Model-Based Testing of Model Transformations

Amr Al Mallah
http ://msdl.cs.mcgill.ca/people/amr

Supervisor : Professor Hans Vangheluwe

School of Computer Science
McGill University
Montréal, Québec, Canada

A thesis submitted to McGill University in partial
fulfilment of the requirements of the degree of
Master of Science in Computer Science

Abstract

Model Driven Engineering (MDE) research has achieved major progress in the past few years. Though MDE research and adoption are moving forward at an increasing pace, there are still few major challenges left to be addressed. Model Transformations (MT) represent an essential part of MDE that is gradually reaching maturity level. Testing MT has been shown to be a challenging task due to a new set of problems. In this thesis we attempt to complement the work done so far by the research community to address MT testing challenges.

We use findings from the research in classical testing to create a prospective view on MT testing challenges and opportunities. More specifically, we focus on two challenges : Model Comparison and automating testing execution through a Testing Framework. First, we introduce a model comparison approach (based an existing graph comparison algorithm) that is customizable, and fine tuned to performs best in testing situations. The performance of our algorithm is throughly investigated against different types of models. Second, we introduce TUnit : a modelled framework for testing Model transformations. We demonstrate the benefit of using TUnit in supporting the process of testing transformations in regression testing and enabling semantic equivalence through extending our case study to perform a comparison of coverability graphs of Petri Nets.

Résumé

La recherche sur le Model Driven Engineering (MDE) a accomplit de grands progrès au cours des dernières années. Bien que la recherche et l'adoption avancent à grands pas, il reste encore plusieurs défis majeurs à adresser. La Transformation de Modèle (TM) représente un élément essentiel du MDE qui atteint graduellement le niveau de maturité. Le test sur les TM s'est démontré être une tâche difficile en raison des nouveaux problèmes survenus. Dans cette thèse, nous essayons de complémenter le travail complété par la communauté de recherche pour adresser les défis restants des tests sur les TM.

Nous utilisons les résultats de la recherche en tests classiques pour créer une vision prospective sur les défis et opportunités des tests sur les TM. Nous nous concentrons plus précisement sur les deux défis suivants : la comparaison des modèles et l'automation des tests exécutés à travers un cadre de tests . Tout d'adord, nous présentons une approche en comparaison de modèles qui peut être personnalisée et atteint de meilleurs résultats dans des situations de tests. La performance de notre algorithme est rigoureusement étudiée contre différents types de modèles. Deuxièmement, nous introduisons Tunit : un cadre de tests en transformation de modèles qui est aussi un modèle. Nous démontrons les avantages d'utiliser TUnit pour donner un support au processus de tests sur les transformations en tests de regression et permettre l'équivalance sémantique.

Acknowledgement

The first and most important thank you goes to my supervisor Hans Vangheluwe. His vast knowledge and constant enthusiasm showed me the way throughout this thesis writing journey. Thank you Hans for this invaluable learning experience. Secondly, I would like to thank my mother for her constant support throughout the years. I would like to thank her and the rest of the family who have made this possible.

Also, I would like to thank my friends at the MSDL (Modelling and Simulation Design Lab) for the stimulating discussions that helped formulate some of my thesis ideas and kept my motivation levels high, and my close friends who had to keep up with me throughout this period.

Finally, a special thanks go to Tibor and Andrea for their encouragement and support during the length of this endeavour, and beyond.

Contents

List of Figures

Introduction

We are solving increasingly complex problems using software everyday. Subsequently, techniques such as Model Driven Engineering (MDE) have surfaced as a viable solution to reduce complexity and increase efficiency. For example, MDE would help in construction of software systems through automatic code generations. It also enables better design and analysis of complex systems in all domains.

Although MDE has showed great potential so far, there are still some underlying challenges before achieving full adoption. Model Transformation is one of the core building blocks of MDE. To help push MDE further we will focus our work on supporting a complete process for building model transformations, through better testing.

The natural first step is to identify the main challenges around testing model transformations using a roadmap. To use a proper roadmap we start off by studying classical software testing approaches i.e coded systems. We then use this classification as a roadmap to categorize current research in model transformation testing, its achievements and remaining challenges.

The few challenges we chose to tackle are complementary to recent advancements in model transformation testing research.

Namely we will focus on building an efficient model comparison algorithm suitable for this context.

We then build a framework to streamline testing process of model transformations similar to existing unit testing frameworks.

Finally we will demonstrate how the framework automates the execution of testing by means of a case study, and further streamline the development process through regression testing, and enabling more complex testing using Semantics Equivalence.

1

Chapter 1 : State Of The Art Software Testing

Software testing was created on the same day when the first program was written. [Het88] points to testing literature starting as early as 1950. Early testing work was concerned with debugging, i.e, finding the bugs and removing them from production systems. Subsequently, testing has evolved into a separate discipline.

This chapter attempts to provide an overview of current foundations in software testing research and practice. A discussion of software requirements is provided next in Section 1.1. Then we proceed by listing the major definitions of software testing throughout the literature, in Section 1.2, and the evolution of the testing discipline in Section 1.3. We examine the three different classification of testing techniques recurring in the literature in Section 1.4. Finally an overview of testing techniques and challenges is thoroughly examined in 1.5.

1.1 Requirements

Requirements, also referred to as specifications, are formal descriptions of the expected system behaviour. Testing usually attempts to validate that the system behaves according to its requirements.

Since requirements guide the testing process, it's important that the requirements are also tested. This procedure is referred to as *Requirements Verification*. Quality attributes which can be used to evaluate requirements include:

- **Completeness:** details about all possible situations should be provided, even exceptional scenarios should be considered and described.
- **Precision and Clearness:** this will allow for exact implementation with minimal room for interpretation by the system implementers.
- **Consistency:** requirements should reflect the same overall functionality, and should not describe conflicting or incompatible functionalities, at any level of the specification.
- **Testability:** requirements should be measurable. Thus enabling checking the correctness of their implementation in the resulting software product. Tthis criteria should be judged by the implementing engineers. Unfeasible requirements cause to

shift the project from success.

1.2 Definitions of Software Testing

We list few of the main definitions of testing, in an attempt to demonstrate the different view points on testing, which in turn reflect its complexity as discussed in [Het88]:

Hetzel 1973:

> Testing is the process of establishing confidence that a program or a system does what is it supposed to.

Myers 1979:

> Testing is the process of executing a program or system with the intent of finding errors.

Hetzel 1983:

> Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results.

Beizer:

> The act of executing tests. Tests are designed and then executed to demonstrate the correspondence between an element and its specification.

IEEE:

> The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results.

1.3 Evolution of Software Testing

The Evolution of software testing as a profound discipline in the software engineering world has been gradual. Software testing research and techniques have been evolving in parallel to the software engineering research since software was first created. A summary of the focus of software engineering research since the 1960' is presented in [Sha90].

In the 1950's programming was restricted to writing small programs in machine languages (e.g: assemblers), there was an elementary understanding of control flow, and back then testing was debugging. Then in the 1960's, compiler development took place, and testing started to emerge separately from debugging.

Throughout the 1970's, software engineering concepts were being introduced and adopted. Programming and Algorithms were the main focus of research, as well as data structures

and types. Testing took the path of a more technical discipline. Subsequently, during the following period the focus switched to interfaces and system structure. Systems grew into more complex specifications with a large structured state space, which lead to the CASE tools into a more developed and evolved stage. Testing then grew into the *verification* and *validation* concepts.

In the 1990's the focus of software processes shifted to shorter development cycles and increasing quality. Demand for specialized testing skills increased, followed by a growing interest in software safety, security and fault tolerance through the recent years.

Today software testing has become a vital part of the software engineering process. Numerous tools and techniques, which support test design and implementation has been developed. They have been applied to almost all phases of the software engineering process.

1.4 Taxonomy of Software Testing

The following is proposed classification of software testing techniques which exist today, both in research and practice. We present three different high level views of this classifications, and discuss each.

1.4.1 Static Vs Dynamic Analysis

This classification is based on the criteria of executing the program [Sch96] [Har00]. *Static* analysis does not involve the execution of the system under test (SUT), whether it's code, a design document or any other system artifact. It may however involve some form of conceptual execution. Tools like **findbugs** [1], could identify dead code segments for example. Static analysis could be manual (code reviews), or automatic: (compilers).

Dynamic analysis on the other hand, involves the execution of the source code. It follows the the traditional approach of testing. The program is executed under certain conditions, and then it's behaviour is observed to conclude correctness.

Static, non-execution based

Techniques such as walkthroughs and inspections by experts has been shown to be highly effective in revealing bugs. Nevertheless, these are relatively high cost if not accompanied with other techniques. Formal approaches such as, correctness proving using mathematical proof are also used. They are intended to show that a product is correctly implementing the specifications. But even then the product needs to be exposed to several forms of execution based testing to ensure quality, which allows exposing other classes of faults which are not related to the overall algorithmic correctness. See Figure 1.1 for an overall view of static testing techniques. Another informal proof technique involves inserting assertions into the code for example.

[KM07] provides the following description of what different static techniques apply through different stages:

1. http://findbugs.sourceforge.net

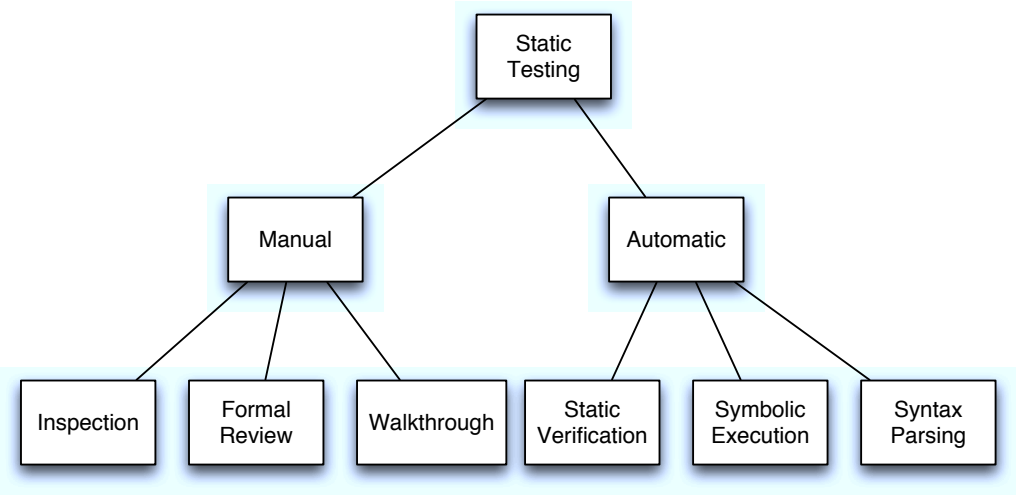


Figure 1.1: Static testing techniques

Stage	Phase	Checks
1	Control Flow Analysis	Loops with multiple exist or entry points Unreachable code
2	Data Use Analysis	Un-initialized variables Variables written twice without intervening assignment Variables declared but never used
3	Interface Analysis	Consistency of procedure declaration and their use
4	Information Flow Analysis	Dependencies of output variables Information for code inspection review
5	Path Analysis	Paths in the program and the statements executed in it.

Dynamic, Execution Based

Dynamic analysis enable testing other aspects of the system, including performance, reliability and correctness. Test cases involve the execution of the system, or parts of it, to achieve the testing verdicts. The main goal of execution based testing is to reveal as many faults as possible. It's understood that there is no way to guarantee the detection of all faults using this approach.2

As concluded in [YT89], this taxonomy is well suited for planning purposes. It can be applied throughout the different stages of the development life-cycle. The following table is a summary to illustrate what testing activities apply for each of the software development stages. For example, model checking is usually based on the requirement, and it does not require executing the implementation.

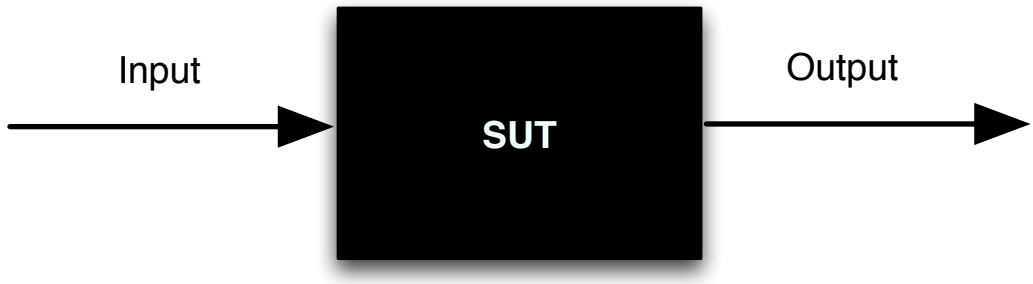


Figure 1.2: Black-box testing

	Execution-Dynamic	Non-Execution-Static
Requirement	Functional testing testing input classes testing output classes	 Requirement validation Model checking
Design	Design/Model Testing	Reviewing/analyzing design artifacts
Coding	Coverage techniques Data-flow-testing Control-flow-testing	Static error analysis (syntax/type checking) Symbolic Execution Peer reviewing

1.4.2 Functional Vs Structural Testing

Another classification is the functional vs structural testing which is the most adopted taxonomy in software testing. Systems being tested usually have several associated artifacts. The number of such artifacts grow as building the system progresses through different levels. The specifications documents are referred to as the first artifact of the system. Other artifact such as design documents, and actual implementation are introduced later.

This classification differentiate testing techniques which are based on which artifact(s) was used to derive test cases [Bei95].

Functional a.k.a Black-Box testing

Functional testing relies on the analysis of requirements of the system under test. Whether it's the complete system, a unit, a component or even the user documentations. The type of testing does not consider the internal implementation, and treats the SUT as a black box (see Figure 1.2). Functional testing implies executing all test cases derived from the requirements. It focuses on the external behaviour of the SUT. Black-box testing include techniques such as: partition analysis and equivalence classes.

Structural a.k.a White-Box testing

In structural testing, selecting test cases depends on the implementation of the system. Design documents, or source code could be inspected to drive test case selection to ensure execution of certain statements or branches of the code. The focus in structural testing

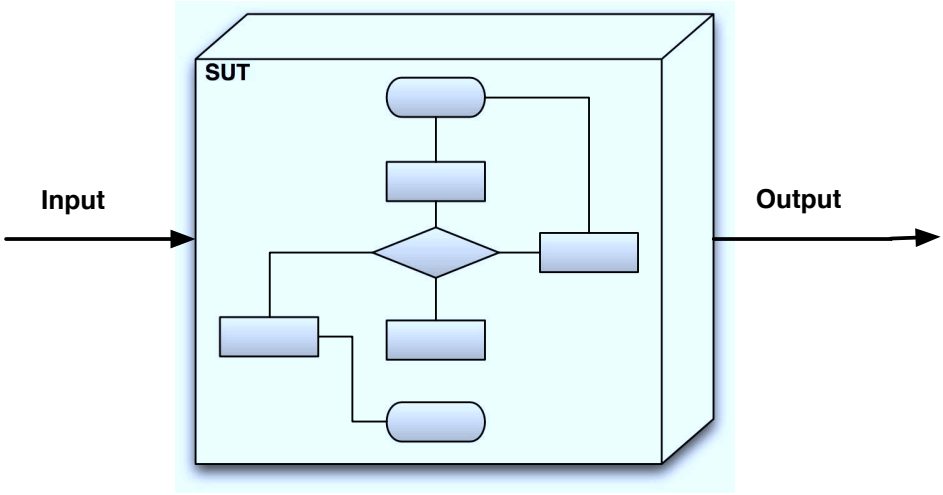


Figure 1.3: White-box testing

is on the internal behaviour (see Figure 1.3). Certain criterion such as coverage represent a qualitative value for the effectiveness of the testing. White-box testing includes techniques such as: branch coverage and path coverage.

1.4.3 Folding Vs Sampling

This classification distinguishes among, on the one hand state space analysis techniques, and on the other, techniques sampling several system behaviours. Such a taxonomy, can relate techniques on one axis, and the relative effort needed on the second axis.

"Does program P obey specification S" is undecidable. Such a statement means that there is a tradeoff between computational cost and accuracy. [YT89] introduces the notion of pessimistic inaccuracy and optimistic inaccuracy. *Pessimistic inaccuracy*: is the failure to accept a correct program. For example by failing to construct a complete proof of correctness. *Optimistic inaccuracy*: is the failure to reject an incorrect program. Could be the case since exhaustive testing can not be performed.

In Figure 1.4 we show on the horizontal axis the type of inaccuracy, and on the vertical axis the effort needed to achieve the activity. Exhaustive testing and formal proof are both intended to be on the top of pyramid. i.e they are both neither an optimistic nor a pessimistic inaccuracy. The effort to achieve either is often unfeasible as it approaches infinity. Threshold of tractability defines the limit beyond which it becomes hard to implement a pragmatic solution from theoretical concepts which could work in most cases. Threshold of decidability defines the limit beyond which it becomes unfeasible to implement a pragmatic solution which could work in most cases.

Folding

Techniques which fold execution states together, using abstraction for example, are called Folding techniques. Folding is when an analysis technique abstract away some details of program execution. It involves transposing the representation of a system onto a

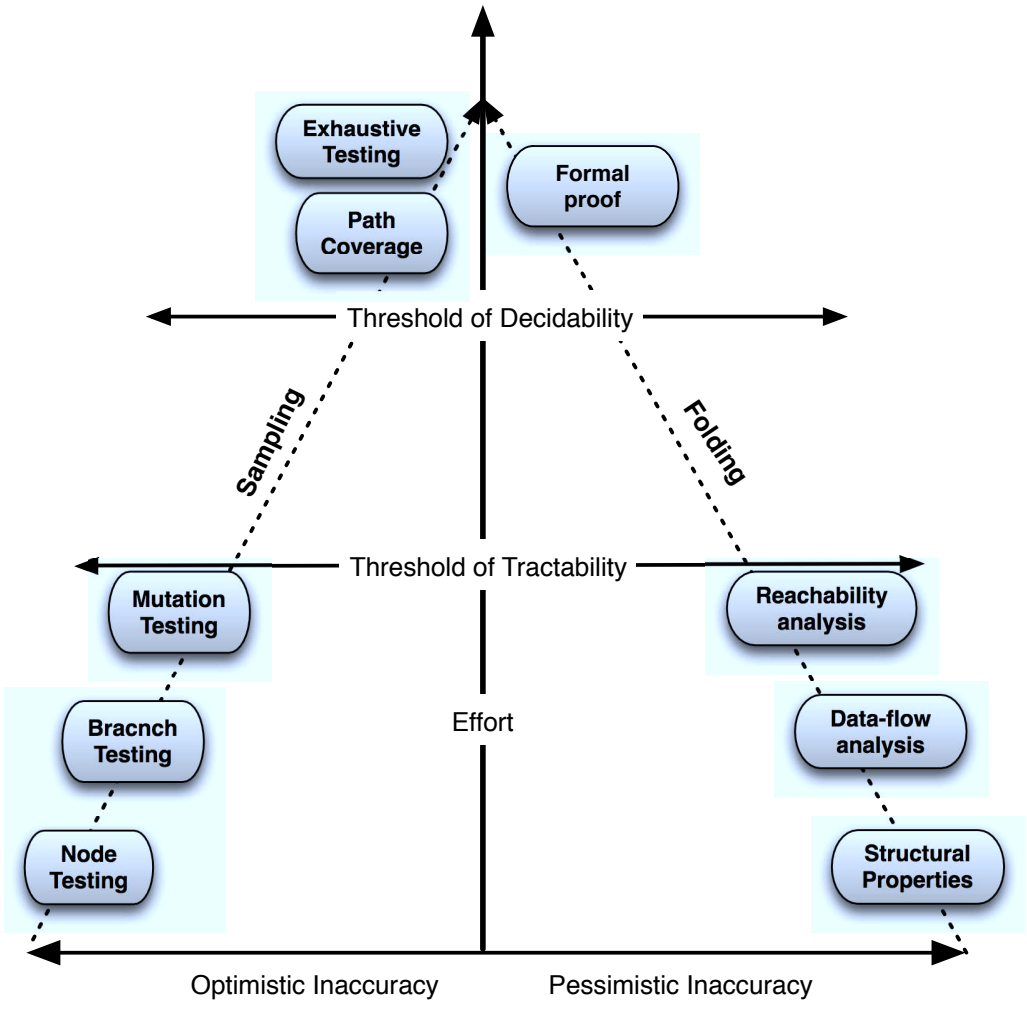


Figure 1.4: The folding and sampling taxonomy

different state space, removing unnecessary details by doing so. It may cause some bad state of the program to be hidden. A thorough procedure is required to implement folding in way which guarantees that no errors will be hidden by the choices of abstraction [YT89]. Folding provides pessimistic inaccuracy. since it might fail to provide a proof of correctness of the system. For example exhaustive analysis petri net could fail with a finite but a large set of states. Instead, running simulations with specific sample space could reveal properties.

Sampling

Sampling, on the other hand, only explores part of the infinite state space, which is what most dynamic analysis techniques do. Picking a sample set/space which guarantees sufficient coverage, and confidence into the system correctness is the challenge. This sample set choice is referred to as *test case selection* and has its own section in this chapter. Random selection of samples could bypass several critical values [YT89]. For example a conditional path value. Systematic approaches provide a better job by taking into consideration such critical values (using partitioning for example). However, recent research suggests that random selection techniques are becoming more and more efficient, mostly in terms of feasibility [Ber07]. Random selection techniques are usually automatic, and can build on feedback information collected while the tests are executed to enhance the test set.

The state space from which sampling and folding occurs, could be decided from the model schemata (petri nets, state charts, flow graphs, etc). It's easier to exploit interaction between techniques when the same model schemata is shared between them.

1.5 Research Foundations and Challenges

In the following we list the main questions, and activities needed to perform software testing in a systematic and engineered matter. For each question, we mention current achievements and any existing related challenges. Such abstraction of activities is reflected, in the practice of software testing, by test plans. Note that we will focus on the dynamic testing, which involves the execution of the System Under Test (SUT), as opposed to other analysis techniques.

1.5.1 Why Are We Testing? (Test Objective)

Defining the *test objective* is the first step to testing. A test objective describes exactly the main motivation of the testing and helps drive the following testing activities. It will help guide the test cases selection process and even the environment in which the test cases should be exercised. We can classify any testing objectives under on of the following three classifications:

Functional Requirements

The term *functional requirements*, in software engineering, is used to describe the formal goals and functionality the system should be built to provide. Functional requirements represent the business logic related behaviour of the system. A *function*, defines for

each input the system receives, the output it will generate. Functional requirements are originated by the customer and then are decomposed along with the system decompositions, and hence can reach as low as small units within the system. In general *functional requirements* describes what the system should do.

Extra-Functional Requirements

Extra-functional requirements tend to describe `How?` the system implements a function rather than defining `What?` the functions should do. For example: The system should have an average response time of less than 1 second. The speech recognition component of the system should achieve an accuracy rate larger than 99 percent. *Extra-functional requirements* are also called quality attributes or constraints on the systems behaviour. Testing within this category tends to require different expertise and set of skills. Examples of extra-functional requirements include : *Accessibility, Performance, Availability, Extensibility and Testability*

Special Purpose

Sometimes the testing objective is not intended towards ensuring any requirement (functional or not). For example: we would like to test the installation of the system on the windows 32bits and 64bits processors. Another example is protocol testing, for example ensuring that a system implements its part of a communication protocol correctly, or according to some standards (pages generated are always strict XHTML)

1.5.2 What Are We Testing? (SUT)

In order to realize the test objective, the next logical step is to identify what should be executed. This is what we refer to as *SUT (System Under Testing)* , that is whatever is being tested. For example considering testing a composite system, several levels of testing could be identified: Unit Testing, Module Testing, Subsystem/Component Testing, System/Acceptance Testing. This classification is seen to be linked to testing phases (unit, integration and system) as well [Gla09]. When we are writing unit tests, the SUT is whatever class or method(s) we are testing; When we are writing customer tests, the SUT is probably the entire application (or at least a major subsystem of it) [Mes07].

Integration testing is defined as ensuring software components can work together properly, through their predefined interfaces.

Challenge: Compositional Testing

As software systems increase in complexity, several new techniques to construct software were introduced to boost productivity, such as promoting reuse in component based development. This lead to many systems today being composed of several individually re-used components. Component in turn are composed of many different modules and units . Classical approach to testing such systems is by *divide and conquer* i.e; by testing each of the composing pieces. Current research provides guidance into how to the organize the execution of testing of the different components. However, more work is needed to minimize the testing efforts especially with the increased adoption of dynamic

system compositions. A survey of recent work in this area can be found in [Ber07]. For example, research questions being investigated such as: how to use the testing results of separate components and units to infer about the global system, and subsequently which test cases need be executed to ensure integration.

1.5.3 How Are We Testing? (Test Case Selection)

After choosing the *test objective* and specifying the *SUT*, the actual execution of the tests can finally take place. However, in order to perform testing, a list of test cases has to be constructed to decide on which behaviour samples to observe. This procedure is referred to as *test selection criteria*, and can be performed in a number of different approaches (random, systematic). Recent work has been pursued in this area reflecting the impact test selection has over test efficacy and success. A test strategy or technique is a systematic method used to select and/or generate tests which will be included in a test suite. In classical research there is two main approaches or strategies, depending on the source of information to derive the test cases [Bei95]:

1. **Behavioural/Functional techniques:** where test case selection strategies are based on requirements. This is the same as black-box testing (e.g: execute all the dirty tests implied by the requirement x). This testing approach applies to all levels of testing. The main technique used in this domain is the *Equivalence Partitioning*, described in [KM07], which help generate much less test case than those generated by exhaustive testing, but with high efficacy in revealing bugs.
2. **Structural Testing:** where strategies are derived from the structure/source code of the SUT (e.g: execute every statement at least once), same as white-box and glass-box techniques.
3. **Hybrid test strategies:** combine both strategies.

Several other and new techniques for test selection have emerged from which we mention the following :

- **Model-Based Testing:** represents the use of models of the system and its environment to drive the testing process. More specifically [UL07] describes the four main approaches to *model-based testing* as:

  1. Generation of test input data from a domain model.
  2. Generation of test cases from an environment model.
  3. Generation of test cases with oracles from a behavioural model.
  4. Generation of test scripts from abstract tests.

  Model based testing is all about automating the design of the testing. It can provide a solution to the oracle problem which is discussed next. As the system complexity increases, the cost of maintenance and testing also increases and hence the use of models (as described in Chapter 2) is desired, and provides a lot of benefits in term of time saving, and even quality, presumably by avoiding low level bugs, and staying closer to the problem domain.

But then *model based testing* enables the reuse of the models which are used to construct the system to help the testing. This is referred to as *white box model based testing*, since it attempts reusing the same models which were used to generate the system to generate test cases. This approach has the risks of missing a critical bug, because the original model of the SUT has missed it, and may prove less effective.

On the other hand, creating a separate model from the requirements of the SUT (black-box), is the other extreme and sort of implies double the work (of modelling that is). This is considered *black box model testing*, where the test cases are generated based on a model of the requirements and not the actual system.

The best approach is to combine both scenarios in a way which can help building the SUT model to generate the test cases, and to reuse some parts of the existing construction models. The book [UL07] presents an extensive information source of model based testing.

- **Anti Model-Based Testing:** represents the other extreme to model based testing, namely using dynamic analysis to synthesis models about the system behaviour. This is a highly useful approach when testing COTS or legacy systems, where models simply do not exist. Dynamic analysis can be usually combined with mutation based testing [SDZ09] techniques to derive and enhance a test suite. Dynamic analysis could include monitoring logs to construct a theory about the system behaviour.
- **Object-Oriented Testing:** this software paradigm, at the beginning gave hopes of overcoming the need of testing, by reducing it to a minimum. However, later it turned out that OO Paradigm introduced a new class of risks and challenges in terms of testing, some of which is testing inheritance code, which needs to be retested extensively to be covered in the inherited context. Also the polymorphism coverage model is relatively different from regular coverage models; Finally, encapsulation increases the difficulty of testing and could cause missing hidden bugs deep in classes. This is highly related to the concept of testability.

Comparison between different test selection criteria has been under heavy research also. It has been identified in [Har00] as a major challenge towards effective software testing research. Since then, analytical studies looked at different factors influencing the ability of techniques to detect faults (systematic versus random techniques for example) [Ber07].

1.5.4 Testing Oracles

Testing Oracles provide a verdict on whether a test case has passed or failed. It accomplishes this by comparing the actual test outcome to the expected outcome. The oracle function is thought to be a magical unit which produce the expected outputs for any test case input, or in other view an engine which annotate each test case with a pass fail verdict according to its output. Subsequently any testing technique requires the presence of an oracle.

Producing such oracles has been one of the barriers to test automation. The two requirements when constructing such oracles is precision and effectiveness in avoiding false positives and false negatives. [Ber07] and [BY01] survey research work regarding oracles

and conclude with the following classifications:

- **Partiality:** represents checking only specific partial criteria of the actual output. This approach has to make a tradeoff between precision and cost.
- **Quantification:** if executable specification languages used to describe oracles, then a tradeoff between efficiency and expressiveness must be achieved.
- **Test case selection:** if using model based testing to select the test cases, then the model could be used to also generate oracles.

1.5.5 Testing Process

Beizer described the process of testing to be of two phases: building a test plan at the requirement gathering phase, and then implementing/executing the test plan only after the software implementation phase is finalized [Har00]. Literature distinguishes among several software development processes. In particular, the waterfall model has been the most dominant form of software construction method in the past. It indicates that the testing process takes place only after the software has been fully implemented.

Much research about testing process has since matured and became more systematic and incorporated into the software development process. This helped better the test-design by thinking about testing at earlier stages of software development process. A widely adopted testing process is the V model [Ber07]. The V model specifies different testing phases (Unit, Integration and system) applied at different stages. Other models like the spiral model, RUP (Rational Unified Process) are also present and widely adopted. Such processes tend to include a testing phase integrated into their progress [Gra08].

Another extreme approach has been gaining grounds lately, especially with the agile community is TDD (Test Driven Development). TDD takes the extreme of writing the testing code before the actual implementation code. Tests in TDD represent executable requirements [Mes07]. But as [Het88] points out, testing is not a phase on its own, but rather there are testing deliverables associated with every phase of the development process.

1.5.6 Test Automation

Test automation is the most important piece of the puzzle in software testing. Software testing methods are trying to keep pace with the software construction tools, as system complexity increases. The need for automation is overwhelmingly critical for many reasons. First, *manual testing* doesn't always work since humans can make mistakes while executing a complicated test case, either on testing inputs, or on observing outputs. Second, It is not economically viable considering the maintenance cost associated to it. Several types of testing such as stress, reliability and performance won't be achievable without automation.

Coverage Criteria Tools

As discussed behavioural testing is based on a model of software, and such models can be wrong; and so to increase confidence in the testing, the structure of the software is

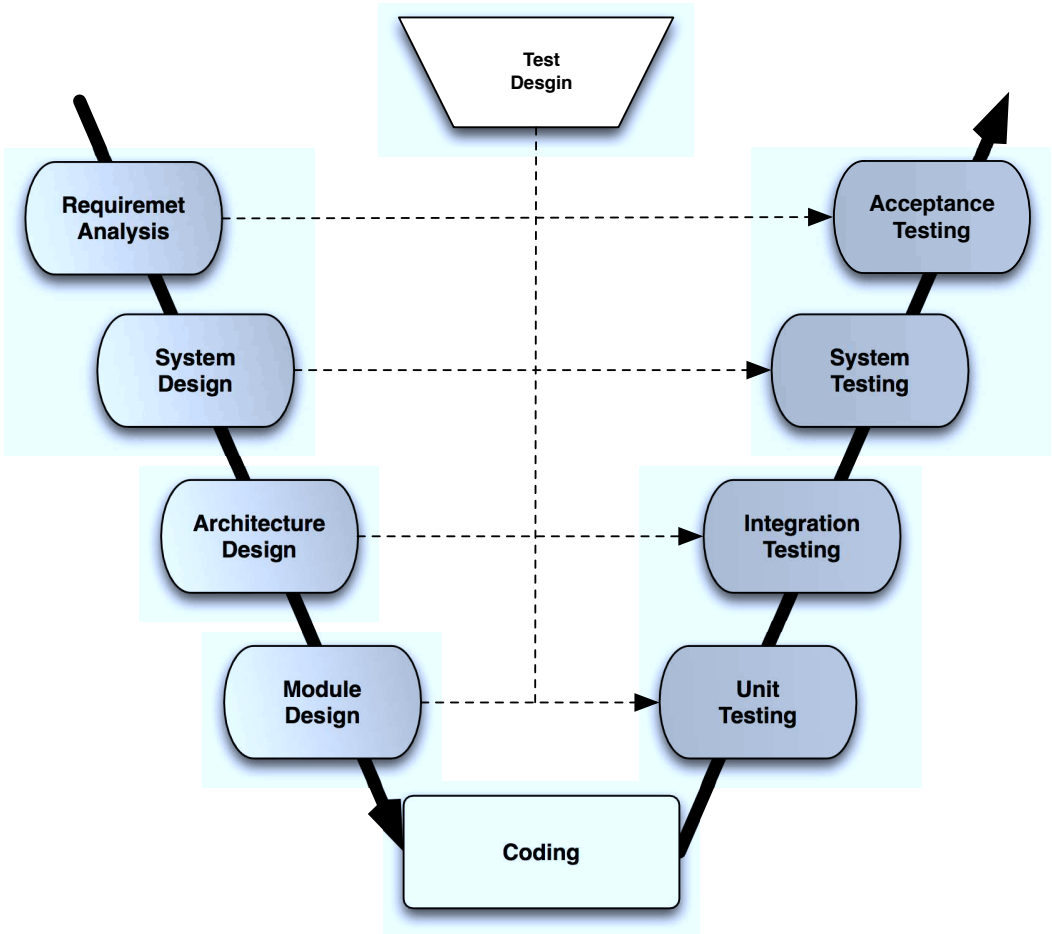


Figure 1.5: The V-Model software process

considered. For example, common sense indicates that each program instruction should be executed while testing at least once. Structural coverage tools provide a quantitative measure of how much of the software was executed by testing, where a target coverage of 100 percent is optimal. The main categories for structural coverage tools as presented in [Bei95] are:

1. **Control-flow Coverage:** the simplest form is source code coverage. However, a more fine-grain coverage such as branch, and predicate condition coverage is also provided. Control-flow coverage tools work (typically) by transparently augmenting the source code of the SUT with instrumentation statements that then will be executed to record which paths and segments were exercised.
2. **Block Coverage (Profilers):** help provide coverage information for the whole system code, and are not limited to the unit test level. Profilers do not interpret the instructions executed (as does a normal coverage tool) but instead records if a given memory location (object-level instruction) has or has not been executed. Profilers may provide this coverage data for individual bytes, individual object instructions, or blocks of bytes or instructions. In [Bei95] two types of coverage are also observed, *deterministic coverage*, where coverage is done for every instruction execution and *statistical coverage* where coverage is achieved by sampling instruction execution periodically. The latter option provide more realistic results but may take thousands of repetitions to reach statistical significance.
3. **Data-flow and Other Coverage Tools:** Although data-flow testing is defined here in terms of behavioural testing, there are corresponding structural concepts and associated coverage tools. There are tools that measure all-uses, all-definitions, all-du paths, and even all-paths.

Test Execution Automation

Test execution is the process in which the input of the test case is exercised on the SUT, and the resulting output is then collected and evaluated using an oracle function. Any test design automation that generates a large number of test cases will be useless without execution automation. The automation of the execution can be achieved in different ways [Bei95]:

- **Testing code**: Writing test programs that specifically test the SUT code is one way to go. However this might not be the best approach, because it introduces the dilemma of testing the test code (since it will have specific logic embedded in it), and testing the testing of the test code, and so on. Also it limits reuse which is counter productive.
- **Testing drivers:** Drivers are appropriate structural tools which can be part of a code package. They can be reused for testing different programs, and provide as part of a framework, an increased automation support. Drivers proceed in three different stages:
    1. **Setup Phase:** for loading initial prerequisites and other hardware or software elements, while also initializing any instrumentation and coverage tools.

2. **Execution Phase:** performs re-initialization as necessary for each test, evaluates assertions and captures output. It also resets instrumentation for every test.
3. **Postmortem Phase:** performs proper test verification through some criterion and reports failures by exceptions. It compares actual to predicted outcomes, using smart comparison methods (e.g. allows you to specify what should and shouldn't be included in the comparison and with what tolerance).

- **Capture/Playback:** a fundamental tool in achieving transition from manual to automation. It could be used in both test design and execution automation. It allows capturing the tester interaction with the SUT interface as part of a test case, and then re-execute the tester role and observe the program's behaviour. The playback phase is just a test driver. Capture/Playback tools have a disadvantage of being dependent on the user interface of the system, and will have to be changed with every change in the interface.

Test Design Automation and Input Generation

Testing automation is achieved by either generating test inputs or automating the testing process. XUnit frameworks for unit testing have helped overcome some of the barriers to unit testing, namely the extra coding necessary for simulating the environment where unit code will run, and the extra checks for the unit's output. However, such frameworks do not help with test generation and environment simulation. The DART project [GKS05] attempts more automation for unit testing by automatically extracting the interface using static code analysis for automatic generation of random test drivers for the interface, followed by dynamic analysis of the program behaviour to drive the generation of new test data to execute along different paths. For more in depth survey refer to [Ber07].

Three main approaches to address the automatic generation of test cases exist today :

- *Model-based test generation*: Incorporates both white box and black box approaches, but most existing tools are state-based and do not focus on input data. One challenge faced when generating test cases from such models, is state explosion. The introduction of symbolism could help overcome this problem, see [Ber07, UL07] for most promising developments in this area.
- *Random test generation*: Traditionally this approach used to be a supplementary approach, since it could not identify critical cases and domain knowledge. However recent clever implementations of random testing appear to outperform systematic test generation. For example random testing can use exploit feedback information observed from the program dynamically. The combination of both random and systematic approaches have great promises in automatic test generation.
- *Search-based test generation*: Explores the set of solutions for a given criteria and using mathematical techniques which helps direct the search towards the potentially most promising area of input space. Reference to work in this area is also provided in [Ber07].

Domain Specific Testing

Domain specific languages are specific languages with restricted domains which helps abstract specification closer to the problem domain. Domain-specific languages provide a special tool to help build better software. Several tools exist today which help translate program specifications expressed in such languages into optimized implementation. Domain specific testing can be considered as testing in specific domains, most likely where domain specific languages are used to construct the software. The Siddhartha framework [RR99] for example help in converting test specifications to domain specific testing drivers. Also the HotTest technique which shows how modelling the SUT using strongly typed domain specific languages allows for automatically embedding domain specific requirements into the test models. Several other techniques and frameworks exist.

2

Chapter 2: Model Transformation Testing

We start by providing a summary of the main work in MDE (Model Driven Engineering). First we provide a high level view of multi-formalism modelling benefits and challenges in Section 2.1, followed by a more detailed description of the different building blocks of MDE, such as meta-modelling, syntax and semantics in Section 2.2. We then introduce model transformation in Section 2.3, and discuss its main design features and approaches with a focus on graph based transformations. Finally, we delve into model transformation testing. We use the testing classification established in Chapter 1, as a basis for classifying techniques in model transformation testing in Section 2.4. We conclude with listing the main challenges need to be addressed to set the basis for the main contribution in the followings chapters.

2.1 Multi-formalism Modelling

Complex systems are not only difficult to code, but also difficult to test and analyze. In most cases they deal with problems at a much different level of abstraction from code. Modelling and Simulation Based Design has surfaced in recent years to address building, maintaining and analyzing such complex systems. However, modelling truly complex systems is a difficult task, since the system structure and semantics overpass a single modelling language, also known as a formalism. Examples of commonly used formalisms are Differential-Algebraic Equations (DAEs), Bond Graphs, Petri Nets, DEVS, Entity-Relationship diagrams, State charts and UML2.0. We present the following introduction to modelling based on the summary provided in [GLV07].

We can achieve modelling such systems in different ways:

- Constructing a super-formalism to include all the formalisms needed for describing the system. However such approach is very complicated and not effective.
- Considering the different components of the system in a way where each can be modelled using the most appropriate formalism for its abstraction. Then using *co-simulation*, the overall modelled system behaviour could be analyzed through simulating each component using the specific formalism simulator. The co-simulation engine orchestrates the flow of input/output data. In this approach, questions about the overall system can only be answered at the level of input/output (state

trajectory) level. It is no longer possible to answer higher-level questions which could be answered within the individual components' formalisms.

- Similarly to the co-simulation, each system component may be modelled using a specific formalism, however, a single formalism is identified and used as a target for symbolically transforming the models of the different components. This process require a transformation function between both formalisms. Some system properties could be changes/lost as a result of the transformation (if the target formalism cant express them), hence the target formalism and transformation function should be chosen wisely to preserve the system properties we intend to investigate.

The model transformation approach seems most elegant and powerful. It enables the synthesis of platform dependent systems from models described in different formalisms. Like generating Java code from UML class and sequence diagrams. However, this approach is most challenging for it carries the difficulty of being tool dependent, and require a great deal of standardization efforts. Object Management Group OMG [1] is one of the largest efforts. It requires interconnecting a plethora of different tools, each designed for a particular formalism. Also, it is desirable to have problem-specific formalisms and tools which is very time consuming. In the light of this we feel the best approach (as we describe in the following sections) is to explicitly model the different formalisms as well as the transformations between them to ensure most compatibility.

2.2 Modelling

Models are an *abstraction* of the real world, they are typically used to describe the structure and behaviour of systems to enable the design and analyses of such systems.

These models, at various *levels of abstraction*, are always described in some *formalism* or *modelling language*. In addition to the *syntax* of a model (how it is represented), one needs to also specify its *meaning* (*i.e.,* assign *semantics*) see Figure 2.1 for a high level view of Modelling Architecture.

On the one hand, we can describe how the system evolves dynamically overtime through specifying its behaviour. On the other hand, we could concentrate purely on the structural aspect of the system that is static and not specifying any behaviour attached to it. Furthermore, we can use models extensively during design to describe both the system structure and behaviour, to enable code generation from these models into different platforms such as hardware specific embedded systems.

In terms of analysis, and as discussed earlier, in many cases, system can be composed of of multiple models representing different views, at various levels of abstraction, and using a plethora of formalisms.

Dissecting a Modelling Language

To "model" modelling languages we will break down a modelling language into its basic constituents [HR00].

1. http://www.omg.org/

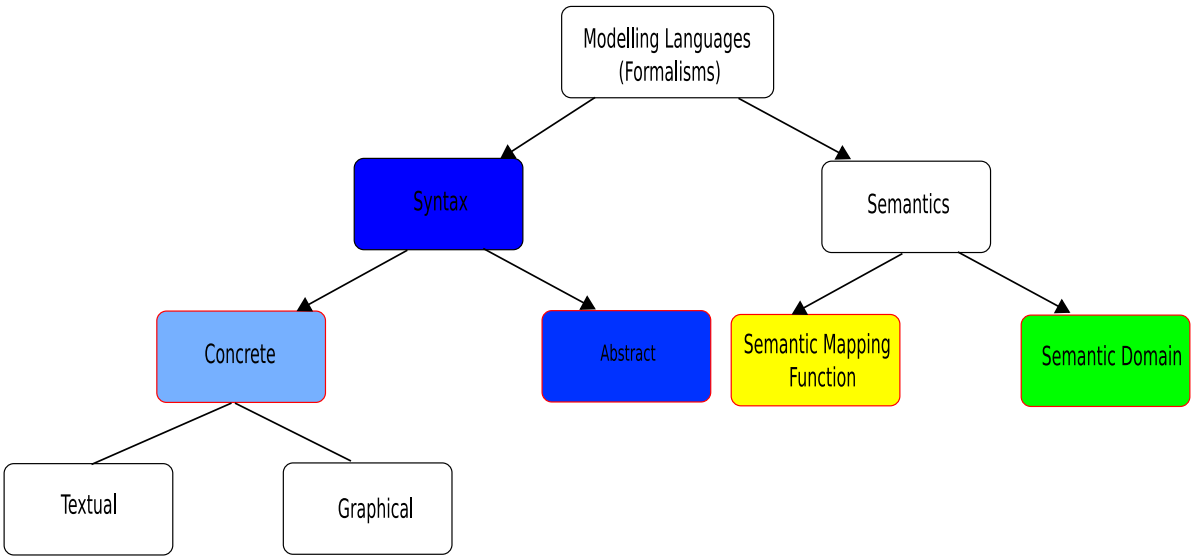


Figure 2.1: Meta Modelling Architecture

As mentioned previously, the two main aspects of a model are its syntax (how it is represented) on the one hand and its semantics (what it means) on the other hand.

Syntax

The syntax of modelling languages is traditionally partitioned into *concrete syntax* and *abstract syntax*. [HR00, GLV07] In textual languages for example, the concrete syntax is made up of sequences of *characters* taken from an *alphabet* These characters are typically grouped into *words* or *tokens*. Certain sequences of words or *sentences* are considered valid (*i.e.,* belong to the language). The (possibly infinite) *set* of all valid sentences is said to make up the language. Costagliola et. al. [CLOP02] present a framework of visual language classes in which the analogy between textual and visual characters, words, and sentences becomes apparent. Visual languages are those languages whose concrete syntax is visual (graphical, geometrical, topological, . . .) as opposed to textual.

The *abstract syntax* represents an "abstract" view of the system capturing the "essence" of the model which is stripped of irrelevant concrete syntax information. We can use multiple concrete syntaxes to represent a single abstract syntax. In programming language compilers, abstract syntax of models (due to the nature of programs) is typically represented in *Abstract Syntax Trees* (ASTs). In the context of general modelling, where models are often graph-like, this representation can be generalized to *Abstract Syntax Graphs* (ASGs) [GLV07].

Semantics

Once the syntactic correctness of a model has been established, its *unique* and *precise* meaning must be specified. Meaning can be expressed by specifying a *semantic mapping function* which maps every model in a language onto an element in a *semantic domain*. For example, the meaning of a Causal Block Diagram is given by mapping it onto an Or-

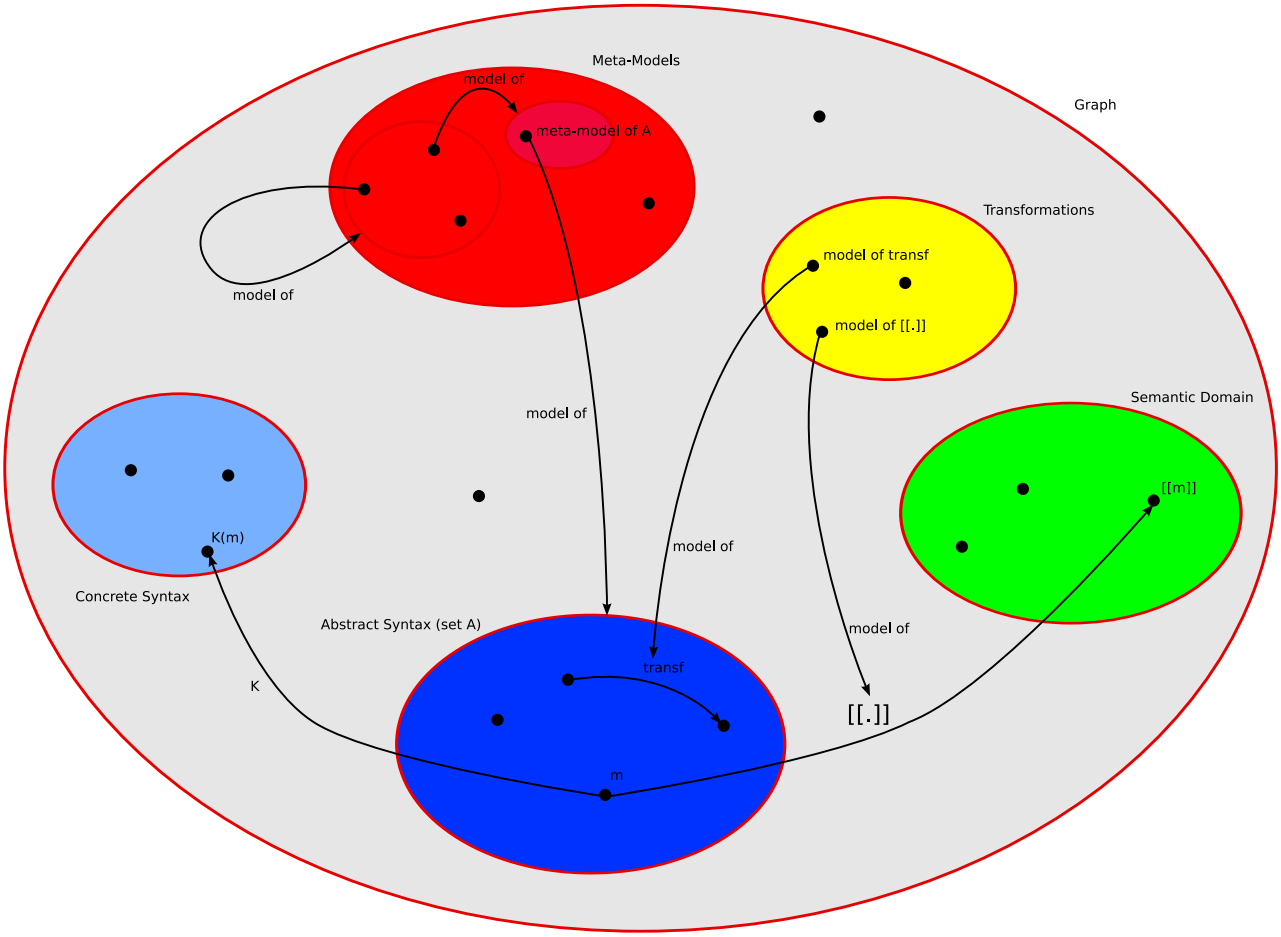


Figure 2.2: Modelling Languages as Sets

dinary Differential Equation. For practical reasons, semantic mapping is usually applied to the abstract rather than to the concrete syntax of a model. Note that the semantic domain is a modelling language in its own right which needs to be properly modelled (and so on, recursively). In practice, the semantic mapping function maps abstract syntax onto abstract syntax.

Meta Modelling Elements As Sets

To continue the introduction of meta-modelling and model transformation concepts, languages will explicitly be represented as (possibly infinite) sets as shown in Figure 2.2.

In the figure, insideness denotes the sub-set relationship.

The dots represent model which are elements of the encompassing set(s).

As one can always, at some level of abstraction, represent a model as a graph structure, all models are shown as elements of the set of all graphs Graph. Though this restriction is not necessary, it is commonly used as it allows for the design, implementation and bootstrapping of (meta-)modelling environments. As such, any modelling language becomes a (possibly infinite) set of graphs. In the bottom centre of Figure 2.2 is the

abstract syntax set A. It is a set of models stripped of their concrete syntax.

Meta-modelling is the explicit description (in the form of a model in an appropriate meta-modelling language) of the abstract syntax set A. Often, meta-modelling also covers a model of the concrete syntax. Semantics is not covered. In the figure, the set A is described by means of the model meta-model of A. On the one hand, a meta-model can be used to *check* whether a general model (a graph) *belongs to* the set A. On the other hand, one could, at least in principle, use a meta-model to *generate* all elements of A.

Several languages are suitable to describe meta-models in. Two approaches are in common use:

1. A meta-model is a *type-graph*. Elements of the language described by the meta-model are instance graphs. There must be a *morphism* between an instance-graph (model) and a type-graph (meta-model) for the model to be in the language. Commonly used meta-modelling languages are Entity Relationship Diagrams (ERDs) and Class Diagrams (adding inheritance to ERDs). However this approach is not sufficient and an extra *constraint language* (such as OCL the Object Constraint Language in the UML) specifying constraints over instances is used to further specify the set of models in a language.
2. A more general approach specifies a meta-model as a transformation (in an appropriate formalism such as Graph Grammars) which, when applied to a model, verifies its membership of a formalism by *reduction*. This is similar to the syntax checking based on (context-free) grammars used in programming language compiler compilers.

Both types of meta-models (type-graph or grammar) can be *interpreted* (for flexibility and dynamic modification) or *compiled* (for performance).

The advantages of meta-modelling are numerous. Firstly, an *explicit* model of a modelling language can serve as *documentation* and as *specification*. Such a specification can be the basis for the *analysis* of properties of models in the language. From the meta-model, a modelling environment may be *automatically* generated. The flexibility of the approach is tremendous: new languages can be designed by simply *modifying* parts of a meta-model. As this modification is explicitly applied to models, the relationship between different variants of a modelling language is apparent. Above all, with an appropriate meta-modelling tool, modifying a meta-model and subsequently generating a possibly visual modelling tool is orders of magnitude *faster* than developing such a tool by hand. The tool synthesis is *repeatable* and *less error-prone* than hand-crafting.

As a meta-model is a model in an appropriate modelling language in its own right, one should be able to meta-model that language's abstract syntax too. Such a model of a meta-modelling language is called a *meta-meta-model*. This is depicted in Figure 2.2.

A model m in the Abstract Syntax set (see Figure 2.2) needs at least one concrete syntax. This implies that a concrete syntax mapping function κ is needed to map an abstract syntax graph onto a concrete syntax model. Such a model could be textual (*e.g.,* an element of the set of all Strings), or visual (*e.g.,* an element of the set of all the 2D vector drawings). Furthermore, concrete models can be modelled in its own right. Often, multiple concrete syntaxes will be defined for a single abstract syntax, depending

on the user. If exchange between modelling tools is intended, an XML-based textual syntax is often used, or a using a binary format for more efficiency. A visual concrete syntax is often used for human consumption, mainly when the formalism is graph-like. The concrete syntax of complex languages is however rarely entirely visual (we represent equations using textual concrete syntax).

Finally, a model m in the Abstract Syntax set (see Figure 2.2) needs a unique and precise meaning. As previously discussed, this is achieved by providing a Semantic Domain and a semantic mapping function `[[.]]`. This mapping can be given informally in English, pragmatically with code or formally with model transformations. Natural languages are not executable. Code is executable, but it is often hard to understand, analyze and maintain. This is why formalisms such as Graph Grammars are often used to specify semantic mapping functions in particular and model transformations in general. Graph Grammars are a visual formalism for specifying transformations. They are defined and at a higher level than code. They express complex behaviour with a few graphical rules. Furthermore, Graph Grammar models can be analyzed and executed. As efficient execution may be an issue, Graph Grammars can often be seen as an executable specification for manual coding. As such, they can be used to automatically generate transformation unit tests if they are are no intermediate structures.

2.3 Model Transformation

Model transformation is a key element of Model Driven Engineering, and in the light of the previous discussion about MDE importance we will introduce model transformations in this section. It can be thought of in software engineering as programs that write programs. In fact any data manipulation can be thought of as a transformation. Code generators represent one form of a Model to Text transformation. Note that the generated text can be thought of as a model to a certain extent (since it will conform to the syntax of the target language). Model transformation accepts a model as input and produces a model as an output, where each model conforms to a specific meta-model as shown in Figure 2.3. Model transformation is defined on meta-model elements, but its implementation transforms models. Endogenous transformation is when the source and the target meta-models are the same, and exogenous is when they are different. Model Transformation can be used to transform models, and also to describe the semantics of modelling languages. For example, defining the operational semantics of a finite state machine.

Model transformation is a concept related to compilers (program transformations). Both disciplines however have evolved separately, and hence have different communities. The subject of the transformation (models versus source code) is the main difference. Program transformation are focused and optimized for specific languages and platforms, where model transformations tend to operate on a different and more diverse set of artifacts including UML models, database schemas and requirement specifications. Another set of differences lie in the fact that model transformation may support multiway transformation (transformation to different levels and back), as well as providing traceability.

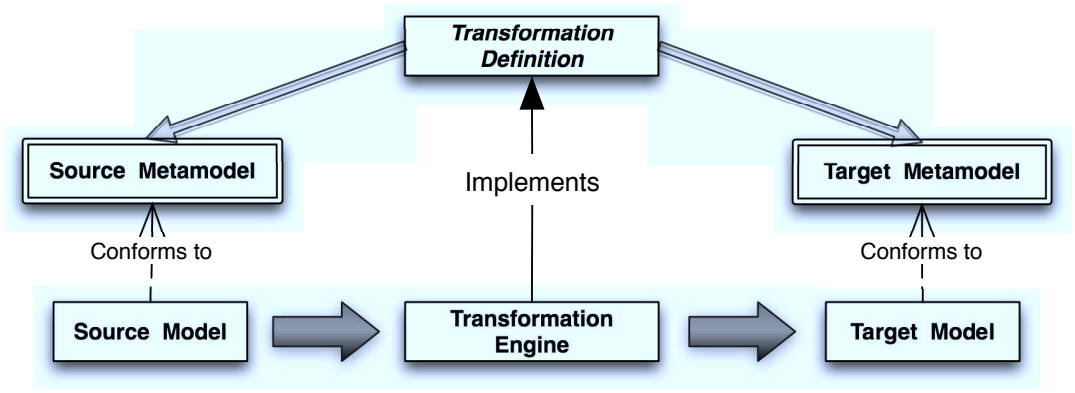


Figure 2.3: Model Transformation basic concepts

2.3.1 Main Design Features

A design-feature based classification of model transformation is introduced in [CH06]. (Note that although the classification we are introducing is a general, we will focus our discussion on graph based methods for model transformations.) When transformation is dealing with models, *Transformation Rules* represent the basic building blocks of model transformation.

Rules have an input meta-model domain and an output meta-model domain, both domains can be identical if the transformation is endogenous. The *domain* expresses how the rules can access and operate on model elements. Models can be described in the rules body using graphs, terms or string-like structures containing variables, constraints and patterns. Rules can have a syntactic separation in terms of a LHS (Left Hand Side) which operate on the source model and a RHS (Right Hand Side) operating on the target model (replacing of original LHS patterns), however when a transformation is described entirely using a programming language, such a separation doesn't exist.

A transformation may involve more than two models and be generalized to an $n \rightarrow m$ relation. This is highly desired when model synchronization is the goal of the transformation.

Parametrization of rules allows for more reuse.

During the transformation execution, the *location determination* of the first occurrence of a a rule's LHS takes place on the source model. Depending on the strategy used, the transformation may produce different outputs in different runs. A deterministic transformation indicates that a repeated execution will always produce the same output. When several choices occur in a non-deterministic transformation, we distinguish concurrent execution where a rule is applied on all choices at the same time, from one-point execution where the rule is applied on only one selected location in a non-deterministic way.

Rule scheduling determines which rules are applied when and can be applied in different approaches. Scheduling can be achieved by explicit control structures or implicitly by the tool. Explicit scheduling can be internal where rules can invoke each other, or external

with a clear separation of the rules from the scheduling logic. Moreover, several rules may be applicable at the same time, or sequentially using a conflict resolution to select rules.

Rules can be organized into modules and packages, and also could allow inheritance and extensions to further facilitate re-use.

Incremental transformations facilitate consistency among models. Furthermore, change detection and change-propagation mechanisms are used.

Tracing transformation execution is crucial for model transformation debugging and analysis. Traces may be automatically generated in the source or the target or kept track of separately.

2.3.2 Model-to-Text Transformations

Model to Text transformation represent mainly code generators (for example generating XML, HMTL, Java . . .), and distinguished mainly because of the target's concrete syntax is textual. It's noted that in most cases the produced text will have a structure which conforms to a specific syntax, and in turn a sort of a meta model.

There exist two main approaches for model to text transformations, summarized in the following:

- **Visitor-Based Approaches:** the approach here is to traverse, or iterate over the elements of the model in a specific order and to write into a text stream. An example of this approach is JAMDA (Java Model driven architecture [2]).
- **Template-Based Approaches:** a textual template resembles the text to be generated, with fields relating to the model, and meta code. textual templates can be independent of the target language. and hence simplify the generation of any textual artefact. No syntactical separation between the LHS and the RHS exists in the transformation rules .

2.3.3 Model-to-Model Transformation

There exist an extensive number of tools and approaches to model to model transformation in the literature, which were surveyed in [CH06], and the following classifications are suggested:

- **Direct-manipulation:** in this approach models of an API to operate on them, usually using a programming language such as Java.
- **Structure-driven:** in this approach the framework provides scheduling and application strategy. The transformation is performed in two distinct phases, first by creating the hierarchical structure of the target model, and second by setting the attributes and references on the target elements.
- **Operational:** extending the meta-models with facilities to extend computations. For example we can add imperative constructs to OCL (Object Constrained Language), and combine that with MOF (Meta Object Facility)[OMG08] to get a full

2. JAMDA, Java Model Driven Architecture 0.2, http://sourceforge.net/projects/jamda

programming language. Examples of this approach are QVT [OMG08], C-SAW [LZG05], and Kermeta [Ker05].

- **Template-based:** uses meta-code and annotations to help directing the transformation.
- **Relational:** includes declarative approaches using mathematical relations between the source and target model, examples include QVT relations.
- **Graph transformation based:** used when we can represent models as typed, attributed and labeled graphs, and hence the theory of graph transformation can be used to execute the transformation.
- **Hybrid:** combining multiple of the above techniques.

2.3.4 Graph Transformations

Blostein et al [BFG96] during the late 90s explored issues regarding the practical use of graph rewriting. Graphs provide high expressiveness and flexibility for data representation, and there are many advantages gained from representing the model transformation using graphs over the the more general programming languages based transformations.

Graph based model transformation is considered the method with the most potential for industrial adoption. Issues such as expressiveness, scale-ability and re-use of models of graph transformation as well as the ability to integrate such models with traditional software components were considered critical enablers for wide-spread use of graph transformations [SV08].

Many of these issues were considered and tackled through a wide range of approaches, most of which are surveyed in the general survey [CH06], and the graph transformation specific survey in [TEG+05].

The programmed graph rewriting main requirements are outlined in [LLMC05]. First, the control structure of the graph transformation should be achievable through control flow primitives such as looping, and conditional branching. Second, encapsulation and organization in a hierarchical scheme allows for more reuse and higher modularity. Some tools add expressiveness through non-determinism and parallel composition. In general, also the control structure should be expressed in a neutral way, from any programming language. Explicit incorporation of time is rarely present in any of the current tools.

Note that our own MSDL lab tool: *AToM3* [dLV02], "A Tool for Multi-formalism and Meta-Modelling" is a visual environment which provides priority-based control structuring.

We will now move into investigating the problem of testing this complex transformations.

2.4 Testing Model Transformation

MDE advocates human involvement at the correct level of abstraction, using model transformation to automate the complex procedures like code generation, aspect weaving, or multi-view modelling and model composition. Writing such a complex transformation is inherently complex and subsequently error-prone. The increased promotion of reuse also

dictates the importance of reliable and validated model transformation implementations. The risks involved are also related to the vast number of existing approaches to implement the model transformations as discussed earlier in the chapter. However, model transformation testing as examined in this work does no attempt to test the correctness of the transformation engine. Instead it assumes that the underlying transformation engine is correct and focuses of testing of the realization of a particular transformation implemented in a particular language or a tool. For example, when using a rule based graph transformation engine such as *AToM*3, we can implement certain transformation by specifying rules and priority. We can assume that the engine can apply the transformation rules on the host graph as expected. Model transformation testing which we focus on pertains only to detecting bugs in the specified rules or priority in such a system.

In the following we present a running example of a model transformation specification since a large number of the literature work refer to it. It also serves as an example problem which model transformation has to address.

2.4.1 UML2RDBMS : UML-to-Schema transformation

This example represents a benchmark for model transformation languages and tools. The OMG group provides a description of UML2RDBMS in the Meta Object Facility (MOF) 2.0 Query/View/Transformation specification document [OMG08]. UML2RDBMS has been the benchmark since it was first introduced during the Model transformation workshop [Bé06] as a mandatory example for all submissions to attempt. The following description of this case study is based on the specifications in [Bé06].

UML Class Diagram Meta Model

In Figure 2.4 we present a simple version of the UML meta model which is relevant to this transformation. The simplified CD meta-model is represented using UML class diagrams for simplicity. It includes the abstract concept of classifiers, which comprises classes and primitive data types. Packages contain classes, and classes contain attributes. All model elements have names, and classes could be labeled as persistent.

The following two constraints are part of the UML meta model :

```
1 context Class inv:
2 allAttributes()->size > 0 and
3 allAttributes()->exists(attr | attr.is-primary = true)
```

Finally, classes can have relationships with other classes called associations. Such relationship is only allowed if both classes are marked as persistent. This can easily be added as an OCL constraint.

RDBMS Meta Model

Figure 2.5 represents the RDBMS (Relational DataBase Management System) meta model. The meta model is also expressed using UML class diagrams. A schema is composed of tables, and tables are composed of columns. Each column has a type which is represented as a string. Every table has one primary-key column. Finally, the foreign keys relates foreign-key columns to tables, and is labeled as Fkey.

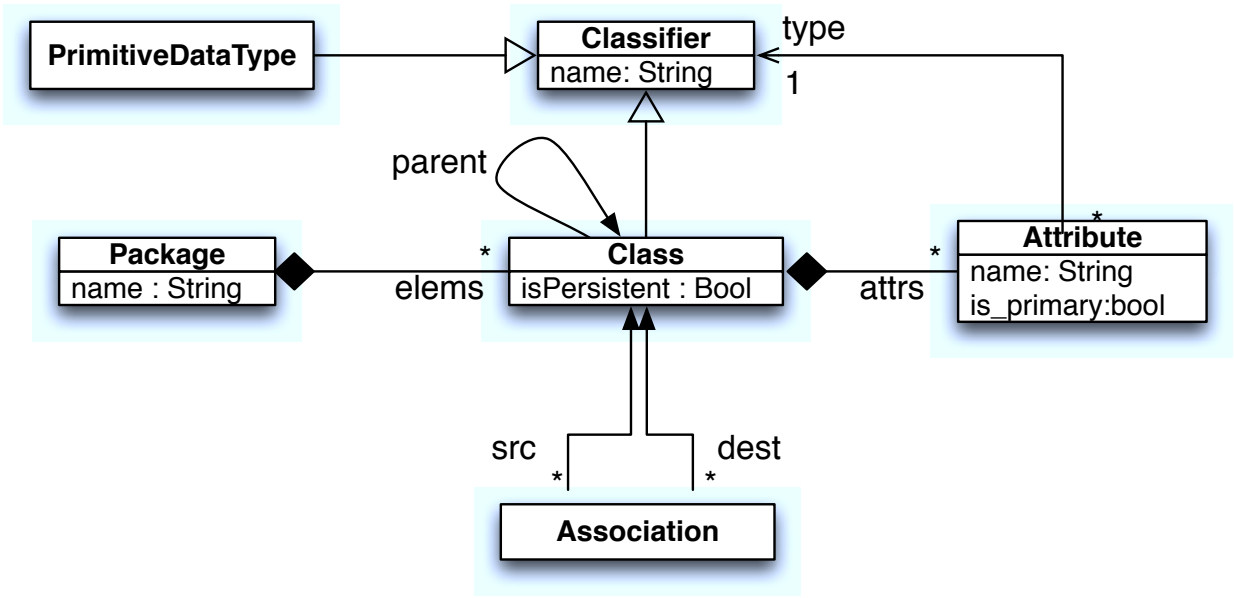


Figure 2.4: Simple UML Meta Model

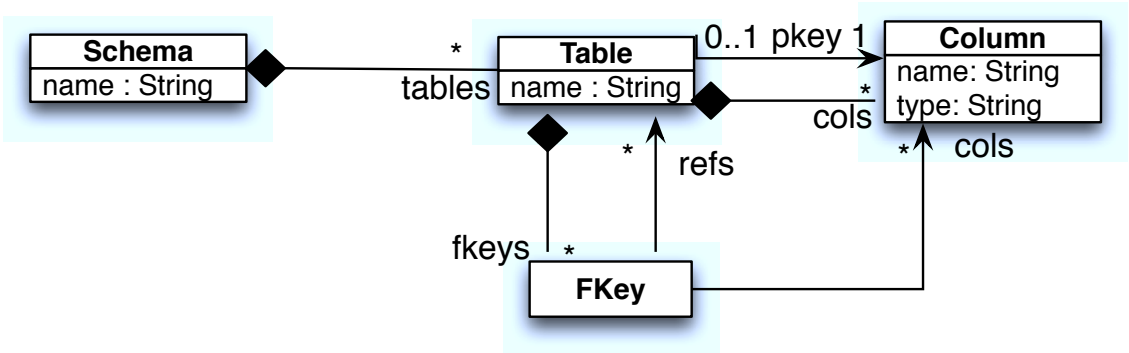


Figure 2.5: Simple RDBMS Meta Model

UML2RDBMS Transformation

The transformation named UML2RDBMS should only accept input models which conform to the UML cd meta model, and only produce models conforming to the RDBMS meta model as described above. The main mappings which need to be performed by the transformation, as presented in [Bé06] are:

1. Every package in the input UML model should be transformed into a schema with the same name in the resulting model.
2. Classes that are marked as persistent in the source model should be transformed into a single table of the same name in the target model. The resultant table should contain one or more columns for every attribute in the class, and one or more columns for every association for which the class is marked as being the source
3. Classes that are marked as non-persistent should not be transformed at the top level. For each attribute whose type is a non-persistent class, or for each association whose *dst* is such a class, each of the classes' attributes should be transformed as per rule 4. The columns should be named name-transformed-attr where name is the name of the attribute or association in question, and transformed-attr is a transformed attribute, the two being separated by an underscore character. The columns will be placed in tables created from persistent classes.
4. Attributes whose type is a primitive data type (e.g. String, Int) should be transformed to a single column whose type is the same as the primitive data type.
5. Attributes whose type is a persistent class should be transformed to one or more columns, which should be created from the persistent classes' primary key attributes. The columns should be named name transformed attr where name is the attributes' name. The resultant columns should be marked as constituting a foreign key; the FKey element created should refer to the table created from the persistent class.
6. Attributes whose type is a non-persistent class should be transformed to one or more columns, as per rule 3. Note that the primary keys and foreign keys of the translated non-persistent class need to be merged in appropriately, taking into consideration that the translated non-persistent class may contain primary and foreign keys from an arbitrary number of other translated classes.
7. When transforming a class, all attributes of its parent classes (which must be recursively calculated), and all associations which have such classes as a *src*, should be considered. Attributes in subclasses with the same name as an attribute in a parent class are considered to override the parent attribute.
8. In inheritance hierarchies, only the top-most parent class should be converted into a table; the resultant table should however contain the merged columns from all of its subclasses.

Since we are interested in testing and not building the transformation, we will not include a realization of the above specification of the UML2RDBMS transformation. Instead we will use the specification to demonstrate testing techniques in the literature. We do provide a complete realization in chapter 4 of another transformation, Traffic2PetriNet,

to demonstrate our framework. We will now provide a literature review of the current model transformation testing approaches. We will use the same classification approach used in Chapter 1 to determine advancements, opportunities, and challenges.

2.4.2 Why Are We Testing? (Test Objective)

As discussed in 1.5 the two main categories in testing are *functional requirements* and *non-functional requirements* testing. Most of the ongoing research in model transformation has been focused on case studies where only functional requirements are examined. For example, the UML2RDBMS which is the standard transformation doesn't examine any extra-functional requirements such as performance or scalability.

Tools such as Kermeta [Ker05] do provide better performance than visual tools like $AToM3$ [dLV02] for example. Very little attention has been paid to such criteria in current model transformation testing research. But as model transformation is being used in more applications and becoming more complex, the need for such extra-functional requirements to be tested is inevitable.

Finally, we predict there will be a need for testing frameworks to support the execution and automation of such extra-functional test cases as well.

2.4.3 What Are We Testing? (SUT)

The System Under Test (SUT) in model transformation testing is the implementation of the complete transformation in most cases, mainly because the intermediate structures do not generally represents valid models. However, depending on the transformation technique (rule based or direct manipulation) and the testing approach (white-box or black-box) compositional testing could be possible.

2.4.4 How Are We Testing? (Test Case Selection)

When testing model transformations, most of the existing test selection techniques in software testing still apply. Testers would have to provide a list of input test models, run the transformation and check the outcome correctness according to the specifications. Coming up with an arbitrary list is achievable, however, as mentioned in the previous chapter, techniques to guide the selection and to qualify the inputs is needed (adequacy criteria). The difference in this case is that test inputs are models which should conform to the input meta model. In an analogy to the testing world, we consider the different approaches to selecting input models representing the test cases :

Black-box

Test cases are based on the requirements (input domain and behaviour), and not based on the specific internals of the implementation of the model transformation. Techniques in this category have the advantage of being applicable to all transformation languages and tools. Subsequently, these techniques can use coverage criteria which is inspired by partition analysis [Bei95], and possibly other coverage criteria specific to the input meta model formalism, to generate input test cases. Such an approach is data centric, as it focuses merely on data structures and values.

Meta-model coverage: The work by Fleurey et al [FSB04] uses partition analysis on the input meta model. They present techniques to explore the input meta model which is expressed using EMOF or UML, based on the following *coverage criteria*:

- **Association End Multiplicities**: for each association end, each of the representative multiplicity must be covered. Then, the representative multiplicity pairs can be computed using the Cartesian product of possible multiplicities at each end.
- **Class Attribute**: for each class attribute each representative value must be covered, if the attribute type is not simple then it needs to be processed as an association, according to the previous rule.

Given the two coverage criteria above, the *representative values* are then created, using two types of partitioning: First, an approach based on the default partitioning which is based on the structure or type of data. The advantage of this approach is the extraction of these values can be fully automated when all the data selection policies are provided. And the second approach, is the knowledge-based portioning where representative values are extracted from the transformation itself (from the requirements since this is a black-box approach). In particular from the pre and post conditions where relevant values may be indicated.

Both techniques can be combined to produce the most coverage possible.

The following table contains examples of representative values for partitioning different model elements from the UML2RDBMS transformation described earlier:

Meta Model Element	**Representative Values**
Class::name : String	Null, "", "something"
Class::isPersistent : Bool	True, False
Class $->$ attribute : [0..*]	[0], [1], [>1]

The next step is creating the *coverage items*, based on the representative values. This process leads to the definition of the most important combinations of the input meta model representative values that should be covered.

Coverage items are computed after eliminating all invalid combinations. Each coverage item represents a constraint on the input models of the transformation. However, it should be noted that the computation complexity increases exponentially for this process as the meta model size increases.

The following table represent a subset of the coverage items for the UML2RDBMS transformation, not that we only list items for the class elements of CA criteria:

Criteria Type	**Meta Model Element**	**Chosen Values**(name,isPersistent)
CA	Class	"",True
CA	Class	"",False
CA	Class	"something",False
CA	Class	"something",True
CA	Class	Null,True
CA	Class	Null,False

Finally, the coverage criteria of input domain is achieved by producing models satisfying a high volume of coverage items.

Transformation rules coverage: Model transformation rarely spans all the input meta model, which means that most of the test cases generated based on the input meta model coverage would be useless. In [FSB04] an *Effective meta model* is proposed to overcome this. It is a subset of the original meta model, computed by selecting only the elements referred to in the transformation and the pre and post conditions. This will allow better test case generation for the relevant subset of the meta-model. For example in the UML2RDBMS transformation, the overall input meta model could include other diagrams like sequence and interaction. The effective meta model would eliminate all the useless cases and help focus on only parts of the meta model which are used.

In [WKC08] authors have implemented a tool which can automate the generation of the effective meta model to restrict the scope of the transformation, followed by the generation of representative values for model elements and finally the coverage items to construct test cases.

Mutation based Test Case Generation: In [SB06], authors describe mutation based approach to generating input test models, using mutation operators. In their work, they describe the process by first automatically extracting mutation operators, which are specific to any meta model that is expressed using EMOF. The Mutation operators include three categories: First, operators to create objects of a particular concrete class in the instance model. Second, operators to create a relationship between two existing objects in the instance model. And finally, operators specifying an attribute for an existing object.

Mutation operators should ensure that mutated models will still be syntactically correct and a valid model. A collection of plans is created such that each plan contains a list of atomic mutation operators. Such plans can then be used to synthesize or evolve models incrementally. The synthesized models serve as input test cases for the transformation. OCL constraints should be used however to validate and guide the model synthesis process.

White-box

White box testing involves examining the model transformation implementation to select the test cases. It is in general more difficult given the numerous approaches for model transformation. It's not clear at the current stage which transformation language, given the diversity, should be considered for white box testing techniques [BGF+09].

However, initial research has been done. For example [KAER07] explains attempts to use white box techniques on testing model transformations. The work focuses on model transformation in the context of business process models. The authors describe using an iterative approach to the design and implementation of the model transformation. Starting with a high level design of the transformation that captures graphically, using the concrete syntax of the underlying modelling language, the main features using a set of conceptual transformation rules. The transformation was then implemented in Java code based on the graphical rules. Testing the transformation is done by creating model templates for each of the conceptual rules, followed by automatically generating model instances which represent valid test cases from each template. Covering the rules

guarantees meta model coverage in a white box approach. The OCL constraints specified on the transformation are used to construct interesting test cases using the following procedure: First, identify the elements changed by the transformation (either from the conceptual rules or from the final implementation). Then, identify constraints that are dependent on theses elements. Finally, for each constraint, construct a test case that checks the validity of the constraint under the transformation.

In conclusion this white-box approach tests the correctness of rules individually to determine correctness. It is highly dependent on the implementation, and test cases will have to change if the rules change.

Hybrid Approaches:

Hybrid approaches involve combining both black-box and white-box techniques. For example Wang et al [WKC08], implemented a tool which integrates both techniques. White box, because the tool can take any model transformation implemented in the Tefkat [LS] language and the input meta model of the transformation to automatically detect the effective meta model. The tool then use black-box techniques to generate input test models using techniques discussed in the previous section.

2.4.5 Testing Oracles

The oracle problem in the context of model transformation testing entails validating the correctness of each of the output models produced by the transformation. An oracle function should exist for each transformation, and accepts the input model and the produced output as parameters to provide a *pass* or *fail* verdict to each case.

Note that all the test generation techniques discussed in Section 2.4.4 lack an oracle function, and need expected output to be specified manually for each test case.

The oracle problem is often thought of as the model comparison problem. However, limiting the oracle problem representation by only considering only the model comparison problem has several disadvantages. For example, expected models are often complex and hard to synthesize. Also in some cases, the verdict of the oracle depends on deep analysis of different properties of the actual model elements. Ultimately, the problem becomes one of checking semantic equivalence of models rather than of syntactic equality (we discuss our solution to this problem in Ch4).

Mottu et. al. in [MBT08] discusses oracle functions in the context of model transformation testing and identify three techniques to implement testing oracles:

- **Model comparison:** The oracle function compares a reference model with a model resulting from the transformation of the test model. However, in the general case the comparison is an NP-complete problem because of the graph isomorphism problem. Several attempts for comparison algorithm have been proposed [LZG05], but most rely on strong assumption such as the presence of unique object identifiers, which is only true when the models are produced in the same environment. Also in [LZG04], authors discuss model comparison as an essential element to model transformation testing and models version control. Some of the issues regarding

model comparison include: What properties of models need to be compared? At which level to compare models ? What are the effective algorithms?. Note that Chapter 3 of this thesis is dedicated to discussing and addressing this challenge.

- **Contracts:** Pre-conditions constrain the set of input models and post-conditions declare a set of expected properties. These can be used to represent partial oracle functions. They can be expressed in OCL or other languages and tools [KAER07].
- **Pattern matching:** Verifying if a certain pattern exist in a model. Patterns can be expressed using OCL assertions or model snippets (subset of a model that conforms to a meta model). Such snippets can be expressed by the tester using the same environment used for writing input test models. Patterns are post-conditions which apply for specific input models, and hence are more specific than the contract post conditions.

Solutions to ease the complexity of oracle problem summarized in the following list [MBT08], where an oracle can be built using:

1. **A reference model transformation:** When another implementation of the transformation exists, the oracle function can compare the output model from the transformation with the output model from the reference transformation to ensure they both are the same. However this approach is complex, and in most cases hard to use.
2. **An inverse transformation:** The oracle can compare the input model to the result model of two steps: first, applying the transformation under test to produce an intermediate model, then feeding that intermediary model into the inverse transformation to produce the result model. The comparison should show that both models are the same once this process is complete. To use this approach the transformation needs to be injective, which is a rare case. For example adding inheritance to our UML2RDBMS would not allow an inverse transformation.
3. **An expected output model:** The oracle compares the actual output test model with a provided output model by the tester for equivalence. The task of creating new expected models is complicated for complex data structures such as models.
4. **A generic contract:** Depending on the inputs, the contract can represent a post condition of the transformation outputs. It can be thought of as a relationship between the input and the output models. For example an OCL contract could validate that the output model has a table with the same name as each persistent class in the input model.
5. **Model snippets:** The oracle can check if the output model of the test transformation contains n model fragments. For example, we can a create model fragment that represent one table called A, another fragment with a table named B, and finally a fragment that contains a table with no specific name. Then certain input models can require certain model fragments to apply on their outputs. This will allow for more reuse of such fragments across different test cases, and greatly reduce the effort of testing.

2.4.6 Testing Process

[Kus04] proposes an analogous process to the water fall process to building transformation, however not much has been done on formalizing software construction processes for the MDE world, and specifically for model transformation. Ideally, a systematic process to build model transformations in the context of MDE should be defined and further it should contain an integrated testing component. Testing Process for Model Transformations could be identified as a research challenge and opportunity.

2.4.7 Test Automation

The automation of the testing process involves several aspects summarized in the following sections. Since the testing process consist of different stages, they can each be automated and implemented using a tool, or compiled into an overall testing framework.

Test design automation: generation of test cases

The ability to automatically generate input test models for different model transformation. The process can use data collected from different sources as discussed in Section 2.4.4.

In [WKC08], authors have implemented a tool which can automatically generate test models, based on the input meta model of the transformation, and the implementation (written in Tefkat) and produce a set of input test cases for the transformation. Their work is based on the original data partitioning for models presented in [FSB04].

Sen et al, in [SBM08], present a tool called Cartier which provides an automatic approach for generating and qualifying input test models. It can interpret and combine knowledge from different sources which are then used to synthesize models. The knowledge is encoded into constraints expressed using the Alloy language. The overall knowledge sources are:

- Meta model expressed in Ecore.
- The transformation pre-conditions as OCL constraints.
- Partitions of meta model as sets of objects expressed in a model fragment language.
- The test model objectives expressed in Alloy.

In [SB06], authors presented an automated transformation to generate mutation operators, and described how it could be compiled into a genetic algorithm to synthesize models which are valid input test cases (as mentioned in Section 2.4.4).

The work in [FBMT09] describes a framework to asses the quality of given input test models for testing a given transformation. The framework examines the coverage of the test models with respect to the meta model and the transformation. Instead of the naive strategy of combining partitions to generate the combinatorial product of all partitions, the notion of object and model fragments is used to define specific combination of ranges for properties that should be covered by test models. The framework works by generating model fragments for the input meta model, and then providing guidance into which fragments still need to be covered by the test set.

Execution and Result collection

Once the test cases have been generated the testing execution process can take place. This step involves executing the transformation with each of the specified input test cases, and collecting the associated resulting models. In some cases this involves automatically loading the input model into a specific state/representation under which it can be ready for execution. In [LZG05] Lin et al. describe a testing framework for model transformations that can automate this task. It focuses on the testing within the context of model-to-model transformation where source models and target models belong to the same meta-model. The framework relies on model comparison as an oracle function, and hence input test models and their expected output models have to be specified beforehand. Their testing framework supports construction/loading of test cases based on test specifications, the execution of test cases, and examination of the produced results, it consists of the following components:

- **Test Case Constructor:** The test case constructor interprets the test specifications to retrieve the necessary information involved in a test case, the input model, the expected model, and finally generates an executable test case. It should be noted these elements are manually prepared, and not created by the framework.
- **Test Engine** Exercises each test case dynamically through the executor. It then collects the generated output models and pass that to the comparator which in turn compares it to the expected output model for this test case.
- **Test Analyzer:** Receives from the test engine comparator the comparison outcomes, namely the difference set. The analyzer then lets the tester visualize the differences and mappings between the actual versus the expected models.

The framework is tool and language specific, and is integrated within the C-SAW model transformation engine and the GME (General Modelling Environment).

2.4.8 General Challenges

Several challenges which face testing model transformations exist, some of which are under heavy research and have promising solutions, and some are caused by the relatively recent nature of MDE research. In recent work, Baudry et al [BGF+09] discusses the following general challenges of testing model transformations:

- **Complex input and output data:** Especially in graph transformation where input and output data is made of complex models represented using graphs. For example these models can have several views (which need to be consistent). This increases the difficulty of generating test models, mainly because the automatic generation translate into a complex constraint solving problem when synthesizing graphs with a lot of multiplicities and OCL constraints. Also such complexity and the lack of historical data makes it difficult to determine the efficacy of test selection criteria. Finally, in terms of the output complexity this also reflect on the complication of finding an oracle. However, several pragmatic solutions to address this problem have been proposed.
- **Model management environments** Since the construction of models for test

cases an error prone process if done manually. The environment needs to provide the right tools to aid in building such models.

- **Heterogeneity of transformation languages and techniques** As discussed earlier in the chapter, there exist numerous tools and techniques for model transformation, which are divided into different categories. This means that black-box approaches to testing transformations can be better candidate since they can apply to all categories. However, a combination with white box techniques would provide better quality and testing coverage. White box techniques have a drawback of being coupled to the transformation language and in most cases need to be completely redefined for another transformation language.

2.4.9 Discussion

Finally we would like to set the focus of the work presented in the coming chapter by referring to Figure 2.6. It shows one axis the SUT (what are we testing: code, transformation or a model) and the other axis as the basis of the method used to achieve the testing (either using code, or using models.) The content is based on the review we presented in the current and previous chapter, to highlight major achievements, challenges and to highlight what is still needed to be addressed.

Section A: Testing code using code is the most studied subject. It refers to techniques mentioned in Chapter 1 like: Black-Box testing through domain partitioning, and white-box techniques to generate test cases. In the same area we notice the automation of execution through testing drivers and XUnit frameworks.

Section B: When using models to test the coded system, techniques surfaced such as, domain specific testing and model-based testing to generate test cases, and in some scenarios oracle functions.

Section C: On the other hand, when we are testing models and their artifacts we could use code to validate the syntax of the models, and check consistency. In fact we also could test models using simulators in some cases.

Section D: We also could use models to test models. For example, we could use transformations which are modelled and apply them on the SUT model to expose certain properties and hence validate it.

Section E: Finally, our focus is testing model transformations. As we discussed in this chapter, most research has focused on addressing functional testing and its test case generation. In fact, we believe it has produced solid findings such as meta-model coverage techniques, effective meta models, and using mutation techniques to enhance the test efficacy. Although, answers to the oracle function problem for model transformation have matured such as using patterns and model fragments, there is still problems with model comparison which need to be further studied. Finally, automating the execution of a test suite, is still needed to support an effective testing process. We believe it needs to further be combined with model comparison techniques and other oracle measures. Such a framework should be flexible and allow for further analysis such as extra-functional requirements and semantic equivalence. We present our proposed solutions to these challenges in the next chapters as follows:

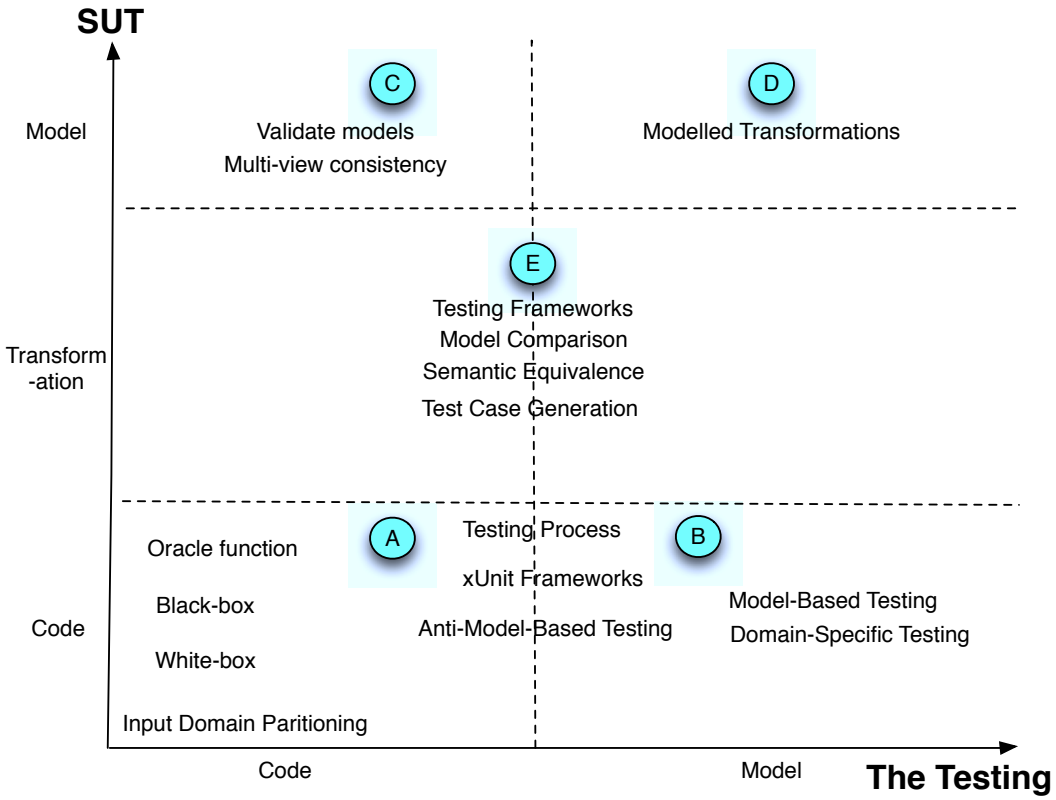


Figure 2.6: Testing Research Matrix

- Model comparison to help solve the oracle problem in chapter 3.
- Automating the execution of the test suite within a framework in chapter 4.
- Enabling semantic equivalence in chapter 4.

3

Chapter 3: Model Comparison aka Model Differencing

In this chapter we address an important challenge of model transformation testing which is model comparison. We start by providing a brief introduction of related work on model comparison and differencing in Section 3.1. We then introduce the problem of Maximum Common Sub-graph (MCS) matching as a possible approach for model comparison in Section 3.2. We discuss the most relevant algorithms to solve MCS matching and subsequently focus on one such backtracking-based algorithm, described in Section 3.3, which we then customize to work optimally in our context as demonstrated by performance experiments. A characterization of the main requirements for model comparison in the context of testing is then presented in Section 3.6. In the last section we describe the performance experiments we run to evaluate the algorithm performance in Section 3.5, and proceed into specifying our enhancements on the search heuristics with their corresponding their impact on performance in Section 3.7. Finally the main advantages and disadvantages of our approach are discussed in Section 3.10.

3.1 Background

Model comparison is an essential element in the MDE world. The ability to perform model comparison and model differencing plays a major role in areas like model evolution and version control, and most importantly in the context of testing [LZG04].

Any testing framework for model transformation requires an implementation of a model comparison technique. The models being compared have to conform to the same meta model.

An example of the difference between model M_1 and model M_2 is described in $M1 - M2$ in Figure 3.1.

Models have abstract syntax (Abstract Syntax Graph), concrete syntax (an XML representation) and semantics as discussed in Chapter 2. Model differencing is applied generally to the models' abstract syntax, and the difference is described according to it.

However in some cases models can be transformed to a concrete syntax representation and then compared. For example, each model can be exported to an equivalent XML rep-

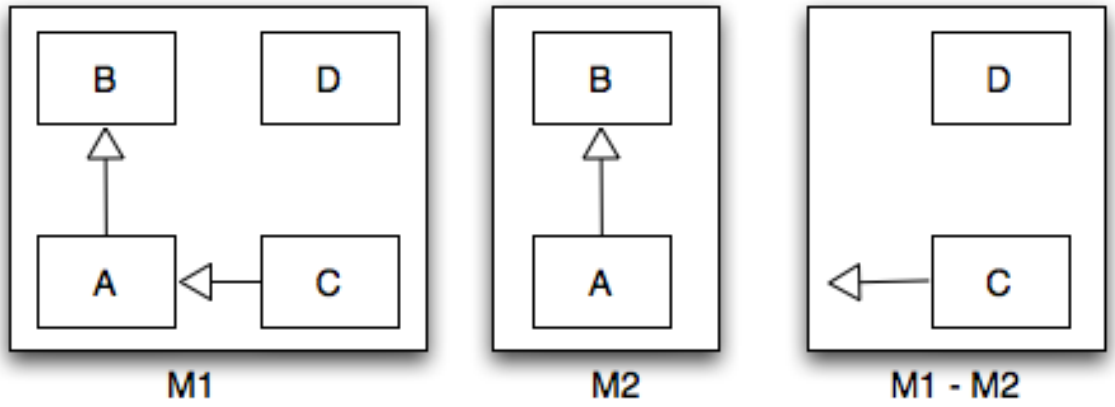


Figure 3.1: An example of model differencing

resentation, and then both models can be compared, either using GNU diff [1] or XMLdiff [WDyC03]. When we compare two XML files using straight GNU diff, we are comparing the concrete syntax representation of the XML files. on the other hand, when comparing two XML files using XMLDiff for example, the files are parsed into their Abstract Syntax Trees, which are then compared.

However, comparison at an inadequate level (like on the XMI file which represent the source code of the mode) produces many conceptually irrelevant deviations. Thus the comparison must be performed on the basis of a conceptual representation (abstract or concrete syntax), at the correct level of abstraction.

In the context of testing, model comparison is used to compare the resulting transformed model to the specified expected model. Even if the transformation implementation has few defects we would still expect the models being compared to be of similar sizes.

Comparing two models involves several activities namely:

1. Identifying matching elements (mapping set between elements of both models);
2. Calculating and representing the difference (usually in the form of an edit script);
3. Visualize those differences appropriately if necessary;

In this chapter we focus on addressing the first two steps: *identifying matching elements* and *calculating the difference*.

Before calculating the difference between two models, a criterion should be defined for matching model elements. For example, if we consider two text files as models, the criterion would be that a full byte by byte diff is run between the model files (Lexical Differencing). Another criterion for comparing models would be to compare only certain properties of the files, like the name, elements and some structural similarities. Once the criterion is defined, then several algorithms can be used to find the actual matching and to calculate the difference.

1. http://www.gnu.org/software/diffutils/

To illustrate this see figure 3.2, which contains two models M_1 and M_2. Both models are represented as graphs which can contain white or black nodes, and connecting edges. If node colour is the criterion used in matching the nodes, then we notice that node 3 in M_2 maps node 1 in M_1. It's visually clear that the difference of M_2 to M_1 is an addition of a black node with a connecting edge to the white node, however given only the previous matching criterion, we notice that node 2 in M_1 can be mapped to either node 4 or node 5 in M_2. Both solution would be correct.

Even a more extreme case would be the following: assume nodes have a state attribute which contains an integer, and that:
node 2 in M_1 has value 5, node 4 in M_2 has value 2, node 5 in M_2 has value 7. Then several solutions exist:

- *first solution* is that node 2 in M1 was deleted, and two new nodes were added (node 4 and 5 in M2).
- *second solution* is that node 2 in M1 has changed its state value from 5 to 7, and that a new node was added (node 4 in M2).
- *third solution* is that node 2 in M1 has changed its state value from 5 to 2, and that a new node was added (node 5 in M2).

Models can be represented using attributed typed graphs on all meta levels. It is not always possible to reduce the representation of such models into trees and run tree comparison algorithms to calculate differences. For example, models can have cyclic dependencies, and depending on the approaches used to generate them, could end up with a different tree representations. Hence, If we treat all models as graphs, then the model comparison problem becomes in turn the graph matching problem. However the graph matching problem can be reduced to the graph isomorphism problem which is NP-Complete as discussed in [KR96]. To overcome this problem of complexity, many techniques have been proposed. Most are tool or language specific, which make use of the semantics. Few techniques claim to be generic and meta model independent. A recent survey of such approaches can be found in [KRPP09] which we will describe next.

When comparing complex models represented as graphs, with many relationships this can become a very complex procedure. This problem is solved easily if there exist a unique identifier which can be used to map nodes, as explained later in this section.

Once the difference has been calculated, it has be to represented using a proper difference model. This step is crucial for analysis and visualization of the differences. However, the difference model tends to be affected by the calculating method. Some of the existing techniques include edit scripts [AP03]. Edit scripts are ordered sets of atomic operations (Create, Delete, Update/Change) which can be applied on the starting model to obtain the target model. They represent low-level implementation which can be optimized to become efficient. However, in most cases, this difference model depends on the presence of a persistent unique identifier attached to model elements for it to be applicable.

Finally, the differences between the two models should be visualized in a meaningful way to the user.

In the context of model evolution, detecting differences between consecutive versions

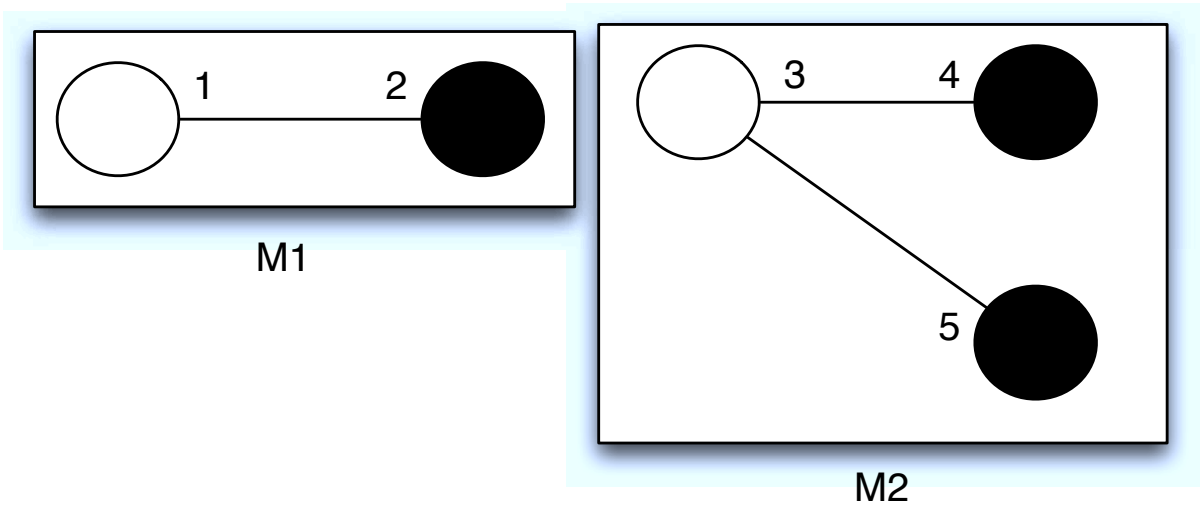


Figure 3.2: Two models being compared

could be solved easily, if the same modelling tool or environment is used to edit or transform the model. If the environment supports traceability on the lowest level, then it is possible to record the trace of the mutating CUD (Create, Update and Delete) operations used to conclude the new version of the model. However, this is not the case in most situations, where models are evolved separately, in difference environments where no unique identifiers are available.

In [KRPP09], the authors propose a categorization for the main existing approaches to model differencing as summarized below:

- **Static Identity Based Matching:** Assumes that model elements always have a unique persistent identifier which is used for matching. This approach has high performance. It however does not work when models were constructed in different environments or tools. [AP03] presents a meta-model independent algorithm to calculate model difference based on uniques identifiers and can be applied in any MOF-based modelling language. And hence it is language specific.
- **Signature Based Matching:** This approach is meant to overcome the assumptions of having a static identity for each model element by the previous approach. It is similar to the last approach in that that elements are compared according to some identity (static or dynamically generated signature) and that is the matching criterion. The identity in this case is calculated based on a user defined function for generating the identity of each model element. It is usually made up of a collection of element properties. The main disadvantage here is that users will have to provide these functions for each of the model types.

  The work in [RFG+05] describes using a signature based, rather than name based, for matching model elements. The name based criterion can lead to conflicts, like matching two classes with the same name but representing different meta concepts or having different properties. Their approach allows for a generic way for

specifying signatures for model elements to be matched. For example, the signature of a class can be its name, meta type as well as the signatures of its operations. The signature is then used to match model elements. This is related to having a canonical form representation of any model in a specific formalism, however in this case the canonical form is restricted to the model elements, whereas in canonical representation means the order of the elements in the representation will be the same.

- **Language specific:** Matching algorithms in this category are tailored towards a specific language or category. For example, UMLDiff [XS05] can compare UML models. Such algorithms can benefit from the domain knowledge to perform more efficiently. However, the full matching algorithm needs to be specified for the given language.
- **Similarity Based Matching:** This approach treats models as attributed typed graphs which allows matching elements based on their feature similarities. It's related to signature based matching, however, depending on the meta model, different features might have different importance in calculating similarity. This is usually provided by the user to indicate the weight of each model feature.

  SiDiff [TBWK07] is a tool which can compare two models based on their similarity model. Authors claim that SiDiff is easily configurable to work almost any model with a graph structure. The tool first transform both models being compared into an internal representation (a directed, typed graph with fixed set of runtime object types to make the system independent of the original model type). The difference is then calculated on the new representation.

  SiDiff uses a set of compare functions to determine the similarity between two nodes (e.g. compare two attributed values, or sets of neighbouring nodes). Each attribute have a defined weight in an external file. The overall similarity of the two elements is calculated as the weighted mean of the similarity relevant properties. If the similarity total number exceeds a specific threshold, both elements are considered a match. SiDiff was demonstrated on UML class diagrams, however the authors claim it is language independent. The disadvantage is that the user will have to define weights for each formalism.

  Another tool, **DSMDiff** [LGJ07], attempts calculating model difference for any domain specific model whose meta model is defined using GME (General Modelling Environment). The authors claim their approach is applicable to any meta modelling tool which represent models as hierarchical graphs. Meta-models in GME consist of a set of *atoms*, *models*, and *connections*. They represent the type of any element of an instance model conforming to this unique meta-model.

  The matching algorithm that is used by DSMDiff does not attempt to find the most optimal solution. It instead employs a greedy strategy to select a match for a node n from a list of candidate nodes x,y,z by simply comparing structural similarities around both nodes for one level. To illustrate see Figure 3.3, when attempting to map node **N** from M1, three candidate nodes with the same property, having white labels, exist **X,Y,Z**. The algorithm, ranks the three candidates according to their

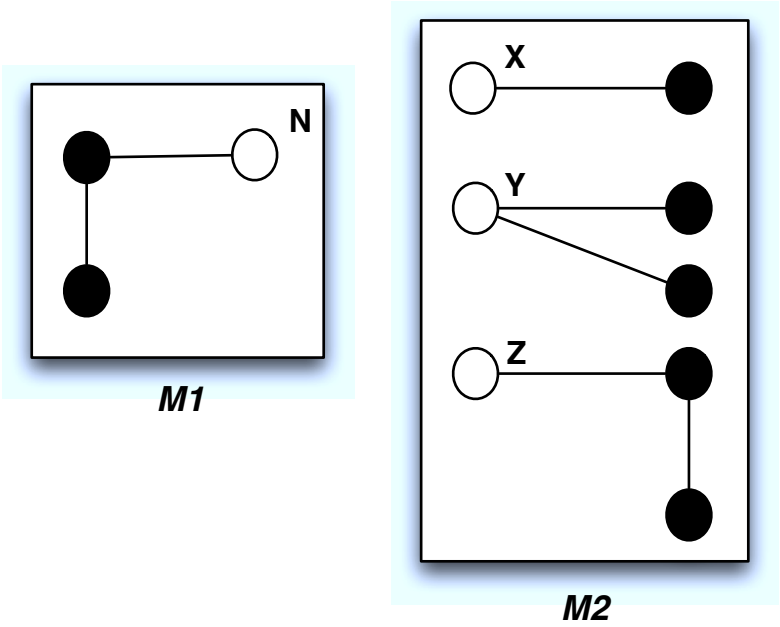


Figure 3.3: Structural similarities in model comparison

edge similarities for node **N**, which has one edge pointing to a black node. The only candidates in M2 which have the same edge similarity are: **X**, and **Z**. The algorithm would pick any of them at this stage, say **X**, as the match of **N**, even though it is clear that the optimal mapping would be **Z**. The reason is that the edge similarity relies only on one level search. This greedy approach is a possible solution to the otherwise NP-complete problem, but we believe it is not enough, as it could produce incorrect results on simple cases as such illustrated in figure 3.3.

In other words, can not guarantee that the solution it finds is most optimal.

Other Approaches like [ACP07] provides a meta model level solution to graph comparison problem. It is however not clear how efficient and usable this approach is yet.

We show how our approach gives the user more control over the accuracy of the result through different customizations, and works without sacrificing accuracy best fit the context of model transformation testing.

3.2 Maximum Common Subgraph Isomorphism

The problem of Maximum Common sub-graph Isomorphism can be defined as follows: Common sub-graphs H1 = [V, E, k] and H2 = [W, F, k] of two given graphs G1, and G2, are those of equal size k, that are isomorphic to each other. Assume we have a function μ to determine if two nodes are equivalent using some criteria (such as label or type). And the function ν checks if two edges are comparable (edge exists or not in both).

This means that there should be such numeration of sub-graphs' vertices x(i) and y(i) where :

$$\mu(v_{x(i)}, w_{y(i)}) = true \ \& \ \nu(g1_{x(i),x(j)}, g2_{y(i),y(j)}) = true \ \forall i, j \in 1...k \tag{3.1}$$

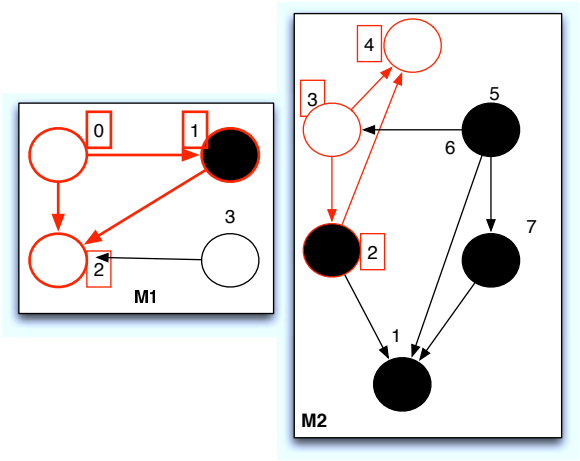


Figure 3.4: A simple example of MCS between two models (highlighted in red)

The above formula implies that for each pair of matching vertices in the sub-graphs, they are connected with matching edges to all other edges in each of the subgraph nodes. For example, the maximum common sub-graph of M1 and M2 is highlighted in Figure 3.4.

Graphs are commonly used to represent structured objects, and models. In applications where graphs need to be compared for similarity, the MCS (Maximum Common Subgraph) problem is used to measure similarity. MCS is used to measure if two graphs share identical parts. It has several essential applications such as: Pattern Recognition, Chemical Reactions and Information handling. [RW02] conducts a review of the many MCS algorithms and makes recommendations regarding their applicability to typical chemo-informatics tasks.

In what follows, we build on the main foundations and recent results achieved in solving the MCS problem to obtain a powerful model comparison algorithm which can be used in the context of testing. The relationship between model comparison and detecting common isomorphic sub-graphs of models is clear. They both rely on detecting a mapping between pairs of nodes, one from each graph, in such a way that the structure (represented by edges between nodes) is similar.

We start by describing the existing approaches to solve the MCS problem. There exist two main categories of algorithms: *exact* and *approximate*.

Approximate algorithms, rely on using heuristics in order to reach a solution within acceptable time constraints. However, this class of algorithms can not provide guarantees to how close the solution is to the true MCS. Nonetheless, such algorithms can prove usable in certain domains where the size of models is upper bounded such as comparing chemical structures [RW02].

One the other hand, exact algorithms attempt to find the optimal solution for the MCS problem. The MCS problem, however, is NP-complete [KR96], and thus such algorithms have a worse-case, exponential-time complexity in most cases. Exact algorithms include two main approaches: backtracking and clique based.

Backtracking based algorithms in general are based on the concept of systematically attempting all possible solutions to guarantee the best solution. In the case of MCS, the algorithm builds a search tree, where at each level it attempts the different possible enumerations between unmapped nodes. To avoid attempting all enumeration (NP-complete), backtracking algorithms increase the performance through the pruning of certain paths in the search tree unless they provide a more optimal solution. In addition to evaluating the isomorphic property described in Equation 3.1, to exclude any violating mapping. One of the original backtrack based algorithms to address MCS was suggested by McGregor in [McG82].

On the other hand, clique based algorithm attempt to reduce the mapping problem to finding the maximum clique in an association graph problem. It can then can use existing clique finding algorithms to reach the result.

For a discussion of the performance of different MCS exact algorithms including clique based and backtracking algorithms, we refer to [DCV07]. The evaluation showed that the run time complexity of these algorithms is similar. However, depending on the type of graphs being compared, some approaches proved more efficient. For example, for very large graphs, creating the association graph, in the clique based approaches, tends to require significantly more memory.

For the purpose of our work, we started from a particular backtracking solution described by Krissenel and Henrick in [KH04] called **CSIA**. The algorithm was derived using results from the ESI (Exact Subgraph Isomorphism) problem, namely the Ullman (UA) ([Ull76]), which implies the backtracking approach may be efficient for MCS at least in cases that are close to ESI (i.e when the two graphs being compared are almost identical and the difference is not significant). We claim that this is common in model transformation testing where the focus is on comparing models with small differences in most cases. The authors showed that the proposed CSIA algorithm is considerably more efficient than existing MCS solutions in this context.

3.3 CSIA Algorithm

We adopted the backtracking algorithm, CSIA, proposed by Krissenel and Henrick in [KH04].

The CSIA algorithm introduces a complexity controlling parameter n_0 representing the minimum size of common sub-graph to be matched (the size is measured by the number of vertices) to be found. In other words if all the common subgraphs of size which exist are of size smaller than n_0, CSIA wont detect any.

In most applications, only sufficiently large common sub-graphs are considered as a useful result of graph matching. For example, pattern recognition tasks normally require MCSs to be found to yield a match, however if the MCS found is insignificant in size, the recognition will return with a no match. Using this parameter, CSIA will reject branches of the recursion tree not leading to acceptable results, from the application point of view, without spending time on finding them. In the context of testing such a parameter can help indicate the accuracy threshold of the solution to be found as will be discussed this

in the next section.

The following describes the CSIA algorithm. We start by setting up two sets X,Y to empty. These sets are used to store an ordered list of mapped vertices from models $m1$, and $m2$ respectively. Also, we set the parameter n_{max} to 0 until we update it when we find solutions. We then call initialize and subsequently start the backtrack search.

Algorithm 1 CSIA : Global

```
1: call Initialize(D)
2: X ⇐ ∅, Y ⇐ ∅
3: n_max = 0
4: n_0 = n_0 //User Provided
5: call Backtrack(D)
```

The procedure **Initialize** examines each pair of nodes from $m1$, $m2$ respectively by executing the function Equivalent(v_i, w_j) which can be customized by the user to incorporate any equivalence criteria (types, labels, attributes) between nodes. The algorithm builds a dictionary of the nodes v_i from $m1$ to their potential candidates in $m2$, alongside the length of the candidate list. The length of the lists is stored in a vector L_i. This mapping dictionary is called **PM** (Possible Mappings).

Algorithm 2 CSIA : procedure **Initialize**[M,L])

```
1: for all v_i ∈ V do
2:     k = 0
3:     for all w_j ∈ W do
4:         if Equivalent(v_i, w_j) then
5:             k++;
6:             M[v_i].append(w_j);
7:         end if
8:     end for
9:     L_i = k;
10: end for
```

The main work and recursion of the overall algorithm takes place in the **Backtrack** procedure. The algorithm first checks if the current search is **Extendable** (we will describe this later), and if it is not, the algorithm considers the MCS it has found so far and stores it in memory if is a new maximum, it also updates n_{max}.

If the current search is extendable, the algorithm will then pick an unmapped vertex from $m1$ through **PickVertex** procedure, and get its possible candidate mappings from $m2$ through procedure **GetMappableVertices**, and try each mapping by one by one. In each iteration the back tracking function call **Refine** procedure to check the isomorphic properties of the unmapped nodes and their candidates and hence reduce the search. The backtracking then restore the solutions and attempts a different candidate.

The procedure **PickVertex** simply picks the unmapped node v_i from $m1$ with the least number of candidate nodes from $m2$. Mapping the most restricted nodes first leads to

Algorithm 3 CSIA : procedure **Backtrack**(PM D)

```
1: if Extendible(D) then
2:    v_i := PickVertex(D)
3:    Candidates:= GetMappableVertices(v_i,D)
4:    for all w_j ∈ Candidates do
5:       X = X + {v_i}
6:       Y = Y + {w_j}
7:       D' = Refine(D)
8:       call Backtrack(D')
9:       X = X − {v_i}
10:      Y = Y − {w_j}
11:   end for
12:   V = V − {v_i}
13:   call Backtrack(D);
14:   V = V + {v_i}
15: else
16:   n_max = max(n_max, | X |)
17:   Print(X, Y)
18: end if
```

less calls to backtrack.

Algorithm 4 CSIA:procedure **PickVertex**(PM D=[M,L])

1: **return** v_i such that $0 < L_i \leq L_k$ for any $v_k \in V$

The procedure **GetMappableVertices** simply returns the list of candidates in $m2$ for a vertex v_i in $m1$.

Algorithm 5 CSIA:procedure **GetMappableVertices**(vertex v_i, VMM D=[M,L])

1: **return** $M[v_i]$ from i to L_i

The procedure **Refine** iterates over the current potential mappings dictionary, to check the isomorphic property of each entry against the current mapping solution. In other words, it filters out the mappings (the candidate list for nodes v_i from $m1$) and excludes them if any violate the isomorphic property with the current in memory mapping solution. It is the **Refine** function which helps reduce the search tree dramatically.

The function Isomorphic($e_{i,x(q)}, f_{j,y(q)}$) takes two nodes v_i, w_j and the current solution set X, Y. It then evaluate if mapping v_i, w_j is allowed under the isomorphic property when added to the current solution X, Y. Note that e, f define the edges in $m1, m2$ respectively.

Finally the procedure **Extenable** examines the current mapping dictionary, and determines how many more vertices (in the best case) could be mapped from such iteration. It then compare that to the size of the MCS found so far, and the minimum size allowed n_0

Algorithm 6 CSIA:procedure **Refine**(PM D=[M,L])

```
1: PM D_1 = [T,N]
2: q := —X—
3: for all v_i ∈ V − X do
4:    l = 0;
5:    for all w_j ∈ M[i] do
6:       if Isomorphic(e_{i,x(q)}, f_{j,y(q)}) then
7:          l++;
8:          T[v_i].append(w_j);
9:       end if
10:   end for
11:   N_i = l;
12: end for
13: return D_1
```

to determine whether to allow the search to continue along this branch or to terminate it.

Algorithm 7 CSIA:procedure **Extendable**(PM D=[M,L])

```
1: q := s := —X—;
2: for all v_i ∈ V − X do
3:    if L_i > 0 then
4:       s++;
5:    end if
6: end for
7: if s ⩾ max(n_0, n_max) and s > q then
8:    return true;
9: else
10:   return false;
11: end if
```

3.4 CSIA Complexity Analysis

We implemented the original CSIA algorithm using the Python language. We then identified and added several enhancements which we describe later in this chapter. These enhancements are specific to model transformation testing.

For the CSIA algorithm, the complexity reduces to that of Ullman algorithm when comparing two models and choosing $n_0 = min(size(m_1), size(m_2))$.

The best case of the algorithm run time is when nodes are uniquely labelled, or in other words, the equivalence criterion uses a global unique identifier. The initialization would set that each node has one candidate, and takes O(nm) comparisons. Then there would be O(n) *Backtrack* calls, and for each call there would be a call to *Refine* which in turn is O(n) giving a best case of O($nn + mn$) = O(mn).

The worst case, *Backtrack* is called O(m) times on each of the O(n) recursion levels giving O(m^n) and for each backtrack there is a call to *Refine* which takes O(mn) comparisons, and hence in total O($mn * m^n + mn$) = O($m^{n+1}n$)

O(mn) $\leq$ CSIA $\leq$ O($m^{n+1}n$) [KH04].

This indicates that the CSIA algorithm is optimal for comparing models of different sizes, and choosing n_0 to be of the size close to the smallest of the two model sizes.

3.5 Performance Analysis

We have implemented the CSIA algorithm in Python in a version which was also customized to handle $AToM_3$ [dLV02] models and be able to compare them. To validate the implementation, we built a small test suite of functional test cases. We ran the algorithm to detect the MCS between the models of each test case. Due to the well described model comparison problem we will not further discuss the functional testing of the algorithm, but rather focus on its performance.

The following step was to measure the performance of the CSIA algorithm under different input graphs, and to get a feeling for its scalability. To achieve this we generated a series of models with increasing sizes.

Due to the exploratory approach we followed in determining heuristics to increase the algorithm performance, we will show different types of experiments.

First type of experiments we attempted uses models made up of blocks, where each block contains nodes with a different density of edges. This experiment does not reflect the complete picture of the effects of increasing edge density of the model since the edges are restricted to individual blocks. It was however necessary to guide the enhancements process. We later scale up this experiment by fixing number of nodes and increasing the edge densities of the models being compared. This will allow us to demonstrate the increase in performance induced by our added heuristics.

Composite Blocks Experiment

Three types of graphs were created (see figure 3.5):

1. "No links": Graphs made up of n composites each made of four nodes with no links.
2. "Two links": Graphs made up of n composites each made of four nodes with two links (50 percent edges).
3. "Three links": Graphs made up of n composites each made of four nodes with three links(75 percent edges).

For each model category ("no-links", "two-links", or "three-links") different instances were created, by varying n from (1 to 30) where each instance is a composed of n blocks of the corresponding composite described above. There were no links between the different blocks.

For example, under the "no link" category, instance where $n = 10$ will have 10 composite blocks each with 4 nodes, and no links (so 40 nodes, 0 edges in total). Instance $n = 20$

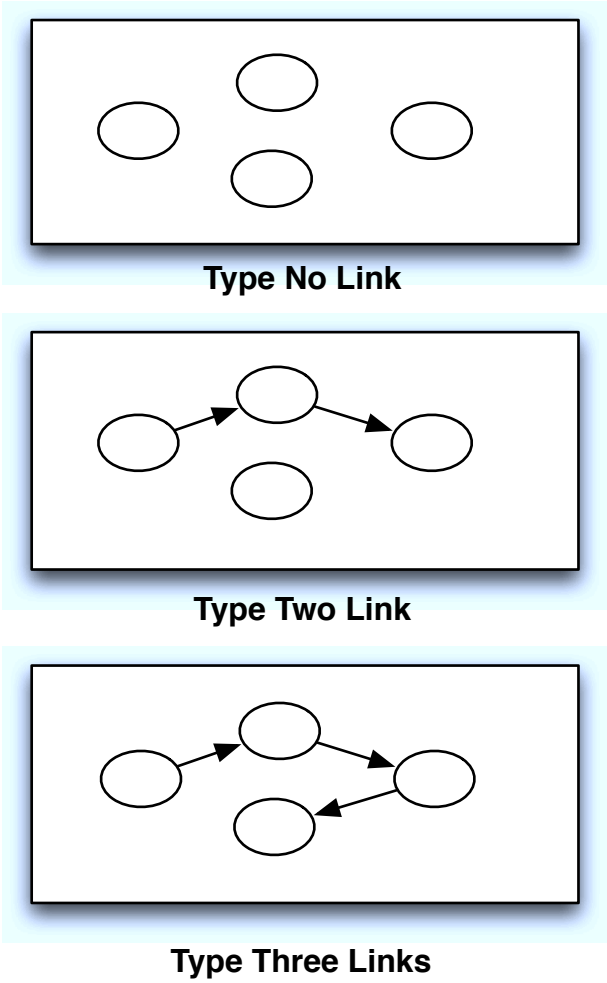


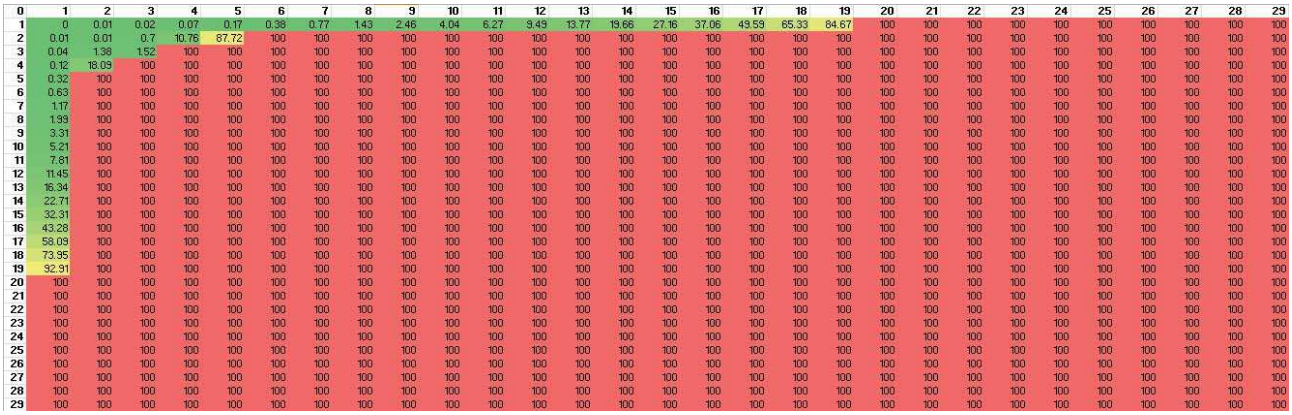
Figure 3.5: The building composite blocks for each type of created test model.

will have 20 composite blocks each with 4 nodes, and no links (so 80 nodes in total) and so on.

In each category, the algorithm was run to compare instances in the same category with different sizes of n, for example comparing "no-link" instance $n = 5$ with instance $n = 10$. The time to complete the comparison was then logged and compared, and the following results was obtained.

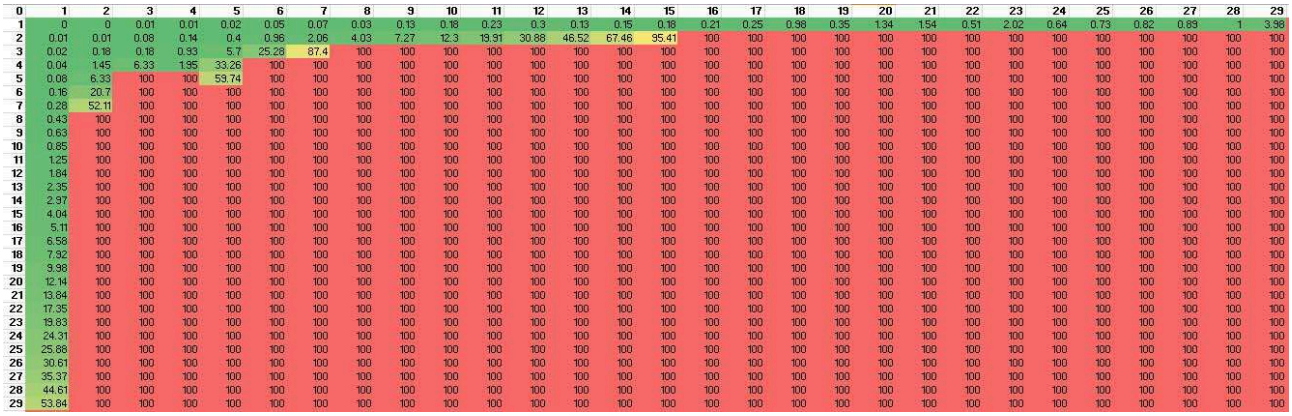
Fixing N_0 to 0, we attempted to run the algorithm on different cases of "no-link" type, in some cases the running time exceeds 100 seconds, at which point we stop the process and set the running time to a 100 seconds. The graph in Figure 3.6 was obtained, with the following observations: When size of g_1 is (n=1, i.e 4 nodes), the algorithm can produce results within the time constraint (100 seconds) up until the size of g_2 reaches (n=29, i.e 116 nodes).

Note the figures are organized as follows: each cell (i,j) contains the time the algorithm in question took to find the MCS between the graphs $g_1(n = i)$ and $g_2(n = j)$. Depending on the time out set, the algorithm wont run beyond such time. The figures are heat



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0.01	0.02	0.07	0.17	0.38	0.77	1.43	2.46	4.04	6.27	9.43	13.77	19.66	27.16	37.06	49.59	65.33	84.67	100	100	100	100	100	100	100	100	100	100
2	0.01	0.01	0.7	10.76	87.72	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
3	0.04	1.38	1.52	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
4	0.12	18.09	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
5	0.32	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
6	0.63	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
7	1.17	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
8	1.99	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
9	3.31	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
10	5.21	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
11	7.81	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
12	11.45	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
13	16.34	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
14	22.71	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
15	32.31	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
16	43.28	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
17	58.09	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
18	73.95	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
19	92.91	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
20	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
21	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
23	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
24	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
25	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
26	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.6: Original algorithm with input type "no-link"



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0.01	0.01	0.02	0.05	0.07	0.03	0.13	0.18	0.23	0.3	0.13	0.15	0.18	0.21	0.25	0.98	0.35	1.34	1.54	0.51	2.02	0.64	0.73	0.82	0.89	1	3.98
2	0.01	0.01	0.08	0.14	0.4	0.96	2.06	4.03	7.27	12.3	19.91	30.88	46.52	67.46	95.41	100	100	100	100	100	100	100	100	100	100	100	100	100	100
3	0.02	0.18	0.18	0.93	5.7	25.28	87.4	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
4	0.04	1.45	6.33	1.35	33.26	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
5	0.08	6.33	100	100	59.74	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
6	0.16	20.7	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
7	0.28	52.11	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
8	0.43	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
9	0.63	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
10	0.85	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
11	1.25	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
12	1.84	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
13	2.35	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
14	2.97	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
15	4.04	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
16	5.11	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
17	6.58	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
18	7.92	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
19	9.98	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
20	12.14	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
21	13.84	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
22	17.35	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
23	19.83	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
24	24.31	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
25	25.88	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
26	30.61	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	35.37	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	44.61	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	53.84	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.7: Original algorithm with input of type "Two-link"

mapped so that they can be better viewed, with red representing a running time of longer than 100 seconds.

Then we attempted to run the "two-links" instances, the results are shown in Figure 3.7. Note how that in this type of models the algorithm performs a little better than the previous type. And even better for the "three-links" models as seen in figure 3.8.

The explanation for this increased performance when models have more structure (represented by higher edge density) is due to the search and pruning mechanism of the CSIA. The fact that the resulting MCS has to satisfy equation 3.1 or the isomorphism property, implies that with more structure more nodes would be eliminated during the search, and hence more pruning and less permutations needs to be tried.

Best Case Experiment

We also created a benchmark for our enhancements, we describe an experiment representing the **best case** running time for the algorithm. Using the same approach to building models as described before, we tweak the equivalence criterion to use a unique identifier for nodes instead of only using the type. The figures show the scalability of both algorithms (original and enhanced) in terms of input models sizes. We do this only

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0	0.01	0.01	0	0.01	0.01	0.02	0.02	0.03	0.02	0.03	0.04	0.03	0.04	0.05	0.05	0.06	0.07	0.07	0.07	0.08	0.08	0.1	0.11	0.09	0.12	0.13
2	0	0.01	0.02	0.04	0.08	0.12	0.2	0.3	0.36	0.48	0.75	0.96	1.22	1.51	1.85	2.25	2.7	2.55	2.93	4.35	5.04	5.78	6.52	7.42	8.38	9.51	10.52	11.84	10.34
3	0.02	0.1	0.05	0.16	0.39	0.86	1.52	2.36	4.31	7.61	11.19	16.06	22.47	28.19	40.39	52.72	54.84	66.76	100	100	100	100	100	100	100	100	100	100	100
4	0.03	0.49	2.72	0.36	1.33	3.8	9.49	19.55	40.14	72.42	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
5	0.05	1.87	27.14	62.15	2.73	11.32	38.12	99.91	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
6	0.09	5.33	100	100	100	22.48	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
7	0.15	11.3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
8	0.22	21.97	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
9	0.29	40.39	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
10	0.41	65.83	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
11	0.54	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
12	0.74	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
13	0.94	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
14	1.22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
15	1.44	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
16	1.74	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
17	2.22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
18	2.71	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
19	2.99	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
20	3.72	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
21	4.53	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
22	5.03	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
23	6.02	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
24	6.87	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
25	8	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
26	8.85	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	9.63	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	11.41	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	13.33	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.8: Original algorithm with input type "three-links"

	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250
10	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.19	0.23	0.19	0.18	0.19	0.19	0.2	0.19	0.19	0.19	0.19	0.18	0.19
20	0.28	1.26	1.28	1.26	1.25	1.27	1.26	1.26	1.28	1.28	1.29	1.26	1.24	1.26	1.25	1.28	1.26	1.27	1.26	1.28	1.28	1.27	1.29	1.29	1.25
30	0.44	1.53	4.03	4	4.03	3.94	4.09	3.97	4.01	4.01	4.05	4.03	4.01	4.05	4.04	4.07	3.98	4	4.03	4.06	4.03	3.99	3.98	3.94	4.09
40	0.69	1.68	4.37	9.27	9.26	9.15	9.19	9.13	9.08	9.2	9.21	9.26	9.26	9.2	9.16	9.22	9.2	9.23	9.16	9.13	9.23	9.15	9.34	9.13	9.11
50	1.04	2.02	4.58	9.62	17.69	17.56	17.67	17.65	17.65	17.71	17.64	17.53	17.52	17.49	17.66	17.54	17.68	17.58	17.53	17.55	17.42	17.65	17.74	17.63	17.75
60	1.49	2.54	5.02	10	18.04	29.81	31.27	31.23	31.98	31.22	31.06	31.01	30.9	32.04	32.15	31.26	30.79	31.77	30.99	30.55	31.14	31.24	31.21	30.7	31.92
70	2.15	3.1	5.69	10.86	19.29	31.9	48.92	49.05	50.39	50.11	48.71	49.55	49	48.12	48.81	48.77	50.05	48.33	48.15	48.94	48.95	49.15	49.26	50.23	49.6
80	2.73	3.66	6.42	11.71	20.55	32.26	51.29	73.95	73.68	74.94	74.05	73.45	75.47	73.79	71.57	70.46	73.13	74.34	73.45	71.39	71.28	71.56	71.28	73.09	73.38
90	3.67	4.64	7.25	12.34	20.95	32.88	50.13	74.21	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	98.27
100	4.38	5.29	7.8	12.83	20.98	33.2	49.13	71.48	98.12	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
110	5.41	6.29	8.82	13.78	21.75	33.8	50.34	72.22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
120	6.49	7.31	9.81	14.84	22.8	34.96	51.24	73.5	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
130	7.97	8.79	11.21	16.25	25.5	38.08	55.27	79.45	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
140	9.64	10.52	13.12	18.32	26.67	38.43	56.94	80.28	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
150	11.23	11.81	14.43	19.43	28.32	40.51	58.38	82.34	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
160	12.48	13.66	15.84	21.04	29.36	41.09	58.8	84.56	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
170	14.77	15.46	18.03	22.42	31.4	44.08	61.18	84.41	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
180	15.71	16.43	18.81	23.67	31.66	43.6	59.46	82.07	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
190	17.67	18.28	20.92	25.44	33.79	45.82	61.73	83.6	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
200	20.47	21.78	24.15	29.19	36.55	49.72	67.35	91.44	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
210	21.94	22.51	25.17	29.6	37.7	49.36	65.87	88.22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
220	25.21	25.75	27.99	32.81	40.98	53.41	72.7	95.36	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
230	26.85	27.37	29.5	34.7	42.13	54.03	70.71	92.45	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
240	29.42	29.9	32.1	36.88	44.74	56.76	73.86	94.89	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
250	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.9: Original algorithm with input type "three-link" using a unique identifier

for the models of type "three-links", on the original CSIA algorithm in Figure 3.9. Note that the algorithm can scale up to a much larger input size graphs (up to graphs of n=250 reflecting a model of size 1000 nodes). Our graphs scale has changed from 1 to 10 for each cell to show more data for this experiment. Another note is that our enhancements do not have direct impact on performance as this is the best case experiment where each node of $m1$ has a candidate list of size 1 nodes from $m2$, and hence, no room for the pruning to help.

Edge Density Scale

The final method of evaluation is based on examining the effects of changing the number of edges on the performance. To do this we build model instances of a fixed number of nodes, and for each instance n detail the number of edges we add to the model (in random fashion). This is important to demonstrate the effects of the greedy enhancements we describe in the following section. For example for instance n of an edge based model with m nodes, it will have 3*n edges. We do this for models of size 20 nodes, and 40 nodes. Note that adding edges randomly produces models representing worst case for the algorithm. We will at the end of the enhancements section run an experiment with

	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250
10	0.16	0.16	0.16	0.15	0.15	0.15	0.16	0.15	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.16	0.15	0.16	0.16	0.16
20	0.15	1.03	1.03	1.04	1.03	1.04	1.04	1.03	1.03	1.04	1.04	1.04	1.04	1.04	1.05	1.04	1.04	1.04	1.05	1.04	1.05	1.04	1.04	1.04	1.04
30	0.16	1.04	3.34	3.25	3.26	3.26	3.26	3.27	3.25	3.26	3.26	3.26	3.26	3.26	3.27	3.26	3.27	3.27	3.27	3.27	3.27	3.27	3.28	3.27	3.27
40	0.16	1.04	3.27	7.51	7.45	7.44	7.43	7.44	7.48	7.44	7.43	7.43	7.44	7.45	7.45	7.45	7.44	7.44	7.44	7.44	7.44	7.44	7.45	7.44	7.51
50	0.15	1.04	3.27	7.44	14.17	14.22	14.16	14.17	14.17	14.22	14.17	14.17	14.24	14.19	14.17	14.23	14.2	14.21	14.22	14.18	14.17	14.24	14.18	14.19	14.26
60	0.16	1.03	3.26	7.44	14.16	24.13	24.09	24.07	24.06	24.07	24.13	24.09	24.35	24.55	24.53	24.82	24.64	24.92	24.58	24.73	24.92	24.53	24.69	24.66	24.57
70	0.15	1.06	3.33	7.47	14.58	24.69	38.78	38.59	38.3	37.97	39.02	38.57	39	38.75	38.56	38.62	38.65	38.52	39.22	39.08	38.55	38.83	39.16	39.71	38.76
80	0.16	1.07	3.34	7.61	14.46	24.46	38.69	57.66	57.91	57.35	56.67	57.36	57.51	57.45	58.04	57.07	56.76	57.71	57.79	58.31	57.44	57.87	57.44	58.48	57.3
90	0.16	1.07	3.43	7.65	14.62	24.53	38.58	57.93	80.98	81.69	82.68	80.99	82.81	81.32	81.18	81.98	80.33	81.9	81.55	83.04	81.61	82.17	82.06	80.42	80.35
100	0.16	1.06	3.31	7.52	14.66	24.75	39.11	57.4	80.77	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
110	0.15	1.04	3.34	7.46	14.17	24.03	37.7	55.82	78.68	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
120	0.16	1.04	3.27	7.44	14.16	24.17	37.77	55.82	78.81	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
130	0.16	1.06	3.36	7.63	14.41	25.05	38.89	57.64	81.21	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
140	0.16	1.04	3.29	7.45	14.71	24.71	38.95	57.92	80.57	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
150	0.16	1.07	3.34	7.64	14.46	24.38	38.32	56.83	80.84	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
160	0.17	1.08	3.31	7.46	14.52	24.93	39.04	57.93	83.77	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
170	0.15	1.04	3.28	7.45	14.18	24.21	37.79	55.93	78.88	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
180	0.16	1.05	3.28	7.61	14.59	24.65	38.68	58.69	82.07	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
190	0.16	1.04	3.27	7.44	14.38	24.14	37.93	55.75	78.88	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
200	0.16	1.05	3.27	7.46	14.22	24.09	37.74	55.97	78.97	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
210	0.16	1.05	3.27	7.45	14.37	24.12	37.93	55.73	79.05	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
220	0.16	1.07	3.35	7.87	14.46	24.64	38.64	57.72	80.02	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
230	0.17	1.08	3.37	7.91	14.44	24.63	39.13	58.26	83.85	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
240	0.16	1.08	3.36	7.68	14.64	25.05	38.93	57.49	80.3	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
250	0.17	1.05	3.28	7.45	14.18	24.11	38	55.76	78.93	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.10: Order algorithm with input "three-links" using a unique identifier

fixed number of nodes and fixed number of edges while increasing the edge density to show how much our final version of the algorithm can scale.

3.6 Specifics of Model Comparison for testing

The authors in [KPP06] proposed a rule based model comparison technique, to enable model transformation testing. It however does not attempt to do a complete model matching using graphs. Also in their testing framework, the authors in [LZG05] assumes a unique identifier exists for comparing models.

The comparison would result in a mapping set (describing which elements of $m1$ map to which elements of $m2$). The work in [KPP06] provides a detailed classification of the mapping and difference sets. The mapping set is equivalent to calculating the maximum common sub-graph. This set in turn is used to generate an edit script to describe the difference between both models. So the size of the edit script is proportional to the difference between the two models being compared, and hence when the models are the same, the edit script is empty.

In the context of testing, the edit script is used to help in debugging activities to help localize bugs in the transformation. We argue that minimizing the edit script becomes less useful beyond a certain threshold. For example consider when comparing two model of size 30 nodes each, then there would be no point in optimizing an edit script of more than 20 edit operations to say 15 operations since the models are very different anyway. This note could be indicative of the usefulness of a controlling parameter for the size of the MCS similar to the one proposed in the CSIA algorithm mentioned before. So in essence we are dealing with a problem close to the ESI (exact (sub)graph isomorphism).

There is an apparent consensus in existing approaches to model comparison (generic or domain/language specific) on the importance of adding domain specific information as part of the node matching criteria. The simplest example is using a static unified identifier to determine node matching. Other approaches use a dynamic function to generate node signatures as a matching criterion. We feel that this should be an important functionality of any model comparison algorithm. Specifically, the algorithm should give

flexibility to the modeller to decide on a matching criteria as needed per domain. It can be a simple type comparison or a complex formula involving each of the nodes attributes. We integrate this flexibility into our algorithm and show in the performance section how using different criteria impact performance.

The algorithm should be optimized to detect exact matches with confidence, as this the most common case especially when running regression testing suites. If however the models are not exact matches (test case fails), then the edit script need not to be most optimal but to achieve a reasonable solution close to the most optimal. At the very least, we feel there should be a mechanism with which the modeller can control how much time can the algorithm spend on searching for the best solution. We discuss our approach on this in the enhancement section.

Finally generating the edit script from the mapping will be discussed in 3.9

To summarize, we identify the following requirements for the algorithm:

- Allows Customizable matching criteria and run time.
- Optimized to compare models of similar size
- Consistent i.e behaves exactly the same if run several times on the same input.

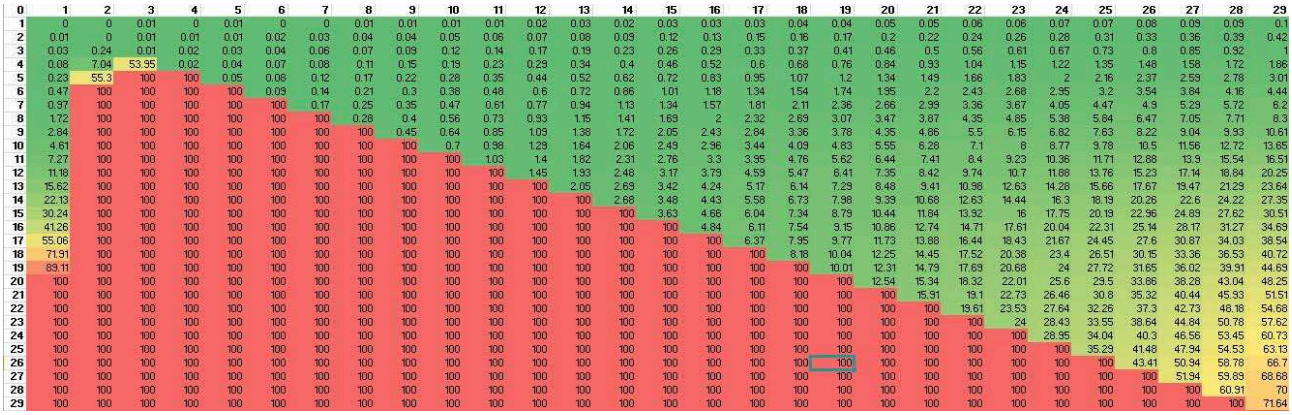
3.7 Performance Enhancements

To enhance the performance of the algorithm to become a good candidate for model transformation testing, we propose to implement and evaluate several modifications to the CSIA algorithm.

There are three possible dimensions (algorithmic enhancements, data structures implementation features and implementation language) to apply enhancements to the performance of the above mentioned algorithm. First, several algorithmic enhancement to take advantage of the mechanics of the algorithm, we will mention such enhancements in this section. Second, the data structures implementations axis, namely how the algorithm is implemented to take advantage of the target language features (for example in Python testing membership in a set is much faster than testing membership is a list). Finally, the language of implementation is another large factor of the overall performance. Switching our implementations from Python to C should increase the performance by about a factor 10. For the sake of this work, and rapid prototyping we will only use Python and describe the algorithmic enhancement we applied to the original algorithm and how they translate in relative performance.

Time-out parameter

Since the nature of the algorithm is to, while it is searching for the best solution (MCS), keep track of the best found solution so far, we are able to introduce a time out control parameter to stop execution when needed. The parameter is only applied as a cut-off for execution once a solution is found and would otherwise be nothing more than a timer with an interrupt from the calling function. The parameter is optional such that if it is not specified, the algorithm will continue executing until it finds the best solution. We have set a time-out on our algorithm as part of the performance analysis we enclose in



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0	0.01	0	0.01	0	0	0.01	0.01	0.01	0.01	0.02	0.03	0.02	0.03	0.03	0.03	0.04	0.04	0.05	0.05	0.06	0.06	0.07	0.07	0.08	0.09	0.09	0.1
2	0.01	0	0.01	0.01	0.01	0.02	0.03	0.04	0.04	0.05	0.06	0.07	0.08	0.09	0.12	0.13	0.15	0.16	0.17	0.2	0.22	0.24	0.26	0.28	0.31	0.33	0.36	0.39	0.42	
3	0.03	0.24	0.01	0.02	0.03	0.04	0.06	0.07	0.09	0.12	0.14	0.17	0.19	0.23	0.26	0.29	0.33	0.37	0.41	0.46	0.5	0.56	0.61	0.67	0.73	0.8	0.85	0.92	1	
4	0.08	7.04	53.35	0.02	0.04	0.07	0.08	0.11	0.15	0.19	0.23	0.29	0.34	0.4	0.46	0.52	0.6	0.68	0.76	0.84	0.93	1.04	1.15	1.22	1.35	1.48	1.58	1.72	1.86	
5	0.23	55.3	100	100	0.05	0.08	0.12	0.17	0.22	0.28	0.35	0.44	0.52	0.62	0.72	0.83	0.95	1.07	1.2	1.34	1.49	1.66	1.83	2	2.16	2.37	2.59	2.78	3.01	
6	0.47	100	100	100	100	0.03	0.14	0.21	0.3	0.38	0.48	0.6	0.72	0.86	1.01	1.18	1.34	1.54	1.74	1.95	2.2	2.43	2.68	2.95	3.2	3.54	3.84	4.16	4.44	
7	0.97	100	100	100	100	100	0.17	0.25	0.35	0.47	0.61	0.77	0.94	1.13	1.34	1.57	1.81	2.11	2.36	2.66	2.99	3.36	3.67	4.05	4.47	4.9	5.29	5.72	6.2	
8	1.72	100	100	100	100	100	100	0.28	0.4	0.56	0.73	0.93	1.15	1.41	1.69	2	2.32	2.69	3.07	3.47	3.87	4.35	4.85	5.38	5.84	6.47	7.05	7.71	8.3	
9	2.84	100	100	100	100	100	100	100	0.45	0.64	0.85	1.09	1.38	1.72	2.05	2.43	2.84	3.36	3.78	4.35	4.86	5.5	6.15	6.82	7.63	8.22	9.04	9.93	10.61	
10	4.61	100	100	100	100	100	100	100	100	0.7	0.98	1.29	1.64	2.06	2.49	2.96	3.44	4.09	4.83	5.55	6.28	7.1	8	8.77	9.78	10.5	11.56	12.72	13.65	
11	7.27	100	100	100	100	100	100	100	100	100	1.03	1.4	1.82	2.31	2.76	3.3	3.95	4.76	5.62	6.44	7.41	8.4	9.23	10.36	11.71	12.88	13.9	15.54	16.51	
12	11.18	100	100	100	100	100	100	100	100	100	100	1.45	1.93	2.48	3.17	3.79	4.59	5.47	6.41	7.35	8.42	9.74	10.7	11.88	13.76	15.23	17.14	18.84	20.25	
13	15.62	100	100	100	100	100	100	100	100	100	100	100	2.05	2.69	3.42	4.24	5.17	6.14	7.29	8.48	9.41	10.98	12.63	14.28	15.66	17.67	19.47	21.29	23.64	
14	22.13	100	100	100	100	100	100	100	100	100	100	100	100	2.68	3.48	4.43	5.58	6.73	7.98	9.39	10.68	12.63	14.44	16.3	18.19	20.26	22.6	24.22	27.35	
15	30.24	100	100	100	100	100	100	100	100	100	100	100	100	100	3.63	4.66	6.04	7.34	8.79	10.44	11.84	13.92	16	17.75	20.19	22.96	24.89	27.62	30.51	
16	41.26	100	100	100	100	100	100	100	100	100	100	100	100	100	100	4.84	6.11	7.54	9.15	10.86	12.74	14.71	17.61	20.04	22.31	25.14	28.17	31.27	34.69	
17	55.06	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	6.37	7.95	9.77	11.73	13.88	16.44	18.43	21.67	24.45	27.6	30.87	34.03	38.54	
18	71.91	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	8.18	10.04	12.25	14.45	17.52	20.38	23.4	26.51	30.15	33.36	36.53	40.72	
19	89.11	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	10.01	12.31	14.79	17.69	20.68	24	27.72	31.65	36.02	39.91	44.69
20	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	12.54	15.34	18.32	22.01	25.6	29.5	33.86	38.28	43.04	48.25
21	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	15.91	19.1	22.73	26.46	30.8	35.32	40.44	45.93	51.51
22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	19.61	23.53	27.64	32.26	37.3	42.73	48.18	54.68
23	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	24	28.43	33.55	38.64	44.84	50.78	57.62
24	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	28.95	34.04	40.3	46.56	53.45	60.73
25	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	35.29	41.48	47.94	54.53	63.13
26	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	43.41	50.94	58.78	66.7
27	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	51.94	59.89	68.68	
28	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	60.91	70	
29	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	71.64

Figure 3.11: Single-MCS enhanced algorithm with input type “no-links”

this work.

Only one MCS is required

The original algorithm was intended to detect and print all maximum common sub-graphs of the graphs being compared. This is relevant if we expect to find more than one common sub-graph, of size $k < min(m, n)$, which could be essential in certain applications. In the context of testing, and according to our discussion in the previous section this is not really needed.

For this purpose we modify the algorithm to only find the single largest (or any if multiple exist) common sub-graph. Specifically in the function **Extendible** we modify the pruning condition related to the N_{max} condition. Line 7 of **Extendible** becomes : $s > max(n_0 - 1, n_{max})$ and $s > q$

To demonstrate the effectiveness of this change we ran the new algorithm on the three types of models and monitored the performance, see figure 3.11, 3.12, and 3.13, for “no-links”, “two-links” and “three-links” type models respectively.

Note the increased performance over the original algorithm, as the algorithm can scale up and process larger size model instances within the time-out criteria. However we can notice that performance is linked to the size of the first input, which inspired us for the next enhancement.

Order of comparison

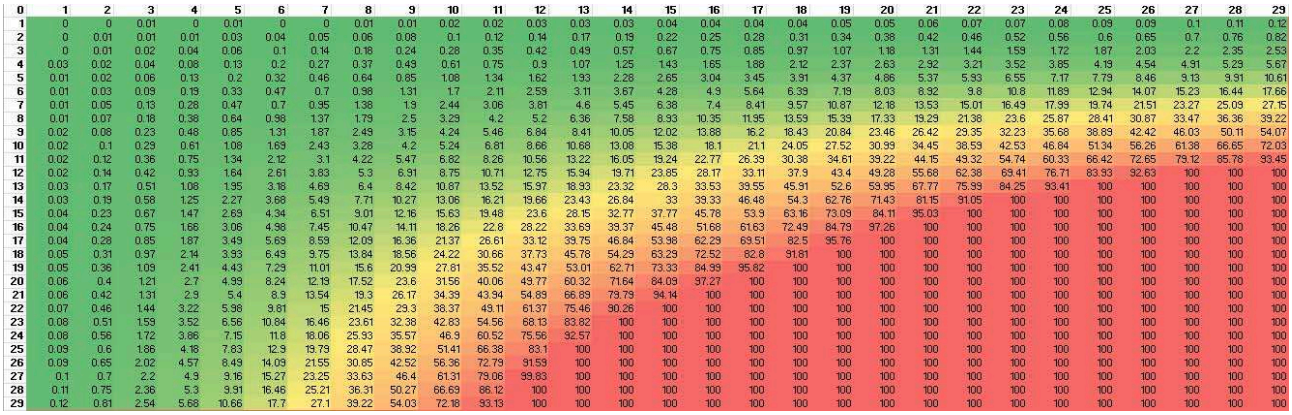
The second enhancement refers to the order of comparison performed on the input graphs. We noted that the algorithm search tree depends on the size of the first input graph, as shown in the previous results. To exploit this dependency, we added an extra step in the initialization to account for the size of the input models and perform the comparison starting with the smaller size input as the first model and mapped it to second model. To demonstrate the speed up the CSIA algorithm was modified to account for the size of the input graphs, and order them appropriately. We ran the new order algorithm on three types of models to measure its performance in Figure 3.14,3.15 and 3.16. Note how the algorithm performance exhibits a more symmetrical graph with respect to inputs

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0	0	0	0	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.03	0.03	0.04	0.05	0.05	0.05	0.05	0.05	0.06	0.06	0.08	0.08
2	0	0.01	0.01	0.01	0.02	0.03	0.03	0.04	0.05	0.07	0.08	0.09	0.11	0.13	0.15	0.17	0.18	0.2	0.17	0.2	0.29	0.3	0.34	0.26	0.39	0.34	0.45	0.48	0.55
3	0.01	0.06	0.02	0.03	0.05	0.05	0.09	0.12	0.15	0.13	0.16	0.26	0.31	0.35	0.42	0.32	0.56	0.62	0.66	0.74	0.81	0.9	0.98	1.15	1.16	1.26	1.37	1.51	1.66
4	0.04	0.72	6.7	0.03	0.09	0.07	0.18	0.23	0.3	0.25	0.46	0.56	0.67	0.76	0.89	1.05	1.14	1.29	0.91	1.64	1.12	1.29	2.18	2.42	1.65	1.84	3.01	3.34	3.58
5	0.07	3.44	42.75	47.2	0.13	0.2	0.29	0.22	0.29	0.65	0.8	0.98	1.18	1.38	1.58	1.82	2.14	2.33	2.64	2.97	1.97	3.61	4	4.37	4.7	5.18	5.64	6.2	4.06
6	0.14	11.49	100	100	100	0.28	0.2	0.29	0.76	0.54	0.69	0.86	1.84	2.17	2.52	2.35	3.35	3.78	4.34	4.74	5.39	5.9	6.62	7.12	7.76	5.04	3.27	10.03	10.85
7	0.22	34.87	100	100	100	100	0.23	0.79	0.51	0.68	1.75	1.1	2.63	3.11	3.68	4.31	4.32	3.17	3.5	7.1	4.47	8.98	9.88	10.63	11.63	12.85	14.09	8.57	9.45
8	0.36	82.21	100	100	100	100	100	1.04	1.41	1.87	2.39	2.93	3.6	4.29	2.61	5.37	6.87	7.79	8.99	10.08	6.09	12.64	13.95	15.35	16.96	3.91	19.31	21.71	23.67
9	0.54	100	100	100	100	100	100	100	1.74	2.37	1.3	3.84	2.14	5.7	6.72	7.88	9.2	10.63	11.96	13.46	15.19	16.88	18.86	11.01	23.07	25.01	14.71	15.75	16.97
10	0.8	100	100	100	100	100	100	100	100	1.03	1.46	4.76	5.95	3.17	8.71	4.72	11.88	13.87	15.66	18.1	20.16	22.6	24.61	27.38	30.38	33.54	36.34	39.15	22.51
11	1.14	100	100	100	100	100	100	100	100	100	4.46	5.76	7.17	3.7	4.54	12.52	6.66	8	9.23	22.11	25.52	28.5	31.46	34.64	19.11	42.24	23.54	50.02	55.36
12	1.56	100	100	100	100	100	100	100	100	100	100	2.37	8.61	10.62	12.72	15.34	17.66	20.96	23.94	12.69	30.93	34.83	39.1	43.24	47.75	52.1	28.36	62.32	33.41
13	2.06	100	100	100	100	100	100	100	100	100	100	100	3.38	12.41	15.05	17.97	21.24	24.76	28.55	32.86	37.31	41.78	47.15	24.25	57.2	62.99	70.06	75.95	82.47
14	2.74	100	100	100	100	100	100	100	100	100	100	100	100	14.24	6.03	20.57	9.57	29.13	33.64	38.47	44.26	21.73	55.75	62.21	69.29	75.66	39.1	91.4	99.77
15	3.66	100	100	100	100	100	100	100	100	100	100	100	100	100	19.76	23.66	28.26	12.66	38.55	17.82	50.58	57.87	28.11	31.33	35.45	89.38	98.33	100	54
16	4.46	100	100	100	100	100	100	100	100	100	100	100	100	100	100	26.58	31.95	37.66	44.11	19.15	23.16	66.71	74.88	83.38	39.87	100	100	100	100
17	6	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	35.13	14.04	17.02	56.56	65.06	74.87	84.07	94.47	100	100	54.52	100	68.06
18	7.02	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	46.41	54.75	63.36	73.05	83.49	94.74	100	100	53.94	100	67.37	74.83
19	8.59	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	53.69	70.15	80.73	92.44	100	100	51.2	100	100	100	100
20	10.37	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	24	83.17	34.41	40.97	100	100	100	100	80.11	89.88
21	12.96	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	97.24	100	100	100	58.17	100	100	100	100
22	15.56	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	51.94	60.97	100	100	100	100
23	17.87	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	97.35	100
24	21.57	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
25	25.74	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	80.97	94.11	100	100
26	28.5	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	32.88	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	40.44	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	45.18	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.12: Single-MCS enhanced algorithm with input type “one-links”

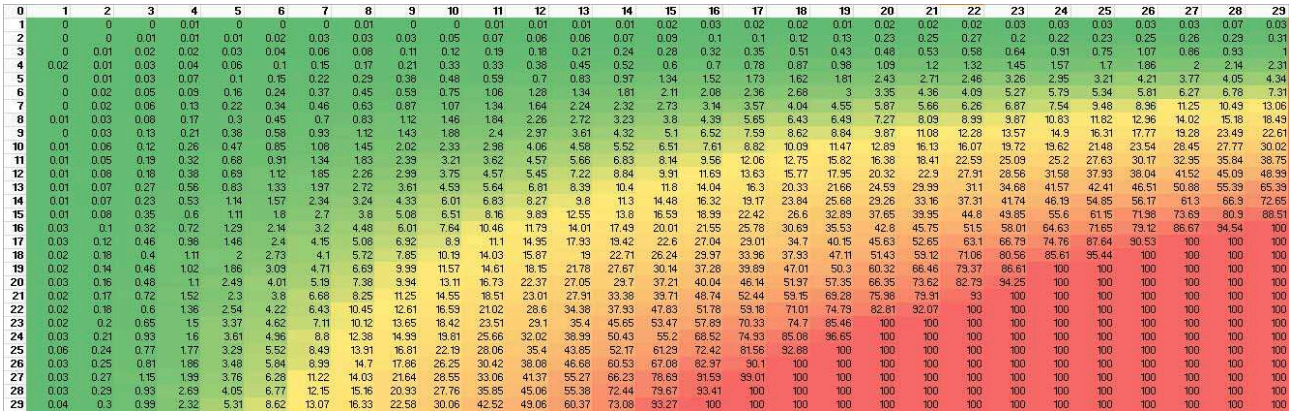
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0.01	0	0	0	0	0	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.01	0.02	0.01	0.02	0.03	0.03	0.03	0.03	0.03	0.03	0.04	0.04	0.04	0.05	0.05
2	0	0.01	0	0.01	0.02	0.02	0.02	0.03	0.04	0.05	0.05	0.06	0.07	0.08	0.09	0.11	0.11	0.13	0.15	0.14	0.18	0.18	0.22	0.24	0.25	0.27	0.3	0.32	0.31
3	0.01	0.03	0.01	0.02	0.03	0.05	0.06	0.07	0.09	0.12	0.13	0.17	0.19	0.21	0.25	0.28	0.32	0.39	0.39	0.45	0.48	0.58	0.61	0.63	0.74	0.73	0.84	0.85	0.94
4	0.02	0.33	1.21	0.04	0.06	0.08	0.12	0.15	0.19	0.24	0.28	0.34	0.41	0.43	0.55	0.64	0.71	0.78	0.89	0.98	1.07	1.15	1.26	1.37	1.54	1.65	1.74	1.92	2
5	0.05	1.35	17.05	24.32	0.08	0.14	0.18	0.26	0.32	0.39	0.5	0.6	0.69	0.83	0.95	1.09	1.26	1.36	1.59	1.74	1.98	2.04	2.25	2.53	2.8	3.01	3.18	3.5	3.72
6	0.08	3.82	73.11	100	100	0.18	0.26	0.35	0.47	0.53	0.73	0.93	1.1	1.28	1.49	1.72	1.91	2.17	2.47	2.8	3.09	3.3	3.63	4.05	4.49	4.9	5.24	5.49	6.01
7	0.12	8.14	100	100	100	100	0.35	0.49	0.65	0.84	1.05	1.28	1.57	1.8	2.15	2.51	2.83	3.19	3.65	4.14	4.49	5.09	5.48	5.91	6.61	7.24	7.72	8.27	9.24
8	0.19	17.75	100	100	100	100	100	0.62	0.85	1.09	1.39	1.72	2.09	2.46	2.91	3.41	3.83	4.38	5.06	5.58	6.3	7	7.64	8.38	9.33	10.18	11.13	11.73	12.76
9	0.29	33.06	100	100	100	100	100	100	1.03	1.37	1.79	2.17	2.7	3.28	3.75	4.48	5.18	5.79	6.77	7.51	8.58	9.38	10.49	11.43	12.49	13.75	14.73	16.14	17.47
10	0.4	57.37	100	100	100	100	100	100	100	1.67	2.21	2.73	3.42	4.06	4.98	5.77	6.58	7.73	8.89	10.05	11.13	12.36	13.55	15.43	16.84	18.14	20.02	21.41	23.34
11	0.52	98.29	100	100	100	100	100	100	100	100	2.64	3.29	4.16	4.99	6.04	7	8.21	9.68	10.9	12.36	14.1	15.53	17.35	18.72	20.84	23.23	24.67	27.32	29.84
12	0.67	100	100	100	100	100	100	100	100	100	100	3.92	4.83	5.97	7.08	8.24	9.71	11.51	13.3	14.86	16.84	18.95	21.08	23.15	25.93	28.25	31.22	33.41	36.87
13	0.94	100	100	100	100	100	100	100	100	100	100	100	5.48	6.74	8.31	9.65	11.41	13.73	15.66	17.72	20.05	22.71	25.64	28.49	31.87	34.53	37.64	40.84	44.07
14	1.14	100	100	100	100	100	100	100	100	100	100	100	100	7.71	9.45	11.26	13.55	15.91	18.22	20.63	23.59	26.71	30.17	33.51	37.48	40.37	44.9	49.11	53.21
15	1.38	100	100	100	100	100	100	100	100	100	100	100	100	100	10.86	13.07	15.54	18.18	21.12	23.82	27.27	31	35.39	38.95	43.44	47.69	52.87	57.02	63.62
16	1.69	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	14.61	17.62	20.42	23.88	27.67	31.24	35.8	39.76	44.96	49.23	55.09	60.58	66.62	73.01
17	2.11	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	19.56	22.52	26.9	31.17	35.33	39.93	45.08	51.07	56.49	62.93	69.81	75.74	84.06
18	2.47	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	24.78	28.97	33.44	38.81	43.85	49.74	55.94	62.58	69.67	77.01	84.6	92.67
19	2.94	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	31.68	37.04	42.32	49.18	55.73	61.98	70.07	77.16	86.65	95.16	100
20	3.34	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	40.48	46.76	53.52	60.96	68.72	76.78	85.35	95.06	100	100
21	4.05	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	51.29	58.03	66.33	75.03	85.02	94.42	100	100	100
22	4.65	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	63.69	72.52	82.12	92.84	100	100	100	100
23	5.35	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	78.76	88.27	99.88	100	100	100	100
24	5.93	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	96.08	100	100	100	100	100
25	6.83	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
26	7.91	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
27	8.89	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
28	10.22	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
29	11.32	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100

Figure 3.13: Single-MCS enhanced algorithm with input type “two-links”



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0.01	0	0.01	0	0	0.01	0.01	0.02	0.02	0.03	0.03	0.03	0.04	0.04	0.04	0.04	0.05	0.05	0.06	0.07	0.07	0.08	0.09	0.09	0.1	0.11	0.12	
2	0	0.01	0.01	0.01	0.03	0.04	0.05	0.06	0.08	0.1	0.12	0.14	0.17	0.19	0.22	0.25	0.28	0.31	0.34	0.38	0.42	0.46	0.52	0.56	0.6	0.65	0.7	0.76	0.82	
3	0	0.01	0.02	0.04	0.06	0.1	0.14	0.18	0.24	0.28	0.35	0.42	0.49	0.57	0.67	0.75	0.85	0.97	1.07	1.18	1.31	1.44	1.59	1.72	1.87	2.03	2.2	2.35	2.53	
4	0.03	0.02	0.04	0.08	0.13	0.2	0.27	0.37	0.49	0.61	0.75	0.9	1.07	1.25	1.43	1.65	1.88	2.12	2.37	2.63	2.92	3.21	3.52	3.85	4.19	4.54	4.91	5.29	5.67	
5	0.01	0.02	0.06	0.13	0.2	0.32	0.46	0.64	0.85	1.08	1.34	1.62	1.93	2.28	2.65	3.04	3.45	3.91	4.37	4.86	5.37	5.93	6.55	7.17	7.79	8.46	9.13	9.91	10.61	
6	0.01	0.03	0.09	0.19	0.33	0.47	0.7	0.98	1.31	1.7	2.11	2.59	3.11	3.67	4.28	4.9	5.64	6.39	7.19	8.03	8.92	9.8	10.8	11.89	12.94	14.07	15.23	16.44	17.66	
7	0.01	0.05	0.13	0.28	0.47	0.7	0.95	1.38	1.9	2.44	3.06	3.81	4.6	5.45	6.38	7.4	8.41	9.57	10.87	12.18	13.53	15.01	16.49	17.99	19.74	21.51	23.27	25.09	27.15	
8	0.01	0.07	0.18	0.38	0.64	0.98	1.37	1.79	2.5	3.29	4.2	5.2	6.36	7.58	8.93	10.35	11.95	13.59	15.39	17.33	19.29	21.38	23.6	25.87	28.41	30.87	33.47	36.36	39.22	
9	0.02	0.08	0.23	0.48	0.85	1.31	1.87	2.49	3.15	4.24	5.46	6.84	8.41	10.05	12.02	13.88	16.2	18.43	20.84	23.46	26.42	29.35	32.23	35.68	38.89	42.42	46.03	50.11	54.07	
10	0.02	0.1	0.29	0.61	1.08	1.69	2.43	3.28	4.2	5.24	6.81	8.66	10.68	13.08	15.38	18.1	21.1	24.05	27.52	30.99	34.45	38.59	42.53	46.84	51.34	56.26	61.38	66.65	72.03	
11	0.02	0.12	0.36	0.75	1.34	2.12	3.1	4.22	5.47	6.82	8.26	10.56	13.22	16.05	19.24	22.77	26.39	30.38	34.61	39.22	44.15	49.32	54.74	60.33	66.42	72.65	79.12	85.78	93.45	
12	0.02	0.14	0.42	0.93	1.64	2.61	3.83	5.3	6.91	8.75	10.71	12.75	15.94	19.71	23.85	28.17	33.11	37.9	43.4	49.28	55.68	62.38	69.41	76.71	83.93	92.63	100	100	100	
13	0.03	0.17	0.51	1.08	1.95	3.18	4.69	6.4	8.42	10.87	13.52	15.97	18.93	23.32	28.3	33.53	39.55	45.91	52.6	59.95	67.77	75.99	84.25	93.41	100	100	100	100	100	
14	0.03	0.19	0.58	1.25	2.27	3.68	5.49	7.71	10.27	13.06	16.21	19.66	23.43	26.84	33	39.33	46.48	54.3	62.76	71.43	81.15	91.05	100	100	100	100	100	100	100	
15	0.04	0.23	0.67	1.47	2.69	4.34	6.51	9.01	12.16	15.63	19.48	23.6	28.15	32.77	37.77	45.78	53.9	63.16	73.09	84.11	95.03	100	100	100	100	100	100	100	100	
16	0.04	0.24	0.75	1.66	3.06	4.98	7.45	10.47	14.11	18.26	22.8	28.22	33.69	39.37	45.48	51.68	61.63	72.49	84.79	97.26	100	100	100	100	100	100	100	100	100	
17	0.04	0.28	0.85	1.87	3.49	5.69	8.59	12.09	16.36	21.37	26.61	33.12	39.75	46.84	53.98	62.29	69.51	82.5	95.76	100	100	100	100	100	100	100	100	100	100	
18	0.05	0.31	0.97	2.14	3.93	6.49	9.75	13.84	18.56	24.22	30.66	37.73	45.78	54.29	63.29	72.52	82.8	91.81	100	100	100	100	100	100	100	100	100	100	100	
19	0.05	0.36	1.09	2.41	4.43	7.29	11.01	15.6	20.99	27.81	35.52	43.47	53.01	62.71	73.33	84.99	95.82	100	100	100	100	100	100	100	100	100	100	100	100	
20	0.06	0.4	1.21	2.7	4.99	8.24	12.19	17.52	23.6	31.56	40.06	49.77	60.32	71.64	84.09	97.27	100	100	100	100	100	100	100	100	100	100	100	100	100	
21	0.06	0.42	1.31	2.9	5.4	8.9	13.54	19.3	26.17	34.39	43.94	54.89	66.89	79.79	94.14	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
22	0.07	0.46	1.44	3.22	5.98	9.81	15	21.45	29.3	38.37	49.11	61.37	75.46	90.26	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
23	0.08	0.51	1.59	3.52	6.56	10.84	16.46	23.61	32.38	42.83	54.56	68.13	83.82	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
24	0.08	0.56	1.72	3.86	7.15	11.8	18.06	25.93	35.57	46.9	60.52	75.56	92.57	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
25	0.09	0.6	1.86	4.18	7.83	12.9	19.79	28.47	38.92	51.41	66.38	83.1	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
26	0.09	0.65	2.02	4.57	8.49	14.09	21.55	30.85	42.52	56.36	72.79	91.59	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
27	0.1	0.7	2.2	4.9	9.16	15.27	23.25	33.63	46.4	61.31	79.06	99.83	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
28	0.11	0.75	2.36	5.3	9.91	16.46	25.21	36.31	50.27	66.69	86.12	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	
29	0.12	0.81	2.54	5.68	10.66	17.7	27.1	39.22	54.03	72.18	93.13	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	

Figure 3.14: Order algorithm with input of type “no-link”



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0	0.01	0	0	0	0	0.01	0	0	0.01	0.01	0.01	0.01	0.02	0.03	0.02	0.02	0.01	0.02	0.02	0.02	0.03	0.03	0.03	0.03	0.07	0.03	
2	0	0	0.01	0.01	0.01	0.02	0.03	0.03	0.03	0.05	0.07	0.06	0.06	0.07	0.09	0.1	0.1	0.12	0.13	0.23	0.25	0.27	0.2	0.22	0.23	0.25	0.26	0.29	0.31	
3	0	0.01	0.02	0.02	0.03	0.04	0.06	0.08	0.11	0.12	0.19	0.18	0.21	0.24	0.28	0.32	0.35	0.51	0.43	0.48	0.53	0.58	0.64	0.91	0.75	1.07	0.86	0.93	1	
4	0.02	0.01	0.03	0.04	0.06	0.1	0.15	0.17	0.21	0.33	0.33	0.38	0.45	0.52	0.6	0.7	0.78	0.87	0.98	1.09	1.2	1.32	1.45	1.57	1.7	1.86	2	2.14	2.31	
5	0	0.01	0.03	0.07	0.1	0.15	0.22	0.29	0.38	0.48	0.59	0.7	0.83	0.97	1.34	1.52	1.73	1.62	1.81	2.43	2.71	2.46	3.26	2.95	3.21	4.21	3.77	4.05	4.34	
6	0	0.02	0.05	0.09	0.16	0.24	0.37	0.45	0.59	0.75	1.06	1.28	1.34	1.81	2.11	2.08	2.36	2.68	3	3.35	4.36	4.09	5.27	5.79	5.34	5.81	6.27	6.78	7.31	
7	0	0.02	0.06	0.13	0.22	0.34	0.46	0.63	0.87	1.07	1.34	1.64	2.24	2.32	2.73	3.14	3.57	4.04	4.55	5.87	5.66	6.26	6.87	7.54	9.48	8.96	11.25	10.49	13.06	
8	0.01	0.03	0.08	0.17	0.3	0.45	0.7	0.83	1.12	1.46	1.84	2.26	2.72	3.23	3.8	4.39	5.65	6.43	6.49	7.27	8.09	8.99	9.87	10.83	11.82	12.96	14.02	15.18	18.49	
9	0	0.03	0.13	0.21	0.38	0.58	0.93	1.12	1.43	1.88	2.4	2.97	3.61	4.32	5.1	6.52	7.59	8.62	8.84	9.87	11.08	12.28	13.57	14.9	16.31	17.77	19.28	23.49	22.61	
10	0.01	0.06	0.12	0.26	0.47	0.85	1.08	1.45	2.02	2.33	2.98	4.06	4.58	5.52	6.51	7.61	8.82	10.09	11.47	12.89	16.13	16.07	19.72	19.62	21.48	23.54	28.45	27.77	30.02	
11	0.01	0.05	0.19	0.32	0.68	0.91	1.34	1.83	2.39	3.21	3.62	4.57	5.66	6.83	8.14	9.56	12.06	12.75	15.82	16.38	18.41	22.59	25.09	25.2	27.63	30.17	32.95	35.84	38.75	
12	0.01	0.08	0.18	0.38	0.69	1.12	1.85	2.26	2.99	3.75	4.57	5.45	7.22	8.84	9.91	11.69	13.63	15.77	17.95	20.32	22.9	27.91	28.56	31.58	37.93	38.04	41.52	45.09	48.99	
13	0.01	0.07	0.27	0.56	0.83	1.33	1.97	2.72	3.61	4.53	5.64	6.81	8.39	10.4	11.8	14.04	16.3	20.33	21.66	24.59	23.99	31.1	34.68	41.57	42.41	46.51	50.88	55.39	65.39	
14	0.01	0.07	0.23	0.53	1.14	1.57	2.34	3.24	4.33	6.01	6.83	8.27	9.8	11.3	14.48	16.32	19.17	23.84	25.68	29.26	33.16	37.31	41.74	46.13	54.85	56.17	61.3	66.9	72.65	
15	0.01	0.08	0.35	0.6	1.11	1.8	2.7	3.8	5.08	6.51	8.16	9.89	12.55	13.8	16.59	16.99	22.42	26.6	32.89	37.65	39.95	44.8	49.85	55.6	61.15	71.98	73.69	80.9	88.51	
16	0.03	0.1	0.32	0.72	1.29	2.14	3.2	4.48	6.01	7.64	10.46	11.79	14.01	17.49	20.01	21.55	25.78	30.69	35.53	42.8	45.75	51.5	58.01	64.63	71.65	79.12	86.67	94.54	100	
17	0.03	0.12	0.46	0.98	1.46	2.4	4.15	5.08	6.92	8.9	11.1	14.95	17.93	19.42	22.6	27.04	29.01	34.7	40.15	45.63	52.65	63.1	66.79	74.76	87.64	90.53	100	100	100	
18	0.02	0.18	0.4	1.11	2	2.73	4.1	5.72	7.85	10.13	14.03	15.87	19	22.71	26.24	29.97	33.96	37.93	47.11	51.43	59.12	71.06	80.56	85.61	95.44	100	100	100	100	
19	0.02	0.14	0.46	1.02	1.86	3.09	4.71	6.69	9.99	11.57	14.61	18.15	21.78	27.67	30.14	37.28	39.89	47.01	50.3	60.32	66.46	79.37	86.61	100	100	100	100	100	100	
20	0.03	0.16	0.48	1.1	2.49	4.01	5.19	7.38	9.94	13.11	16.73	22.37	27.05	29.7	37.21	40.04	46.14	51.97	57.35	66.35	73.62	82.73	94.25	100	100	100	100	100	100	
21	0.02	0.17	0.72	1.52	2.3	3.8	6.68	8.25	11.25	14.55	18.51	23.01	27.91	33.38	39.71	48.74	52.44	59.15	69.28	75.98	79.91	93	100	100	100	100	100	100	100	
22	0.02	0.18	0.6	1.36	2.54	4.22	6.43	10.45	12.61	16.59	21.02	28.6	34.38	37.93	47.83	51.78	53.18	71.01	74.79	82.81	92.07	100	100	100	100	100	100	100	100	
23	0.02	0.2	0.65	1.5	3.37	4.62	7.11	10.12	13.65	18.42	23.51	29.1	35.4	45.65	53.47	57.89	70.33	74.7	85.46	100	100	100	100	100	100	100	100	100	100	
24	0.03	0.21	0.93	1.6	3.61	4.96	8.8	12.38	14.99	19.81	25.66	32.02	38.99	50.43	55.2	66.52	74.93	85.08	96.65	100	100	100	100	100	100	100	100	100	100	
25	0.06	0.24	0.77	1.77	3.29	5.52	8.49	13.91	16.81	22.19	28.06	35.4	43.85	52.17	61.29	72.42	81.56	92.88	100	100	100	100	100	100	100	100	100	100	100	
26	0.03	0.25	0.81	1.86	3.48	5.84	8.99	14.7	17.86	26.25	30.42	38.08	46.68	60.53	67.08	82.97	90.1	100	100	100	100	100	100	100	100	100	100	100	100	
27	0.03	0.27	1.15	1.99	3.76	6.28	11.22	14.03	21.64	28.55	33.06	41.37	55.27	66.23	78.69	91.59	99.01	100	100	100	100	100	100	100	100	100	100	100	100	
28	0.03	0.29	0.93	2.69	4.05	6.77	12.15	15.16	20.93	27.76	35.85	45.06	55.38	72.44	79.67	93.41	100	100	100	100	100	100	100	100	100	100	100	100	100	
29	0.04	0.3	0.99	2.32	5.31	8.62	13.07	16.33	22.58	30.06	42.52	49.06	60.37	73.08	93.27	100	100	100	100	100	100	100	100	100	100	100	100	100	100	

Figure 3.15: Order algorithm with input of type “two-links”

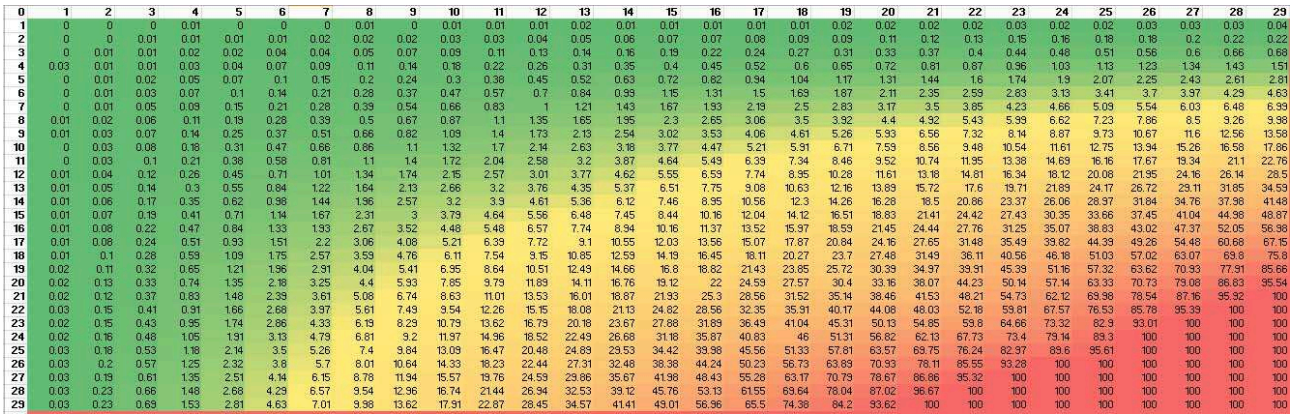
sizes. Also we ran the best case experiment using the *ordering* enhancement described in the enhancements section in Figure 3.10.

Greedy choices

In accordance with our discussion about the characteristics of model comparison in the context of testing, we attempt to modify the algorithm to satisfy such requirements. Namely, optimizing the algorithm to detect exact matches quickly with confidence, and in the case of a mismatch attempting to find the most optimal edit script possible, within any of the constraints (time or minimum size n_0 of the match). We implemented two main greedy strategies, then we show the effects they have in performance.

GetMappableVertices

First, in the function **GetMappableVertices** which is responsible for returning for node v from **M1**, a list of vertices $v_1\prime \ldots v_n\prime$ from **M2** which are candidates mapping for v. The original algorithm returns the list of candidates with no particular order, which could lead to exploring less promising branches first. We modify the function to return a sorted list of candidate nodes according to their structural similarity to the vertex v we



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
1	0	0	0	0.01	0	0	0	0.01	0	0.01	0.01	0.01	0.02	0.01	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.02	0.02	0.03	0.03	0.03	0.04
2	0	0	0.01	0.01	0.01	0.01	0.02	0.02	0.02	0.03	0.03	0.04	0.05	0.06	0.07	0.07	0.08	0.09	0.09	0.11	0.12	0.13	0.15	0.16	0.18	0.18	0.2	0.22	0.22
3	0	0.01	0.01	0.02	0.02	0.04	0.04	0.05	0.07	0.09	0.11	0.13	0.14	0.16	0.19	0.22	0.24	0.27	0.31	0.33	0.37	0.4	0.44	0.48	0.51	0.56	0.6	0.66	0.68
4	0.03	0.01	0.01	0.03	0.04	0.07	0.09	0.11	0.14	0.18	0.22	0.26	0.31	0.35	0.4	0.45	0.52	0.6	0.65	0.72	0.81	0.87	0.96	1.03	1.13	1.23	1.34	1.43	1.51
5	0	0.01	0.02	0.05	0.07	0.1	0.15	0.2	0.24	0.3	0.36	0.45	0.52	0.63	0.72	0.82	0.94	1.04	1.17	1.31	1.44	1.6	1.74	1.9	2.07	2.25	2.43	2.61	2.81
6	0	0.01	0.03	0.07	0.1	0.14	0.21	0.28	0.37	0.47	0.57	0.7	0.84	0.99	1.15	1.31	1.5	1.69	1.87	2.11	2.35	2.59	2.83	3.13	3.41	3.7	3.97	4.29	4.63
7	0	0.01	0.05	0.09	0.15	0.21	0.28	0.39	0.54	0.66	0.83	1	1.21	1.43	1.67	1.93	2.19	2.5	2.83	3.17	3.5	3.85	4.23	4.66	5.09	5.54	6.03	6.48	6.99
8	0.01	0.02	0.06	0.11	0.19	0.28	0.39	0.5	0.67	0.87	1.1	1.35	1.65	1.95	2.3	2.65	3.06	3.5	3.92	4.4	4.92	5.43	5.99	6.62	7.23	7.86	8.5	9.26	9.98
9	0.01	0.03	0.07	0.14	0.25	0.37	0.51	0.66	0.82	1.09	1.4	1.73	2.13	2.54	3.02	3.53	4.06	4.61	5.26	5.93	6.56	7.32	8.14	8.87	9.73	10.67	11.6	12.56	13.58
10	0	0.03	0.08	0.18	0.31	0.47	0.66	0.86	1.1	1.32	1.7	2.14	2.63	3.18	3.77	4.47	5.21	5.91	6.71	7.59	8.56	9.48	10.54	11.61	12.75	13.94	15.26	16.58	17.86
11	0	0.03	0.1	0.21	0.38	0.58	0.81	1.1	1.4	1.72	2.04	2.58	3.2	3.87	4.64	5.49	6.39	7.34	8.46	9.52	10.74	11.95	13.38	14.69	16.16	17.67	19.34	21.1	22.76
12	0.01	0.04	0.12	0.26	0.45	0.71	1.01	1.34	1.74	2.15	2.57	3.01	3.77	4.62	5.55	6.53	7.74	8.95	10.28	11.61	13.18	14.81	16.34	18.12	20.08	21.95	24.16	26.14	28.5
13	0.01	0.05	0.14	0.3	0.55	0.84	1.22	1.64	2.13	2.66	3.2	3.76	4.35	5.37	6.51	7.75	9.08	10.63	12.16	13.89	15.72	17.6	19.71	21.89	24.17	26.72	29.11	31.85	34.59
14	0.01	0.06	0.17	0.35	0.62	0.98	1.44	1.96	2.57	3.2	3.9	4.61	5.36	6.12	7.46	8.95	10.56	12.3	14.26	16.28	18.5	20.86	23.37	26.06	28.97	31.84	34.76	37.98	41.48
15	0.01	0.07	0.19	0.41	0.71	1.14	1.67	2.31	3	3.79	4.64	5.56	6.48	7.45	8.44	10.16	12.04	14.12	16.51	18.83	21.41	24.42	27.43	30.35	33.66	37.45	41.04	44.98	48.87
16	0.01	0.08	0.22	0.47	0.84	1.33	1.93	2.67	3.52	4.48	5.48	6.57	7.74	8.94	10.16	11.37	13.52	15.97	18.59	21.45	24.44	27.76	31.25	35.07	38.83	43.02	47.37	52.05	56.98
17	0.01	0.08	0.24	0.51	0.93	1.51	2.2	3.06	4.08	5.21	6.39	7.72	9.1	10.55	12.03	13.56	15.07	17.87	20.84	24.16	27.65	31.48	35.49	39.82	44.39	49.26	54.48	60.68	67.15
18	0.01	0.1	0.28	0.59	1.09	1.75	2.57	3.59	4.76	6.11	7.54	9.15	10.85	12.59	14.19	16.45	18.11	20.27	23.7	27.48	31.49	36.11	40.56	46.18	51.03	57.02	63.07	69.8	75.8
19	0.02	0.11	0.32	0.65	1.21	1.96	2.91	4.04	5.41	6.95	8.64	10.51	12.49	14.66	16.8	18.82	21.43	23.85	25.72	30.39	34.97	39.91	45.39	51.16	57.32	63.62	70.93	77.91	85.66
20	0.02	0.13	0.33	0.74	1.35	2.18	3.25	4.4	5.93	7.85	9.79	11.89	14.11	16.76	19.12	22	24.59	27.57	30.4	33.16	38.07	44.23	50.14	57.14	63.33	70.73	79.08	86.83	95.54
21	0.02	0.12	0.37	0.83	1.48	2.39	3.61	5.08	6.74	8.63	11.01	13.53	16.01	18.87	21.93	25.3	28.56	31.52	35.14	38.46	41.53	48.21	54.73	62.12	69.98	78.54	87.16	95.92	100
22	0.03	0.15	0.41	0.91	1.66	2.68	3.97	5.61	7.49	9.54	12.26	15.15	18.08	21.13	24.82	28.56	32.35	35.91	40.17	44.08	48.03	52.18	59.81	67.57	76.53	85.78	95.39	100	100
23	0.02	0.15	0.43	0.95	1.74	2.86	4.33	6.19	8.29	10.79	13.62	16.79	20.18	23.67	27.88	31.89	36.49	41.04	45.31	50.13	54.85	59.8	64.66	73.32	82.9	93.01	100	100	100
24	0.02	0.16	0.48	1.05	1.91	3.13	4.79	6.81	9.2	11.97	14.96	18.52	22.49	26.68	31.18	35.87	40.83	46	51.31	56.82	62.13	67.73	73.4	79.14	89.3	100	100	100	100
25	0.03	0.18	0.53	1.18	2.14	3.5	5.26	7.4	9.84	13.09	16.47	20.48	24.89	29.53	34.42	39.98	45.56	51.33	57.81	63.57	69.75	76.24	82.97	89.6	95.61	100	100	100	100
26	0.03	0.2	0.57	1.25	2.32	3.8	5.7	8.01	10.64	14.33	18.23	22.44	27.31	32.48	38.38	44.24	50.23	56.73	63.89	70.93	78.11	85.55	93.28	100	100	100	100	100	100
27	0.03	0.19	0.61	1.35	2.51	4.14	6.15	8.78	11.94	15.57	19.76	24.59	29.86	35.67	41.98	48.43	55.28	63.17	70.79	78.67	86.86	95.32	100	100	100	100	100	100	100
28	0.03	0.23	0.66	1.48	2.68	4.29	6.57	9.54	12.96	16.74	21.44	26.94	32.53	39.12	45.76	53.13	61.55	69.64	78.04	87.02	96.67	100	100	100	100	100	100	100	100
29	0.03	0.23	0.69	1.53	2.81	4.63	7.01	9.98	13.62	17.91	22.87	28.45	34.57	41.41	49.01	56.96	65.5	74.38	84.2	93.62	100	100	100	100	100	100	100	100	100

Figure 3.16: Order algorithm with input of type “three-links”

are trying to match. More specifically we use as an indication the absolute value of the difference between the number of incoming edges between the node and its candidate in addition to the absolute value of the difference in the number of outgoing edges. This approach ensures choosing the most promising vertices first, and hence has the potential of increasing the performance. The reason for this is simple: the algorithm proceeds in searching for a solution only if the branch can achieve a better than current solution.

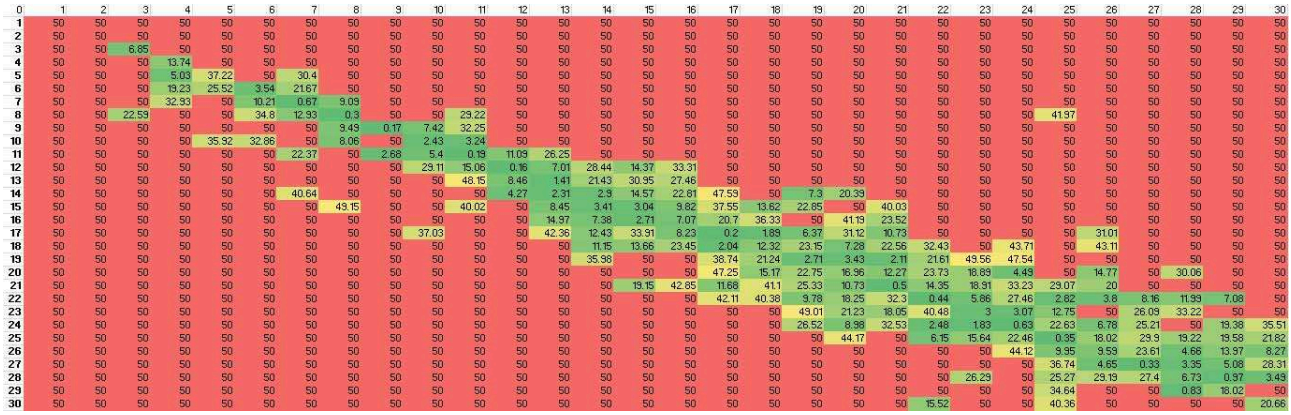
The previous enhancement is one step towards ensuring that repeated runs on the same inputs converge on the best solution within the same required timing, i.e; since there is no specific order in examining the candidate vertices, the algorithm could run longer in cases where it attempts a different order especially in the initial depth first search. We show the performance enhancement of this first strategy grouped with the second once, and with the third in another.

The cost is that for each call to *GetMappableVertices* is n^2, which is called once for every *Backtrack* call.

In the worst case there is $O(m^n)$ *Backtrack* calls so total cost is $O(n^2 * m^n) = O(n * m^n * n)$. but we know that $m \leq n$ because of the order enhancement. When compared to the worst case of the original CSIA algorithm : $O(m^{n+1}n)$ we notice that the worst case complexity gets shifted to: $O(n^2 * m^n)$.

PickVertix

We implemented another greedy strategy regarding the function **PickVertix** which decides in the original algorithm which vertex v from **M1** to attempt mapping next. The heuristic used in the original CSIA algorithm was to choose the vertex with the least number of candidates (i.e the most restricted vertex) which is a very good performance enhancement but is not optimal for the context of testing where exact matches are more important. It does not give any indication of what to do when you get two vertices of equal least number of candidates. This could lead to different running times by the algorithm on the same set of inputs where such case arises. Furthermore, we feel that a greedy strategy should be in place to help the function in choosing which vertex to process next in such scenarios.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	50	50	6.85	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	50	13.74	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	50	5.03	37.22	50	30.4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
6	50	50	50	19.23	25.52	3.54	21.67	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
7	50	50	50	32.93	50	10.21	0.67	9.09	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
8	50	50	22.59	50	50	34.8	12.93	0.3	50	50	23.22	50	50	50	50	50	50	50	50	50	50	50	50	50	50	41.97	50	50	50	50	50
9	50	50	50	50	50	50	50	9.49	0.17	7.42	32.25	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
10	50	50	50	50	35.32	32.86	50	8.06	50	2.43	3.24	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
11	50	50	50	50	50	50	22.37	50	2.68	5.4	0.19	11.09	26.25	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
12	50	50	50	50	50	50	50	50	50	29.11	15.06	0.16	7.01	28.44	14.37	33.31	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	50	50	50	50	50	50	50	50	50	50	48.15	8.46	1.41	21.43	30.95	27.46	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	50	50	50	50	50	50	40.64	50	50	50	50	4.27	2.31	2.9	14.57	22.61	47.59	50	7.3	20.39	50	50	50	50	50	50	50	50	50	50	50
15	50	50	50	50	50	50	50	49.15	50	50	40.02	50	8.45	3.41	3.04	9.82	37.55	13.62	22.85	50	40.03	50	50	50	50	50	50	50	50	50	50
16	50	50	50	50	50	50	50	50	50	50	50	50	14.37	7.38	2.71	7.07	20.7	36.33	50	41.19	23.52	50	50	50	50	50	50	50	50	50	50
17	50	50	50	50	50	50	50	50	50	37.03	50	50	42.36	12.43	33.31	8.23	0.2	1.88	6.37	31.12	10.73	50	50	50	50	31.01	50	50	50	50	50
18	50	50	50	50	50	50	50	50	50	50	50	50	50	11.15	13.66	23.45	2.04	12.32	23.15	7.28	22.56	32.43	50	43.71	50	43.11	50	50	50	50	50
19	50	50	50	50	50	50	50	50	50	50	50	50	50	50	35.98	50	50	38.74	21.24	2.71	3.43	2.11	21.61	43.56	47.54	50	50	50	50	50	50
20	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	47.25	15.17	22.75	16.96	12.27	23.73	18.89	4.49	50	14.77	50	30.06	50	50
21	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	19.15	42.85	11.68	41.1	25.33	10.73	0.5	14.35	18.91	33.23	29.07	20	50	50	50	50
22	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	42.11	40.38	9.78	18.25	32.3	0.44	5.86	27.46	2.82	3.8	8.16	11.99	7.08	50
23	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	49.01	21.23	18.05	40.48	3	3.07	12.75	50	26.09	33.22	50	50
24	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	26.52	8.98	32.53	2.48	1.83	0.63	22.63	6.78	25.21	50	19.38	35.51
25	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	44.17	50	6.15	15.64	22.46	0.35	18.02	23.9	19.22	19.58	21.82
26	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	44.12	9.95	9.59	23.61	4.68	13.97	8.27
27	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	36.74	4.65	0.33	3.35	5.08	28.31
28	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	26.29	50	25.27	29.19	27.4	6.73	0.97	3.43
29	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	34.64	50	50	0.83	18.02	50
30	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	15.52	50	50	40.36	50	50	50	50	20.66

Figure 3.17: Original algorithm with input size 20 nodes with varying edges

To achieve this, we implement a custom **PickVertixGreedy** function to extend the function **PickVertix**. The main idea is to consider the a greedy optimization within each candidate list when deciding which vertex to pick. We would like to choose a vertex v_i from **M1** to process such that for its candidate list there is a candidate which is most similar according to the in,out-degree difference. If we find more than one such node in **M1**, we then further choose the one which has the least number of similar candidates. If we find more than one such node in **M1** satisfying both previous criteria, we choose the one which has a larger sum of differences among all of its candidate nodes from **M2**. This seems complex, but it ensures we follow the most promising path first (where only few candidate seem optimal and the rest seem non-optimal).

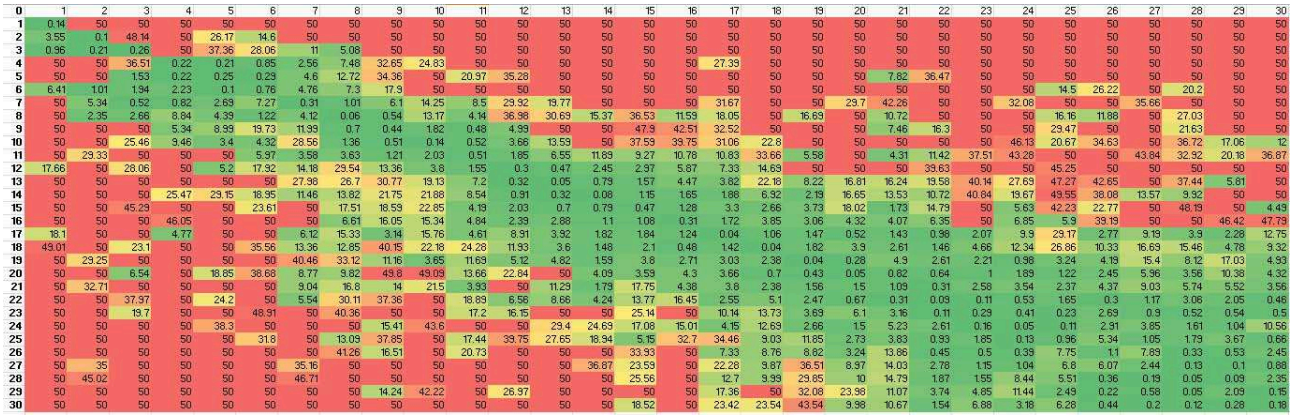
This logic provides ground for choosing more deterministically in which order to process the nodes. Combined with the previous greedy heuristic **GetMappableVertices** the performance increases. Note that this enhancement does not change the complexity of the algorithm but make it converge to the best solution in a faster and more persistent manner

The impact on performance for this enhancement is negligible since the degree difference does not change per vertex and calculated only once.

To demonstrate the performance gains we compare the results from this version of the algorithm to the original version using the varying edge models.

First, we use the 20 nodes fixed models to run the original algorithm in figure 3.17, the order based algorithm in figure 3.18 and the greedy based algorithm in figure 3.19. Note that in this case the greedy strategy does not show a significant enhancement in performance.

Second we use the 40 nodes fixed models to run the order based algorithm (original algorithm will be even worse running time) in Figure 3.20 , and the greedy based algorithm in Figure 3.21. Note how the greedy algorithm performs best in situations where there is an exact match (i.e the diagonal of the map). The order based algorithm suffers greatly in this type of models, this is due to the fact that the fixed size models with varying random edges represent the worst case models for MCS to deal with.

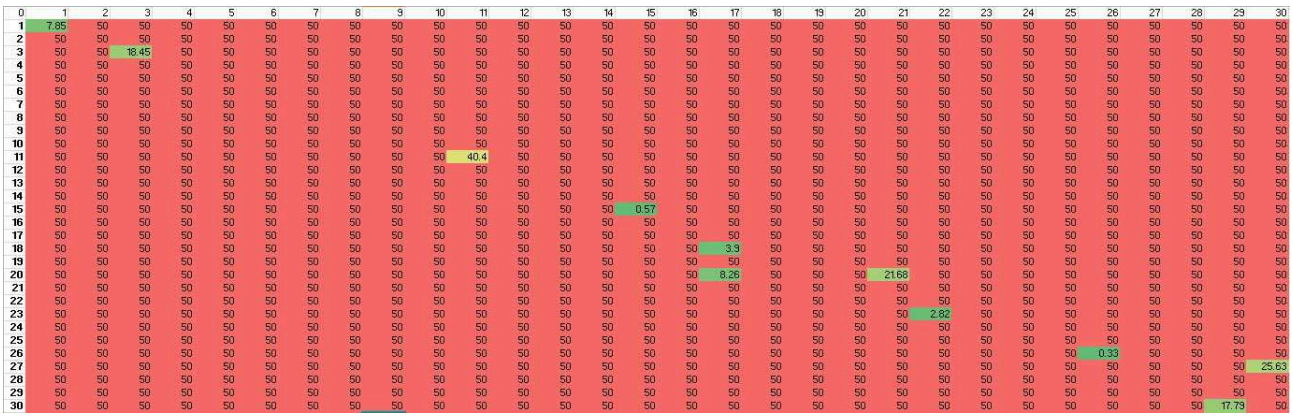


	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0.14	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	3.55	0.1	48.14	50	26.17	14.6	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	0.96	0.21	0.26	50	37.36	28.06	11	5.08	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	36.51	0.22	0.21	0.85	2.56	7.48	32.65	24.83	50	50	50	50	50	50	50	27.39	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	1.53	0.22	0.25	0.29	4.6	12.72	34.36	50	20.97	35.28	50	50	50	50	50	50	50	50	50	7.82	36.47	50	50	50	50	50	50	50	50
6	6.41	1.01	1.94	2.23	0.1	0.76	4.76	7.3	17.9	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	14.5	26.22	50	20.2	50	50	50
7	50	5.34	0.52	0.82	2.63	7.27	0.31	1.01	6.1	14.25	8.5	29.92	19.77	50	50	50	50	31.67	50	50	29.7	42.26	50	50	32.08	50	50	35.66	50	50	50
8	50	2.35	2.66	8.84	4.39	1.22	4.12	0.06	0.54	13.17	4.14	36.98	30.69	15.37	36.53	11.59	18.05	50	16.69	50	10.72	50	50	50	16.16	11.88	50	27.03	50	50	50
9	50	50	50	5.34	8.99	19.73	11.99	0.7	0.44	1.62	0.48	4.99	50	50	47.9	42.51	32.52	50	50	50	7.46	16.3	50	50	29.47	50	50	21.63	50	50	50
10	50	50	25.46	9.46	3.4	4.32	28.56	1.36	0.51	0.14	0.52	3.66	13.53	50	37.53	33.75	31.06	22.8	50	50	50	50	50	46.13	20.67	34.63	50	36.72	17.06	12	50
11	50	29.33	50	50	50	5.97	3.58	3.63	1.21	2.03	0.51	1.85	6.55	11.83	9.27	10.78	10.83	33.66	5.58	50	4.31	11.42	37.51	43.28	50	50	43.84	32.92	20.18	36.87	50
12	17.66	50	28.06	50	5.2	17.32	14.18	29.54	13.36	3.8	1.55	0.3	0.47	2.45	2.37	5.87	7.33	14.69	50	50	50	33.63	50	50	45.25	50	50	50	50	50	50
13	50	50	50	50	50	50	27.96	26.7	30.77	19.13	7.2	0.32	0.05	0.79	1.57	4.47	3.82	22.18	8.22	16.81	16.24	19.58	40.14	27.69	47.27	42.65	50	37.44	5.81	50	50
14	50	50	50	25.47	29.15	18.95	11.46	13.82	21.75	21.88	8.54	0.91	0.32	0.08	1.15	1.65	1.88	6.32	2.19	16.65	13.53	10.72	40.84	19.67	43.55	38.08	13.57	9.92	50	50	50
15	50	50	45.29	50	50	23.61	50	17.51	18.59	22.85	4.19	2.03	0.7	0.79	0.47	1.28	3.3	2.66	3.73	18.02	1.73	14.79	50	5.63	42.23	22.77	50	48.19	50	4.49	50
16	50	50	50	46.05	50	50	50	6.61	16.05	15.34	4.84	2.39	2.88	1.1	1.08	0.31	1.72	3.85	3.06	4.32	4.07	6.35	50	6.85	5.9	39.19	50	50	46.42	47.79	50
17	18.1	50	50	4.77	50	50	6.12	15.33	3.14	15.76	4.61	8.91	3.92	1.82	1.84	1.24	0.04	1.06	1.47	0.52	1.43	0.98	2.07	9.9	29.17	2.77	9.19	3.9	2.28	12.75	50
18	49.01	50	23.1	50	50	35.56	13.36	12.85	40.15	22.18	24.28	11.93	3.6	1.48	2.1	0.48	1.42	0.04	1.82	3.9	2.61	1.46	4.66	12.34	26.86	10.33	16.69	15.46	4.78	9.32	50
19	50	29.25	50	50	50	50	40.46	33.12	11.16	3.65	11.69	5.12	4.82	1.59	3.8	2.71	3.03	2.38	0.04	0.28	4.9	2.61	2.21	0.98	3.24	4.19	15.4	8.12	17.03	4.93	50
20	50	50	6.54	50	18.85	38.68	8.77	9.82	43.8	49.09	13.66	22.84	50	4.09	3.53	4.3	3.66	0.7	0.43	0.05	0.82	0.64	1	1.89	1.22	2.45	5.96	3.56	10.38	4.32	50
21	50	32.71	50	50	50	50	9.04	16.8	14	21.5	3.93	50	11.29	1.79	17.75	4.38	3.8	2.38	1.56	1.5	1.09	0.31	2.58	3.54	2.37	4.37	9.03	5.74	5.52	3.56	50
22	50	50	37.37	50	24.2	50	5.54	30.11	37.36	50	18.89	6.56	8.66	4.24	13.77	16.45	2.55	5.1	2.47	0.67	0.31	0.09	0.11	0.53	1.65	0.3	1.17	3.06	2.05	0.46	50
23	50	50	19.7	50	50	48.91	50	40.36	50	50	17.2	16.15	50	50	25.14	50	10.14	13.73	3.69	6.1	3.16	0.11	0.29	0.41	0.23	2.69	0.9	0.52	0.54	0.5	50
24	50	50	50	50	38.3	50	50	50	15.41	43.6	50	50	29.4	24.69	17.08	15.01	4.15	12.69	2.68	1.5	5.23	2.61	0.16	0.05	0.11	2.31	3.85	1.61	1.04	10.56	50
25	50	50	50	50	50	31.8	50	13.09	37.85	50	17.44	39.75	27.65	18.94	5.15	32.7	34.46	9.03	11.85	2.73	3.83	0.93	1.85	0.13	0.96	5.34	1.05	1.79	3.67	0.66	50
26	50	50	50	50	50	50	50	41.26	16.51	50	20.73	50	50	50	33.93	50	7.33	8.76	8.82	3.24	13.86	0.45	0.5	0.39	7.75	1.1	7.83	0.33	0.53	2.45	50
27	50	35	50	50	50	50	35.16	50	50	50	50	50	50	50	36.87	23.59	50	22.28	9.87	36.51	8.37	14.03	2.78	1.15	1.04	6.8	6.07	2.44	0.13	0.1	0.88	50
28	50	45.02	50	50	50	50	46.71	50	50	50	50	50	50	50	50	25.56	50	12.7	9.99	23.85	10	14.79	1.87	1.55	8.44	5.51	0.36	0.19	0.05	0.09	2.35	50
29	50	50	50	50	50	50	50	50	14.24	42.22	50	26.97	50	50	50	50	50	17.36	50	32.08	23.98	11.07	3.74	4.85	11.44	2.49	0.22	0.58	0.05	2.09	0.15	50
30	50	50	50	50	50	50	50	50	50	50	50	50	50	50	18.52	50	23.42	23.54	43.54	9.98	10.67	1.54	6.88	3.18	6.28	0.44	0.2	0.12	0.28	0.18	50

Figure 3.18: Order algorithm with input size 20 nodes with varying edges

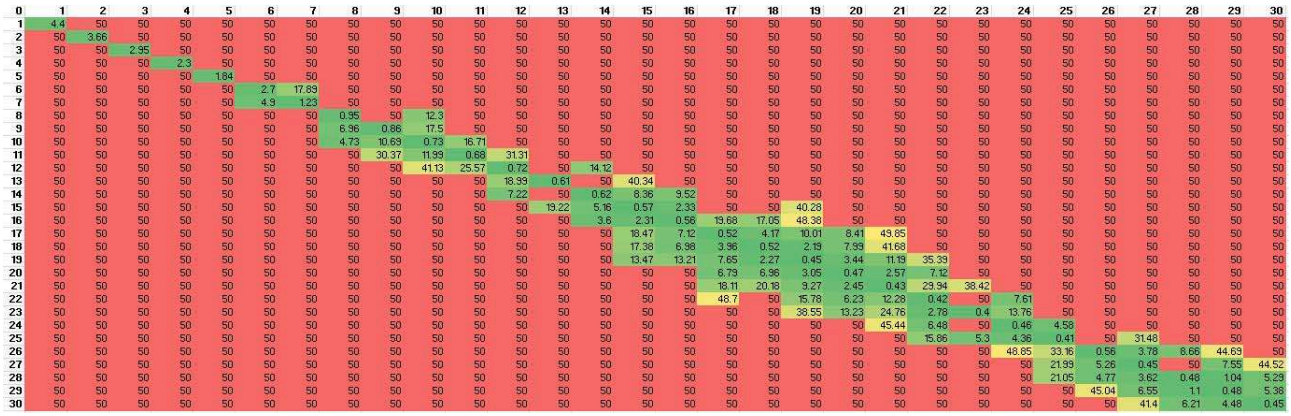
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	0.15	34.05	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	0.11	0.42	33.25	4.68	4.87	38.62	20.64	50	50	26.31	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	22.95	0.12	0.09	7.83	3.56	6.43	2.53	2.46	6.66	32.18	15.6	50	50	50	50	41.04	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	8.79	1.09	0.08	0.09	0.17	0.25	3.84	13.26	14.55	50	46.7	50	50	50	50	50	44.9	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	1	0.92	0.09	0.09	0.23	1.63	8.94	10.54	12.04	15.75	17.02	42.73	27.14	50	50	50	50	50	50	12.3	50	50	50	50	50	50	50	50	50	50
6	50	0.34	0.32	0.17	0.21	0.09	1.72	6	7.63	6.72	12.47	27.64	30.91	50	50	50	50	50	50	50	11.53	50	50	50	50	50	50	50	50	50	50
7	50	26.04	0.21	0.16	1.05	1.06	0.08	1.01	2.73	5.07	4.42	7.53	21.27	11.74	36.26	50	50	50	50	20.39	18.15	50	46.82	50	50	50	50	40.84	41.51	41.62	50
8	50	2.77	0.23	0.46	1.25	1.79	0.61	0.06	0.33	1.75	4.55	32.37	11.64	11.76	14.84	18.18	15	50	50	16.42	11.42	7.99	32.66	41.83	41.5	27.38	50	46.81	47.49	47.53	50
9	50	50	2.09	3.09	2.5	5.14	1.31	0.19	0.06	0.62	0.59	12.94	8.65	10.11	11.53	18.16	15.13	50	50	50	20.25	50	50	50	50	48.78	39.9	27.61	28.68	28.61	27.37	50
10	50	10.47	9.84	1.61	0.64	1.14	12.05	0.47	0.54	0.06	0.61	3.53	11.25	24.15	19.75	17.04	7.13	26.08	18.2	50	50	50	50	50	50	11.38	43.55	33.35	35.19	35.21	40.08	50
11	50	50	6.29	5.97	2.12	3.15	2.88	1.56	0.18	0.21	0.07	1.22	2.94	5.13	3.63	4.04	9.12	17.73	9.03	7.02	9.76	24.18	37.27	50	50	14.19	36.95	50	50	47.93	50
12	50	32.41	17.35	17.65	3.06	20.81	2.41	12.83	6.71	3.44	1.01	0.06	0.18	0.67	1.38	4	4.53	9.36	6.26	16.41	13.93	4.68	13.57	10.07	49.91	50	50	50	50	50	50
13	50	50	47.01	37.38	12.2	50	9.24	8.45	17.97	5.32	1.82	0.17	0.06	0.27	0.7	1.83	3.66	10.77	7.62	19.71	7.16	5.73	15.42	12.1	50	50	50	50	50	50	50
14	50	50	50	11.41	10	12.68	5.43	6.81	12.46	16.85	2.55	0.42	0.27	0.05	0.18	0.27	1.23	1.43	1.27	1.93	2.23	4.02	15.52	18.41	50	43.73	50	50	50	50	50
15	50	50	17.37	24.48	25.09	28.16	14.34	5.68	11.93	15.02	2.54	0.76	0.32	0.15	0.04	0.24	1.33	1.41	1.36	2.19	2.3	4.31	17.64	6.17	14.96	43.85	36.22	48.38	43.08	34.9	50
16	50	50	50	11.93	29.91	30.96	21.7	4.88	4.61	12.36	2.04	1.92	1.05	0.21	0.16	0.05	0.5	0.54	1.39	2.5	4.22	6.87	19.92	9.58	15.66	25.68	42.11	50	50	43.27	50
17	50	50	50	50	50	50	8.53	5.32	5.14	14.13	9.78	1.8	1.53	0.78	0.82	0.46	0.04	0.24	0.83	0.48	0.82	0.9	3.04	2.67	17.21	7.08	6.11	5.73	6.32	12.77	50
18	50	50	50	50	50	50	6.61	7.17	36.09	14.77	8.14	4.08	4.72	0.72	0.86	0.47	0.14	0.04	0.22	0.42	0.42	1.6	3.57	3.17	3.9	5.91	6.42	7.53	6.32	14.38	50
19	50	50	50	50	50	50	12.71	13.1	38.32	14.46	6.93	2.98	3.9	0.7	0.72	1.14	0.47	0.21	0.04	0.29	0.15	0.55	2.25	1.51	1.44	2.98	6.27	6.18	6.82	7.42	50
20	50	50	50	50	15.26	17.84	3.14	12.09	36.17	39.94	6.33	29.63	25.72	1.91	1.89	2.37	0.46	0.63	0.32	0.04	0.28	0.22	0.56	0.43	1.47	1.06	2.75	2.63	2.41	2.39	50
21	50	50	50	50	50	50	13.83	5.11	37.32	18.25	10.17	11.33	5.36	1.49	1.58	2.62	0.48	0.67	0.26	0.29	0.04	0.25	0.53	0.6	2.08	1.38	6.61	3.33	3.42	5.95	50
22	50	50	50	50	50	50	29.96	27.09	36.07	25.41	23.01	7.11	27.03	8.49	9.47	14.49	1.4	1.76	0.68	0.21	0.25	0.05	0.07	0.07	0.19	0.14	2.21	0.35	0.39	0.5	50
23	50	50	50	50	50	50	50	50	50	43.24	13.46	15.24	18.49	40.88	49.01	50	4.23	5.67	2.15	0.61	0.8	0.06	0.05	0.06	0.21	0.14	0.99	0.91	0.87	1.04	50
24	50	50	50	50	50	50	50	50	50	47.62	29.38	6.07	13.51	19.36	13.17	12.1	7.99	4.65	2.14	0.61	0.7	0.06	0.06	0.06	0.07	0.27	0.91	0.63	0.64	0.57	50
25	50	50	50	50	50	50	34.29	10.38	10.46	20.9	22.84	10.78	11.03	15.75	13.7	10.86	21.2	6.14	4.6	3.2	2.39	0.2	0.23	0.08	0.06	0.34	1.7	1.47	1.43	1.53	50
26	50	50	50	50	50	50	50	50	50	42.75	9.66	40.41	50	37.54	50	50	15.47	15.66	6.53	2.9	4.64	0.21	0.32	0.35	0.32	0.05	0.07	0.52	0.42	0.48	50
27	50	50	50	50	42.73	42.61	35.4	41.96	40.22	45.53	29.09	50	50	50	38.67	50	19.84	19.93	17.99	6.39	7.52	1.89	0.84	0.7	1.79	0.07	0.06	0.06	0.06	0.06	50
28	50	50	50	50	50	50	36.18	42.91	40.16	46.32	26.25	50	50	45.57	43.94	50	18.98	21.95	20.11	4.85	10.42	4.24	0.76	1.05	5.63	0.5	0.07	0.05	0.06	0.07	50
29	50	50	50	50	50	50	36.27	42.59	35.02	40.82	26.4	50	50	50	43.95	50	20.37	19.57	20.25	8.43	8.02	2.12	0.77	0.81	1.68	0.36	0.06	0.05	0.06	0.06	50
30	50	50	50	50	50	50	34.62	39.02	37.26	38.85	23.44	50	50	50	41.94	50	17.84	17.41	17.8	5.09	8.22	1.95	0.95	0.8	1.91	0.43	0.09	0.06	0.06	0.06	50

Figure 3.19: Greedy algorithm with input size 20 nodes with varying edges



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	7.85	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	50	50	18.45	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
6	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
7	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
8	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
9	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
10	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
11	50	50	50	50	50	50	50	50	50	50	50	40.4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
12	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
15	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	0.57	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
16	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
17	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
18	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	3.9	50	50	50	50	50	50	50	50	50	50	50	50	50
19	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
20	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	8.26	50	50	50	21.68	50	50	50	50	50	50	50	50	50
21	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
22	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
23	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	2.82	50	50	50	50	50	50	50	50
24	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
25	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
26	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	0.33	50	50	50	50
27	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	25.63
28	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
29	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
30	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	17.79	50

Figure 3.20: Order enhanced algorithm with input size 40 nodes with varying edges



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	4.4	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	3.66	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	50	50	2.95	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	50	2.3	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	50	50	1.84	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
6	50	50	50	50	50	2.7	17.89	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
7	50	50	50	50	50	4.9	1.23	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
8	50	50	50	50	50	50	50	0.95	50	12.3	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
9	50	50	50	50	50	50	50	6.96	0.86	17.5	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
10	50	50	50	50	50	50	50	4.73	10.69	0.73	16.71	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
11	50	50	50	50	50	50	50	50	30.37	11.99	0.68	31.31	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
12	50	50	50	50	50	50	50	50	50	41.13	25.57	0.72	50	14.12	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	50	50	50	50	50	50	50	50	50	50	50	18.39	0.61	50	40.34	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	50	50	50	50	50	50	50	50	50	50	50	7.22	50	0.62	8.36	9.52	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
15	50	50	50	50	50	50	50	50	50	50	50	50	19.22	5.16	0.57	2.33	50	50	40.28	50	50	50	50	50	50	50	50	50	50	50	50
16	50	50	50	50	50	50	50	50	50	50	50	50	50	3.6	2.31	0.58	19.68	17.05	48.38	50	50	50	50	50	50	50	50	50	50	50	50
17	50	50	50	50	50	50	50	50	50	50	50	50	50	50	18.47	7.12	0.52	4.17	10.01	8.41	49.85	50	50	50	50	50	50	50	50	50	50
18	50	50	50	50	50	50	50	50	50	50	50	50	50	50	17.36	6.96	3.96	0.52	2.19	7.99	41.68	50	50	50	50	50	50	50	50	50	50
19	50	50	50	50	50	50	50	50	50	50	50	50	50	50	13.47	13.21	7.65	2.27	0.45	3.44	11.19	35.39	50	50	50	50	50	50	50	50	50
20	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	6.73	6.96	3.05	0.47	2.57	7.12	50	50	50	50	50	50	50	50	50
21	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	18.11	20.18	9.27	2.45	0.43	29.94	38.42	50	50	50	50	50	50	50	50
22	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	48.7	50	15.78	6.23	12.28	0.42	50	7.61	50	50	50	50	50	50	50
23	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	38.55	13.23	24.76	2.78	0.4	13.76	50	50	50	50	50	50	50
24	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	45.44	6.48	50	0.46	4.58	50	50	50	50	50	50
25	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	15.86	5.3	4.36	0.41	50	31.48	50	50	50	50
26	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	48.85	33.16	0.56	3.78	8.66	44.69	50	50
27	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	21.99	5.26	0.45	50	7.55	44.52	
28	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	21.05	4.77	3.62	0.48	1.04	5.23	
29	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	45.04	6.55	1.1	0.48	5.38	
30	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	41.4	6.21	4.48	0.45	

Figure 3.21: Greedy enhanced algorithm with input size 40 nodes with varying edges

The cost of *PickVertix* will not affect the previously found worst case complexity caused by the *getMappableVertices* enhancement, since *PickVertix* is called once for every backtrack and not for every recursion level.

Delayed n_0 application

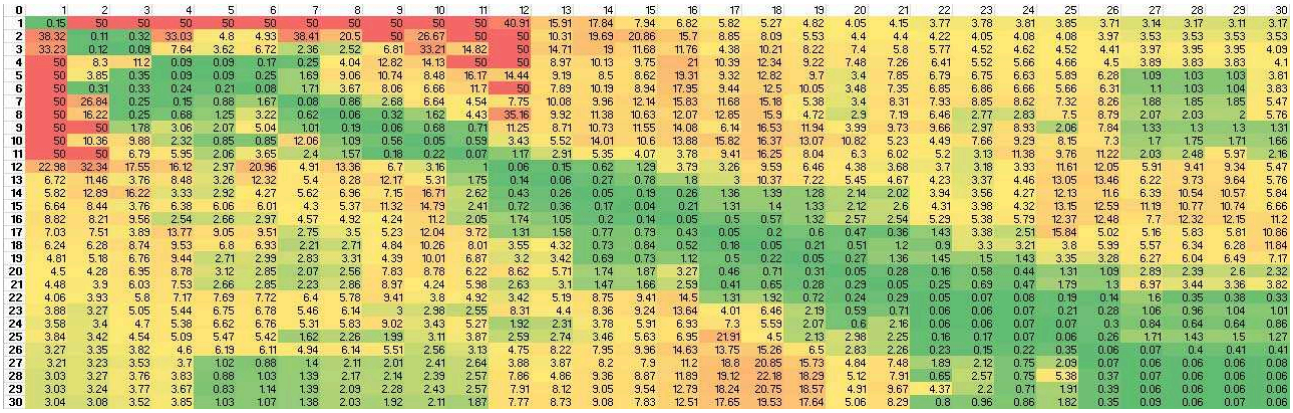
Finally we make another modifications to help further accomplish the testing requirements. We introduce the option to delay the use of the pruning condition n_0 (representing the minimum MCS size allowed) until a solution is found.

To demonstrate the usefulness of such a delay, imagine the following scenario. We have to compare two models of size n each. We know if there is an exact match then we should get an MCS with size n, and hence if we set $n_0 = n$ then we will find the MCS very fast. However, if there is a mismatch (even a very small one) then the actual MCS size will be less than n_0, and hence the algorithm will ignore it and return that no MCS of minimum size n_0 exists.

Subsequently it would be useful if we only start looking if there is a MCS of size $\geq n_0$ only after some solution has been found. Furthermore, this always guarantees the algorithm returning a solution either larger than n_0 if it exists, or smaller if it does not.

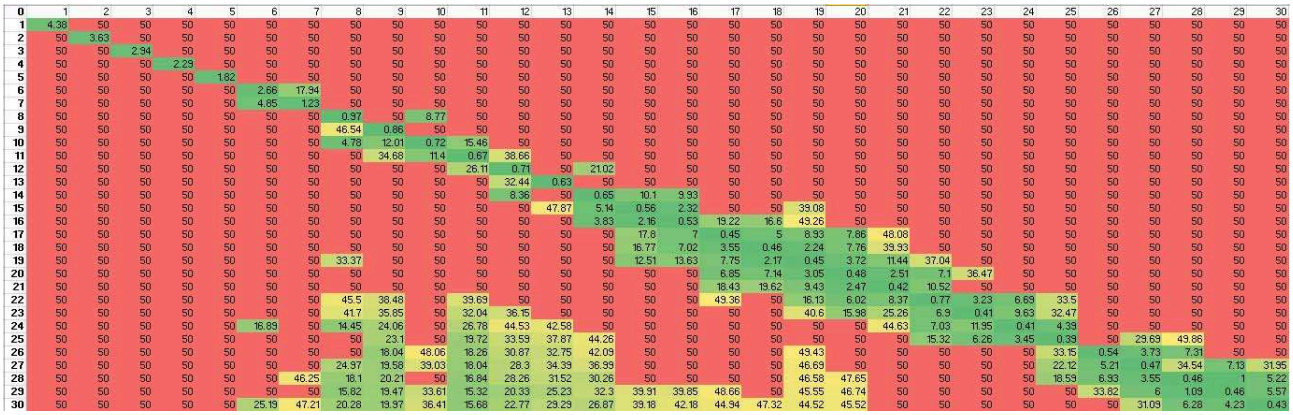
For example, we can use this feature when we are most interested in a complete match, where we know that it would be of size n. Then we can set $n_0 = n$ and run the algorithm with the delayed option specified here. The algorithm would find some solution in the initial depth first search iteration (note that we expect this solution to be a good solution in most cases since we are using a greedy ordering strategy), then it will start using the pruning condition n_0 which would make it as fast as possible converge on the complete match if it exists and terminate quickly. Also, note that when size of $\mathbf{M1}$,$\mathbf{M2} = n$ then there is no solution larger than n.

To demonstrate the use of the delayed application on n_0 we run two two experiments using the fixed size of nodes models. In the first case we run models of 20 nodes with setting $n0 = 15$ in Figure 3.22. Note that this is using the greedy enhanced algorithm, and the data should be compared to Figure 3.19.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	0.15	50	50	50	50	50	50	50	50	50	50	40.91	15.91	17.84	7.94	6.82	5.82	5.27	4.82	4.05	4.15	3.77	3.78	3.81	3.85	3.71	3.14	3.17	3.11	3.17
1	38.32	0.11	0.32	33.03	4.8	4.93	38.41	20.5	50	26.67	50	50	10.31	19.69	20.86	15.7	8.85	8.09	5.53	4.4	4.4	4.22	4.05	4.08	4.08	3.97	3.53	3.53	3.53	3.53
2	33.23	0.12	0.09	7.64	3.62	6.72	2.36	2.52	6.81	33.21	14.82	50	14.71	19	11.68	11.76	4.38	10.21	8.22	7.4	5.8	5.77	4.52	4.62	4.52	4.41	3.97	3.95	3.95	4.09
3	50	8.3	11.2	0.09	0.09	0.17	0.25	4.04	12.82	14.13	50	50	8.97	10.13	9.75	21	10.39	12.34	9.22	7.48	7.26	6.41	5.52	5.66	4.66	4.5	3.89	3.83	3.83	4.1
4	50	3.85	0.35	0.09	0.09	0.25	1.69	9.06	10.74	8.48	16.17	14.44	9.19	8.5	8.62	19.31	9.32	12.82	9.7	3.4	7.85	6.79	6.75	6.63	5.89	6.28	1.09	1.03	1.03	3.81
5	50	0.31	0.33	0.24	0.21	0.08	1.71	3.67	8.06	6.66	11.7	50	7.89	10.19	8.94	17.95	9.44	12.5	10.05	3.48	7.35	6.85	6.86	6.66	5.66	6.31	1.1	1.03	1.04	3.83
6	50	26.84	0.25	0.15	0.88	1.67	0.08	0.86	2.68	6.64	4.54	7.75	10.08	9.96	12.14	15.83	11.68	15.18	5.38	3.4	8.31	7.93	8.85	8.62	7.32	8.26	1.88	1.85	1.85	5.47
7	50	16.22	0.25	0.68	1.25	3.22	0.62	0.06	0.32	1.62	4.43	35.16	9.92	11.38	10.63	12.07	12.85	15.9	4.72	2.9	7.19	6.46	2.77	2.83	7.5	8.79	2.07	2.03	2	5.76
8	50	50	1.78	3.06	2.07	5.04	1.01	0.19	0.06	0.68	0.71	11.25	8.71	10.73	11.55	14.08	6.14	16.53	11.94	3.99	9.73	9.66	2.97	8.93	2.06	7.84	1.33	1.3	1.3	1.31
9	50	10.36	9.88	2.32	0.85	0.85	12.06	1.09	0.56	0.05	0.59	3.43	5.52	14.01	10.6	13.88	15.82	16.37	13.07	10.82	5.23	4.49	7.66	9.29	8.15	7.3	1.7	1.75	1.71	1.66
10	50	50	6.79	5.95	2.06	3.65	2.4	1.57	0.18	0.22	0.07	1.17	2.91	5.35	4.07	3.78	9.41	16.25	8.04	6.3	6.02	5.2	3.13	11.38	9.76	11.22	2.03	2.48	5.97	2.16
11	22.98	32.34	17.55	16.12	2.97	20.96	4.91	13.36	6.7	3.16	1	0.06	0.15	0.62	1.29	3.79	3.26	3.53	6.46	4.38	3.68	3.7	3.18	3.93	11.61	12.05	5.91	9.41	9.34	5.47
12	6.72	11.46	3.76	8.48	3.26	12.32	5.4	8.28	12.17	5.31	1.75	0.14	0.06	0.27	0.78	1.8	3	10.37	7.22	5.45	4.67	4.23	3.37	4.46	13.05	13.46	6.22	9.73	9.64	5.76
13	5.82	12.89	16.22	3.33	2.92	4.27	5.62	6.96	7.15	16.71	2.62	0.43	0.26	0.05	0.19	0.26	1.36	1.39	1.28	2.14	2.02	3.94	3.56	4.27	12.13	11.6	6.39	10.54	10.57	5.84
14	6.64	8.44	3.76	6.38	6.06	6.01	4.3	5.37	11.32	14.79	2.41	0.72	0.36	0.17	0.04	0.21	1.31	1.4	1.33	2.12	2.6	4.31	3.38	4.32	13.15	12.59	11.19	10.77	10.74	6.66
15	8.82	8.21	9.56	2.54	2.66	2.97	4.57	4.32	4.24	11.2	2.05	1.74	1.05	0.2	0.14	0.05	0.5	0.57	1.32	2.57	2.54	5.29	5.38	5.79	12.37	12.48	7.7	12.32	12.15	11.2
16	7.03	7.51	3.89	13.77	9.05	3.51	2.75	3.5	5.23	12.04	3.72	1.31	1.58	0.77	0.79	0.43	0.05	0.2	0.6	0.47	0.36	1.43	3.38	2.51	15.84	5.02	5.16	5.83	5.81	10.86
17	6.24	6.28	8.74	9.53	6.8	6.93	2.21	2.71	4.84	10.26	8.01	3.55	4.32	0.73	0.84	0.52	0.18	0.05	0.21	0.51	1.2	0.9	3.3	3.21	3.8	5.39	5.57	6.34	6.28	11.84
18	4.81	5.18	6.76	9.44	2.71	2.99	2.83	3.31	4.39	10.01	6.87	3.2	3.42	0.69	0.73	1.12	0.5	0.22	0.05	0.27	1.36	1.45	1.5	1.43	3.35	3.28	6.27	6.04	6.49	7.17
19	4.5	4.28	6.95	8.78	3.12	2.85	2.07	2.56	7.83	8.78	6.22	8.62	5.71	1.74	1.87	3.27	0.46	0.71	0.31	0.05	0.28	0.16	0.58	0.44	1.31	1.09	2.89	2.39	2.6	2.32
20	4.48	3.9	6.03	7.53	2.66	2.85	2.23	2.86	8.97	4.24	5.98	2.63	3.1	1.47	1.66	2.59	0.41	0.65	0.28	0.29	0.05	0.25	0.69	0.47	1.79	1.3	6.97	3.44	3.36	3.82
21	4.06	3.93	5.8	7.17	7.69	7.72	6.4	5.78	9.41	3.8	4.92	3.42	5.19	8.75	9.41	14.5	1.31	1.92	0.72	0.24	0.29	0.05	0.07	0.08	0.19	0.14	1.6	0.35	0.38	0.33
22	3.88	3.27	5.05	5.44	6.75	6.78	5.46	6.14	3	2.98	2.55	8.31	4.4	8.36	9.24	13.64	4.01	6.46	2.19	0.53	0.71	0.06	0.06	0.07	0.21	0.28	1.06	0.96	1.04	1.01
23	3.58	3.4	4.7	5.38	6.62	6.76	5.31	5.83	9.02	3.43	5.27	1.92	2.31	3.78	5.91	6.93	7.3	5.59	2.07	0.6	2.16	0.06	0.06	0.07	0.07	0.3	0.84	0.64	0.64	0.86
24	3.84	3.42	4.54	5.09	5.47	5.42	1.62	2.26	1.99	3.11	3.87	2.59	2.74	3.46	5.63	6.95	21.91	4.5	2.13	2.38	2.25	0.16	0.17	0.07	0.06	0.26	1.71	1.43	1.5	1.27
25	3.27	3.35	3.82	4.6	6.19	6.11	4.94	6.14	5.51	2.56	3.13	4.75	8.22	7.95	9.96	14.63	13.75	15.26	6.5	2.83	2.26	0.23	0.15	0.22	0.35	0.06	0.07	0.4	0.41	0.41
26	3.21	3.23	3.53	3.7	1.02	0.88	1.4	2.11	2.01	2.41	2.64	3.88	3.87	8.2	7.9	11.2	18.8	20.85	15.73	4.84	7.48	1.89	2.12	0.75	2.09	0.07	0.06	0.06	0.06	0.08
27	3.03	3.27	3.76	3.83	0.88	1.03	1.39	2.17	2.14	2.39	2.57	7.86	4.86	9.36	8.87	11.89	19.12	22.18	18.29	5.12	7.91	0.85	2.57	0.75	5.38	0.37	0.07	0.06	0.06	0.06
28	3.03	3.24	3.77	3.67	0.83	1.14	1.39	2.09	2.28	2.43	2.57	7.91	8.12	9.05	9.54	12.79	18.24	20.75	18.57	4.91	9.67	4.37	2.2	0.71	1.91	0.39	0.06	0.06	0.06	0.06
29	3.04	3.08	3.52	3.85	1.03	1.07	1.38	2.03	1.92	2.11	1.87	7.77	8.73	9.08	7.83	12.51	17.65	19.53	17.64	5.06	8.29	0.8	0.96	0.86	1.62	0.35	0.09	0.06	0.07	0.06

Figure 3.22: Order algorithm with input size 20 nodes, varying edges and $n0 = 15$



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	4.38	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
1	50	3.63	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
2	50	50	2.94	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
3	50	50	50	2.29	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
4	50	50	50	50	1.82	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
5	50	50	50	50	50	2.66	17.94	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
6	50	50	50	50	50	4.85	1.23	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
7	50	50	50	50	50	50	50	0.97	50	8.77	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
8	50	50	50	50	50	50	50	46.54	0.86	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
9	50	50	50	50	50	50	50	4.78	12.01	0.72	15.46	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
10	50	50	50	50	50	50	50	50	34.68	11.4	0.67	38.66	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
11	50	50	50	50	50	50	50	50	50	50	26.11	0.71	50	21.02	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
12	50	50	50	50	50	50	50	50	50	50	50	32.44	0.63	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
13	50	50	50	50	50	50	50	50	50	50	50	8.36	50	0.65	10.1	9.93	50	50	50	50	50	50	50	50	50	50	50	50	50	50
14	50	50	50	50	50	50	50	50	50	50	50	50	47.87	5.14	0.56	2.32	50	50	39.08	50	50	50	50	50	50	50	50	50	50	50
15	50	50	50	50	50	50	50	50	50	50	50	50	50	3.83	2.16	0.53	19.22	16.6	49.26	50	50	50	50	50	50	50	50	50	50	50
16	50	50	50	50	50	50	50	50	50	50	50	50	50	50	17.8	7	0.45	5	8.93	7.86	48.08	50	50	50	50	50	50	50	50	50
17	50	50	50	50	50	50	50	50	50	50	50	50	50	50	16.77	7.02	3.55	0.46	2.24	7.76	33.93	50	50	50	50	50	50	50	50	50
18	50	50	50	50	50	50	50	33.37	50	50	50	50	50	50	12.51	13.63	7.75	2.17	0.45	3.72	11.44	37.04	50	50	50	50	50	50	50	50
19	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	6.85	7.14	3.05	0.48	2.51	7.1	36.47	50	50	50	50	50	50	50
20	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	18.43	19.62	9.43	2.47	0.42	10.52	50	50	50	50	50	50	50	50
21	50	50	50	50	50	50	50	45.5	38.48	50	39.69	50	50	50	50	50	49.36	50	16.13	6.02	8.37	0.77	3.23	6.69	33.5	50	50	50	50	50
22	50	50	50	50	50	50	50	41.7	35.85	50	32.04	36.15	50	50	50	50	50	50	40.6	15.98	25.26	6.9	0.41	9.63	32.47	50	50	50	50	50
23	50	50	50	50	50	16.89	50	14.45	24.06	50	26.78	44.53	42.58	50	50	50	50	50	50	50	44.63	7.03	11.95	0.41	4.39	50	50	50	50	50
24	50	50	50	50	50	50	50	50	23.1	50	19.72	33.59	37.87	44.26	50	50	50	50	50	50	50	15.32	6.26	3.45	0.39	50	29.69	49.86	50	50
25	50	50	50	50	50	50	50	50	18.04	48.06	18.26	30.87	32.75	42.09	50	50	50	50	49.43	50	50	50	50	50	33.15	0.54	3.73	7.31	50	50
26	50	50	50	50	50	50	50	24.37	19.58	33.03	18.04	28.3	34.39	36.99	50	50	50	50	46.69	50	50	50	50	50	22.12	5.21	0.47	34.54	7.13	31.95
27	50	50	50	50	50	50	46.25	18.1	20.21	50	16.84	28.26	31.52	30.26	50	50	50	50	46.58	47.65	50	50	50	50	18.53	6.93	3.55	0.46	1	5.22
28	50	50	50	50	50	50	50	15.82	13.47	33.61	15.32	20.33	25.23	32.3	39.91	39.85	48.66	50	45.55	46.74	50	50	50	50	50	33.82	6	1.03	0.46	5.57
29	50	50	50	50	50	25.19	47.21	20.28	13.37	36.41	15.68	22.77	23.29	26.87	33.18	42.18	44.94	47.32	44.52	45.52	50	50	50	50	50	50	31.09	6.28	4.23	0.43

Figure 3.23: Greedy algorithm with input size 40 nodes, varying edges and $n0 = 35$

In the second case we run models of size 40 nodes with setting $n0 = 35$ in Figure 3.23. Note that this is using the greedy enhanced algorithm, and the data should be compared to Figure 3.21.

Note that in both of the previous cases, the algorithm would converge quickly in case of n_0 values are most accurate. The main advantage of the delayed application is that a solution is returned even if it is of size less n_0. In complex cases, choosing n_0 wont help (as shown in the previous graphs) since the algorithm could still have to explore search branches promising potentially larger MCS, in which case the time out parameter could serve as the stopping condition.

3.8 Edge Density Scalability Test

To further understand the effects of edge density on our enhanced algorithm versions we run a different set of experiments. We generate models with the same number of nodes but with different edge density percentage. The edges are directed and a node can point to it self. i.e a model with n nodes can have from 0 to n^2 edges.

We perform the experiment using our enhanced algorithm version, on models of size (100,

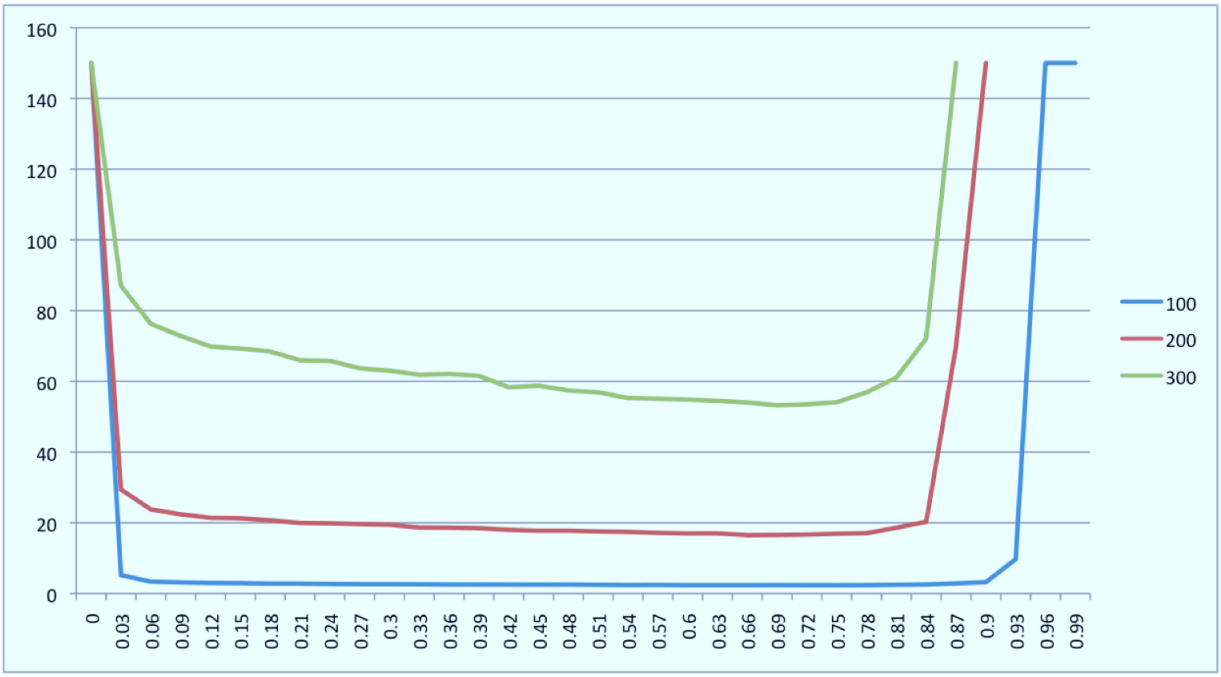


Figure 3.24: Effects of edge density on the algorithm's performance

200, 300) nodes. As shown in Figure 3.24, each point(x,y) represents comparing a model with density percentage x to it self, and y is the time for the algorithm to finish with a timeout of 150 seconds. We plot line by line the three sizes: (100, 200, 300) nodes, We notice how the worst running time happens when the edge density is 0 or above 90 percent. We discussed how higher edge density helps in pruning. However when all nodes have similar connectivity (as in the case of a complete clique) then this situation becomes similar to 0 density case. Since all nodes are similar, and pruning is minimal. This is shown in Figure 3.24.

Finally we note that when we tried the original CSIA implementation on the same type of models as above, results were almost always representing worst case (more than 150 seconds) for all densities of the 100 nodes models. This indicates the effect of our added heuristics.

3.9 Edit Script

In model comparison, the desired outcome of the algorithm is to represent the difference between the first and the second input models if it exists. In most cases the difference is represented with and edit script.

The CSIA algorithm (and any MCS algorithm) produce a mapping set (MS) which indicate which nodes of g1 are mapped to which nodes of g2. The Mapping set is then used tos generate the edit script which contains a Removed Elements Set (RS) and a Created Elements Set (CS).

The overall process generates three sets: MS, RS and CS describing an edit script which

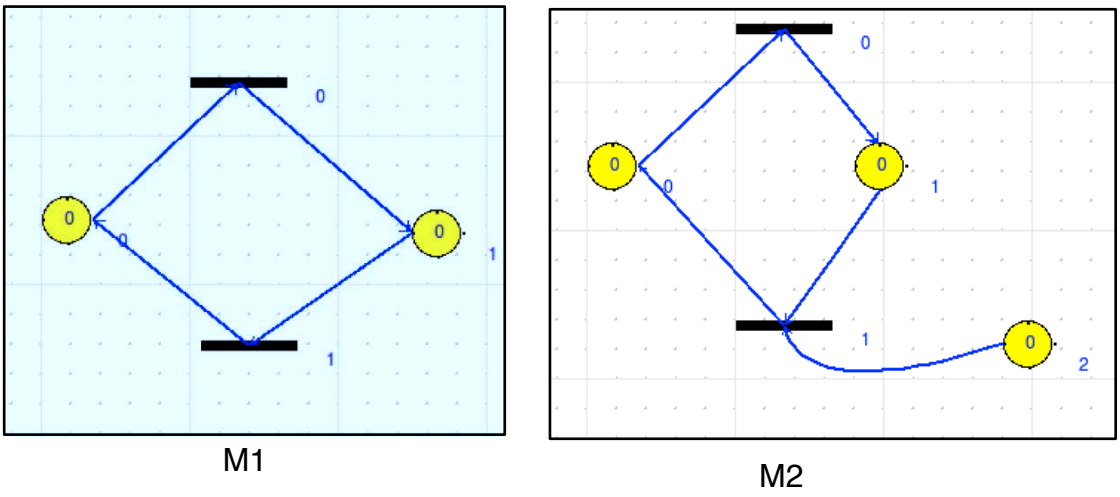


Figure 3.25: An example of An Edit Script

as a list of actions to produce M_2 form M_1 (the order matter), and there is two types of actions:

- Remove(e): indicate to remove the element e form M_1, where e could be a node or an edge between two nodes.
- Create(e): indicate to add the element e to M_1, where e could be a node or an edge between two nodes.

It should also be noted that the order of the actions in the edit script is important. An action should never reference an element which does not "yet" exist in the current model. For example the sequence: $[Remove(n_1), Remove(e(n_1, n_2))]$ is not correct, because n_1 would not exist after the first remove was executed, and the order should be reversed.

For example an edit script form M_1 to M_2 in Figure 3.25, is the following:

$[Create(place2), Create(transition1, place2)]$

Finally, sometimes the edit script could include a change/update action indicating that two elements match but some of their attributes need to be updated. This is very related to the equivalence criterion used when comparing nodes. The criterion could check the type equivalence, or go on into checking internal attributes of the nodes being compared.

3.10 Discussion

The problem of MCS greatly resembles the model comparison problem, as long as models are considered as graphs, with nodes and connections.

In model comparison, the desired outcome of the algorithm is to represent the difference between the first and the second input models. In most cases the difference is represented with and edit script.

The nodes different attributes could be used as in the nodes comparison function. For example, the function could determine if two nodes are comparable if both share the

same type (place or a transition in Petri Nets). Using only the type of the nodes as an indicator would clearly increase the number of possibilities to explore. It may be possible however, in some domains to include other properties of the nodes in the comparison function. In the context of comparing Petri Nets for example, two nodes are equal if they have the same type (place, transition) and the same name. One can go further and include the number of tokens a place has as part of the comparison criteria.

Our approach aims at giving the user more control over the accuracy of the comparison results through different algorithm parameters. Since our approach is based on MCS we argue that it is produces more accurate results, as opposed to using only a greedy strategy, since the algorithm examines all possible solutions through backtracking to find the most optimal one. However, this comes at a cost of longer processing times than greedy algorithms, which we address through customizations by the user such as: nodes compare function customizability, *time out* parameter, n_0 parameter (with delayed application). Finally a disadvantage to our algorithm is that it may exercise different running times on the same input. The reason is that greedy choices are not always deterministic enough for certain models. Also, when calculating a MCS, situations can arise where that the isomorphic rule is too restrictive and leads to a larger (though still correct) edit script. For example the edit script could be: (note n1, n2 have the same properties here and considered equivalent): $[remove(noden1), remove(edge(n1, n7), create(noden1), create(edge(n1, n5))]$, where the most optimal edit script is: $[remove(edge(n1, n7), create(edge(n1, n5))]$. This situation can be resolved by running specific optimization on the edit script.

3.11 Conclusion

In this chapter we have presented a new customizable approach to model comparison. We discussed the overlap between graph similarity and model comparison and proposed using Maximum Common Subgraph (MCS) algorithms to solve the model comparison problem. We chose one such algorithm and extended it using heuristics, and implemented it in Python language. Performance of the algorithm and various heuristics were evaluated using extensive experiments of model instances with different attributes. We attempted to characterize the requirements of model comparison in the context of testing model transformations. We then identified and implemented potential enhancements to the algorithm to increase its performance taking into consideration that it will be used in the context of model transformation testing. The results are promising, taken into account that model differences are typically small, within the context of model transformation testing. We will use this algorithm in our TUnit framework described in Chapter 4.

4

Chapter 4: TUnit, A Framework for testing Model Transformations

In this chapter we describe TUnit, our proposed framework for testing model transformations. We start off by describing the underlying formalism used in our framework in Section 4.1. Then we introduce the framework TUnit and describe the overall architecture in Section 4.2, where we describe the functionality of each of the components in more details. We present a case study (Traffic2PN transformation) in Section 4.3 which we use to demonstrate the effectiveness of TUnit in reducing testing efforts in Section 4.4, and in achieving semantic equivalence in Section 4.5. Finally we end the chapter with a discussion in Section 4.6.

4.1 Underlying Formalism

To implement the TUnit testing framework, and following the multi-paradigm philosophy of modelling everything explicitly, at the most appropriate level of abstraction, using the most appropriate formalism, we chose to use the Discrete Event system Specification (DEVS) rather than using a general object oriented code using a specific language. Recent work has shown its power and modularity in the context of complicated tasks such as model transformation [SV08]. Each of the DEVS blocks represent an encapsulated unit with a defined interface though its ports to communicate with other blocks. Using a well known formalism such as DEVS supports a platform independent approach. Platform dependent code generators can be used to generate code from a DEVS model description. Furthermore, the formalism introduces the notion of time, and hence can help formulate complex testing scenarios. Finally, several distributed environments for DEVS models simulation exist [SPB+04, CSPZ04, ZZH06, SKHP07], which can be used to enhance the performance of simulation (which in our approach corresponds to executing a test suite) among other benefits.

This section introduces the *Discrete EVent system Specification* (DEVS) formalism. The (DEVS) formalism was introduced in the late seventies by Bernard Zeigler as a rigorous basis for the compositional modelling and simulation of discrete event systems [Zei84], and been successfully applied to the design, performance analysis and implementation of a plethora of complex systems.

A DEVS model is either *atomic* or *coupled*. An atomic model describes the behaviour

of a reactive system. A coupled model is the composition of several DEVS sub-models which can be either atomic or coupled. Each sub-model could have *ports*, which are connected by channels, and are either *input* or *output*. Ports and channels allow a model to send and receive signals (events) between models. A channel must either connect:

- An output port of some model to an input port of another.
- An output port of a sub-model to one of its parent model output ports.
- An input port of a coupled model to an input port of one of its sub-models.

An **atomic DEVS** [1] model is a tuple $(S, X, Y, \delta^{int}, \delta^{ext}, \lambda, \tau)$ where S is a set of sequential **states**, one of which is the *initial* state. X is a set of allowed **input events**. Y is a set of allowed **output events**. There are two types of transitions between states: $\delta^{int} : S \to S$ is the **internal transition function**, $\delta^{ext} : Q \times X \to S$ is the **external transition function**, Associated with each state are $\tau : S \to \mathbb{R}_0^+$, the **time-advance** function and $\lambda : S \to Y$, the **output function**. In this definition, $Q = \{(s, e) \in S \times \mathbb{R}^+ \mid 0 \leq e \leq \tau(s)\}$ is called the **total state space**. For each $(s, e) \in Q$, e is called the **elapsed time**. $\mathbb{R}_0^+$ denotes the positive reals with zero included.

Informally, the operational semantics of an atomic model is as follows: the model starts in its initial state. It remains in any given state for as long as specified by the *time-advance* function result for state or until input is received on some port. If no input is received, after the state time-advance expires, the model first sends the output specified by the *output function* and then instantaneously jumps to a new state specified by the *internal transition function*. If input is received before the time for the next internal transition however, then it is the *external transition function* which is applied. The external transition depends on the current state, the time elapsed since the last transition and the inputs from the input ports.

The following definition formalizes the concept of coupled DEVS models. A **coupled DEVS**[1] model named D is a tuple $(X, Y, N, M, I, Z, select)$ where X is a set of allowed **input events** and Y is a set of allowed **output events**. N is a set of **component names** (or labels) such that $D \notin N$. $M = \{M_n \mid n \in N, M_n$ is a DEVS model (atomic or coupled) with input set X_n and output set $Y_n\}$ is a set of DEVS **sub-models**. $I = \{I_n \mid n \in N, I_n \subseteq N \cup \{D\}\}$ is a set of **influencer** sets for each component named n. I encodes the connection topology of sub-models. $Z = \{Z_{i,n} \mid \forall n \in N, i \in I_n . Z_{i,n} : Y_i \to X_n \; or \; Z_{D,n} : X \to X_n \; or \; Z_{i,D} : Y_i \to Y\}$ is a set of **transfer functions** from each component i to some component n. $select : 2^N \to N$ is the **select** or tie-breaking function. 2^N denotes the powerset of N (the set of all sub-sets of N).

The connection topology of sub-models is expressed by the influencer set of each component. Note that for a given model n, this set includes not only the external models that provide inputs to n, but also its own internal sub-models that produce its output (if n is a coupled model.) Transfer functions represent output-to-input translations between components, and can be thought of as channels that make the appropriate type translations. For example, a "departure" event output of one sub-model is translated to an"arrival" event on a connected sub-model's input. The *select* function takes care of

1. For simplicity, we do not present a formalization of the concept of "ports".

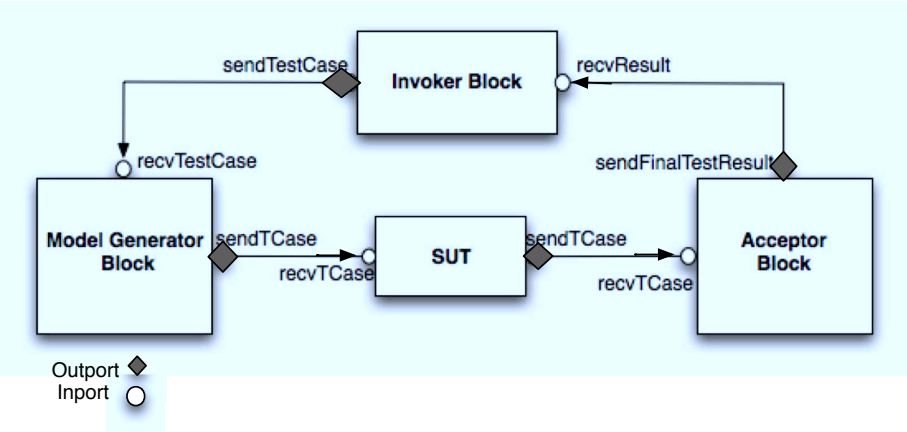


Figure 4.1: An Overview of TUnit

conflicts as explained below.

The semantics for a coupled model is, informally, the parallel composition of all the sub-models. A priori, each sub-model in a coupled model is assumed to be an independent process, concurrent to the rest. There is no explicit method of synchronization between processes. Blocking does not occur except if it is explicitly modelled by the output function of a sender, and the external transition function of a receiver. There is however a *serialization* whenever there are multiple sub-models that have an internal transition scheduled to be performed at the same time. The modeller controls which of the conflicting sub-models undergoes its transition first by means of the *select* function.

For this work, we use our own DEVS simulator called `pythonDEVS` [BV01], grafted onto the object-oriented scripting language Python.

4.2 Overall Components

Our DEVS-based testing framework contains several building blocks each made up of atomic or coupled DEVS models. Each block has a specific function as specified in this section. Communication between the blocks is achieved using the events *TestCase*, *TestResult*, and *FinalTestResult*.

Such events capture and encapsulate different attributes of a "test case" to be communicated throughout the framework. The overall framework components are shown in figure 4.1

4.2.1 Events

In the following we list all the DEVS events communicated in TUnit alongside their attributes.

TestCase

TestCase is the main event which encodes the test case specification needed for the framework to execute a test case. It contains the following attributes:

- **Input Model Name**: The name of the file containing the input model which will undergo the transformation.
- **Input Model**: An instantiated input model object to be transformed. Initially this attribute is Null until it gets set by the model generator block.
- **Expected Model Name**: The name of the file containing the expected output model. This is the output model which will be compared to the actual transformation output model. The attribute is set to Null initially to indicate that the test case has no expected model yet.
- **Expected Model**: The instantiated instance of the expected model. Initially the value is Null until it is set by the model comparator block (maybe by the model generator block).
- **Fragment Model Name**: The name of the file containing the model fragment which the output model should be tested against. The attribute is set to Null initially to indicate that the test case has no model fragment for the resulting model to be satisfied.
- **Output Model:** an attribute initially set to Null until is set to include the actual output model produced by the transformation in the SUT block.

ComparisonResult

This event is used to encode the outcome result of comparing an actual output model to and expected output model. ComparisonResult encodes the following information:

- **Input Model Name**: The name of the file containing the input model which constituted the test case. It is needed to match all the results events into a final test result event as we will show later.
- **Verdict**: A pass/fail attribute indicating whether the comparison was successful.
- **Edit Script**: The edit script generated by the comparison algorithm to indicate the mismatches if any.

CriteriaResult

This event is used to encode the outcome results of comparing an actual output model to a specified model fragment or to a post-condition. CriteriaResult encodes the following information:

- **Input Model Name**: The name of the file containing the input model which constituted the test case.
- **Verdict**: A pass/fail attribute indicating whether the comparison was successful.
- **Matching Errors**: Debugging errors to indicate which parts of the model did not conform to the fragment elements, if any.
- **Fragment Name**: The name of the file containing the model fragment to be used. In the case of a post-condition block, this attribute will reflect the block's name.

FinalTestResults

This event is used to encode the outcome results of a test case by composing the individual results from comparison block, fragment block, and the post conditions. It is the event which gets transmitted back to the invoker to finalize the test case. FinalTestResult encodes the following information:

- **Input Model Name**: The name of the file containing the input model which constituted the test case.
- **Comparison Result**: The event resulting from executing a comparison test on the output, if any.
- **Criteria Result**: The event resulting from executing a fragment test on the output, if any.
- **Post Condition Results**: A list of the events representing the results of applying a check of each post condition.

4.2.2 Invoker Block

The invoker block is the main controller of the framework execution. It initiates the testing process, by reading and executing the test specifications. The test suite is specified as a group of test case events. The invoker attempts to execute each test case, and collects the test results for final reporting.

It contains a list of test cases (encoded as events) to be executed by the framework. Each test case event contains the information needed perform the test evaluation. If the test case does not have any specified expected output model, or a model fragment, it will be evaluated solely based on post condition evaluations.

The Invoker block is an Atomic DEVS block which represents a scheduling for executing a list of test cases, where each test case contains the names of the files of the input model and their corresponding expectations.

It has two ports: *sendTestcase* (out-port) and *recvResult* (in-port).

The block proceeds to send the first test case containing the name of the file to be tested in *testCase* event from its *sendTestcase* out-port. Then it waits until it receives on its*recvResult* in-port, the test result event, containing the results and the verdict (pass/fail) with some debug information. This result received is encapsulated in an event called *finalTestResult*. The block stores all these *finalTestResult* events list for the executed test cases. The event's attributes were listed above in the events section. They contain information about the result and a description message of any mismatch errors, to help debugging. The Invoker block keeps sending test cases and collecting results until the list of all test cases has been fully executed.

4.2.3 Model Generator Block

This block is responsible for the first step in the process of executing a test case. It represents an abstraction of the procedure for loading input models (In general, the block could also generate input models on the fly randomly or according to some strategy),

which could be complex data structures, and may require complex loading procedures. It helps load the input model M using the specified file name in the test case event. The model is loaded to be compatible with the SUT under test. The model generator block could be customized to parse model files from any format it chooses to support, into an object that is compatible with the SUT. In the context of our case study, the SUT is a MoTif [SV08] model which understands $AToM^3$ [dLV02] Python representation of models. Such models could be represented as XML or using other format; and then translated into $AToM^3$ models by the model generator block. The tester could create these input models visually in a modelling environment and automatically generate their code, or could code them manually.

The model generator block is an atomic DEVS block which receives on its in-port *recvTestcase*, the testCase event. This event contains the file name of the input model that needs to be tested. This block is responsible for parsing the input model from the file with the specified name in the input. It then sends this parsed model object encoded in the *testCase* event to its out port *sendTestCase*.

4.2.4 SUT Block

The system under test is a model transformation implementation, and is treated as a function which takes a model M as input, and produces a transformed model M' as output. The SUT block serves as a wrapper for the Model Transformation function allowing it to be integrated into the TUnit framework. For the purpose of this case study, the model transformation is implemented using MoTif [SV08]. The block is intended as an abstraction of the transformation procedure, which in its own right could be composed of multiple steps or a chain of transformations.

The SUT container block is an Atomic DEVS block that represents a container for the transformation function/engine that need to be tested. It accepts on its input port *recvTestCase* the event testCase. The block then extracts the model object which is encoded in the event, and triggers the transformation function execution. It then waits for the transformation to finish, and encodes the resulting transformed model (output model) into the event testCase and sends it to its outport *sendOutputModel*.

4.2.5 Acceptor Block

The acceptor block is responsible for examining the transformed model M' to produce a verdict for the test case, namely pass or fail. The block uses the test specifications encoded in the test case event to determine which conditions the output model should satisfy. It contains the three types of oracle blocks that are used to evaluate a test case as discussed earlier: A comparison of the output model, a test against a model fragment and post conditions to satisfy.

The acceptor block is not an atomic DEVS instance, Rather it is a coupled DEVS instance which is a composite of several atomic DEVS building blocks. The actual testing and assertions on M' are performed in this block. A verdict on the test is determined in general by the collective results of the different evaluation criteria, and then encoded into a *finalTestResult* event to be transmitted back to the invoker.

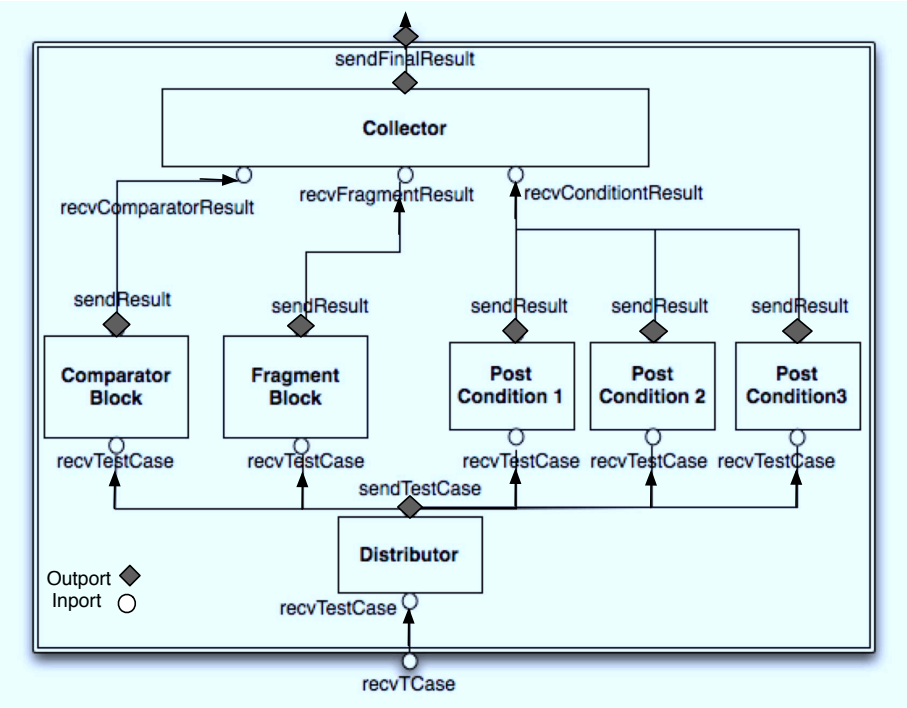


Figure 4.2: An overview of Acceptor block in TUnit

The block receives the *testCase* event, on its *recvTestCase* in-port. It then generates the verdict, and finally sends the *finalTestResult* event to its out-port *sendResult*. Figure 4.2) illustrates the different components of the acceptor block:

Distributor Block

The distributor block's job is to distribute the test case event to all evaluating blocks in order for the testing oracle configurations to be executed. In particular, the block will forward on its outport *sendTestCase* the test case event to:

- Comparator block;
- Fragment block;
- Each of the post-condition blocks.

The Distributer block will send the event to all blocks even those which do not apply. For example, even if a test case does not have a specified expected output model M_{exp}, it will be sent to the comparator block. It is the job of the individual blocks, comparator and fragment, to determine how to interpret and handle the specifications.

Comparator Block

The comparator block compares the actual output model M' with the expected model M_{exp}. The block waits in idle mode until it receive the testCase event on its *recvTestCase* in-port. If the event specifies the expected output model M_{exp}, it is then compared to the actual output model M', and a verdict is calculated and encoded into a **ComparatorResult** event (described earlier). The event is then sent in turn to out-port

sendResult. The block uses the modified CSIA algorithm described in chapter 3 to conduct the comparison and produce the edit script if any. However, if the testCase which the comparator block receives has no specified expected output model M_{exp}, then an empty **ComparatorResult** event is built and sent instantly to the collector block.

Fragment Block

The block load the fragment file, and checks if the output model M' corresponds to the specified model fragment. Model fragments are described as an approach to help address the oracle function problem in MDE [MBT08]. For our purposes we consider model fragments to be pattern rules which specify a set of models, similar to regular expressions for strings. They can also be thought of as a query which is applied on a model for checking the presence of a specific pattern.

The main motivation for using fragments is the complexity nature of models, as in most cases it is very hard to build exact expected models for each test case. Fragments can help in detecting specific errors in the transformation by considering very specific scenarios, and they can be reused and applied to a large group of models. The block provide flexibility as it can load fragments specifications from files applicable to certain test cases. This abstraction could be modified to allow testing a list of fragments against a single test case.

When the *testCase* event specifies a fragment to be evaluated, the block proceeds by loading the fragment, evaluating whether the output model conforms to the fragment and encodes the result in a *criteriaResult* event (described earlier). Once done, the block sends the event to its out-port *sendResult*. However, if no fragment was specified, then the block builds an empty criteriaResult event instantly and forwards it through the same out-port.

Post-Condition Block(s)

These blocks are the same as the fragment block, however they use a fixed fragment per block to represent a post condition which applies to all models.

Each post-condition block is an Atomic DEVS, it contains an in-port *recvTestCase* to receive the model thats encoded in the event, and execute the criteria check.

The block encodes a *criteriaResult* event with the result of the checks as a verdict and the error messages if applicable. It also has an out-port *sendResult* which it uses to send the event. An excerpt of the containsNoTrafficLight post-condition is shown below

```
1
2   class Containstrafficlight:
3     def check(self, graph):
4       ''' Returns Result, a description string of what went wrong
5       '''
6
7       C = {}
8       N = {}
9       C[1] = 'TR_TrafficLight'
10      N[1] = []
11
12      # with the LHS label as key
13      for label in C:
```

```
14          if graph.listNodes.has_key(C[label]):
15              N[label] += graph.listNodes[C[label]]
16
17
18      # Check if all nodes types exist in the host graph
19      for k in N:
20          if not N[k]:
21              return False,"Criteria "+ self.name + "failed:missing nodes"
22
23      # Verify links between nodes
24      M = {} # holds all the matched nodes
25      for tr_trafficlight1 in N[1]:
26          M[1] = tr_trafficlight1
27          break
28
29      # check if all the nodes are matched
30      if len(M) != len(N):
31          return False,"Criteria "+self.name+" failed: mismatch"
32
33      return True, "Criteria " +self.name+" was matched successfully."
```

Listing 4.1: Post-condition as a query in TUnit

Collector Block

The collector block acts a synchronizer to ensure that all evaluating blocks have finished processing so a verdict on a test case can be made. The total number of postconditions is indicated to the block at initiation time. It is important for this block to expect the correct number of post conditions events, in order for the framework to work correctly. The block is represented by an Atomic DEVS block, that contains three in-ports:

- **recvComparatorResult** receives the test result outcome of the model comparator block.
- **recvFragmentResult** receives the fragment evaluator result from the corresponding block.
- **recvConditionResult** receives *all* the of the post-conditions results.

The block builds the event *finalTestResult* by composing the above results, only when it receives all expected events from all blocks. Once all the final result are ready the block transmits the event onto its out-port *sendFinalTestResult* that is connected to the Acceptor block's out-port *sendResult* to make its way back to the invoker block. It then waits in idle mode for the next test case.

4.2.6 Handling Errors

When dealing with models and transformations, running the test suite becomes more complicated than in normal code. It is a multi-stage process with many dependencies, and several potential points of failure. TUnit deals with errors in any test case in the following manner: If an exception happens in any stage of the process then the flag valid, in the test case, is set to *False* to indicate to any further processing that this test case is invalid. The framework will continue processing any other test cases from the test suite after logging the errors. Also, the exception type and message is pushed onto the test case event for easier debugging.

Potential errors in the framework execution include:

1. **Loading of models:** Since models are complex data structures with many dependencies. The model generator block could fail to load the input model properly. The framework encodes the exception message within the test case event, and sets a valid flag to False. Subsequent processing of this test case by other blocks would detect the flag and don't attempt further processing as the result gets propagated back to the invoker block.
2. **Transformation Errors:** For the purpose of this work we will treat the transformation as a black box, there is however recent work describing in details exception handling in model transformation [SKV].
3. **Comparison Errors:** The comparator block is responsible for loading the expected output model of the transformation and executing the comparison algorithm, both of which could lead to exceptions. Exceptions are handled in the same way though setting a valid flag to *False* and encapsulate it in the test case event.
4. **Fragment and post conditions Errors:** Same as above.

4.3 Case study

In this case study we describe transformation from a domain specific language we created, called "Traffic" to the well known Petri-Net formalism. We start by describing the meta models of the source and target formalisms, followed by describing the transformation. An example of a simple transformation is then shown.

4.3.1 Petri-Net meta model

Petri nets is a modelling formalism that is most useful for analysis of concurrent systems [Mur89].

Petri nets consist of two types of entities: places, and transitions. Places have two attributes:

- a name to distinguish different instances,
- a positive integer representing the number of tokens within the place.

Transitions on the other hand have only a name to distinguish each instance. Finally, a place can have links to transitions through the relationship pl2tran, and transitions can have links to places through the relationship tran2pl, as shown in the Petri-Nets meta model in Figure 4.3. The PN meta model shown is described using Entity Relational Diagrams formalism.

See also Figure 4.4 for example instances of the visual syntax of Petri nets. The places are represented visually as circles with an integer in the centre representing the number of tokens the place currently contains. The transitions are represented using horizontal (or vertical) bars. Note that a place can never be linked directly with a place, and the transitions can never be directly linked to transitions either.

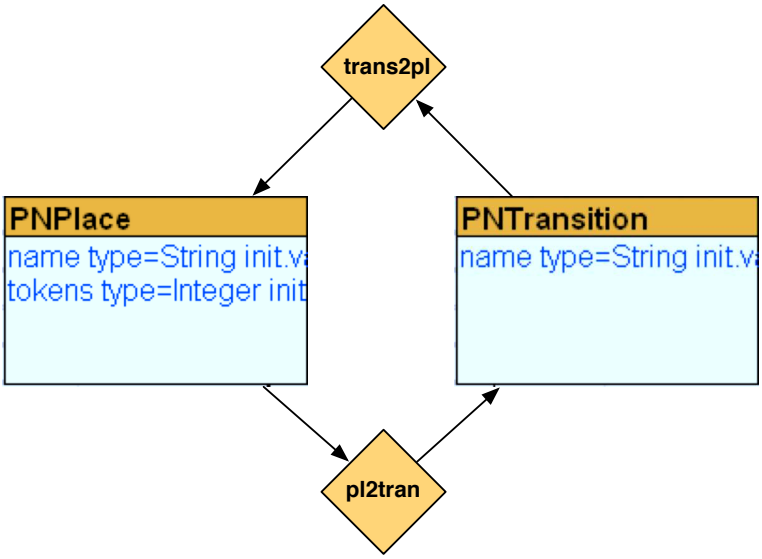


Figure 4.3: Petri-Nets Meta Model, expressed as an E/R model

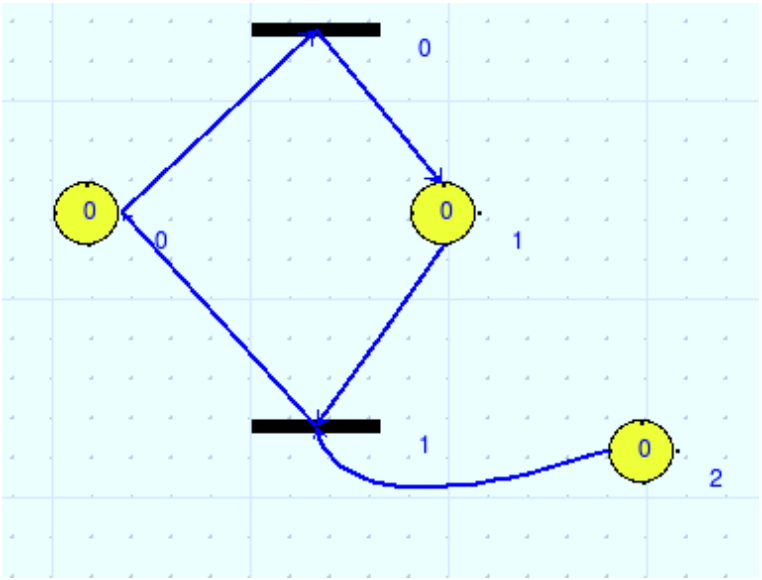


Figure 4.4: Petri-Nets visual concrete syntax

4.3.2 Traffic meta model

Traffic is a specific domain we created to model simple traffic networks. The Meta Model in figure 4.5 (described using a UML class diagram) contains the following entities:

- *TR-RoadSegment*: Represents the simplest building block of traffic roads, namely the road segment which contains the following attributes: `CarCapacity` as an integer to indicate how many cars can fit on the specified road segment, `Occupied` as a boolean to represent if the segment is occupied with cars, name to distinguish the road segment instance, and finally `numCars` as an integer representing the number of cars currently present in the segment. When a carCapacity is set to -1, we call infinite capacity road segments, this indicate a road segment that can accept cars indefinitely.
- *TR-Outport*: Represents the out-port for road segments which helps to identify the exit point where the road segment can link to other entities.
- *TR-Inport*: Similar to the previous except that it represents the entry point where other entities can link to the attached segment. Also, it gives a sense of direction of car movement over the different segments.
- *TR-Generator*: Represent a source for generating cars in the traffic network. It has a `name` attribute to distinguish its identity.
- *TR-Join* : A special type of road segment, it represents a lane merge between two road segments into one. It also has knowledge about direction through its links relationships.
- *TR-TrafficLight*: represents a traffic light as barrier for entering cars. It has a `name` attribute to distinguish it from other instances.

The Meta Model also describes the following relationships, which determine the allowed associations among the previous list of entities:

- RoadSegmentOutPort: indicates that a road segment entity can be linked to not more than one out-port entity.
- RoadSegmentInPort: indicates that an out-port entity can be linked to not more than one road segment entity.
- RoadConnection: indicates that each out-port entity can be linked to not more than one in-port entity.
- GeneratorOutPort: indicates that a car generator entity can be linked to not more than one out-port entity.
- JoinInPorts: indicates that up to two in-ports entities can be linked to a segment join entity.
- JoinOutPort: indicates that each segment join entity can be linked to not more than one out-port entity.
- TrafficLightBarrier: indicates that a traffic light barrier entity could be linked to control many in-port entities.

The concrete visual syntax of the traffic formalism is shown in Figure 4.6.

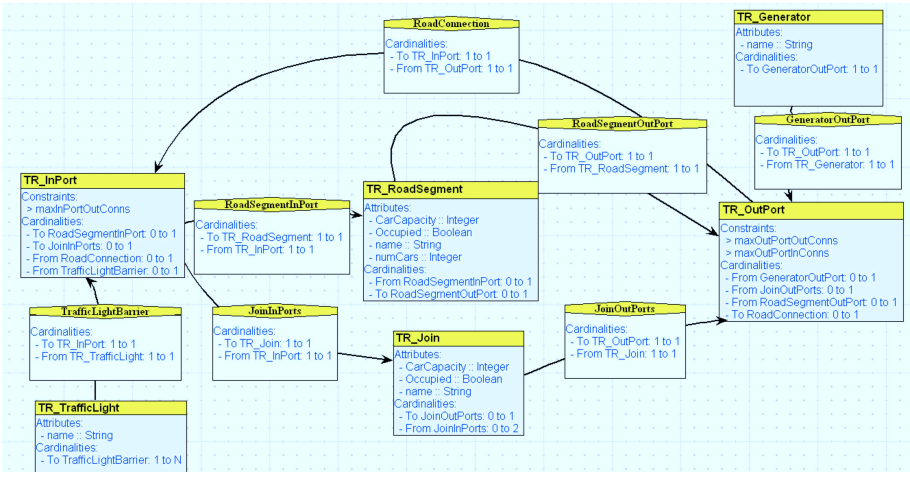


Figure 4.5: The Traffic formalism Meta Model

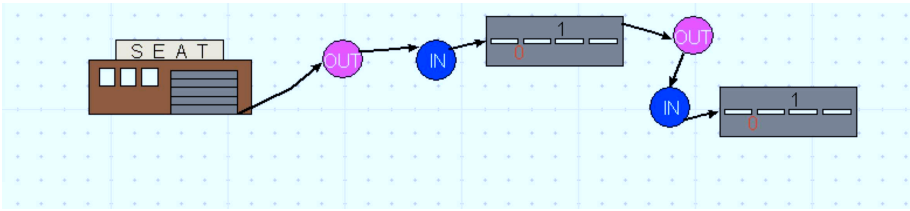


Figure 4.6: Traffic formalism visual concrete syntax example

4.3.3 Traffic to Petri-Net transformation

To define the semantics of the traffic formalism we choose to map it to the PN formalism which allows the use of existing analysis techniques for PN models.

This transformation accepts instance models of the Traffic formalism and produces models of the PN formalism. It is described using 11 rules illustrated in Figure 4.7 and Figure 4.8. Note that the transformation makes use of a helper formalism *Generic Graph* which allows one to connect arbitrary entities to accommodate intermediate steps during the process. The rules used are described in the following:

- Light to PN: The rule appends the equivalent of a traffic light as a Petri Nets model. It will also keep a temporary generic graph link attached to the traffic light.
- RoadSegment 2 PN: The rule appends the equivalent of a road segment as a PN model. It keeps two generic links to the road segment in-port and out-ports to keep direction.
- Infinity to PN: Appends the PN representation of an infinite capacity road segment, and keeps two generic links to the segment's ports.
- Generator to PN: Does the same with a generator.
- Join Generator: Will attempt to link the PN representation of the generator to the PN representation of a road segment entry.

- Join Roads: attempts to link the PN representation of the road segments.
- Join Light: attempts to link the PN representation of the traffic light to the PN representation of a road segment entry.
- Complete Collector: finishes the collector entities transformation.
- Remove Generator: removes any traces of the traffic generator entities left in the starting model.
- Remove Roads: remove any traffic road entities left in the starting model.
- Remove Lights: remove any traffic light entities left in the starting model.

The rules are applied in order of priority, in the same order as listed above. For example, the transformation engine attempts to apply Rule1 until it cannot be applied any more. Then it applies Rule2, until it can't be applied any more and so on. It terminates when no rule can be applied on the model. The final model is the resulting model.

See Figure 4.9 for a demonstration of a simple example to show how the transformation applies.

For now we will treat the transformation mechanism as a black box, namely it will be executed using the above mentioned rules embedded in MoTif. MoTif will specify the control flow of the different rule application until the final model is completely produced and the transformation is done.

4.3.4 Example Test Cases

To test the Traffic2PN transformation we need to list test cases and demonstrate how the TUnit framework could help in testing. This will include specifying some input models, fragments, and expected output models. Also, we will include post conditions which will apply to all test cases.

To test the correctness of the implementation of this transformation, we attempt to build a test suite based on different criteria.

We will build the models for these test cases visually and generate their code using $AToM^3$. We presented in chapter 2 existing approaches to test case generation, we don't follow an existing method exactly step by step. However, we use a similar approach to cover different attributes and association values. Test case generation is a complementary companion to TUnit (which focuses on execution framework). TUnit abstraction blocks can load models generated by other means, such as a model generation tool, as discussed earlier.

In the following we list the test cases we will use to test the Traffic2PN transformation.

Comparator Test Cases

We build a few test cases to validate the Traffic to Petri-nets transformation. Each test case has a specified input model and a specified output model. The following is a list of those test cases:

1. Generator to Segment to Infinity Traffic model: This model involves three traffic entities linked together through single links representing a generator which gener-

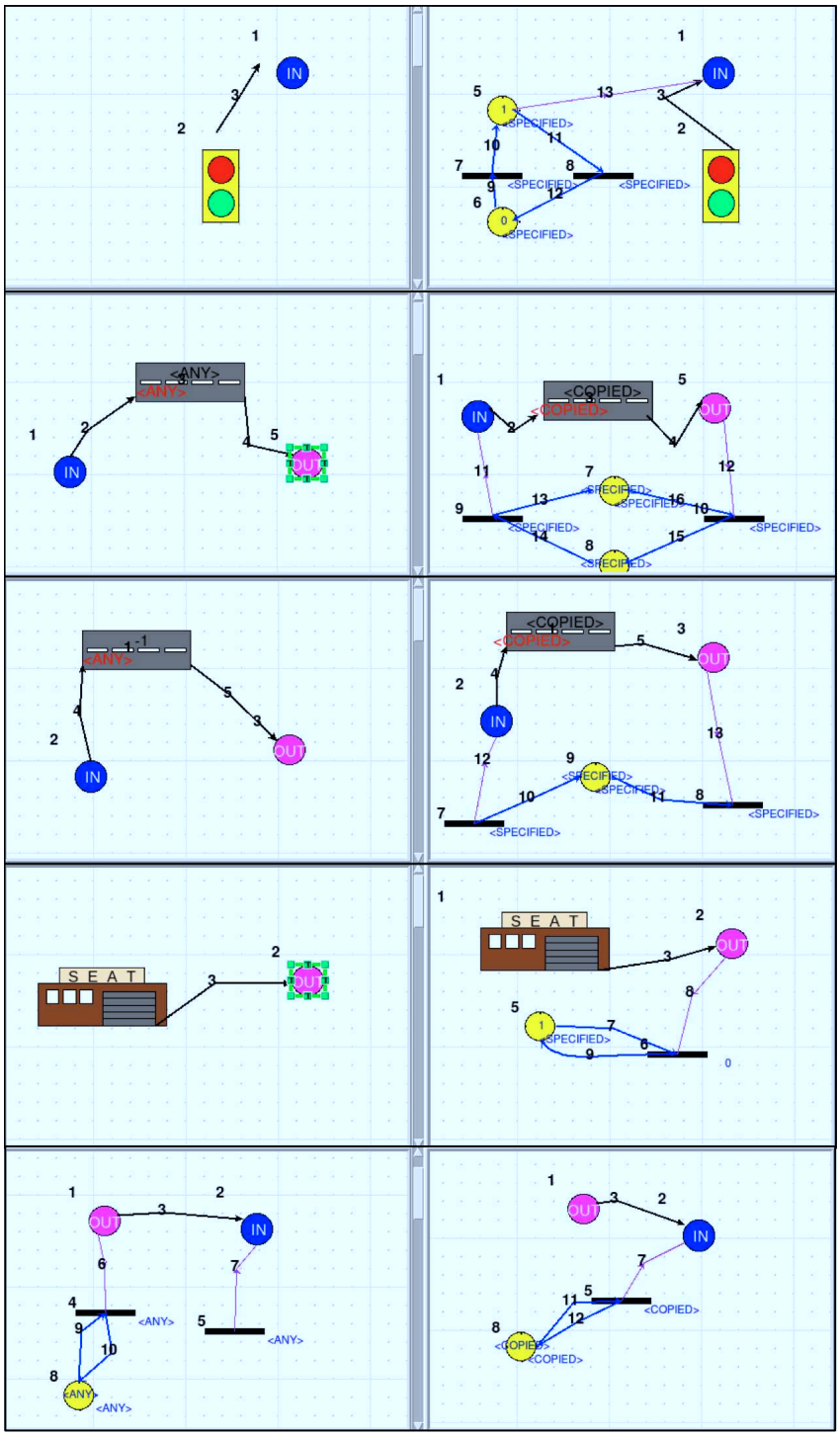


Figure 4.7: Traffic to Petri-Nets transformation rules part 1

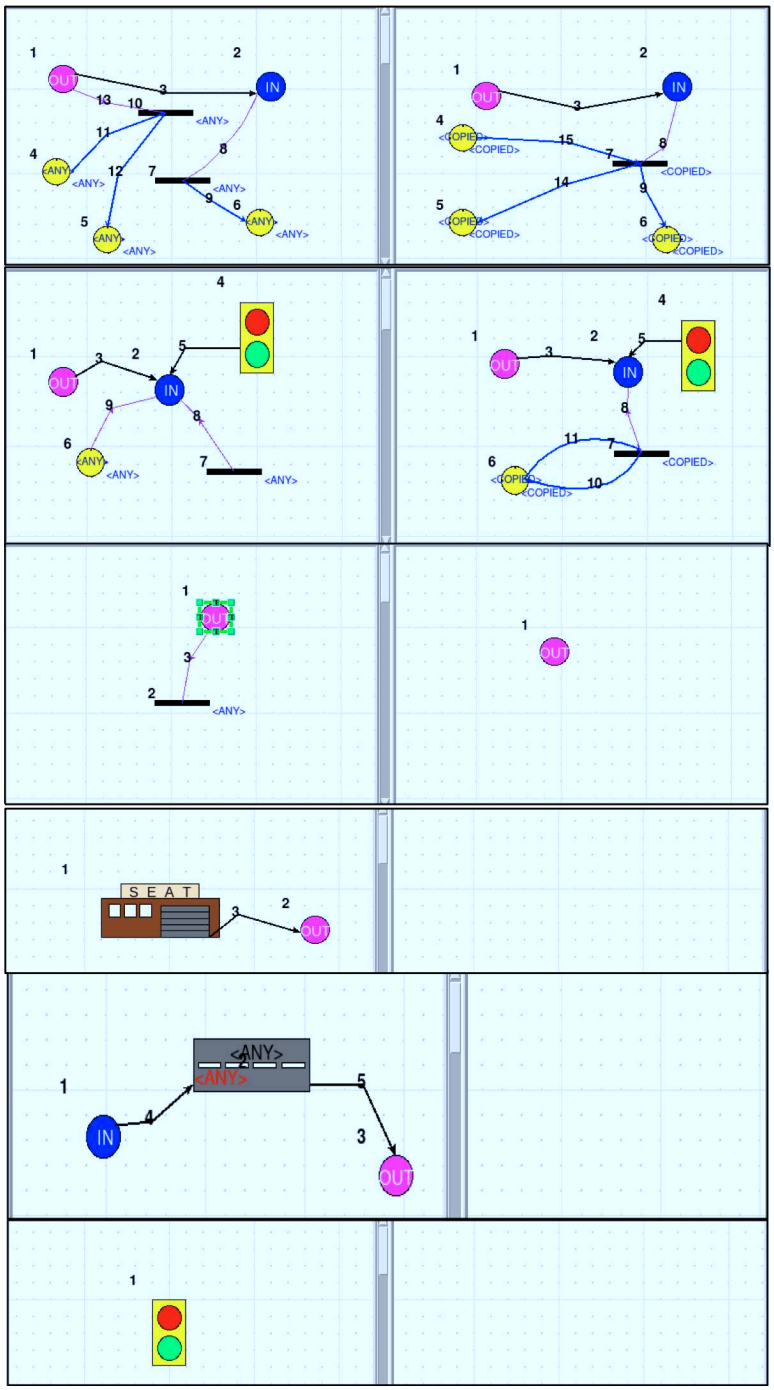


Figure 4.8: Traffic to Petri-Nets transformation rules part 2

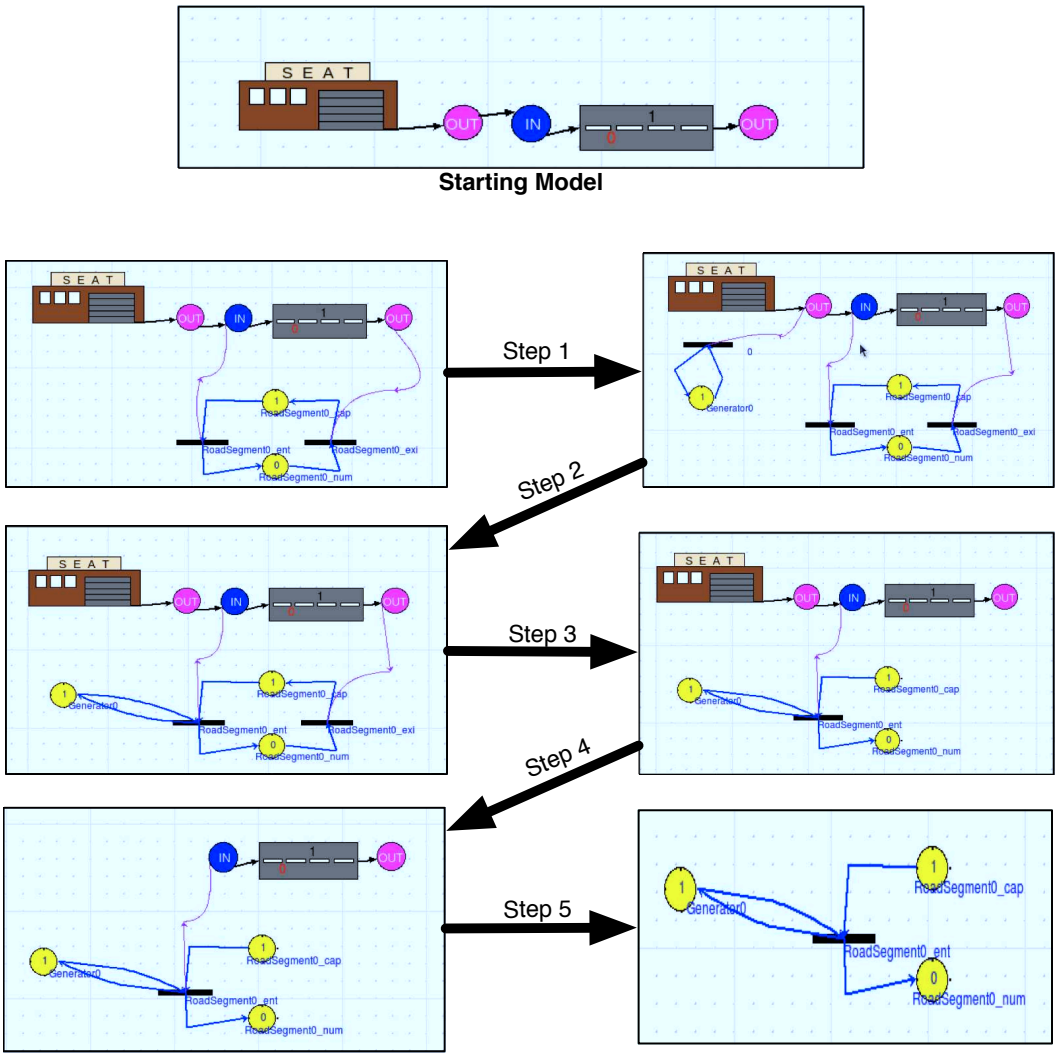


Figure 4.9: Traffic to Petri-Nets Transformation Example

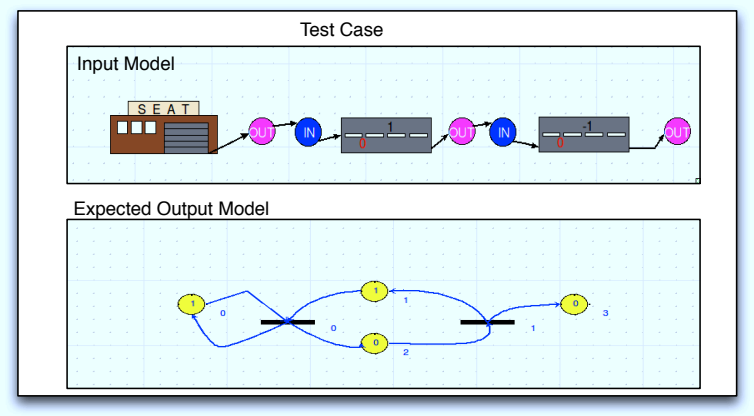


Figure 4.10: Test Case 1 : Input and Expected Output models

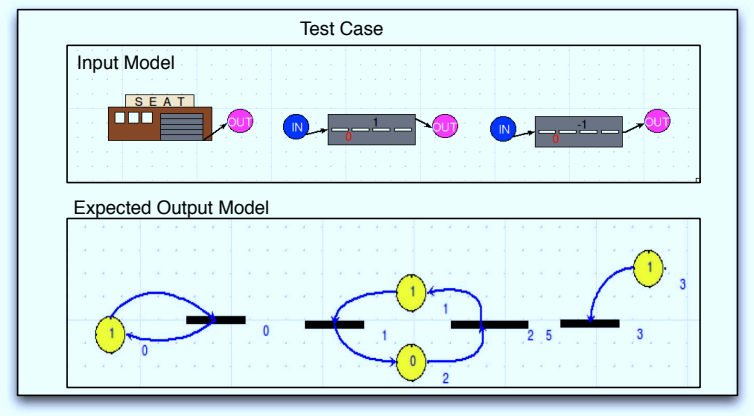


Figure 4.11: Test Case 2 : Input and Expected Output models

ates cars into a road segment and then to an infinity segment in Figure 4.10

2. Generator Segment Infinity No Links Traffic model: Same as before but no links exist between the elements in Figure 4.12.
3. Generator To Light To Segment To Infinity: Same as the first model, but the generator to segment link is controlled by a Traffic Light element in Figure 4.12.

Post Conditions Test Cases

We specify the following post conditions to be applied towards all test cases output models:

1. No Traffic model elements (light, road segment, road connectors, generators or other link entities) present.
2. All elements are Petri Nets instances (overlaps with the previous condition).
3. All elements have a specified id attribute.

The post conditions can be expressed in different ways, in most cases using OCL (Object Constraints Language) [KW00]. OCL is also used in most cases to describe any

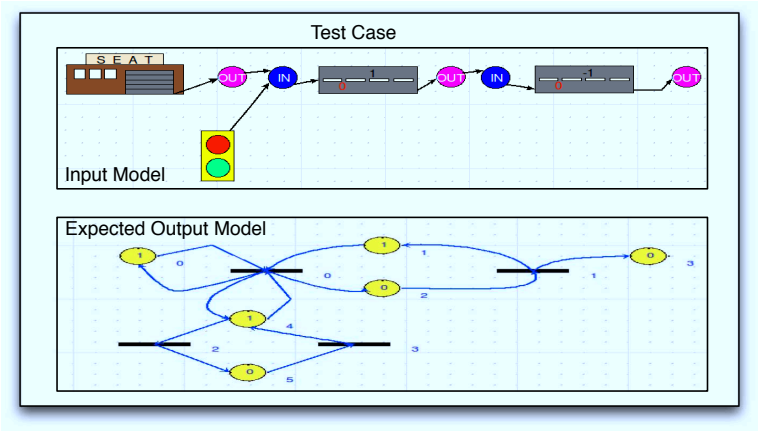


Figure 4.12: Test Case 3 : Input and Expected Output models

constraints on the meta models. Depending on the expressiveness of the language used to describe them, post conditions or model fragment can be very powerful. For this work, we implement post conditions and fragments in two ways: Python code which can navigate model elements to determine a matching criterion, and using a LHS (left hand side) Rule pattern which is similar to conducting a Query on the model to determine existence of some criteria (we show an example in the next section for a fragment test case).

All test cases will be tested against these post conditions.

Fragment Test Cases

The fragment we choose to implement has the input model: "Generator to segment to infinity traffic two tokens" which has a generator linked to a road segment capacity of two cars, linked to an infinity segment. For this test case we don't specify an expected output model, but instead we specify a model fragment: "place with two tokens" to match it. The fragment will look for a PN place in the output model which has two tokens.

The output model will also be tested against the previously mentioned post-conditions.

4.4 Framework Demonstration

After deciding on the test suite to be used, including for each test case an input model, and an expected outcome (model or fragment), we will demonstrate how to configure the framework and show traces of the test suite execution. As we will show in this section, we demonstrate the advantage of having TUnit in a realistic "test-fix-retest" engineering process, where we find bugs after running tests, fix them and re run the test suite to ensure tests pass.

4.4.1 Configuration

Models are modelled visually in $AToM^3$ and then compiled into files with specific names. Test cases specified in the format including input file name, expected model file name, fragment file name. They are encoded into a list of *Test Case* events each representing a simple case. The code specifying the test suite, describes four test cases for each in the form: TestCase(input model name, expected output model name, expected fragment name)

We show the python code used to specify them in the following Listing:

```
# Test Case 1
input_list.append(
  TestCase("generator_to_seg_to_infinity_traffic",
   "generator_to_seg_to_infinity_pn",
   None))
# Test Case 2
input_list.append(
  TestCase("generator_seg_infinity_no_links_traffic",
   "generator_seg_infinity_no_links_pn",
   None))
# Test Case 3
input_list.append(
  TestCase("generator_to_light_to_seg_to_infinity_traffic",
   "generator_to_light_to_seg_to_infinity_pn",
   None))
# Test Case 4
input_list.append(
  TestCase("generator_seg_to_infinity_traffic_two_tokens",
   None,
   "has_places_two_tokens"))
```

Listing 4.2: Python code showing the specification of the tests to be run

Note that in Listing 4.2, we specified 4 test cases, 3 of which have a specified expected output model and no fragments, and one has a fragment with no expected model specified. This indicates that:

- Test 4 will not be processed by the model comparator block since it does not have a specified model
- Test 4 will be the only test to be processed by the fragment block, using the specified fragment name.
- All Tests will be processed by every post condition.

In the model comparator block we use the model comparison algorithm described earlier in chapter 3. The algorithm lets us define the minimum size allowed for the match n_0, and the time we allow the comparison to run.

We pass to the SUT block a callable reference to the transformation function which is being tested, namely our rule based implementation of the "Traffic2PN" transformation. The callable should be able to return the resulting transformed graph from the transformation.

4.4.2 Execution

We start executing the framework environment to process the test cases, and collect the results. The framework as earlier specified proceeds to execute the test cases in the test suite one by one. We ran the framework using our Python DEVS platform. We will present the output we got in the following sections

4.4.3 Results

We ran the framework on the specified input test cases, and the Traffic2PN transformation described earlier, the following was a sample output of framework:

```
1  ********************************************************
2  Invoker Block: Sending Next Test Case
3    Loading Model.....
4      Caught an exception in block Model_Generator_Block
5        Type: <type exceptions.ImportError>
6        Message: No module named generator_seg_to_infinity_traffic_two_tokens
7    Transforming None.....Invalid Test Case
8    Post-Condition::No_Traffic_Elements_Rule::.....None,Invalid Test Case
9    Post-Condition::All_PN_Elements_Rule::.....None,Invalid Test Case
10   Post-Condition::All_Elements_ids_Rule::.....None,Invalid Test Case
11   ModelComparator::.....None,Invalid Test Case
12   FragmentComparator::.....None,Invalid Test Case
13 Invoker Block : Received Test Result:
14       ********************************************************
15 Invoker Block: Sending Next Test Case
16   Loading Model.........done
17   Transforming <generator_to_light_to_seg_to_infinity_traffic >.....done
18   Post-Condition::No_Traffic_Elements_Rule::..........Fail,Check Errors
19       Errors:Found Traffic Nodes of type:
20        TrafficLightBarrier, RoadConnection,
21   Post-Condition::All_PN_Elements_Rule::..........Fail,Check Errors
22       Errors:Found Non PN Nodes of type:
23        TrafficLightBarrier, RoadConnection, GenericGraphEdge,
24   Post-Condition::All_Elements_ids_Rule::..........Fail,Check Errors
25       Errors:Found Nodes with no ID of type:
26        pl2tran, tran2pl, GenericGraphEdge, PNPlace, PNTransition,
27   ModelComparator::.........Failed, check Edit Script
28   FragmentComparator::.........none
29 Invoker Block : Received Test Result
30   ********************************************************
31 Invoker Block: Sending Next Test Case
32   Loading Model.........done
33   Transforming <generator_seg_infinity_no_links_traffic >.....done
34   Post-Condition::No_Traffic_Elements_Rule::..........Pass
35   Post-Condition::All_PN_Elements_Rule::..........Fail,Check Errors
36       Errors:Found Non PN Nodes of type:
37        GenericGraphEdge,
38   Post-Condition::All_Elements_ids_Rule::..........Fail,Check Errors
39       Errors:Found Nodes with no ID of type:
40        pl2tran, tran2pl, PNPlace, GenericGraphEdge, PNPlace, PNTransition,
41   ModelComparator::.........Failed, check Edit Script
42   FragmentComparator::.........none
43 Invoker Block : Received Test Result
44   ********************************************************
45 Invoker Block: Sending Next Test Case
46   Loading Model.........done
47   Transforming <generator_to_seg_to_infinity_traffic >.....done
48   Post-Condition::No_Traffic_Elements_Rule::..........Fail,Check Errors
49       Errors:Found Traffic Nodes of type:
50        RoadConnection,
```

```
51   Post-Condition::All_PN_Elements_Rule::.....Fail,Check Errors
52      Errors:Found Non PN Nodes of type:
53       RoadConnection, GenericGraphEdge,
54   Post-Condition::All_Elements_ids_Rule::.....Fail,Check Errors
55      Errors:Found Nodes with no ID of type:
56       pl2tran, tran2pl, PNPlace, GenericGraphEdge, PNTransition,
57   ModelComparator::.....Failed, check Edit Script
58   FragmentComparator::.....none
59 Invoker Block : Received Test Result
60   ******************************************************
61 Invoker Block: Finished Processing All Test Cases
```

Listing 4.3: The TUnit output trace running the sample test suite

The output is formatted as follows: Indicate the processing stage for each test case and its results, i.e., sending Test Case, Transformation, Post-Conditions, Expected Model Comparison, Fragment Comparison. If some steps are not applicable, print none. Also, if an error occurs, indicate the exception type and message. For example in Listing 4.3 line 3-6, when TUnit failed to find the specified input model named: "segment to infinity traffic two tokens", it indicated the exception and skipped over the subsequent steps for that specific test case.

Bug1-Visual Links

After examining the output log we got when we ran the framework on our test suite, we noticed few errors. We start by examining the post conditions failures in our test cases, since they apply to all test cases. Note that in the above listing the edit scripts were not shown for simplicity. However, we have shown a sample edit script in Section 3.9.

First notice the post condition "No-Traffic-Elements-Rule" is failing, indicating that there are traffic elements left in the output of the transformed model. Notice how the elements types are links, such as *e.g.,* TrafficLightBarrier, RoadConnection and GenericGraphEdge.

Intuitively, we consider that the transformation is working on the concrete syntax layer of the models. In particular, the following two rules: **Remove Roads** and **Remove Light**, proceed to remove the visual traffic elements, but do not specify what happens to the "link elements" pointing to the removed elements explicitly. Visually these links are not shown since we need both ends of a link to be present for it to be visible. Subsequently when the rules remove these elements, they remove one end of the link while leaving the other present.

We modify the rules **Remove Roads** and **Remove Light** to ensure removing any attached linking elements (Traffic links and Generic Graph links). We then re-run our test suite to get the Listing 4.4. Note that now the first two post conditions pass on all valid test cases.

```
1 ******************************************************
2 Invoker Block: Sending Next Test Case
3   Loading Model.....
4     Caught an exception in block Model_Generator_Block
5       Type: <type exceptions.ImportError>
6       Message: No module named generator_seg_to_infinity_traffic_two_tokens
7   Transforming None..... Invalid Test Case
8   Post-Condition::No_Traffic_Elements_Rule::.....None,Invalid Test Case
9   Post-Condition::All_PN_Elements_Rule::.....None,Invalid Test Case
```

```
10   Post-Condition::All_Elements_ids_Rule::.....None,Invalid Test Case
11   ModelComparator::.....None,Invalid Test Case
12   FragmentComparator::.....None,Invalid Test Case
13 Invoker Block : Received Test Result:
14   ***************************************************
15 Invoker Block: Sending Next Test Case
16   Loading Model.....done
17   Transforming <generator_to_light_to_seg_to_infinity_traffic >.....done
18   Post-Condition::No_Traffic_Elements_Rule::.....Pass
19   Post-Condition::All_PN_Elements_Rule::.....Pass
20   Post-Condition::All_Elements_ids_Rule::.....Fail, Check Errors
21      Errors:Found Nodes with no ID of type:
22       pl2tran, tran2pl, PNPlace, PNTransition,
23   ModelComparator::.....Pass
24   FragmentComparator::.....none
25 Invoker Block : Received Test Result:
26   ***************************************************
27 Invoker Block: Sending Next Test Case
28   Loading Model.....done
29   Transforming <generator_seg_infinity_no_links_traffic >.....done
30   Post-Condition::No_Traffic_Elements_Rule::.....Pass
31   Post-Condition::All_PN_Elements_Rule::.....Pass
32   Post-Condition::All_Elements_ids_Rule::.....Fail, Check Errors
33      Errors:Found Nodes with no ID of type:
34       pl2tran, tran2pl, PNPlace, PNTransition,
35   ModelComparator::.....Failed, check Edit Script
36   FragmentComparator::.....none
37 Invoker Block : Received Test Result:
38   ***************************************************
39 Invoker Block: Sending Next Test Case
40   Loading Model.....done
41   Transforming <generator_to_seg_to_infinity_traffic >.....done
42   Post-Condition::No_Traffic_Elements_Rule::.....Pass
43   Post-Condition::All_PN_Elements_Rule::.....Pass
44   Post-Condition::All_Elements_ids_Rule::.....Fail, Check Errors
45      Errors:Found Nodes with no ID of type:
46       pl2tran, tran2pl, PNPlace, PNTransition,
47   ModelComparator::.....Pass
48   FragmentComparator::.....none
49 Invoker Block : Received Test Result:
50   ***************************************************
51 Invoker Block: Finished Processing All Test Cases
```

Listing 4.4: The TUnit output trace running the test suite after fixing bug1

Bug2-No-Links

After fixing the first bug, we notice in the Listing 4.4, we notice that the model comparator step fails for the test case "generator segment infinity no links traffic" (the test case is shown in Figure 4.11).

After investigation, we notice that the Rule **Complete Collector** is not specific enough. We modify it to check for outports only attached to an infinity segment.

This will avoid it being applied to outports attached to road segments. This is only visible in test case "Generator Segment Infinity no links".

We fix it and re-run the test suite to find that the test cases pass now. Furthermore, our changes did not break the test cases which passed before the changes, as shown in Listing 4.5.

```
1 ***************************************************
```

```
2 Invoker Block: Sending Next Test Case
3   Loading Model.....
4     Caught an exception in block Model_Generator_Block
5         Type: <type 'exceptions.ImportError'>
6         Message: No module named generator_seg_to_infinity_traffic_two_tokens
7   Transforming None..... Invalid Test Case
8   Post-Condition::No_Traffic_Elements_Rule::.....None..... Invalid Test Case
9   Post-Condition::All_PN_Elements_Rule::.....None..... Invalid Test Case
10  Post-Condition::All_Elements_ids_Rule::.....None..... Invalid Test Case
11  ModelComparator::.....None..... Invalid Test Case
12  FragmentComparator::.....None..... Invalid Test Case
13 Invoker Block : Received Test Result:
14  ******************************************************
15 Invoker Block: Sending Next Test Case
16  Loading Model.........done
17  Transforming <generator_to_light_to_seg_to_infinity_traffic >.....done
18  Post-Condition::No_Traffic_Elements_Rule::..........Pass
19  Post-Condition::All_PN_Elements_Rule::..........Pass
20  Post-Condition::All_Elements_ids_Rule::..........Fail, Check Errors
21     Errors:Found Nodes with no ID of type:
22      pl2tran, tran2pl, PNPlace, PNTransition,
23  ModelComparator::.........Pass
24  FragmentComparator::..........none
25 Invoker Block : Received Test Result:
26  ******************************************************
27 Invoker Block: Sending Next Test Case
28  Loading Model.........done
29  Transforming <generator_seg_infinity_no_links_traffic instance >.....done
30  Post-Condition::No_Traffic_Elements_Rule::..........Pass
31  Post-Condition::All_PN_Elements_Rule::..........Pass
32  Post-Condition::All_Elements_ids_Rule::..........Fail, Check Errors
33     Errors:Found Nodes with no ID of type:
34      pl2tran, tran2pl, PNPlace, PNTransition,
35  ModelComparator::.........Pass
36  FragmentComparator::..........none
37 Invoker Block : Received Test Result:
38  ******************************************************
39 Invoker Block: Sending Next Test Case
40  Loading Model.........done
41  Transforming <generator_to_seg_to_infinity_traffic >.....done
42  Post-Condition::No_Traffic_Elements_Rule::..........Pass
43  Post-Condition::All_PN_Elements_Rule::..........Pass
44  Post-Condition::All_Elements_ids_Rule::..........Fail, Check Errors
45     Errors:Found Nodes with no ID of type:
46      pl2tran, tran2pl, PNPlace, PNTransition,
47  ModelComparator::.........Pass
48  FragmentComparator::..........none
49 Invoker Block : Received Test Result:
50  ******************************************************
51 Invoker Block: Finished Processing All Test Cases
```

Listing 4.5: The TUnit output trace running the test suite after fixing bug2

Bug3-Model-Fragment

We finally fix the file not found in the model fragment test case, by adding the right file in the test directory. A re-run of the test suite shows all test cases as passing except the post conditions 3, regarding the unique id of elements as shown in Listing 4.6. This is called the traceability link aspect of model transformations, namely we can use the id of an element in the transformed output model, to figure out which elements of the starting model it was generated from.

We can fix this post-condition by enforcing the rules which create PN elements to specify the id attribute, but we won't do it here as we have demonstrated the effectiveness of the TUnit framework in performing regression testing. (Note by regression testing we mean an implementation change due to a bug fix rather than a requirement change)

```
1  ******************************************************
2  Invoker Block: Sending Next Test Case
3    Loading Model ..... done
4    Transforming <generator_seg_to_infinity_traffic_two_tokens instance >..... done
5    Post-Condition :: No_Traffic_Elements_Rule :: ..... Pass
6    Post-Condition :: All_PN_Elements_Rule :: ..... Pass
7    Post-Condition :: All_Elements_ids_Rule :: ..... Fail, Check Errors
8       Errors:Found Nodes with no ID of type:
9        pl2tran, tran2pl, PNPlace, PNTransition,
10   ModelComparator :: ..... None
11   FragmentComparator :: ..... pass
12 Invoker Block : Received Test Result:
13   ******************************************************
14 Invoker Block: Sending Next Test Case
15   Loading Model ..... done
16   Transforming <generator_to_light_to_seg_to_infinity_traffic >..... done
17   Post-Condition :: No_Traffic_Elements_Rule :: ..... Pass
18   Post-Condition :: All_PN_Elements_Rule :: ..... Pass
19   Post-Condition :: All_Elements_ids_Rule :: ..... Fail, Check Errors
20      Errors:Found Nodes with no ID of type:
21       pl2tran, tran2pl, PNPlace, PNTransition,
22   ModelComparator :: ..... Pass
23   FragmentComparator :: ..... none
24 Invoker Block : Received Test Result:
25   ******************************************************
26 Invoker Block: Sending Next Test Case
27   Loading Model ..... done
28   Transforming <generator_seg_infinity_no_links_traffic instance >..... done
29   Post-Condition :: No_Traffic_Elements_Rule :: ..... Pass
30   Post-Condition :: All_PN_Elements_Rule :: ..... Pass
31   Post-Condition :: All_Elements_ids_Rule :: ..... Fail, Check Errors
32      Errors:Found Nodes with no ID of type:
33       pl2tran, tran2pl, PNPlace, PNTransition,
34   ModelComparator :: ..... Pass
35   FragmentComparator :: ..... none
36 Invoker Block : Received Test Result:
37   ******************************************************
38 Invoker Block: Sending Next Test Case
39   Loading Model ..... done
40   Transforming <generator_to_seg_to_infinity_traffic instance >..... done
41   Post-Condition :: No_Traffic_Elements_Rule :: ..... Pass
42   Post-Condition :: All_PN_Elements_Rule :: ..... Pass
43   Post-Condition :: All_Elements_ids_Rule :: ..... Fail, Check Errors
44      Errors:Found Nodes with no ID of type:
45       pl2tran, tran2pl, PNPlace, PNTransition,
46   ModelComparator :: ..... Pass
47   FragmentComparator :: ..... none
48 Invoker Block : Received Test Result:
49   ******************************************************
50 Invoker Block: Finished Processing All Test Cases
```

Listing 4.6: The TUnit output trace running the test suite after fixing fragment bug

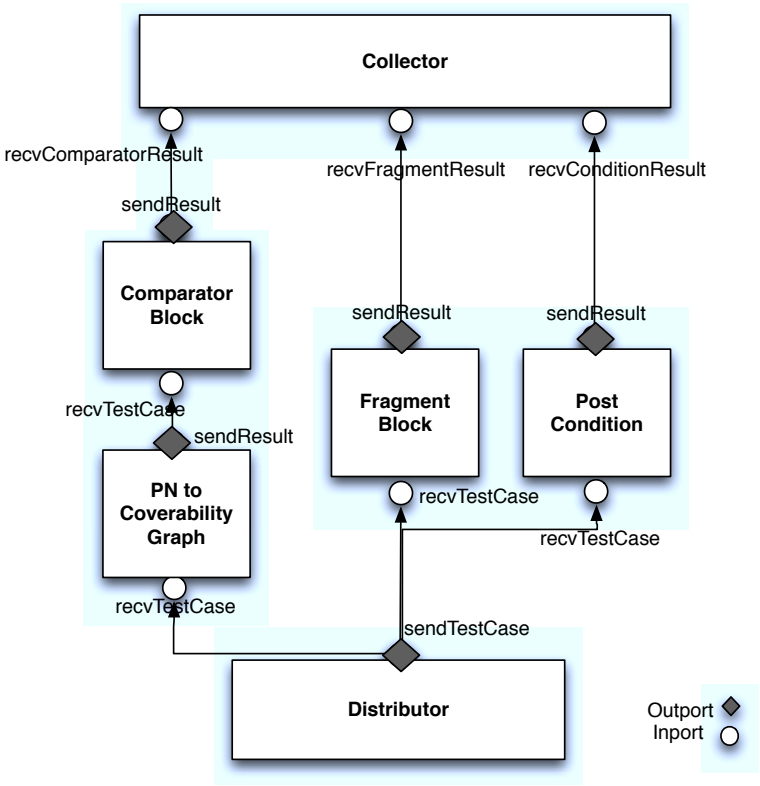


Figure 4.13: Acceptor Block Semantic Equivalence

4.5 Semantic Equivalence

Comparing models is usually done at the syntax level (abstract or concrete). For example, in the TUnit demonstration we showed that we were comparing PN models (expected vs. actual transformed models) at the syntax level. But ultimately, we would like to check if the models have the same meaning, and hence compare at the semantic level. For example, PN models have a reachability/coverability graphs as semantic domain. So instead of giving a PN expected, we could specify an expected a coverability graph to do the comparison at the semantic level. TUnit enables such changes with minimal efforts. For example, the acceptor block would have an extra block to do the conversion between PN and coverability graphs as shown in Figure 4.13. Note that in this scenario, the expected output could be specified as a coverability graph or as a PN, where in the latter case the block will convert both, the actual PN output to a coverability graph and the expected PN to its coverability graph. The Comparator block will then compare the coverability graphs of both PN models.

For example as shown in Figure 4.14, PN models shown are different in direct comparison, however they are semantically the same as shown by the coverability analysis. TUnit lets us specify test cases pertaining not only to the syntactic, but also to the semantic level.

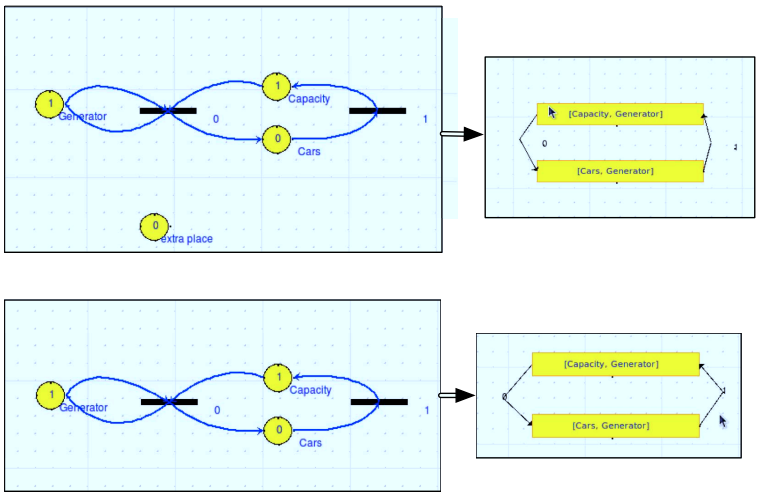


Figure 4.14: Coverability Graph Example

4.6 Discussion

In this section we discuss the advantages and disadvantages of the choices we made to address the main challenges we chose to tackle. To reiterate, the problem we are dealing with is automating the execution of testing for model transformations. We choose to build a framework to enable the automation of repetitive testing tasks. The framework was modelled at a level of abstraction which we feel is most appropriate for testing transformations.

In what follows, we further elaborate on the three aspects of this framework: *model based*, *DEVS formalism choice* and *automating execution*.

In the "model everything" spirit, we feel that modelling the testing framework could best realize this notion by focusing on the right level of abstraction. It can help promoting reuse and productivity through code generation. And hence allow for more complex scenarios. Furthermore, since we are dealing with model objects, it makes sense to have a modelled framework to deal with models.

The event based DEVS formalism is a natural candidate for our testing. DEVS is a powerful simulation and analysis formalism, and testing can be seen as running a simulation experiment of the SUT. We can encode graphs in events to be transmitted and processed through blocks. DEVS has a well established research community to provide support for different DEVS tools and simulators.

Most work in model transformation testing has been focusing on areas such as generating input models and building oracles as discussed in chapter 2. In software testing, frameworks like XUnits (JUnit, NUnit, PyUnit .. etc) are well established and represent an essential part of the engineering process of creating software. These frameworks represent testing drivers (as described in Chapter1), to allow seamless repetitive execution of the test suites in areas such as functional and regression testing. We feel that for MDE to realize its potential as a well established engineering process, we need parallels to the

XUnit testing frameworks in the MDE world. However, a new level of challenges arise when dealing with models and their transformations as models are complex objects which could have dependencies. The process also depends on model comparison (as discussed in Chapter 3) to be integrated and fully automated.

We built TUnit to address the challenges of automation when testing model transformations and to complement the work done by others in the test case generation area as discussed in detail in section 2.4. A major advantage of using TUnit, is that we could test several different implementations of the same transformation in parallel. Given the advantages of the model based nature of TUnit, we could modify the acceptor block to compare the output of the two transformations to check for any discrepancies for example. We could also build complex verdict criteria in the acceptor block like mapping to a semantic domain as discussed in Section 4.5. Furthermore, we could test the performance of different model comparison algorithms by applying minimal modifications.

4.7 Conclusion

In this chapter we presented our "modelled" TUnit framework to help enable the automatic testing of model transformations. Our contribution lies in providing a hands on practical experience with using a framework to enable complex testing. For example we enable different types of oracle functions used for model transformation testing. We further show how our framework helps in regression testing. Finally, we show how TUnit enables semantic domain equivalence testing by introducing the proper framework hooks and demonstrate it in our case study.

Conclusion

In conclusion, this thesis has made an attempt to resolve several outstanding issues in model transformation testing. Our approach was to start by studying the foundations of software testing research and identifying its main achievements, and remaining challenges. Using this knowledge we built a categorization for studying software testing in different contexts. We then briefly introduced the foundations of Model Driven Engineering (MDE) and Model Transformation (MT) in particular. Afterwards we surveyed current research related to testing model transformations. The categorization we built in Chapter 1 for software testing was used to classify MT testing techniques, and subsequently build a roadmap of achievements and outstanding research challenges.

The main contribution of this thesis is summarized in three items.

First, in chapter 3 we presented the first challenge we tackled, model comparison. We built on existing techniques and algorithms in other domains to derive a lean and mean algorithm which is most suitable for comparing models in the context of testing. The algorithm is mean in that it provide exact non-approximate results, and lean in terms of customizability. We ran extensive experiments to qualify the performance of our customized algorithm according to a set of different input model types.

Second, we presented our testing framework TUnit (Transformation Unit) to enable the automation of execution of the test suites. TUnit's model based architecture was described in details along with its underlying DEVS formalism. We described new MDE related challenges and our solutions through TUnit. The model comparison algorithm was integrated into TUnit to serve as one of its different oracle function capabilities. Using a case study, we demonstrated the frameworks's advantages to streamline the testing process, specifically by allowing regressions testing due to code changes or bug fixes.

Finally, our last contribution was to demonstrate achieving semantic equivalence in the context of MT testing, due to the model based nature of TUnit. Semantic equivalence checking enables a new level of model comparison, which could scale beyond current testing scenarios in terms of accuracy.

We believe our work would complement the current MT testing research which is focused on test case generations, and should be combined with such techniques in future work as have been discussed.

Bibliography

[ACP07] Davide Di Ruscio Antonino Cicchetti and Alfonso Pierantoni. A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185, Jan 2007.

[AP03] M. Alanen and I. Porres. Difference and union of models. *Lecture Notes in Computer Science*, 2863:2–17, Jan 2003.

[Bei95] Boris Beizer. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, 1995.

[Ber07] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[BFG96] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Issues in the practical use of graph rewriting. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Selected papers from the 5th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 38–55, Williamsburg (USA), November 1996. Springer-Verlag.

[BGF+09] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 2009.

[BV01] Jean-Sébastien Bolduc and Hans Vangheluwe. The modelling and simulation package pythonDEVS for classical hierarchical DEVS. MSDL technical report msdl-tr-2001–01, McGill University, June 2001.

[BY01] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[Bé06] Rumpe B Schürr A Tratt L . Bézivin, J. Model transformations in practice workshop. *Lecture Notes in Computer Science*, 3844:120–127, 2006.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[CLOP02] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese. A classification framework to support the design of visual languages. *J. Vis. Lang. Comput.*, 13(6):573–600, 2002.

[CSPZ04] Saehoon Cheon, Chungman Seo, Sunwoo Park, and Bernard P. Zeigler. Design and implementation of distributed DEVS simulation in a peer to peer network system. In Herwing Unger, editor, *ASTC'04*, pages 18–22, Arlington (USA), April 2004. Society for Modeling and Simulation International.

[DCV07] Pasquale Foggia Donatello Conte and Mario Vento. Challenging complexity of maximum common subgraph detection algorithms: A performance analysis of three algorithms on a wide database of graphs. *Journal of Graph Algorithms and Applications*, 11(1):99–143, 2007.

[dLV02] Juan de Lara and Hans Vangheluwe. Atom^3: A tool for multi-formalism and meta-modelling. In *FASE*, pages 174–188, 2002.

[FBMT09] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying input test data for model transformations. *Software and Systems Modeling*, 8(2):185–203, April 2009.

[FSB04] F Fleurey, J Steel, and B Baudry. Validation in model-driven engineering: testing model transformations. *Model, Design and Validation, 2004. Proceedings. 2004 First International Workshop on*, pages 29–40, 2004.

[GKS05] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, Jun 2005.

[Gla09] Robert L. Glass. A classification system for testing, part 2. *IEEE Software,*, 26(1):104–104, Jan.-Feb. 2009.

[GLV07] Holger Giese, Tihamér Levendovszky, and Hans Vangheluwe. *Summary of the Workshop on Multi-Paradigm Modeling: Concepts and Tools*, volume 4364/2007 of *Lecture Notes in Computer Science*, pages 252–262. Springer Berlin / Heidelberg, 2007.

[Gra08] Paul Baker Ôø Zhen Ru Dai Ôø Jens Grabowski. *Model-Driven Testing Using the UML Testing Profile*. Springer, 2008.

[Har00] Mary Jean Harrold. Testing: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 61–72, New York, NY, USA, 2000. ACM.

[Het88] Bill Hetzel. *The complete guide to software testing (2nd ed.)*. QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[HR00] D. Harel and B. Rumpe. Modeling languages: Syntax, semantics and all that stuff, part i: The basic stuff. Technical report, Jerusalem, Israel, 2000.

[KAER07] J Kuster and M Abd-El-Razik. Validation of model transformations-first experiences using a white box approach. *Lecture Notes in Computer Science*, Jan 2007.

[Ker05] Kermeta. Kermeta: The kermeta project home page. available from: http://www.kermeta.org, 2005.

[KH04] EB Krissinel and K Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6), 2004.

[KM07] R.A. Khan K Mustafa. *Software Testing Concepts and Practices*. Narosa, 2007.

[KPP06] Dimitrios Kolovos, Richard Paige, and Fiona Polack. Model comparison: a foundation for model composition and model transformation testing. *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, May 2006.

[KR96] Samir Khuller and Balaji Raghavachari. Graph and network algorithms. *Computing Surveys (CSUR*, 28(1), Mar 1996.

[KRPP09] D Kolovos, D Di Ruscio, A Pierantonio, and R Paige. Different models for model matching: An analysis of approaches to support model differencing. *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, pages 1 – 6, May 2009.

[Kus04] J. M. Kuster. Systematic validation of model transformations. *WiSME'04 (associated to UML'04)*, 2004.

[KW00] Anneke Kleppe and Jos Warmer. An introduction to the object constraint language (ocl). In *TOOLS (33)*, page 456, 2000.

[LGJ07] Yuehua Lin, Jeff Gray, and Frédéric Jouault. DSMDiff: A differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, Jan 2007.

[LLMC05] L. Lengyel, T. Levendovszky, G. Mezei, and H. Charaf. Control flow support in metamodel-based model transformation frameworks. *Computer as a Tool, 2005. EUROCON 2005.The International Conference on*, 1:595–598, Nov. 2005.

[LS] M Lawley and J Steel. Practical declarative model transformation with tefkat. *Satellite Events at the MoDELS 2005 Conference*, pages 139–150.

[LZG04] Yuehua Lin, Jing Zhang, and Jeff Gray. Model comparison: A key challenge for transformation testing and version control in model driven software development. *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, Jan 2004.

[LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray. A testing framework for model transformations. *Model-Driven Software Development - Research and Practice in Software Engineering. 2005*, pages 219–236, 2005.

[MBT08] J.-M. Mottu, B. Baudry, and Y.L. Traon. Model transformation testing: oracle issue. *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on*, pages 105–112, April 2008.

[McG82] JJ McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1), 1982.

[Mes07] Gerard Meszaro. *xUnit Test Patterns : Refactoring Test Code*. Addison-Wesley, 2007.

[Mur89] Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541 – 580, 1989.

[OMG08] OMG. Meta object facility (mof) 2.0 query/view/transformation, v1.0, April 2008.

[RFG+05] R Reddy, R France, S Ghosh, F Fleurey, and B Baudry. Model composition-a signature-based approach. *Aspect Oriented Modeling (AOM) Workshop, in conjunction with MoDELS'05*, Jan 2005.

[RR99] Arthur Reyes and Debra Richardson. Siddhartha: A method for developing domain-specific test driver generators. *ASE '99: Proceedings of the 14th IEEE international conference on Automated software engineering*, Oct 1999.

[RW02] J Raymond and P Willett. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, Jan 2002.

[SB06] S. Sen and B. Baudry. Mutation-based model synthesis in model driven engineering. *Mutation Analysis, 2006. Second Workshop on*, pages 13–13, Nov. 2006.

[SBM08] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *ICST '08: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, pages 328–337. IEEE Computer Society, 2008.

[Sch96] Stephen R. Schach. Testing: principles and practice. *ACM Comput. Surv.*, 28(1):277–279, 1996.

[SDZ09] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. Technical report, Universität des Saarlandes, Saarbrücken, Germany, January 2009.

[Sha90] M. Shaw. Prospects for an engineering discipline of software. *Software, IEEE*, 7(6):15–24, Nov 1990.

[SKHP07] Park Sunwoo, Sean H. J. Kim, C. Anthony Hunt, and Dongsun Park. DEVS peer-to-peer protocol for distributed and parallel simulation of hierarchical and decomposable DEVS models. In Hyongsuk Kim and Benjamin Wah, editors, *ISITC'07*, pages 91–95, Jeonju (Korea), November 2007. IEEE Computer Society.

[SKV] Eugene Syriani, Jörg Kienzle, and Hans Vangheluwe. Exceptional transformations. In Laurence Tratt, editor, *Theory and Practice of Model Transformations (ICMT 2010)*.

[SPB+04] Chungman Seo, Sunwoo Park, Kim Byounguk, Saehoon Cheon, and Bernard P. Zeigler. Implementation of distributed high-performance DEVS simulation framework in the grid computing environment. In Herwing Unger, editor, *ASTC'04*, Arlington (USA), April 2004. Society for Modeling and Simulation International.

[SV08] Eugene Syriani and Hans Vangheluwe. Programmed graph rewriting with devs. pages 136–151, 2008.

[TBWK07] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. Difference computation of large models. *ESEC-FSE '07: Proceedings of the the 6th*

joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, Sep 2007.

[TEG+05] Gabriele Taentzer, Karsten Ehrig, Esther Guerra, Juan de Lara, László Lengyel, Tihamér Levendovszky, Ulrike Prange, Dániel Varró, , and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. October 2005.

[UL07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing, a tools approaches*. Elsevier Inc, 2007.

[Ull76] J. Ullman. An algorithm for subgraph isomorphism. *Journal of the ACM*, 23:31–42, 1976.

[WDyC03] Yuan Wang, David J. DeWitt, and Jin yi Cai. X-diff: An effective change detection algorithm for xml documents. In *ICDE*, pages 519–530, 2003.

[WKC08] Junhua Wang, Soon-Kyeong Kim, and D. Carrington. Automatic generation of test models for model transformations. *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*, pages 432–440, March 2008.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, Nov 2005.

[YT89] Michael Young and Richard N. Taylor. Rethinking the taxonomy of fault detection techniques. In *ICSE '89: Proceedings of the 11th international conference on Software engineering*, pages 53–62, New York, NY, USA, 1989. ACM.

[Zei84] Bernard P. Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, 1984.

[ZZH06] Ming Zhang, Bernard P. Zeigler, and Phillip Hammonds. DEVS/RMI-an auto-adaptive and reconfigurable distributed simulation environment for engineering studies. *Journal of Test and Evaluation*, 27(1):49–60, April 2006.