","t8","t7","t6","t5","t3","t4"},
                 {"t11","b12","t10"},{"t7","t8","t5","t6","t11","t10","t9"}}
```

6.5.2 C++ Tests

The C++ code has been tested using an open source unit testing framework called *Catch* [9]. For all of the test results, see Appendix D. The catch framework provides an easy way of defining unit tests. A test is defined using the `TEST_CASE` methods. This method can contain nested sections that enable tests of different scenarios. An assertion is defined using the `REQUIRE` function. The unit test in the next page corresponds to the first to RSL test shown in the previous section. The test checks both disconnections of sections and extraction (reachability) of networks.

```
TEST_CASE("Disconnecting linears", "[disconnect, linears]"){
    string xmlInputFile = "xml/mini-e.xml";
    string xmlCutFile  = "xml/cut.xml";
    string xmlOutputFile_dis = "xml/mini-e_disconnected.xml";
    string xmlOutputFile_down = "xml/mini-e_disconnected_down.xml";
    string xmlOutputFile_up = "xml/mini-e_disconnected_up.xml";

    DecompositionParser *decParser = new DecompositionParser();
    DecompositionCommon *common = new DecompositionCommon();
    DecompositionXmlWriter *xmlWriter = new DecompositionXmlWriter();

    decParser -> parse(xmlInputFile,xmlCutFile);
    RttTgenDkIxlInterlocking* interlocking = decParser -> getInterlocking();
    RttTgenDkIxlNetworkLayout* network = interlocking -> getNetworkLayout();
```

```
    map<string, RttTgenDkIxlLinear*>& linears = network -> getLinears();

    SECTION("Disconnect between two given linears"){
        REQUIRE(linears["t2"] -> isNeighborWith("t3"));

        common -> disconnectNetwork(network, network -> findTrackSection("t2") ,
         ↪  network -> findTrackSection("t3"));
        REQUIRE(!linears["t2"] -> isNeighborWith("t3"));
        REQUIRE(linears["t2"] -> getNeighborOids().size() == 1);
        REQUIRE(linears["t3"] -> getNeighborOids().size() == 1);

        SECTION ("Test of function getReachableSections"){
            std::queue<string> sectionsToVisit;
            std::set<std::string> visited;

            SECTION("Down"){
                sectionsToVisit.push("t2");
                std::set<std::string> sections_down =
                    common ->  getReachableSections(network, sectionsToVisit,
                     ↪  visited);
                REQUIRE(sections_down.size() == 2);
            }

            SECTION("Up"){
                sectionsToVisit.push("t3");
                std::set<std::string> sections_up =
                    common ->  getReachableSections(network, sectionsToVisit,
                     ↪  visited);
                REQUIRE(sections_up.size() == 10);
            }
        }
    }
}
```

The result of the tests are obtained by creating an executable and running it, a list of successful and unsuccessful assertions are printed. An example of one of the assertion output is shown below.

```
PASSED:
  REQUIRE( sections_up.size() == 10 )
with expansion:
  10 == 10
```

The overall result of the tests is also obtained. The result is:

```
All tests passed (43 assertions in 9 test cases)
```

CHAPTER 7

Experiments

In this chapter, the tool is examined with different cases. The experiments will test if the tool can handle different networks, both self-created and real-world examples. A successful decomposition should result in a statically correct network that will be accepted by the verification tool.

7.1 Goal of the Experiments

The main goal of the experiments is to see how well the decomposition tool handles networks of different types and sizes. So essentially, the main goal is to see if the tool works. The second goal is to see if the decomposed networks are in fact easier on the verification tool and if they are executed successfully.

7.2 Experimental Approach

For each of the case, the original network is verified using the verification tool (monolithically). Then the sub-networks created using the decomposition tool are also verified (compositionally). If a statically correct output is acquired from the decomposition tool, then the main goal is achieved. The results from both executions are compared by measuring time (in seconds) and memory usage (in MB). Besides, statistical information about the network layouts is also provided such as: number of *linears*, *points*, *marker boards* and *routes*.

The experiments are performed on a server running Ubuntu (14.04.1) with Intel(R) Core(TM) i5 CPU 750 @ 2.67GHz and 16GB RAM.

7.3 Mini Extended

One of the most heavily used and tested network layout is called *mini-e*. It is an extension of the self-created *mini* layout previously showed in Figure 2.2. The *mini-e* network can be seen down below in Figure 7.1. The reason for not using the *mini* layout for experiments is because of its limitations regarding applicable cuts. This extended layout allows cuts in the middle of two the points and on the sections stretching both sides. The XML definition for this layout can be seen in Appendix C.1.1.

For this particular case, the following applicable border cuts exists: {t2,t3}, {t5,t6}, {t7,t8} and {t10,t11}. For best possible results with the verification tool, the combination that generates most sub-networks is chosen. All of the border cuts listed

can be applied. However, {t5,t6} and {t7,t8} must be applied as a cluster cut to output valid sub-networks. A multi cut is defined as seen below which will enable us to generate all of the sub-networks in one go.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <multiCut id="miniEMultiCut">
        <clusterCut id="miniEClusterCut1">
            <borderCut id="miniEClusterCut1_BC1">
                <trackSection id="t5" side="down" type="linear"/>
                <trackSection id="t6" side="up" type="linear"/>
            </borderCut>
            <borderCut id="miniEClusterCut1_BC2">
                <trackSection id="t7" side="down" type="linear"/>
                <trackSection id="t8" side="up" type="linear"/>
            </borderCut>
        </clusterCut>
        <borderCut id="miniRBorderCut1">
            <trackSection id="t2" side="down" type="linear"/>
            <trackSection id="t3" side="up" type="linear"/>
        </borderCut>
        <borderCut id="miniBorderCut2">
            <trackSection id="t10" side="down" type="linear"/>
            <trackSection id="t11" side="up" type="linear"/>
        </borderCut>
    </multiCut>
</xmi:XMI>
```

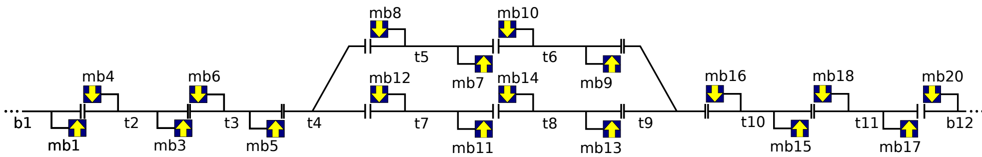


Figure 7.1: The *mini-e* network layout.

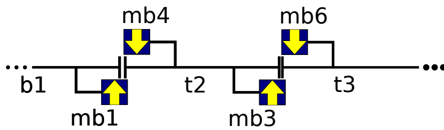


Figure 7.2: Sub-network *mini-e_mc_1*.

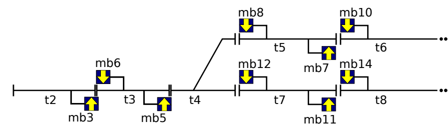


Figure 7.3: Sub-network *mini-e_mc_2*.

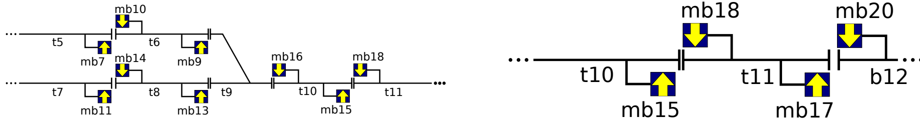


Figure 7.4: Sub-network *mini-e_mc_3*.

Figure 7.5: Sub-network *mini-e_mc_4*.

By applying the multi cut, four sub-networks are successfully produced. The tool is lightweight and the decomposition happens instantaneously. The full XML definitions for the sub-networks can be seen in Appendix C and the Figures 7.2 to 7.5 show sub-networks generated.

The verification tool successfully verified the safety properties for all of the networks. Table 7.1 shows the verification metrics for the compositional analysis of the sub-networks. The metrics for the compositional analysis are obtained by summing the results from the sub-network metrics, except the memory usage, which is the maximum memory usage of any verification.

The results show that the compositional verification of the network requires much less memory than the monolithic run, only third in this case. The running time does also show positive results, which is only 8.47 seconds opposed to the 25.31 seconds in the monolithic run. The table shows the result for a synchronized run of the compositional verification, if all of the sub-networks were to be executed concurrently, then the memory usage would increase. In that case, the memory usage is 240.80 MB, which is still less than the monolithic run.

	Linears	Points	Signals	Routes	Time (s)	Memory (MB)
mini-e_mc_1	3	0	4	2	0.09	19.1
mini-e_mc_2	6	1	9	9	4.59	101.4
mini-e_mc_3	6	1	9	9	3.70	101.2
mini-e_mc_4	3	0	4	2	0.09	19.0
Compositional	130	39	152	191	8.47	101.4
mini-e	10	2	18	22	25.31	309.5

Table 7.1: Comparison between monolithic and compositional execution of the verification tool.

7.4 EDL

EDL stands for Early Deployment Line and is the first the first regional line in Denmark to be commissioned in the Danish Signalling Programme. The line stretches from Roskilde- to Næstved station and is 55km in total. The layout of the EDL line is confidential and can therefore not be included in this paper. A compositional analysis has already been done in the paper [7], where a manual decomposition was done by hand.

The same decomposition has been successfully replicated by the developed tool. The existence of intermediate stations between Roskilde and Næstved adds up to eight sub-networks. Where each sub-network is a station. The metrics from the paper are shown here in Table 7.2. This table contains an extra row containing the state space dimension (in logarithmic scale).

	Linears	Points	Signals	Routes	$log_{10}(\|S\|)$	Time	Memory
Gadstrup	14	3	16	21	73	86	513
Havdrup	10	2	12	14	51	20	263
L. Skensved	20	4	22	28	101	223	1212
Køge	52	22	54	66	306	6581	9393
Herfølge	6	2	10	14	39	13	191
Tureby	6	2	10	14	39	12	180
Haslev	10	2	12	14	51	22	261
Holme-Olstrup	12	2	16	20	63	27	350
Compositional	130	39	152	191	682	6984	9393
EDL	116	39	138	191	682	22793	26484

Table 7.2: Comparison between monolithic and compositional verification of the early development line. Metrics are obtained from another machine.[7].

Another successful verification of safety properties of all the networks shows the same picture. The compositional verification of the eight sub-networks shows that the verification time is approximately a third of the monolithic analysis. The memory usage of a synchronized run shows a maximum of 9393 MB, whereas a concurrent run would result in 12363 MB. Both are significantly less than the 26484 MB used by the monolithic analysis.

7.5 Roskilde station

The Roskilde station is another network that has been experimented with. It is also confidential, so the same conditions as EDL do apply regarding including it in the paper. The station is huge and does result in an out-of-memory error when executed monolithically because of the state explosion problem.

Unfortunately, the complexity of the station made it very difficult to find applicable cuts, and only one cluster cut was found applicable. This big cluster cut contains 7 single cuts that together cuts in the middle of the station. The decomposition successfully decomposed the station resulting in two sub-networks *Roskilde (Down)* and *Roskilde (Up)*. The cut was unfortunately not enough to overcome the state explosion problem. However, the cut remarkably improved the time before termination for the smaller network *Roskilde (Up)*. See Table 7.3 for the measured time before termination.

	Linears	Points	Signals	Routes	Time (before termination)
Roskilde (Down)	76	43	93	310	221
Roskilde (Up)	40	22	50	142	6783
Compositional	116	65	143	452	-
Roskilde	102	65	129	468	415

Table 7.3: Comparison between monolithic and compositional execution run time before termination.

CHAPTER 8 Discussion

This chapter contains a discussion of the project and the developed tool. The results of the experiments are recalled and discussed if the tool delivers what it promises. The current limitations of the tool and the consequences it may possess are discussed. Finally, potential improvements are presented that can be included in a future work of the tool.

8.1 Results

Different networks have been used for experimental purposes. The tool did certainly satisfy its main goal of decomposing all given networks. The self-created network *mini-e* showed that it could be applied a border cut with different types of decomposition methods, such as single-, cluster and multi cut. Successful decomposition was also achieved with the EDL and Roskilde network. Both *mini-e* and EDL showed better certifications metrics after the decomposition. A cluster cut decomposition of Roskilde, however, was not enough to overcome the state explosion problem. The limited time with this confidential network was not enough to find more than one applicable cut. A multi cut specification with more cuts could have resulted in smaller sub-networks, thus resulting in a successful verification. If other cut types other than border cut was included in this tool, such as the *linear cut* and *horizontal cut*, then the possibility of specifying such multi cut could have been easier.

The performance of the decomposition tool has been great, and all executions were instantaneous. This is due to the tool being lightweight and developed in C++. The user experience is smooth and easy. A non-technical user can easily learn the use of the command line tool. The tool shows great potential to be used together with the verification tool in the future.

8.2 Limitations

The tool does have some limitations that can be tedious in some cases. Some of them are already mentioned in the paper. These limitations are:

- *Limited cut types.* Only border cut is included in this project. This may become a limitation depending on the network that has to be decomposed.
- *Code dependence.* The tool uses a shared library from RobustRailS' verification project (see Appendix A). This is a limitation only if the tool will be kept separate

from rest of the tool-chain. The shared library can influence the decomposition tool if changed and recompiled.

- *Not considered network elements.* For simplicity, some elements of the network layouts were not considered, such as level crossings.

8.3 Directions for Future Work

Potential improvements for the decomposition tool are listed below. These improvements can extend the tool to have more functionalities.

- *A Static checker.* A static checker can be used ensure that the given- and generated network is statically correct. Currently, the RobustRailS' tool provides this check for network inputs, however, manually running this tool for every sub-network can be tedious.
- *Automatic find of cuts.* Develop an algorithm that searches for applicable cuts in the network. A function as such will fully automate decomposition of networks and free the user of manually defining cut specifications.
- *Graphical user interface.* A graphical user interface can make it easier for the user to specify where to apply the cut. A good option here would be to integrate the decomposition tool with the graphical tool developed by Andreas [5].

CHAPTER 9
Conclusion

The goal of this project was to support the RobustRailS' compositional verification by developing a decomposition tool. The verification tool suffered from the state explosion problem that is seen with model-checkers. This made it difficult to verify large networks.

To support the verifications tool, a decomposition tool working in the command line environment is developed. Before implementing, an analysis of the network layouts and cut types have been done to gain knowledge of the domain and develop decomposition methods. Then, requirements have been determined, from abstract to concrete. RSL has been used both for requirement- and design specification of the software. With the specifications in place, the implementation of the tool has been made using C++.

The tool uses the cut type *border cut* with a variety of decomposition methods such as *single-*, *cluster-* and *multi cut*. Network layouts and cut specifications in XML format are used by the tool to generate the sub-networks. The sub-networks are written as XML files to the file system which can be used together with the verification tool.

The tool has been tested using unit tests, and by experiments. The unit tests have been solely used to test the functionalities of the tool. The experiments, however, has been a combination of experimenting the decomposition- and verification tool. Comparison between the monolithic and compositional verification reveals that the decomposition tool can support the verification tool and potentially solve the state explosion problem.

Appendix A
Installing the tool

A.1 Prerequisites

To be able to install/build the project, some libraries must first be installed. These libraries are common to C++ and are often available through the default system package manager by OS. In case the operating system is Ubuntu/Linux, the libraries can be installed with:

```
sudo apt-get install cmake
sudo apt-get install build-essential g++ python-dev autotools-dev libicu-dev li
bbz2-dev libboost-all-dev libclang-dev
sudo apt-get install libxml2 libxml2-dev
sudo apt-get install doxygen
sudo apt-get install zlibc
sudo apt-get install sqlite3
sudo apt-get install libsqlite3-0 libsqlite3-dev
sudo apt-get install libc++1 libc++-dev
sudo apt-get install cvs
```

It is important that the library versions are up to date and are the same during the build and executions. When the libraries are updated, then the project must be built again, otherwise it will look after older library versions which no longer exist.

A.1.1 Needed files

The first thing that must be ensured is that we have all of the files needed to build. The whole project folder will be archived in a tar file. Running `ls` reveals the following files in the root project folder:

```
build-aux          CMakeCPackOptions.cmake  decomposers  lib                    parsers
build.bat          CMakeFiles               dkixl        librttmbtlib.so        README.cmake.txt
build.Debug        cmake_install.cmake      exceptions   libsonolar-shared.so   rtt-mbt
build_dk_gcc.sh    CMakeLists.txt           executables  logging                symboltable
build.Release      copySharedLibs.sh        IMR          Makefile               testprocs
build.sh           CPackConfig.cmake        include      math                   types
cmake              CPackSourceConfig.cmake  intvallib    memory                 utils
CMakeCache.txt     CTestTestfile.cmake      latticelib   memorymodel            visitors
```

Notice that only the folder `decomposers`, `dkixl` and `parsers` have been used in development. All the other files and folders are dependencies.

A.2 Building

The project's building process is handled by CMake. To build the project, do the following in exact order:

1. Go to the root folder of the project.
2. Run `sh Build.sh` in the terminal - needed only first time
3. Run `make`. The compiler will end with some *undefined reference* errors, however they are expected since not all of the project folders are available due to confidentiality.
4. Run the command `sh copSharedLibs.sh`. This will copy the shared libs from the folder `lib` to the root folder.
5. Rerun `make` in terminal. No errors should appear this time and the executables `Decompose` and `tests` should be generated successfully in the paths:
   a) `[rr-project]/decomposers/networkdecomposition/decompose`
   b) `[rr-project]/decomposers/networkDecomposition/tests`

Where `[rr-project]` is the root folder of the project. When a change is made to the source code, rerunning `make` a single time should be sufficient.

A.2.1 Making shared libraries accessible

The decomposition tool uses two shared files from the verification tool. These shared libraries are called `librttmbtlib.so` and `libsonolar-shared.so`. Copies of the shared libraries are already included in the `[rr-project]/lib` folder. In the building process, the user already made sure that the shared libraries are also copied to the root folder, this ensures that the tool can be executed in its original path listed above.

However, if the user wants to move the binaries somewhere else in the system, then the shared libraries must be copied to a system library `PATH`. A good candidate for a library path is `/usr/lib/`.

APPENDIX B
Using the tool

B.1 Prerequisites

This user guide assumes that you already have installed the tool by building the project and installing its dependencies, if not, take a look at Appendix A for how to do so.

B.2 Usage

To see the usage list simply run `./Decompose` without any options or with the `-h` flag. The user must be in the path of `Decompose` binary file.

```
Usage:
        -h,--help               Show this help message
        -c,--cut                Specify the cut file path
        -n,--network            Specify the network file path
```

If this message does not appear, then the tool was not properly installed. If it appears, you may continue with the guide. Let us start with an example. Let us decompose a network called *mini-e*. The XML definitions of the cut specification and the network layout before and after the decomposition are included in the Appendix C.

The user must provide a relative path, both for the network layout and the cut specification. The `-n` and `-c` options are used to accomplish this. The following command is executed to decompose the mini-e network with singe cut:

```
./Decompose -n mini-e.xml -c xml/mini-e_sc.xml
```

If successful, the detected cut type and a list of generated XML files will be printed:

```
BorderCut Detected
Generated xml file: ./mini-e_down.xml
Generated xml file: ./mini-e_up.xml
```

Another example is shown below. To decompose the mini-e network with a multi cut specification, the same command is executed but with a different cut path:

```
./Decompose -n mini-e.xml -c xml/mini-e_mc.xml
```

If successful, the detected cut type and a list of generated XML files will be printed:

```
MultiCut Detected
Generated xml file: ./xml/mini-e_mc_1.xml
Generated xml file: ./xml/mini-e_mc_2.xml
Generated xml file: ./xml/mini-e_mc_3.xml
Generated xml file: ./xml/mini-e_mc_4.xml
```

As it can be seen, the usage of the tool is very straightforward.

APPENDIX C
Networks and Cuts in XML

This appendix shows network layouts and cut specifications in XML formats. The XML definition before

C.1 Original network

C.1.1 mini-e.xml


```xml
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <trackSection id="b1" length="100" type="linear">
                <neighbor ref="t2" side="up"/>
            </trackSection>
            <trackSection id="t2" length="100" type="linear">
                <neighbor ref="b1" side="down"/>
                <neighbor ref="t3" side="up"/>
            </trackSection>
            <trackSection id="t3" length="100" type="linear">
                <neighbor ref="t2" side="down"/>
                <neighbor ref="t4" side="up"/>
            </trackSection>
            <trackSection id="t4" length="100" type="point">
                <neighbor ref="t5" side="plus"/>
                <neighbor ref="t7" side="minus"/>
                <neighbor ref="t3" side="stem"/>
            </trackSection>
            <trackSection id="t5" length="100" type="linear">
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t6" side="up"/>
            </trackSection>
            <trackSection id="t6" length="100" type="linear">
                <neighbor ref="t5" side="down"/>
                <neighbor ref="t9" side="up"/>
            </trackSection>
            <trackSection id="t7" length="100" type="linear">
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t8" side="up"/>
            </trackSection>
            <trackSection id="t8" length="100" type="linear">
```

```
                <neighbor ref="t7" side="down"/>
                <neighbor ref="t9" side="up"/>
            </trackSection>
            <trackSection id="t9" length="100" type="point">
                <neighbor ref="t6" side="plus"/>
                <neighbor ref="t8" side="minus"/>
                <neighbor ref="t10" side="stem"/>
            </trackSection>
            <trackSection id="t10" length="100" type="linear">
                <neighbor ref="t9" side="down"/>
                <neighbor ref="t11" side="up"/>
            </trackSection>
            <trackSection id="t11" length="100" type="linear">
                <neighbor ref="t10" side="down"/>
                <neighbor ref="b12" side="up"/>
            </trackSection>
            <trackSection id="b12" length="100" type="linear">
                <neighbor ref="t11" side="down"/>
            </trackSection>
            <markerboard distance="50" id="mb1" mounted="up" track="b1"/>
            <markerboard distance="50" id="mb3" mounted="up" track="t2"/>
            <markerboard distance="50" id="mb4" mounted="down" track="t2"/>
            <markerboard distance="50" id="mb5" mounted="up" track="t3"/>
            <markerboard distance="50" id="mb6" mounted="down" track="t3"/>
            <markerboard distance="50" id="mb7" mounted="up" track="t5"/>
            <markerboard distance="50" id="mb8" mounted="down" track="t5"/>
            <markerboard distance="50" id="mb9" mounted="up" track="t6"/>
            <markerboard distance="50" id="mb10" mounted="down" track="t6"/>
            <markerboard distance="50" id="mb11" mounted="up" track="t7"/>
            <markerboard distance="50" id="mb12" mounted="down" track="t7"/>
            <markerboard distance="50" id="mb13" mounted="up" track="t8"/>
            <markerboard distance="50" id="mb14" mounted="down" track="t8"/>
            <markerboard distance="50" id="mb15" mounted="up" track="t10"/>
            <markerboard distance="50" id="mb16" mounted="down" track="t10"/>
            <markerboard distance="50" id="mb17" mounted="up" track="t11"/>
            <markerboard distance="50" id="mb18" mounted="down" track="t11"/>
            <markerboard distance="50" id="mb20" mounted="down" track="b12"/>
        </network>
        <routetable id="miniroutetable" network="mininetwork">
        </routetable>
    </interlocking>
</xmi:XMI>
```

C.2 Cut Specifications

C.2.1 mini-e_sc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <borderCut id="miniBorderCut" version="0.1">
            <trackSection id="t2" side="down" type="linear"/>
            <trackSection id="t3" side="up" type="linear"/>
```

```
    </borderCut>
</xmi:XMI>
```

C.2.2 mini-e_cc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <clusterCut id="miniClusterCut">
        <borderCut id="miniBorderCut1">
            <trackSection id="t5" side="down" type="linear"/>
            <trackSection id="t6" side="up" type="linear"/>
        </borderCut>
        <borderCut id="miniBorderCut1">
            <trackSection id="t7" side="down" type="linear"/>
            <trackSection id="t8" side="up" type="linear"/>
        </borderCut>
    </clusterCut>
</xmi:XMI>
```

C.2.3 mini-e_mc.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <multiCut id="miniEMultiCut">
        <clusterCut id="miniEClusterCut1">
            <borderCut id="miniEClusterCut1_BC1">
                <trackSection id="t5" side="down" type="linear"/>
                <trackSection id="t6" side="up" type="linear"/>
            </borderCut>
            <borderCut id="miniEClusterCut1_BC2">
                <trackSection id="t7" side="down" type="linear"/>
                <trackSection id="t8" side="up" type="linear"/>
            </borderCut>
        </clusterCut>
        <borderCut id="miniRBorderCut1">
            <trackSection id="t2" side="down" type="linear"/>
            <trackSection id="t3" side="up" type="linear"/>
        </borderCut>
        <borderCut id="miniBorderCut2">
            <trackSection id="t10" side="down" type="linear"/>
            <trackSection id="t11" side="up" type="linear"/>
        </borderCut>
    </multiCut>
</xmi:XMI>
```

C.3 Sub-networks

C.3.1 After Single Cut

mini-e_sc_down.xml

```
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="b1" length="100" type="linear">
                <neighbor ref="t2" side="up"/>
            </trackSection>
            <trackSection id="t2" length="100" type="linear">
                <neighbor ref="b1" side="down"/>
                <neighbor ref="t3" side="up"/>
            </trackSection>
            <trackSection id="t3" length="100" type="linear">
                <neighbor ref="t2" side="down"/>
            </trackSection>
            <!--Markerboards-->
            <markerboard id="mb1" distance="50" mounted="up" track="b1"/>
            <markerboard id="mb3" distance="50" mounted="up" track="t2"/>
            <markerboard id="mb4" distance="50" mounted="down" track="t2"/>
            <markerboard id="mb6" distance="50" mounted="down" track="t3"/>
        </network>
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
</xmi:XMI>
```

mini-e_sc_up.xml

```
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="b12" length="100" type="linear">
                <neighbor ref="t11" side="down"/>
            </trackSection>
            <trackSection id="t10" length="100" type="linear">
                <neighbor ref="t9" side="down"/>
                <neighbor ref="t11" side="up"/>
            </trackSection>
            <trackSection id="t11" length="100" type="linear">
                <neighbor ref="t10" side="down"/>
                <neighbor ref="b12" side="up"/>
            </trackSection>
            <trackSection id="t2" length="100" type="linear">
```

```
            <neighbor ref="t3" side="up"/>
        </trackSection>
        <trackSection id="t3" length="100" type="linear">
            <neighbor ref="t2" side="down"/>
            <neighbor ref="t4" side="up"/>
        </trackSection>
        <trackSection id="t5" length="100" type="linear">
            <neighbor ref="t4" side="down"/>
            <neighbor ref="t6" side="up"/>
        </trackSection>
        <trackSection id="t6" length="100" type="linear">
            <neighbor ref="t5" side="down"/>
            <neighbor ref="t9" side="up"/>
        </trackSection>
        <trackSection id="t7" length="100" type="linear">
            <neighbor ref="t4" side="down"/>
            <neighbor ref="t8" side="up"/>
        </trackSection>
        <trackSection id="t8" length="100" type="linear">
            <neighbor ref="t7" side="down"/>
            <neighbor ref="t9" side="up"/>
        </trackSection>
        <!--Points-->
        <trackSection id="t4" length="100" type="point">
            <neighbor ref="t5" side="plus"/>
            <neighbor ref="t7" side="minus"/>
            <neighbor ref="t3" side="stem"/>
        </trackSection>
        <trackSection id="t9" length="100" type="point">
            <neighbor ref="t6" side="plus"/>
            <neighbor ref="t8" side="minus"/>
            <neighbor ref="t10" side="stem"/>
        </trackSection>
        <!--Markerboards-->
        <markerboard id="mb10" distance="50" mounted="down" track="t6"/>
        <markerboard id="mb11" distance="50" mounted="up" track="t7"/>
        <markerboard id="mb12" distance="50" mounted="down" track="t7"/>
        <markerboard id="mb13" distance="50" mounted="up" track="t8"/>
        <markerboard id="mb14" distance="50" mounted="down" track="t8"/>
        <markerboard id="mb15" distance="50" mounted="up" track="t10"/>
        <markerboard id="mb16" distance="50" mounted="down" track="t10"/>
        <markerboard id="mb17" distance="50" mounted="up" track="t11"/>
        <markerboard id="mb18" distance="50" mounted="down" track="t11"/>
        <markerboard id="mb20" distance="50" mounted="down" track="b12"/>
        <markerboard id="mb3" distance="50" mounted="up" track="t2"/>
        <markerboard id="mb5" distance="50" mounted="up" track="t3"/>
        <markerboard id="mb6" distance="50" mounted="down" track="t3"/>
        <markerboard id="mb7" distance="50" mounted="up" track="t5"/>
        <markerboard id="mb8" distance="50" mounted="down" track="t5"/>
        <markerboard id="mb9" distance="50" mounted="up" track="t6"/>
    </network>
    <routetable id="miniroutetable" network="mini-e-network"/>
  </interlocking>
</xmi:XMI>
```

C.3.2 After Cluster Cut

mini-e_cc_down.xml

```xml
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="b1" length="100" type="linear">
                <neighbor ref="t2" side="up"/>
            </trackSection>
            <trackSection id="t2" length="100" type="linear">
                <neighbor ref="b1" side="down"/>
                <neighbor ref="t3" side="up"/>
            </trackSection>
            <trackSection id="t3" length="100" type="linear">
                <neighbor ref="t2" side="down"/>
                <neighbor ref="t4" side="up"/>
            </trackSection>
            <trackSection id="t5" length="100" type="linear">
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t6" side="up"/>
            </trackSection>
            <trackSection id="t6" length="100" type="linear">
                <neighbor ref="t5" side="down"/>
            </trackSection>
            <trackSection id="t7" length="100" type="linear">
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t8" side="up"/>
            </trackSection>
            <trackSection id="t8" length="100" type="linear">
                <neighbor ref="t7" side="down"/>
            </trackSection>
            <!--Points-->
            <trackSection id="t4" length="100" type="point">
                <neighbor ref="t5" side="plus"/>
                <neighbor ref="t7" side="minus"/>
                <neighbor ref="t3" side="stem"/>
            </trackSection>
            <!--Markerboards-->
            <markerboard id="mb1" distance="50" mounted="up" track="b1"/>
            <markerboard id="mb10" distance="50" mounted="down" track="t6"/>
            <markerboard id="mb11" distance="50" mounted="up" track="t7"/>
            <markerboard id="mb12" distance="50" mounted="down" track="t7"/>
            <markerboard id="mb14" distance="50" mounted="down" track="t8"/>
            <markerboard id="mb3" distance="50" mounted="up" track="t2"/>
            <markerboard id="mb4" distance="50" mounted="down" track="t2"/>
            <markerboard id="mb5" distance="50" mounted="up" track="t3"/>
            <markerboard id="mb6" distance="50" mounted="down" track="t3"/>
            <markerboard id="mb7" distance="50" mounted="up" track="t5"/>
            <markerboard id="mb8" distance="50" mounted="down" track="t5"/>
        </network>
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
```

```
</xmi:XMI>
```

mini-e_cc_up.xml


```xml
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="b12" length="100" type="linear">
                <neighbor ref="t11" side="down"/>
            </trackSection>
            <trackSection id="t10" length="100" type="linear">
                <neighbor ref="t9" side="down"/>
                <neighbor ref="t11" side="up"/>
            </trackSection>
            <trackSection id="t11" length="100" type="linear">
                <neighbor ref="t10" side="down"/>
                <neighbor ref="b12" side="up"/>
            </trackSection>
            <trackSection id="t5" length="100" type="linear">
                <neighbor ref="t6" side="up"/>
            </trackSection>
            <trackSection id="t6" length="100" type="linear">
                <neighbor ref="t5" side="down"/>
                <neighbor ref="t9" side="up"/>
            </trackSection>
            <trackSection id="t7" length="100" type="linear">
                <neighbor ref="t8" side="up"/>
            </trackSection>
            <trackSection id="t8" length="100" type="linear">
                <neighbor ref="t7" side="down"/>
                <neighbor ref="t9" side="up"/>
            </trackSection>
            <!--Points-->
            <trackSection id="t9" length="100" type="point">
                <neighbor ref="t6" side="plus"/>
                <neighbor ref="t8" side="minus"/>
                <neighbor ref="t10" side="stem"/>
            </trackSection>
            <!--Markerboards-->
            <markerboard id="mb10" distance="50" mounted="down" track="t6"/>
            <markerboard id="mb11" distance="50" mounted="up" track="t7"/>
            <markerboard id="mb13" distance="50" mounted="up" track="t8"/>
            <markerboard id="mb14" distance="50" mounted="down" track="t8"/>
            <markerboard id="mb15" distance="50" mounted="up" track="t10"/>
            <markerboard id="mb16" distance="50" mounted="down" track="t10"/>
            <markerboard id="mb17" distance="50" mounted="up" track="t11"/>
            <markerboard id="mb18" distance="50" mounted="down" track="t11"/>
            <markerboard id="mb20" distance="50" mounted="down" track="b12"/>
            <markerboard id="mb7" distance="50" mounted="up" track="t5"/>
            <markerboard id="mb9" distance="50" mounted="up" track="t6"/>
        </network>
```

```xml
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
</xmi:XMI>
```

C.3.3 After Multi Cut

mini-e_mc_1.xml

```xml
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="b1" length="100" type="linear">
                <neighbor ref="t2" side="up"/>
            </trackSection>
            <trackSection id="t2" length="100" type="linear">
                <neighbor ref="b1" side="down"/>
                <neighbor ref="t3" side="up"/>
            </trackSection>
            <trackSection id="t3" length="100" type="linear">
                <neighbor ref="t2" side="down"/>
            </trackSection>
            <!--Markerboards-->
            <markerboard id="mb1" distance="50" mounted="up" track="b1"/>
            <markerboard id="mb3" distance="50" mounted="up" track="t2"/>
            <markerboard id="mb4" distance="50" mounted="down" track="t2"/>
            <markerboard id="mb6" distance="50" mounted="down" track="t3"/>
        </network>
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
</xmi:XMI>
```

mini-e_mc_2.xml

```xml
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="t2" length="100" type="linear">
                <neighbor ref="t3" side="up"/>
            </trackSection>
            <trackSection id="t3" length="100" type="linear">
                <neighbor ref="t2" side="down"/>
                <neighbor ref="t4" side="up"/>
            </trackSection>
            <trackSection id="t5" length="100" type="linear">
```

```
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t6" side="up"/>
            </trackSection>
            <trackSection id="t6" length="100" type="linear">
                <neighbor ref="t5" side="down"/>
            </trackSection>
            <trackSection id="t7" length="100" type="linear">
                <neighbor ref="t4" side="down"/>
                <neighbor ref="t8" side="up"/>
            </trackSection>
            <trackSection id="t8" length="100" type="linear">
                <neighbor ref="t7" side="down"/>
            </trackSection>
            <!--Points-->
            <trackSection id="t4" length="100" type="point">
                <neighbor ref="t5" side="plus"/>
                <neighbor ref="t7" side="minus"/>
                <neighbor ref="t3" side="stem"/>
            </trackSection>
            <!--Markerboards-->
            <markerboard id="mb10" distance="50" mounted="down" track="t6"/>
            <markerboard id="mb11" distance="50" mounted="up" track="t7"/>
            <markerboard id="mb12" distance="50" mounted="down" track="t7"/>
            <markerboard id="mb14" distance="50" mounted="down" track="t8"/>
            <markerboard id="mb3" distance="50" mounted="up" track="t2"/>
            <markerboard id="mb5" distance="50" mounted="up" track="t3"/>
            <markerboard id="mb6" distance="50" mounted="down" track="t3"/>
            <markerboard id="mb7" distance="50" mounted="up" track="t5"/>
            <markerboard id="mb8" distance="50" mounted="down" track="t5"/>
        </network>
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
</xmi:XMI>
```

mini-e_mc_3.xml

```
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
    <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
    <interlocking id="mini-e" version="0.1">
        <network id="mini-e-network">
            <!--Linears-->
            <trackSection id="t10" length="100" type="linear">
                <neighbor ref="t9" side="down"/>
                <neighbor ref="t11" side="up"/>
            </trackSection>
            <trackSection id="t11" length="100" type="linear">
                <neighbor ref="t10" side="down"/>
            </trackSection>
            <trackSection id="t5" length="100" type="linear">
                <neighbor ref="t6" side="up"/>
            </trackSection>
            <trackSection id="t6" length="100" type="linear">
                <neighbor ref="t5" side="down"/>
```

```
            <neighbor ref="t9" side="up"/>
        </trackSection>
        <trackSection id="t7" length="100" type="linear">
            <neighbor ref="t8" side="up"/>
        </trackSection>
        <trackSection id="t8" length="100" type="linear">
            <neighbor ref="t7" side="down"/>
            <neighbor ref="t9" side="up"/>
        </trackSection>
        <!--Points-->
        <trackSection id="t9" length="100" type="point">
            <neighbor ref="t6" side="plus"/>
            <neighbor ref="t8" side="minus"/>
            <neighbor ref="t10" side="stem"/>
        </trackSection>
        <!--Markerboards-->
        <markerboard id="mb10" distance="50" mounted="down" track="t6"/>
        <markerboard id="mb11" distance="50" mounted="up" track="t7"/>
        <markerboard id="mb13" distance="50" mounted="up" track="t8"/>
        <markerboard id="mb14" distance="50" mounted="down" track="t8"/>
        <markerboard id="mb15" distance="50" mounted="up" track="t10"/>
        <markerboard id="mb16" distance="50" mounted="down" track="t10"/>
        <markerboard id="mb18" distance="50" mounted="down" track="t11"/>
        <markerboard id="mb7" distance="50" mounted="up" track="t5"/>
        <markerboard id="mb9" distance="50" mounted="up" track="t6"/>
    </network>
    <routetable id="miniroutetable" network="mini-e-network"/>
  </interlocking>
</xmi:XMI>
```

mini-e_mc_4.xml

```
<?xml version="1.0" encoding="UTF8"?>
<xmi:XMI xmi:version="2.4.1" xmlns:xmi="http://www.omg.org/spec/XMI/2.4.1">
  <xmi:Documentation exporter="DK-IXL" exporterVersion="0.1"/>
  <interlocking id="mini-e" version="0.1">
    <network id="mini-e-network">
        <!--Linears-->
        <trackSection id="b12" length="100" type="linear">
            <neighbor ref="t11" side="down"/>
        </trackSection>
        <trackSection id="t10" length="100" type="linear">
            <neighbor ref="t11" side="up"/>
        </trackSection>
        <trackSection id="t11" length="100" type="linear">
            <neighbor ref="t10" side="down"/>
            <neighbor ref="b12" side="up"/>
        </trackSection>
        <!--Markerboards-->
        <markerboard id="mb15" distance="50" mounted="up" track="t10"/>
        <markerboard id="mb17" distance="50" mounted="up" track="t11"/>
        <markerboard id="mb18" distance="50" mounted="down" track="t11"/>
        <markerboard id="mb20" distance="50" mounted="down" track="b12"/>
    </network>
```

```
        <routetable id="miniroutetable" network="mini-e-network"/>
    </interlocking>
</xmi:XMI>
```

APPENDIX D

Tests

This appendix contains the test files for RSL and their results. The C++ test cases are summarized in a table because the source code can be overwhelming.

D.1 Decomposition_TEST

This file contains all of the RSL tests.

```
/*=======================================================
*     File: $Name: Decomposition_TEST.rsl $
*  Created: $Date: 2017-04-10 15:16:14 $
*   Author: $Author: Cebrail Erdogan<s113414@student.dtu.dk>$
* Description: Test cases for decomposition of networks.
*=======================================================
*/

Decomposition_DESIGN
scheme Decomposition_TEST =
  with T in
  class
    object D : Decomposition_DESIGN

    value
      /**
      *  Extended mini layout.
      *  The extended mini layout contains extra linear sections compared to the mini layout.
      *
      *
      *                               t5     t6
      *                          /-|-----|------|-\
      *                         /                  \
      *  |-----|-----|-----|---/---|-----|------|---\--|------|------|------|
      *    b1    t2    t3    t4      t7    t8     t9    t10      t11    b12
      *
      *
      */

      -- linears
      b1 : D.C.ITG.I.L.Linear = D.C.ITG.I.L.mk_Linear([UP +> "t2"], 40),
      t2 : D.C.ITG.I.L.Linear =
        D.C.ITG.I.L.mk_Linear([DOWN +> "b1", UP +> "t3"], 100),
      t3 : D.C.ITG.I.L.Linear =
        D.C.ITG.I.L.mk_Linear([DOWN +> "t2", UP +> "t4"], 100),
      t5 : D.C.ITG.I.L.Linear =
```

```
      D.C.ITG.I.L.mk_Linear([DOWN +> "t4", UP +> "t6"], 100),
    t6 : D.C.ITG.I.L.Linear =
      D.C.ITG.I.L.mk_Linear([DOWN +> "t5", UP +> "t9"], 100),
    t7: D.C.ITG.I.L.Linear =
      D.C.ITG.I.L.mk_Linear([DOWN +> "t4", UP +> "t8"], 100),
    t8 : D.C.ITG.I.L.Linear =
      D.C.ITG.I.L.mk_Linear([DOWN +> "t7", UP +> "t9"], 100),
    t10 : D.C.ITG.I.L.Linear =
      D.C.ITG.I.L.mk_Linear([DOWN +> "t9", UP +> "t11"], 100),
    t11 : D.C.ITG.I.L.Linear =
      D.C.ITG.I.L.mk_Linear([DOWN +> "t10", UP +> "b12"], 100),
    b12 : D.C.ITG.I.L.Linear = D.C.ITG.I.L.mk_Linear([DOWN +> "t11"], 100),

    -- points
    t4 : D.C.ITG.I.L.Point =
      D.C.ITG.I.L.mk_Point(
         [NB_STEM +> "t3", NB_PLUS +> "t7", NB_MINUS +> "t5"], 100),
    t9 : D.C.ITG.I.L.Point =
      D.C.ITG.I.L.mk_Point(
         [NB_STEM +> "t10", NB_PLUS +> "t8", NB_MINUS +> "t6"], 100),

    -- signals (marker boards)
    mb1 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("b1", UP, 5),
    mb3 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t2", UP, 50),
    mb4 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t2", DOWN, 50),
    mb5 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t3", UP, 50),
    mb6 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t3", DOWN, 50),
    mb7 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t5", UP, 50),
    mb8 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t5", DOWN, 50),
    mb9 :  D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t6", UP, 50),
    mb10 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t6", DOWN, 50),
    mb11 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t7", UP, 50),
    mb12 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t7", DOWN, 50),
    mb13 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t8", UP, 50),
    mb14 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t8", DOWN, 50),
    mb15 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t10", UP, 50),
    mb16 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t10", DOWN, 50),
    mb17 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t11", UP, 50),
    mb18 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("t11", DOWN, 50),
    mb20 : D.C.ITG.I.L.MarkerBoard = D.C.ITG.I.L.mk_MarkerBoard("b12", DOWN, 50),

    -- all elements
    all_linears : SecId -m-> D.C.ITG.I.L.Linear =
      ["b1" +> b1, "t2" +> t2, "t3" +> t3, "t5" +> t5, "t6" +> t6,
       "t7" +> t7, "t8" +> t8, "t10" +> t10, "t11" +> t11, "b12" +> b12],
    all_points : SecId -m-> D.C.ITG.I.L.Point = ["t4" +> t4, "t9" +> t9],
    all_sigs : MbId -m-> D.C.ITG.I.L.MarkerBoard =
      ["mb1"  +> mb1,  "mb3"  +> mb3,  "mb4"  +> mb4,  "mb5"  +> mb5,
       "mb6"  +> mb6,  "mb7"  +> mb7,  "mb8"  +> mb8,  "mb9"  +> mb9,
       "mb10" +> mb10, "mb11" +> mb11, "mb12" +> mb12, "mb13" +> mb13, "mb14" +>
↪  mb14,
               "mb15" +> mb15, "mb16" +> mb16, "mb17" +> mb17, "mb18" +> mb18, "mb20"
↪  +> mb20],
    mini_ext : D.C.ITG.I.L.NetworkLayout =
      D.C.ITG.I.L.mk_NetworkLayout(all_linears, all_points, all_sigs),

    empty_n : D.C.ITG.I.L.NetworkLayout =
```

```
        D.C.ITG.I.L.mk_NetworkLayout([], [], []),

     -- cut specs
      sc1: D.C.SingleCut = D.C.mk_BorderCut("t2","t3"),
      sc2: D.C.SingleCut = D.C.mk_BorderCut("t10","t11"),
      sc3: D.C.SingleCut = D.C.mk_BorderCut("t5", "t6"),
      sc4: D.C.SingleCut = D.C.mk_BorderCut("t7", "t8"),
      sc5: D.C.SingleCut = D.C.mk_BorderCut("b1", "t2"), -- invalid
      sc6: D.C.SingleCut = D.C.mk_BorderCut("t20", "t21"), -- invalid
      sc7: D.C.SingleCut = D.C.mk_BorderCut("t30", "t31"), -- invalid
      cc1: D.C.ClusterCut = {sc3,sc4},
      cc2: D.C.ClusterCut = {sc2,sc4},
      cut1: D.C.Cut = D.C.Cut_from_SingleCut(sc1),
      cut2: D.C.Cut = D.C.Cut_from_SingleCut(sc2),
      cut3: D.C.Cut = D.C.Cut_from_SingleCut(sc7),
      cut4: D.C.Cut = D.C.Cut_from_ClusterCut(cc1),
      mc1: D.C.MultiCut = {cut1,cut2},
      mc2: D.C.MultiCut = {cut3} -- invalid


    -- Variables used
    test_case
      [single_cut]
        sc1,

      [cluster_cut]
        cc1,

      [multi_cut]
        mc1,

      [mini_ext]
        mini_ext

    -- Well-formedness tests
    test_case

      [mini_ext_is_wf]
        D.C.ITG.I.L.is_wf(mini_ext),

      [sc_wf]
        D.C.cut_wf(mini_ext, sc1),

      [cc_wf]
        D.C.cut_wf(mini_ext, cc1),

      [mc_wf]
        D.C.cuts_wf(mini_ext, mc1)

    -- Auxilliary function tests
    test_case
      [remove_nb_down]
        let s = D.C.remove_nb(t2, DOWN) in DOWN ~isin dom D.C.ITG.I.L.neighbors(s)
↪    end,

      [remove_nb_up]
        let s = D.C.remove_nb(t2, UP) in UP ~isin dom D.C.ITG.I.L.neighbors(s)  end,
```

```
      [remove_nb_minus]
        let s = D.C.remove_nb(t4, NB_MINUS) in  NB_MINUS ~isin dom
↪   D.C.ITG.I.L.neighbors(s) end,

      [remove_nb_plus]
        let s = D.C.remove_nb(t4, NB_PLUS) in  NB_PLUS ~isin dom
↪   D.C.ITG.I.L.neighbors(s) end,

      [remove_nb_stem]
        let s = D.C.remove_nb(t4, NB_STEM) in  NB_STEM ~isin dom
↪   D.C.ITG.I.L.neighbors(s) end,

      [isOverlapDistanceSatisfied_False]
        D.C.isOverlapDistanceSatisfied(mb1,b1) = false,

      [isOverlapDistanceSatisfied_True]
        D.C.isOverlapDistanceSatisfied(mb1,t2)

    -- Getters
    test_case
      [get_applicableCuts_True]
        let cuts = D.C.get_applicableCuts(mini_ext, mc1) in
            cuts = mc1
        end,

      [get_applicableCuts_False]
        let cuts = D.C.get_applicableCuts(mini_ext, mc2) in
            cuts = {}
        end,

      [get_all_section]
        let ss = D.C.get_all_sections(mini_ext, {"b1"}, {}) in
           card ss = card D.C.ITG.I.L.sections(mini_ext)
        end,

      [get_all_linears]
        let ls = D.C.get_all_linears(mini_ext,  D.C.ITG.I.L.sections(mini_ext)) in
           ls = D.C.ITG.I.L.linears(mini_ext)
        end,

      [get_all_points]
        let ps = D.C.get_all_points(mini_ext,  D.C.ITG.I.L.sections(mini_ext)) in
           ps = D.C.ITG.I.L.points(mini_ext)
        end,

      [get_all_signals]
        let mbs = D.C.get_all_signals(mini_ext, D.C.ITG.I.L.sections(mini_ext)) in
           mbs =  D.C.ITG.I.L.marker_boards(mini_ext)
        end,

      [get_reachableNetwork]
        let n = D.C.get_disconnectedNetwork(mini_ext, D.C.section_down(sc1),
↪   D.C.section_up(sc1)) in
            D.C.get_reachableNetwork(mini_ext, "b1") ~=
            D.C.get_reachableNetwork(n, "b1")
        end,
```

```
    [get_disconnectNetwork_sc1]
      let n = D.C.get_disconnectedNetwork(mini_ext, D.C.section_down(sc1),
       ↪  D.C.section_up(sc1)) in
          D.C.ITG.I.L.are_neighbors(D.C.section_down(sc1), D.C.section_up(sc1),
           ↪  mini_ext) /\
          D.C.ITG.I.L.are_neighbors(D.C.section_down(sc1), D.C.section_up(sc1), n) =
           ↪  false
      end


  -- Decompose function tests
  test_case
    --[decompose_sc1]
    --  D.decompose(mini_ext, sc1),

    [decomposed_sc1]
      D.C.decomposed_sec_repr(D.decompose(mini_ext, sc1)),

    [decomposed_sc1_wf]
      let (n1,n2) = D.decompose(mini_ext, sc1) in
          D.C.ITG.I.L.is_wf(n1) /\ D.C.ITG.I.L.is_wf(n2)
      end,

    [decomposed_cc1]
      D.C.decomposed_sec_repr(D.decompose(mini_ext, cc1)),

    [decomposed_cc1_wf]
      let (n1,n2) = D.decompose(mini_ext, cc1) in
          D.C.ITG.I.L.is_wf(n1) /\ D.C.ITG.I.L.is_wf(n2)
      end,

    [decompose_mc1]
      D.C.decomposed_sec_repr(D.decompose(mini_ext, mc1), {}),

    [decomposed_mc1_wf]
      let ns = D.decompose(mini_ext, mc1) in
          (all n : D.C.ITG.I.L.NetworkLayout :- n isin ns => D.C.ITG.I.L.is_wf(n))
      end,

    [decomposed_cluster_cut_apply_cut]
      D.C.apply_cut(mini_ext, cc1, DOWN)

  -- BCC tests
  -- Takes significant time to execute
  test_case
    [bcc1]
      D.C.BCC1(mini_ext, sc1),

    [bcc2_M]
      D.C.BCC2_M(mini_ext, sc1),

    [bcc2_T]
    let (n_down,n_up) = D.decompose(mini_ext, sc1) in
      D.C.BCC2_T(mini_ext, n_down, n_up)
    end,
```

```
    [bcc3]
    let (n_down,n_up) = D.decompose(mini_ext, sc1) in
      D.C.BCC3(mini_ext, n_down, n_up)
    end,

    [bcc4]
    let (n_down,n_up) = D.decompose(mini_ext, sc1) in
      D.C.BCC4(mini_ext, n_down, n_up)
    end,

    [bcc5]
    let (n_down,n_up) = D.decompose(mini_ext, sc1) in
      D.C.BCC5(mini_ext, n_down, n_up)
    end
end
```

D.2 RSL Results

```
[mini_ext_is_wf] true
[sc_wf] true
[cc_wf] true
[mc_wf] true
[remove_nb_down] true
[remove_nb_up] true
[remove_nb_minus] true
[remove_nb_plus] true
[remove_nb_stem] true
[isOverlapDistanceSatisfied_False] true
[isOverlapDistanceSatisfied_True] true
[get_applicableCuts_True] true
[get_applicableCuts_False] true
[get_all_section] true
[get_all_linears] true
[get_all_points] true
[get_all_signals] true
[get_reachableNetwork] true
[get_disconnectNetwork_sc1] true
[get_disconnectNetwork_with_points] true
[decomposed_sc1]
↪   ({"t2","t3","b1"},{"t2","b12","t11","t10","t7","t8","t6","t5","t3","t4","t9"})
[decomposed_sc1_wf] true
[decomposed_cc1]
↪   ({"b1","t2","t3","t6","t5","t8","t7","t4"},{"t7","b12","t11","t10","t5","t6","t8","t9"})
[decomposed_cc1_wf] true
[decompose_mc1] {{"t2","t3","b1"},{"t2","t8","t7","t6","t5","t3","t4"},
{"t11","b12","t10"},{"t7","t8","t5","t6","t11","t10","t9"}}
[decomposed_mc1_wf] true
[bcc] true
[sc1_M] true
[sc1] true
[sc2] true
[sc3] true
[sc4] true
```

D.3 C++ Results

Test Section	Nr. of Assertions	Result
Disconnecting linears	6	PASSED
Decomposition of networks with single cut	5	PASSED
Parsing of cuts	9	PASSED
Decomposition of networks with cluster cut	2	PASSED
Decomposition of networks with multi cut	4	PASSED
Parsing of network	4	PASSED
Writing of networks	3	PASSED
Network wellformed (Under development)	4	PASSED
Decomposition of Roskilde network with cluster cut	2	PASSED
Minor functions	2	PASSED

Table D.1: Simplified overview of test sections.

APPENDIX E
RSL Specifications

E.1 Decomposition_DESIGN

```
/*=======================================================
*     File: $Name: Decomposition_DESIGN.rsl $
*  Created: $Date: 2017-03-26 16:12:06 $
*   Author: $Author: Cebrail Erdogan<s113414@student.dtu.dk>$
* Description: Decomposition of Railway Networklayouts
*=======================================================
*/

Decomposition_COMMON
scheme Decomposition_DESIGN =
    with T in
    class
        object C: Decomposition_COMMON
        value
          /*
          *  Single-cut decomposition.
          *  Decompose a network into two sub-networks, given a single cut
↪   specification.
          */
          decompose: C.ITG.I.L.NetworkLayout >< C.SingleCut -~->
                     C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
          decompose(n,sc) is
            let
                l_down = C.section_down(sc),
                l_up = C.section_up(sc),

                cut_applied_down = C.apply_bc(n,sc, DOWN),
                cut_applied_up = C.apply_bc(n,sc, UP),

                n_down = C.get_reachableNetwork(cut_applied_down,l_down),
                n_up = C.get_reachableNetwork(cut_applied_up,l_up)
            in
                (n_down,n_up)
            end
            pre C.ITG.I.L.is_wf(n) /\ C.cut_wf(n,sc),

          /*
          *  Cluster-cut decomposition.
          *  Decompose a network into two sub-networks, given a cluster-cut
↪   specification.
          */
          decompose: C.ITG.I.L.NetworkLayout >< C.ClusterCut -~->
                     C.ITG.I.L.NetworkLayout >< C.ITG.I.L.NetworkLayout
```

```
      decompose(n,cc) is
          let
            l_ups =   { C.section_up(c)   | c:C.SingleCut :- c isin cc },
            l_downs = { C.section_down(c) | c:C.SingleCut :- c isin cc },

            cut_applied_down = C.apply_bc(n,cc, DOWN),
            cut_applied_up = C.apply_bc(n,cc, UP),

            n_down = C.get_reachableNetwork(cut_applied_down, hd l_downs),
            n_up = C.get_reachableNetwork(cut_applied_up, hd l_ups)
          in
            (n_down, n_up)
          end
          pre C.ITG.I.L.is_wf(n) /\ C.cut_wf(n, cc),

      /*
      *  Multi-Cut decomposition.
      *  Decompose a network into multiple networks, given a set of cuts.
      *  The input can be a mix of single- and cluster cuts.
      */
      decompose: C.ITG.I.L.NetworkLayout >< C.MultiCut >< Bool -~->
                 C.ITG.I.L.NetworkLayout-set
      decompose(n,mc, isOriginalNetwork) is
          let
            applicableCuts = C.get_applicableCuts(n,mc)
          in
            if applicableCuts = {}
            then {n}
            else let cut= hd applicableCuts in
               let
                    (n1,n2) = case cut of
                                C.Cut_from_SingleCut(sc)  ->  decompose(n,sc),
                                C.Cut_from_ClusterCut(cc) ->  decompose(n,cc)
                              end,
                    ns1 = decompose(n1, mc \ {cut}, false),
                    ns2 = decompose(n2, mc \ {cut}, false)
               in
                    ns1 union ns2
               end
            end
            end
          end
          pre C.ITG.I.L.is_wf(n) /\ if isOriginalNetwork then C.cuts_wf(n, mc) else
           ↪   true end
  end
```

E.2 Decomposition_COMMON

```
/*=======================================================
*     File: $ Name: Decomposition_COMMON.rsl $
*  Created: $ Date: 2017-05-12 11:12:06 $
*   Author: $ Author: Cebrail Erdogan<s113414@student.dtu.dk> $
* Description: Common types and functions
*=======================================================
*/
InterlockingTableGenerator
scheme Decomposition_COMMON =
    with T in
    class
        object ITG : InterlockingTableGenerator
        type
             BorderCut  ::
                section_down:SecId
                section_up:SecId,

             SingleCut = BorderCut,
             ClusterCut = SingleCut-set,
             MultiCut = Cut-set,
             Cut == Cut_from_SingleCut(cut_to_singlecut: SingleCut) |
                    Cut_from_ClusterCut(cut_to_clustercut: ClusterCut)
        value
          /* Constant values */
          MIN_SAFETY_DISTANCE : Distance = 50,
          empty_n : ITG.I.L.NetworkLayout =
                    ITG.I.L.mk_NetworkLayout([], [], []),
          empty_mc : MultiCut =  {}

        /* Well formedness check of cuts */
        value
          /* Check if single cut spec is well-formed */
          cut_wf: ITG.I.L.NetworkLayout >< SingleCut -> Bool
          cut_wf(n,sc) is
            let l_up = section_up(sc),
                l_down = section_down(sc)
            in
                /* Does linear section up exists in network? */
                ITG.I.L.l_exists(l_up,n) /\
                /* Does linear section down exists in network? */
                ITG.I.L.l_exists(l_down,n) /\
                /* Is BCC rules satisfied? */
                BCC(n,sc)
            end,

          /* Check if cluster cut spec is well-formed */
          cut_wf: ITG.I.L.NetworkLayout >< ClusterCut -> Bool
          cut_wf(n,cc) is
                    (all sc: SingleCut :- sc isin cc =>
                         /* Does linear section up exists in network? */
                         ITG.I.L.l_exists(section_up(sc),n) /\
                         /* Does linear section down exists in network? */
                         ITG.I.L.l_exists(section_down(sc),n)
                    ) /\
```

```
          /* Is BCC rules satisfied? */
          BCC(n,cc),

  /* Check if multi-cut specs are well-formed */
  cuts_wf: ITG.I.L.NetworkLayout >< MultiCut -> Bool
  cuts_wf(n , mc) is
          (all c: Cut:- c isin mc =>
               case c of
                    Cut_from_SingleCut(sc)  -> cut_wf(n, sc),
                    Cut_from_ClusterCut(cc) -> cut_wf(n, cc)
               end)

/* Border Cut Condition rules */
value
  /* BCC
  *  Type: Pre condition
  *  Check if sc sections are neighbours
  *  Check if sc sections contain one up- and one down signal
  *  Check if sc sections are already boundaries
  *  Check if sc sections are still reachable after disconnecting
  */
  BCC: ITG.I.L.NetworkLayout >< SingleCut -> Bool
  BCC(n, sc) is
    let l_up = section_up(sc),
        l_down = section_down(sc)
    in
        /* Are the sections neighbours? */
        (ITG.I.L.are_neighbors(l_down,l_up, n) /\
        /* Is down-linear a boundary? */
        ~ITG.I.L.is_boundary(l_down,n) /\
        /* Is up-linear a boundary? */
        ~ITG.I.L.is_boundary(l_up,n) /\
        /* Do we have a down- and up signal from linears in cut specification?
         ↪   */
        DOWN isin dom ITG.I.L.signals(l_up,n) /\ UP isin dom
         ↪   ITG.I.L.signals(l_down,n)) /\
        /* Check if linears are is still reachable when disconnected */
        let disconnected_network = get_disconnectedNetwork(n,l_down,l_up) in
            l_up ~isin get_reachableNetworkSet(disconnected_network,l_down) /\
            l_down ~isin get_reachableNetworkSet(disconnected_network,l_up)
        end
     end,

  /* BCC modified for cluster cut
  *  Check for reachability after last disconnection.
  */
  BCC: ITG.I.L.NetworkLayout >< ClusterCut -> Bool
  BCC(n, cc) is
          (all sc: SingleCut :- sc isin cc =>
              let l_up = section_up(sc),
                  l_down = section_down(sc)
              in
                  /* Are the sections neighbours? */
                  (ITG.I.L.are_neighbors(l_down,l_up, n) /\
                  /* Is down-linear a boundary? */
                  ~ITG.I.L.is_boundary(l_down,n) /\
                  /* Is up-linear a boundary? */
```

```
                        ~ITG.I.L.is_boundary(l_up,n) /\
                        /* Do we have a down- and up signal from linears in cut
                         ↪   specification? */
                        DOWN isin dom ITG.I.L.signals(l_up,n) /\ UP isin dom
                         ↪   ITG.I.L.signals(l_down,n)) /\
                        /* Check if linears are is still reachable when
                         ↪   disconnected -- Disconnects all cuts every time */
                        let disconnected_network = get_disconnectedNetwork(n,cc)
                         ↪   in
                            l_up ~isin
                             ↪   get_reachableNetworkSet(disconnected_network,l_down)
                             ↪   /\
                            l_down ~isin
                             ↪   get_reachableNetworkSet(disconnected_network,l_up)
                        end
                    end
                ),
        /*
        *  SC1_M
        *  Obj.: Check for overlaps requirements, is the minimum distance satisfied?
        *  Type: (Manual Check) - Pre condition
        */
        SC1_M: ITG.I.L.NetworkLayout >< SingleCut -> Bool
        SC1_M(n,sc) is
          let l_up_id = section_up(sc),
              l_down_id = section_down(sc),
              l_up = ITG.I.L.get_linear(l_up_id,n),
              l_down = ITG.I.L.get_linear(l_down_id,n),
              mb_up = ITG.I.L.get_markerboard(ITG.I.L.usig(l_down_id,n), n),
              mb_down = ITG.I.L.get_markerboard(ITG.I.L.dsig(l_up_id,n), n)

          in
              (isOverlapDistanceSatisfied(mb_down,l_down) /\
               isOverlapDistanceSatisfied(mb_up, l_up))
          end,

        /* SC1:
        *  Objective: Found overlap sections must not reside in the other generated
↪   table.
        */
        SC1: ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout
↪   -> Bool
        SC1(n,n_down,n_up) is
                  let
                    t = ITG.mk_table(n),
                    down_routes = get_sub_routes(n_down,t),
                    up_routes = get_sub_routes(n_up,t)
                  in
                    (all r : ITG.I.Route :- r isin rng down_routes =>
                         (all s: SecId :- s isin ITG.I.overlap(r) =>
                              s isin ITG.I.L.sections(n_down)))
                     /\
                    (all r : ITG.I.Route :- r isin rng up_routes =>
                        (all s: SecId :- s isin ITG.I.overlap(r) =>
                             s isin ITG.I.L.sections(n_up)))
                  end,
```

```
        /* SC2:
         *  Objective: Found point requirements must not reference points that reside
↪  in the other generated table.
         */
        SC2: ITG.I.L.NetworkLayout ><  ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout
↪  -> Bool
        SC2(n, n_down,n_up) is
                  let
                    t = ITG.mk_table(n),
                    down_routes = get_sub_routes(n_down,t),
                    up_routes = get_sub_routes(n_up,t)
                  in
                    (all r : ITG.I.Route :- r isin rng down_routes =>
                        (all s_id : SecId :- s_id isin dom ITG.I.points(r) =>
                             s_id isin ITG.I.L.sections(n_down)))
                     /\
                    (all r : ITG.I.Route :- r isin rng up_routes =>
                        (all s_id : SecId :- s_id isin dom ITG.I.points(r) =>
                             s_id isin ITG.I.L.sections(n_up)))
                  end,

        /* SC3:
         *  Objective: Found marker boards must not reside in the other generated
↪  table.
         */
        SC3: ITG.I.L.NetworkLayout ><  ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout
↪  -> Bool
        SC3(n, n_down,n_up) is
                  let
                    t = ITG.mk_table(n),
                    down_routes = get_sub_routes(n_down,t),
                    up_routes = get_sub_routes(n_up,t)
                  in
                    (all r : ITG.I.Route :- r isin rng down_routes =>
                        (all m_id : MbId :- m_id isin ITG.I.signals(r) =>
                             m_id isin dom ITG.I.L.marker_boards(n_down)))
                     /\
                    (all r : ITG.I.Route :- r isin rng up_routes =>
                        (all m_id : MbId :- m_id isin ITG.I.signals(r) =>
                             m_id isin dom ITG.I.L.marker_boards(n_up)))
                  end,
        /* SC4:
         *  Objective: Found conflicting routes must not reside in the other generated
↪  table.
         */
        SC4: ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout
↪  -> Bool
        SC4(n, n_down,n_up) is
                  let
                    t = ITG.mk_table(n),
                    down_routes = get_sub_routes(n_down,t),
                    up_routes = get_sub_routes(n_up,t)
                  in
                    (all r : ITG.I.Route :- r isin rng down_routes =>
                        (all r_id : RouteId :- r_id isin ITG.I.conflicts(r) =>
                             r_id isin dom down_routes))
                     /\
```

```
                (all r : ITG.I.Route :- r isin rng up_routes =>
                    (all r_id : RouteId :- r_id isin ITG.I.conflicts(r) =>
                         r_id isin dom up_routes))
              end

      /* Getters and adders */
      value
         get_applicableCuts :  ITG.I.L.NetworkLayout >< MultiCut -> Cut-set
         get_applicableCuts(n,mc) is
                        {c | c:Cut :- c isin mc /\
                         case c of
                           Cut_from_SingleCut(sc)  -> cut_wf(n, sc),
                           Cut_from_ClusterCut(cc) -> cut_wf(n, cc)
                         end},

         /* Apply cut according to direction (up or down) */
         apply_bc : ITG.I.L.NetworkLayout >< SingleCut >< Direction -~->
↪  ITG.I.L.NetworkLayout
         apply_bc(n,sc,d) is
           let
              l_down_id = section_down(sc),
              l_up_id = section_up(sc),
              l_down = ITG.I.L.get_linear(l_down_id,n),
              l_up = ITG.I.L.get_linear(l_up_id,n),

              disconnect_down = if d = DOWN then l_up_id else ITG.I.L.down(l_down)
              ↪  end,
              disconnect_up = if d = DOWN then ITG.I.L.up(l_up) else l_down_id end,
              disconnected_network = get_disconnectedNetwork(n, disconnect_down,
              ↪  disconnect_up),

              -- Remove invalid marker boards at the interested side of the
              ↪  disconnection
              extra_mb_removed_n =
                  if d = DOWN then remove_extra_mb(disconnected_network,l_up_id,d)
                  else remove_extra_mb(disconnected_network,l_down_id,d)
                  end
           in
              extra_mb_removed_n
           end,

         apply_bc : ITG.I.L.NetworkLayout >< ClusterCut >< Direction -~->
↪  ITG.I.L.NetworkLayout
         apply_bc(n,cc,d) is
              if cc = {} then n else
                let
                  sc = hd cc,
                  sc_appliedNetwork = apply_bc(n,sc,d)
                in
                  apply_bc(sc_appliedNetwork,cc \ {sc},d)
                end
              end,



         /* Get disconnected network
         */
```

```
        get_disconnectedNetwork : ITG.I.L.NetworkLayout >< SecId >< SecId -~->
↪   ITG.I.L.NetworkLayout
        get_disconnectedNetwork(n,secIdDown, secIdUp ) is
          let
             points = ITG.I.L.points(n),
             linears = ITG.I.L.linears(n),

             linearsToDisconnect =
                     -- if down section is a linear
                     [ l +> remove_nb(ITG.I.L.get_linear(l,n), UP) | l:SecId :- l
                     ↪   isin ITG.I.L.linears(n) /\ l = secIdDown] !!
                     -- if up section is a linear
                     [ l +> remove_nb(ITG.I.L.get_linear(l,n), DOWN) | l:SecId :- l
                     ↪   isin ITG.I.L.linears(n) /\ l = secIdUp],

             pointsToDisconnect =
                     -- if down section is a point
                     -- point end to disconnect:  ITG.I.L.get_p_end_by_nb_id(s,s,n)
                     [ p +> remove_nb(ITG.I.L.get_point(secIdDown,n),
                     ↪   ITG.I.L.get_p_end_by_nb_id(secIdDown, secIdUp, n)) |
                     ↪   p:SecId :- p isin ITG.I.L.points(n) /\ p = secIdDown] !!
                     -- if up section is a linear
                     [ p +> remove_nb(ITG.I.L.get_point(secIdUp,n),
                     ↪   ITG.I.L.get_p_end_by_nb_id(secIdUp, secIdDown, n)) |
                     ↪   p:SecId :- p isin ITG.I.L.points(n) /\ p = secIdUp]
          in
             -- Update the network by adding the new boundaries
             ITG.I.L.mk_NetworkLayout(linears !! linearsToDisconnect, points !!
              ↪   pointsToDisconnect, ITG.I.L.marker_boards(n))
          end
          pre ITG.I.L.are_neighbors(secIdDown, secIdUp, n),

        /* Get disconnected network
        */
        get_disconnectedNetwork : ITG.I.L.NetworkLayout >< ClusterCut -~->
↪   ITG.I.L.NetworkLayout
        get_disconnectedNetwork(n,cc) is
             if cc = {} then n
             else  let
                     sc = hd cc,
                     sc_disconnected = get_disconnectedNetwork(n, section_down(sc),
                      ↪   section_up(sc))
                   in
                     get_disconnectedNetwork(sc_disconnected,cc \ {sc})
                   end
             end
          pre (all sc : SingleCut :- sc isin cc =>
           ↪   ITG.I.L.are_neighbors(section_down(sc), section_up(sc), n)),

        remove_extra_mb :  ITG.I.L.NetworkLayout >< SecId >< Direction ->
↪   ITG.I.L.NetworkLayout
        remove_extra_mb(n,l,d) is
          let
             -- Get the extra marker boards to be removed (if exists)
             extra_mbs = { ITG.I.L.signals(l,n)(-d)}
          in
             -- Update the network by deleting the extra marker board
```

```
                ITG.I.L.mk_NetworkLayout(ITG.I.L.linears(n), ITG.I.L.points(n),
                 ↪  ITG.I.L.marker_boards(n) \ extra_mbs)
              end,


            /*
            * Get all sections by discovering through neighbors.
            * Recoursively accumalate all the sections in both directions.
            */
            get_all_sections: ITG.I.L.NetworkLayout >< SecId-set >< SecId-set -~->
↪   SecId-set
            get_all_sections(n,tv,v) is
               if tv = {} then v                                             -- If no
                ↪  unvisited sections left, then finish
               else
                 let
                   current = hd tv,                                          --
                    ↪  Select an element from to visit set
                   visited = {current} union v,                              -- Add
                    ↪  current section to visited
                   toVisit = (tv union ITG.I.L.get_neighbors(current,n)) \ visited   --
                    ↪  Update toVisit and prevent endless loop
                   in
                     get_all_sections(n,toVisit,visited)                     -- Call
                      ↪  function with updated args
                 end
               end,

            /* Get all linears from given sections */
            get_all_linears:ITG.I.L.NetworkLayout >< SecId-set -~-> SecId -m->
↪   ITG.I.L.Linear
            get_all_linears(n,secs) is
               [l +> ITG.I.L.get_linear(l,n) | l:SecId :- l isin secs /\ l isin
                ↪  ITG.I.L.linears(n)],

            /* Get all points from given sections */
            get_all_points : ITG.I.L.NetworkLayout >< SecId-set -~-> SecId -m->
↪   ITG.I.L.Point
            get_all_points(n,secs) is
               [p +> ITG.I.L.get_point(p,n) | p:SecId :- p isin secs /\ p isin
                ↪  ITG.I.L.points(n)],

            /* Get all the signals from given sections */
            get_all_signals : ITG.I.L.NetworkLayout >< SecId-set -~-> MbId -m->
↪   ITG.I.L.MarkerBoard
            get_all_signals(n,secs) is
               let
                 signal_ids = flatten({rng ITG.I.L.signals(sec,n) | sec:SecId :- sec
                  ↪  isin secs /\ sec isin dom ITG.I.L.linears(n) })
               in
                 [m +> ITG.I.L.get_markerboard(m,n) | m:MbId :- m isin signal_ids /\
                                                      m isin ITG.I.L.marker_boards(n)]
               end,

            /* Get reachable network */
            get_reachableNetwork : ITG.I.L.NetworkLayout >< SecId -~->
↪   ITG.I.L.NetworkLayout
```

```
        get_reachableNetwork(n,s) is
          let
             -- Get all sections
             sections = get_all_sections(n,{s},{}),
             -- Get all linears from given sections
             linears = get_all_linears(n,sections),
             -- Get all points from given sections
             points  = get_all_points(n, sections),
             -- Get all signals from given sections (signals only exists in linears)
             signals = get_all_signals(n, sections)
          in
             -- Instantiate a new network layout
             ITG.I.L.mk_NetworkLayout(linears,points,signals)
          end
          pre ITG.I.L.s_exists(s,n),

        get_reachableNetworkSet : ITG.I.L.NetworkLayout >< SecId ~~~> SecId-set
        get_reachableNetworkSet(n,l) is
            get_all_sections(n,{l},{}),

        get_sub_routes: ITG.I.L.NetworkLayout >< ITG.I.InterlockingTable ~~~>
↪   ITG.I.InterlockingTable
        get_sub_routes(n,t) is
            [r_id +> ITG.I.get_route(r_id,t) | r_id: RouteId :- r_id isin dom t /\
                                        ITG.I.source(t(r_id)) isin
                                        ↪   ITG.I.L.marker_boards(n) /\
                                        ITG.I.dest(t(r_id)) isin
                                        ↪   ITG.I.L.marker_boards(n)]
        pre ITG.I.L.is_wf(n)

     /* Auxiliary functions */
     value
       remove_nb: ITG.I.L.Linear >< LinearEnd -> ITG.I.L.Linear
       remove_nb(l,le) is
              ITG.I.L.mk_Linear(ITG.I.L.neighbors(l) \ {le}, ITG.I.L.length(l)),

       remove_nb: ITG.I.L.Point >< PointEnd -> ITG.I.L.Point
       remove_nb(p,pe) is
              ITG.I.L.mk_Point(ITG.I.L.neighbors(p) \ {pe}, ITG.I.L.length(p)),

       /*
       * Simpler representations of sections and signals
       * For two sub-networks
       */
       decomposed_sec_repr: ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout ->
↪   SecId-set >< SecId-set
       decomposed_sec_repr(n1,n2) is
            (ITG.I.L.sections(n1), ITG.I.L.sections(n2)),

       /*
       * For multi cut
       */
       decomposed_sec_repr: ITG.I.L.NetworkLayout-set >< (SecId-set)-set ->
↪   (SecId-set)-set
       decomposed_sec_repr(n, secs) is
            if n = {} then secs
            else
```

```
              let h = hd n in
                  decomposed_sec_repr(n \ {h}, secs union {ITG.I.L.sections(h)})
              end
             end,

        decomposed_sig_repr: ITG.I.L.NetworkLayout >< ITG.I.L.NetworkLayout ->
↪   MbId-set >< MbId-set
        decomposed_sig_repr(n1,n2) is
             (dom ITG.I.L.marker_boards(n1), dom ITG.I.L.marker_boards(n2)),


        /* Check if overlap distance is satisfied */
        isOverlapDistanceSatisfied: ITG.I.L.MarkerBoard >< ITG.I.L.Linear -> Bool
        isOverlapDistanceSatisfied(mb, l) is
             let mb_distance =  ITG.I.L.distance(mb),
                 border_distance = ITG.I.L.length(l)
             in
                (MIN_SAFETY_DISTANCE <= mb_distance + border_distance)
             end

    end
```

APPENDIX F

C++ Code

This appendix contains the C++ files of the most importants parts of the decomposition tool. The cut types and the main decomposition methods are part of the shown listings.

F.1 Cut Types

The listed header files are sufficient to observe the data types and relations between the cuts.

F.1.1 Cut.h

```
#ifndef CUT_H
#define        CUT_H

#include <iostream>
#include <sstream>
#include <string>
#include <map>
#include <set>
#include <list>
#include "RttTgenDkIxlComponent.h"


struct CutTypes {
    typedef enum{
        UNDEF = 0,
        BORDERCUT = 1,
        CLUSTERCUT = 2,
        MULTICUT = 3
    } cut_type_t;
};

class Cut : public RttTgenDkIxlComponent {
 protected:
    const CutTypes::cut_type_t type;

 public:
        Cut(std::string n, std::string oid = "<undefined id>" , CutTypes::cut_type_t t
     ↪   = CutTypes::UNDEF)
                : RttTgenDkIxlComponent(n, oid), type(t) {}

    virtual CutTypes::cut_type_t getCutType() {return type;}

    bool isA(int) const;
```

```
};

#endif
```

F.1.2 SingleCut.h

```
#ifndef SINGLECUT_H
#define         SINGLECUT_H

#include <iostream>
#include <sstream>
#include <string>
#include <map>
#include <set>
#include <list>
#include "RttTgenDkIxlLinear.h"
#include "Cut.h"

class SingleCut : public Cut {
 public:
        SingleCut(std::string n, std::string oid, CutTypes::cut_type_t t =
        ↪   CutTypes::UNDEF)
                : Cut(n, oid, t) {}
};

#endif
```

F.1.3 BorderCut.h

```
#ifndef BORDERCUT_H
#define         BORDERCUT_H

#include <iostream>
#include <sstream>
#include <string>
#include <map>
#include <set>
#include <list>
#include "RttTgenDkIxlLinear.h"
#include "SingleCut.h"


class BorderCut : public SingleCut {
 private:
        RttTgenDkIxlLinear* linear_up;
        RttTgenDkIxlLinear* linear_down;

 public:
        BorderCut(std::string oid)
                : SingleCut("BorderCut", oid, CutTypes::BORDERCUT) {}
```

```
        RttTgenDkIxlLinear* getLinearUp() {return linear_up;}
        RttTgenDkIxlLinear* getLinearDown(){return linear_down;}

        void setLinearUp(RttTgenDkIxlLinear* linear) { linear_up = linear; }
        void setLinearDown(RttTgenDkIxlLinear* linear){ linear_down = linear;}
};

#endif
```

F.1.4 ClusterCut.h

```
#ifndef CLUSTERCUT_H
#define        CLUSTERCUT_H

#include <iostream>
#include <sstream>
#include <string>
#include <list>
#include "Cut.h"
#include "BorderCut.h"


class ClusterCut : public Cut {
 private:
    std::list<BorderCut*> borderCuts;

 public:
        ClusterCut(std::string oid)
                : Cut("CluterCut", oid, CutTypes::CLUSTERCUT) {}

        std::list<BorderCut*> getBorderCuts(){return borderCuts;}

        void setBorderCuts(std::list<BorderCut*> bcs) { borderCuts = bcs;}
};

#endif
```

F.1.5 MultiCut.h

```
#ifndef MULTICUT_H
#define        MULTICUT_H

#include <iostream>
#include <sstream>
#include <list>
#include "Cut.h"
#include "BorderCut.h"
#include "ClusterCut.h"
```

```
class MultiCut : public Cut {
 private:
    std::list<BorderCut*> borderCuts;
    std::list<ClusterCut*> clusterCuts;

 public:
        MultiCut(std::string oid)
                : Cut("MultiCut", oid, CutTypes::MULTICUT) {}

        std::list<BorderCut*> getBorderCuts(){return borderCuts;}
        std::list<ClusterCut*> getClusterCuts(){return clusterCuts;}

    BorderCut* getBorderCut(std::string oid);
    ClusterCut* getClusterCut(std::string oid);
    Cut* getCut(std::string oid);

        void setBorderCuts(std::list<BorderCut*> bcs) { borderCuts = bcs; }
        void setClusterCuts(std::list<ClusterCut*> ccs) { clusterCuts = ccs; }
};

#endif
```

F.2 Main decomposition files

The *main.cpp* file will be the executable that triggers the main decomposition methods placed in *Decomposition.cpp*.

F.2.1 main.cpp

```
#include "Decomposition.h"
#include "parser/DecompositionParser.h"
#include "xmlWriter/DecompositionXmlWriter.h"
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <unistd.h>
#include <cstdlib>

static void show_usage(){
    std::cerr
        << "Usage:\n"
        << "\t-h,--help\t\tShow this help message" << endl
        << "\t-d,--destination\tSpecify the output destination path" << endl
        << "\t-c,--cut\t\tSpecify the cut file path"  << endl
        << "\t-n,--network\t\tSpecify the network file path"  << endl
        << "\t-v,--verbose\t\tVerbose mode"   << endl
        << std::endl;
}

bool verbose = false;
string input_cut_path;
string input_network_path;
```

```
string out_path;

int main(int argc, char** argv){
    int c ;
    if (argc == 1) {
        show_usage();
        return 1;
    }

    // Manually check for path arguments first

    while ((c = getopt (argc, argv, "hvd:c:n:")) != -1)
        switch (c)
        {
            case 'h':
                show_usage();
                break;
            case 'd':
                cout << optarg << endl;
                break;
            case 'c':
                input_cut_path = optarg;
                break;
            case 'n':
                input_network_path = optarg;
                break;
            case 'v':
                verbose = true;
                break;
            case '?':
                /*
                if (optopt == 'c')
                    fprintf (stderr, "Option -%c requires an argument.\n", optopt);
                else
                    fprintf (stderr,
                            "Unknown option character `\\x%x'.\n",
                            optopt);
                */
                return 1;
            default:
                abort ();
        }
    for (int index = optind; index < argc; index++){
        printf ("Non-option argument %s\n", argv[index]);
    }

    Decomposition* decomposition = new Decomposition();
    DecompositionParser* parser = new DecompositionParser();
    DecompositionCommon*  common = new DecompositionCommon();
    DecompositionXmlWriter* xmlWriter = new DecompositionXmlWriter();

    parser -> parse(input_network_path, input_cut_path);
    RttTgenDkIxlInterlocking* interlocking = parser -> getInterlocking();
    RttTgenDkIxlNetworkLayout* network = interlocking -> getNetworkLayout();
    Cut* cut = parser -> getCut();
    upDownNetworks_t upDownNetworks;
    networks_t networks;
```

```
    switch(cut -> getCutType())
    {
        case CutTypes::UNDEF:
            cout << "Cut Undefined, aborting..." << endl;
            return 0;
            break;
        case CutTypes::BORDERCUT:
            cout << "BorderCut Detected" << endl;
            upDownNetworks = decomposition -> decompose(network, (BorderCut*) cut);

            // write down network
            interlocking -> setNetworkLayout(upDownNetworks.down);
            xmlWriter -> xmlWriteMainDoc(out_path + interlocking -> getOid() +
             ↪  "_down.xml", interlocking);

            // write up network
            interlocking -> setNetworkLayout(upDownNetworks.up);
            xmlWriter -> xmlWriteMainDoc(out_path + interlocking -> getOid() +
             ↪  "_up.xml", interlocking);
            break;
        case CutTypes::CLUSTERCUT:
            cout << "ClusterCut Detected" << endl;
            upDownNetworks = decomposition -> decompose(network, (ClusterCut*) cut);

            // write down network
            interlocking -> setNetworkLayout(upDownNetworks.down);
            xmlWriter -> xmlWriteMainDoc(out_path + interlocking -> getOid() +
             ↪  "_down.xml", interlocking);

            // write up network
            interlocking -> setNetworkLayout(upDownNetworks.up);
            xmlWriter -> xmlWriteMainDoc(out_path + interlocking -> getOid() +
             ↪  "_up.xml", interlocking);
            break;

        case CutTypes::MULTICUT:
            cout << "MultiCut Detected" << endl;
            networks = decomposition -> decompose(network, (MultiCut*) cut);
            int cnt = 0;
            for(const auto& n : networks){
                interlocking -> setNetworkLayout(n);
                xmlWriter -> xmlWriteMainDoc(out_path + interlocking -> getOid() + "_"
                 ↪  + std::to_string(++cnt) +  ".xml", interlocking);
            }
            break;
    }
    return 0;
}
```

F.2.2 Decomposition.cpp

```
#include "Decomposition.h"
#include <stdio.h>
#include <string.h>

using namespace std;

Decomposition::Decomposition(){
    common = new DecompositionCommon();
}


upDownNetworks_t Decomposition::decompose(RttTgenDkIxlNetworkLayout* network,
 ↪  BorderCut* sc){
    RttTgenDkIxlLinear* l_down = sc -> getLinearDown();
    RttTgenDkIxlLinear* l_up = sc -> getLinearUp();

    string l_down_id = sc -> getLinearDown() -> getOid();
    string l_up_id = sc -> getLinearUp() -> getOid();

    RttTgenDkIxlNetworkLayout* downNetwork = common -> clone(network);
    RttTgenDkIxlNetworkLayout* upNetwork = common -> clone(network);

    upDownNetworks_t networks;
    bool isDownSubnetCreated = common -> createSubnet(downNetwork, sc,
     ↪  direction_t::DOWN);
    bool isUpSubnetCreated = common -> createSubnet(upNetwork, sc, direction_t::UP);
    if (isDownSubnetCreated && isUpSubnetCreated){
        networks.down = common -> reachableNetwork(downNetwork, l_down);
        networks.up = common -> reachableNetwork(upNetwork, l_up);
    }
    return networks;
}

upDownNetworks_t Decomposition::decompose(RttTgenDkIxlNetworkLayout* network,
 ↪  ClusterCut* cc){
    RttTgenDkIxlLinear* l_down = cc -> getBorderCuts().front() -> getLinearDown();
    RttTgenDkIxlLinear* l_up = cc -> getBorderCuts().front() -> getLinearUp();

    RttTgenDkIxlNetworkLayout* downNetwork = common -> clone(network);
    RttTgenDkIxlNetworkLayout* upNetwork = common -> clone(network);

    bool isDownSubnetCreated = common -> createSubnet(downNetwork, cc,
     ↪  direction_t::DOWN);
    bool isUpSubnetCreated = common -> createSubnet(upNetwork, cc, direction_t::UP);

    upDownNetworks_t networks;
    if (isDownSubnetCreated && isUpSubnetCreated){
        networks.down = common -> reachableNetwork(downNetwork, l_down);
        networks.up = common -> reachableNetwork(upNetwork, l_up);
    }

    networks.down = downNetwork;
    networks.up = upNetwork;
    return networks;
}
```

```
networks_t Decomposition::decompose(RttTgenDkIxlNetworkLayout* network, MultiCut* mc){
    list<string> applicableCutOids = common -> getApplicableCutOids(network, mc);

    if (applicableCutOids.size() == 0){
        networks_t n = {network};
        return n;
    }
    else{
        string cutOid = applicableCutOids.front();
        applicableCutOids.pop_front();
        CutTypes::cut_type_t cutType = mc -> getCut(cutOid) -> getCutType();

        upDownNetworks_t ns;
        if (cutType == CutTypes::BORDERCUT){
            ns = decompose(network, mc -> getBorderCut(cutOid));
            list<BorderCut*> borderCuts = mc -> getBorderCuts();
            borderCuts.remove(mc -> getBorderCut(cutOid));
            mc -> setBorderCuts(borderCuts);
        }
        else if (cutType == CutTypes::CLUSTERCUT){
            ns = decompose(network, mc -> getClusterCut(cutOid));
            list<ClusterCut*> clusterCuts = mc -> getClusterCuts();
            clusterCuts.remove(mc -> getClusterCut(cutOid));
            mc -> setClusterCuts(clusterCuts);
        }
        else{
            //Throw error
        }

        // make a deep of multi cut?
        networks_t ns1 = decompose(ns.down, mc);
        networks_t ns2 = decompose(ns.up, mc);

        networks_t mergedNetworks = ns1;
        mergedNetworks.splice(mergedNetworks.end(), ns2);

        return mergedNetworks;
    }
}
```

Bibliography

[1] *A selection of projects which have used RAISE.* URL: http://spd-web.terma.com/Projects/RAISE/project.html (visited on July 14, 2017) (cited on page 8).

[2] *About the project - Robustrails.* URL: http://www.robustrails.man.dtu.dk/About-the-project (visited on July 13, 2017) (cited on page 3).

[3] *Depth First Search and Breadth First Search in Python.* URL: http://eddmann.com/posts/depth-first-search-and-breadth-first-search-in-python/ (visited on July 21, 2017) (cited on page 40).

[4] Alessandro Fantechi, Anne E. Haxthausen, and Hugo Daniel Macedo. “Compositional Verification of Interlocking Systems for Large Stations”. In: *Proceedings of SEFM’ 2017.* Edited by Alessandro Cimatti and Marjan Sirjani. Volume 1046. Lecture Notes in Computer Science. Springer International Publishing, 2017 (cited on page 18).

[5] Andreas Foldager. “A Graphical Domain-specific Language for Railway Interlocking Systems , Et Grafisk Domænespecifikt Sprog for Jernbane-sikringsanlæg”. und. 2015 (cited on pages 7, 68).

[6] Anne E. Haxthausen and Peter H. Østergaard. “On the Use of Static Checking in the Verification of Interlocking Systems”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications: 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part II.* Edited by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2016, pages 266–278. ISBN: 978-3-319-47169-3. DOI: 10.1007/978-3-319-47169-3_19. URL: http://dx.doi.org/10.1007/978-3-319-47169-3_19 (cited on pages 1, 6).

[7] Hugo D. Macedo, Alessandro Fantechi, and Anne E. Haxthausen. “Compositional Verification of Multi-Station Interlocking Systems”. In: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications, Part II.* Volume 9953. Lecture Notes in Computer Science. Springer International Publishing AG, 2016, pages 279–293 (cited on pages 4, 10, 63, 64).

[8] Hugo Daniel Macedo, Alessandro Fantechi, and Anne E. Haxthausen. “Compositional Model Checking of Interlocking Systems for Lines with Multiple Stations”. In: *NASA Formal Methods: 9th International Symposium, NFM 2017, Proceedings.* Edited by Clark Barrett, Misty Davies, and Temesghen Kahsai. Springer International Publishing, 2017, pages 146–162. ISBN: 978-3-319-57288-8. DOI:

10.1007/978-3-319-57288-8_11. URL: http://dx.doi.org/10.1007/978-3-319-57288-8_11 (cited on page 17).

[9] Phil Nash. *Catch: A modern, C++-native, header-only, test framework for unit-tests, TDD and BDD - using C++98, C++03, C++11, C++14 and later.* original-date: 2010-11-08T18:22:56Z. July 2017. URL: https://github.com/philsquared/Catch (cited on page 59).

[10] *Signalling Programme / UkBane.* URL: http://uk.bane.dk/Projects/Signalling-Programme (visited on July 12, 2017) (cited on page 1).

[11] *std::variant - cppreference.com.* URL: http://en.cppreference.com/w/cpp/utility/variant (visited on July 22, 2017) (cited on page 44).

[12] *The Danish Council for Strategic Research.* Ministry of Higher Education and Science. 2015. URL: http://ufm.dk/en/research-and-innovation/councils-and-commissions/former-councils-and-commissions/the-danish-council-for-strategic-research (visited on May 27, 2015) (cited on page 1).

[13] Linh Hong Vu. "Formal Development and Verification of Railway Control Systems - In the context of ERTMS/ETCS Level 2". PhD thesis. Technical University of Denmark, DTU Compute, 2015 (cited on pages 1, 6, 12, 22, 29, 30, 42, 55, 56).

[14] Linh Hong Vu, Anne E. Haxthausen, and Jan Peleska. "Formal modelling and verification of interlocking systems featuring sequential release". In: *Science of Computer Programming* 133, Part 2 (2017). http://dx.doi.org/10.1016/j.scico.2016.05.010, pages 91–115. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2016.05.010. URL: http://www.sciencedirect.com/science/article/pii/S0167642316300570 (cited on page 6).

[15] Linh Hong Vu, Anne Elisabeth Haxthausen, and Jan Peleska. "A Domain-Specific Language for Railway Interlocking Systems". eng. In: *Proceedings of the 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, Forms/format 2014* (2014), pages 200–209 (cited on pages 1, 3).