MASTER'S THESIS | LUND UNIVERSITY 2018

Improved precision and verification for test selection in Modelica

Markus Olsson, Filip Stenström

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2018-08

Improved precision and verification for test selection in Modelica

Markus Olsson
markus.iluvatar@gmail.com

Filip Stenström
filip.stenstrom@hotmail.com

June 7, 2018

Master's thesis work carried out at Modelon AB.

Supervisors:
Jon Sten, jon.sten@modelon.com
Niklas Fors, niklas.fors@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Regression testing is a key concept to keep software in good shape, yet it can be a time consuming process. Testing time can be reduced by using a test selection technique which selects only the subset of tests that might have been affected.

We have defined and implemented a high precision regression test selection technique for the modeling language Modelica by using static dependency analysis. Our test selection technique provides better time savings compared to a previous technique. The time savings were computed for tests in certain Modelica libraries when one file or class was changed.

We verified our dependency analysis by finding a subset of the actual dependencies and making certain they were found by our analysis. The actual dependencies were found by mutating classes and seeing which tests were affected. Furthermore, we evaluated our verification by analyzing the effectiveness of the different kinds of mutations.

Keywords: Regression test selection, Modelica, static analysis, verification, mutation testing

Acknowledgements

We would like to thank Jon Sten for helping us to learn Modelica and for taking his time to ensure that the thesis was advancing in the right direction, and for the feedback on the report.

We would also like to thank Niklas Fors, our supervisor at LTH, for giving us continued advice on how to proceed with our thesis and for the feedback on the report.

Division of labor

For this thesis project we have both worked together at Modelon AB the whole time. We have discussed every part of the project and how to proceed, and we have defined the rules for the class dependencies together. The implementation of the dependency analysis, the mutation framework and the general mutations was done by Markus. Filip implemented the specialized mutations. We both worked on python scripts to generate the graphs. Filip also ported the results of the mutation dependency analysis to work with python and did the evaluation of the mutations. Markus also implemented the Modelica class dependency test suite. In addition to this we have both worked on things that did not work out and therefore was not included in this report.

For the report we initially worked on different chapters and sections. When we had finished a first draft of all sections we both read the whole report and made improvements on all sections regardless of who originally wrote them. The initial division was as follows:

- Introduction - Filip
- Background, Regression test selection - Filip
- Background, Modelica - Markus
- Background, OPTIMICA Compiler Toolkit - Both
- Background, Mutation testing - Filip
- Rules for dependencies - Markus
- Method - Filip
- Implementation - Filip
- Evaluation - Filip
- Discussion - Filip
- Related work - Filip
- Conclusions - Filip

Contents

1 Introduction **9**

2 Background **11**
2.1 Regression test selection 11
2.2 Modelica 12
2.2.1 Example 12
2.2.2 Modelica classes 14
2.2.3 Dot notation 14
2.2.4 Name lookup and class access 14
2.2.5 Inheritance 15
2.2.6 Modifications and redeclare 16
2.2.7 Modelica tests 16
2.3 OPTIMICA Compiler Toolkit 18
2.3.1 Source tree 18
2.3.2 Instance tree 18
2.3.3 Flat tree 19
2.4 Mutation testing 19

3 Rules for dependencies **21**
3.1 Descriptions and motivations 22
3.1.1 Rule 1 22
3.1.2 Rule 2 22
3.1.3 Rule 3 24
3.1.4 Rule 4 27

4 Method **31**
4.1 Dependency analysis 31
4.1.1 External files 31
4.2 Verification 32
4.2.1 Mutation dependency analysis 32

4.2.2 Test suite 32

5 Implementation **33**
5.1 Dependency analysis 33
5.2 Verification 34

6 Evaluation **39**
6.1 Test selection performance 39
6.1.1 Precision 39
6.1.2 Savings when one class or file changed 40
6.1.3 MSL commit history 43
6.2 Verification results 43
6.3 Mutation type results 44
6.4 Missing dependencies 46

7 Discussion **49**
7.1 Comparison with source tree analysis 49
7.2 Alternative technique 51
7.3 Replace functions mutation 53
7.4 Improvements to general mutations 53
7.5 Threat to validity 53

8 Related work **55**
8.1 Runtime and static analysis 55
8.2 Runtime analysis 55
8.3 Static analysis 56
8.4 Discussion 56

9 Conclusions **57**
9.1 Future work 57

Chapter 1
Introduction

A software system under development experiences frequent updates, where each update introduces a risk of breaking existing functionality. Regression tests attempt to ensure that the system still works as expected after changes are made. Running the whole regression test suite can be a time-consuming process though. In order to minimize the testing time, *regression test selection (RTS)* techniques select only a subset of the tests. A *safe* RTS technique ensures that all tests that might fail will always be selected [1].

Modelica is a language for modeling dynamical systems. Modelica tests generally require the simulation of models, which is a time costly process. It is therefore of great importance to reduce the number of unnecessary tests that run. As an example, testing the *Modelica Standard Library (MSL)* takes about two to three hours when tested using the *Modelica Testing Toolkit* [2].

The main goal of our thesis is to define a RTS technique for the Modelica language with high precision, and to verify that the technique is safe. The aim is also for the technique to have greater time savings compared to other techniques.

We have implemented our technique by performing *class dependency analysis* in a Modelica compiler. This analysis finds dependencies between Modelica classes and tests are selected if they have a dependency to any changed class. The rules for class dependencies are implementation independent.

To our knowledge there is only one other RTS technique for Modelica, which has been defined by Hedblom and Rundquist [3]. This technique also uses class dependency analysis to select tests. The implementation of our technique is done in the same compiler as this technique, however our technique is implemented in a later phase of the compilation process and define stricter rules for the dependency analysis to get a higher precision.

In order to verify the safety of our RTS technique we have performed mutations in MSL to find dependencies from test classes. We have then ensured that our dependency analysis found all those dependencies. Although this method could not guarantee the safety of our technique, it increased the confidence that it was safe.

As a result of this thesis, we contribute to the body of knowledge three artifacts for

further research on Modelica test selection:

- A set of dependency rules for Modelica classes
- An open source test suite for class dependency analysis in Modelica
- An open source database of dependencies for test classes in MSL

Both the test suite and the database of dependencies can be found online [1] for anyone who wants to develop Modelica test selection techniques.

[1] https://github.com/modelon/MCDTS

Chapter 2
Background

In this chapter we will provide the necessary background to understand this report. This includes an introduction to RTS techniques, an overview of Modelica, and information about the compiler we have used for our implementation.

2.1 Regression test selection

The purpose of an RTS technique is to select a subset of tests that might have been affected by a change in order to reduce the testing time. For the technique to be efficient, it is also important that its execution time is short relative to the actual testing time.

In this report we denote an original version of a program as P, and the same program but with changes as P'. A test is considered changed if it has a different execution path in P' compared to P. This is under the assumption that the test case is deterministic.

An RTS technique can be measured by its *inclusiveness* [1]. Given all changed tests in P', the inclusiveness of an RTS technique is measured as the percent of changed tests it selects. If all changed tests always are selected, the technique will have 100% inclusiveness and is then called *safe*.

The *precision* of an RTS technique is determined by its ability to exclude tests that are not changed [1]. The precision is calculated as the percent of selected tests that are changed. A technique with 100% precision will only select changed tests. A higher precision means that fewer tests are selected, however the analysis execution time might be longer since it is usually involves more decisions.

In order to verify that an RTS is safe, one can exhaustively prove that all changes that might result in a changed test is accounted for by the technique. Another way to verify the technique, is by implementing it and thoroughly testing that all changed tests always are selected.

Regression test selection is usually done with either static analysis [3, 4] or with runtime analysis [5]. The runtime analysis uses *code instrumentation* to monitor the execution

of the program. Extra instructions are then inserted into the program to get information such as the execution path and code coverage [6]. It can also be used to find which files a program has accessed, it depends on those files [5]. Static analysis can for example be done with class dependency analysis.

2.2 Modelica

Modelica is an object oriented modeling language for simulation of dynamical systems [7]. It can be used to simulate everything from electrical circuits to airplane dynamics. The benefit of this wide application area is that subsystems from different domains can be connected in the same model. For instance, the output from the model for an electrical circuit can be used as input for the model of a motor, and the whole system can be simulated.

The modeling in Modelica is declarative, but it is also possible to specify algorithms that are imperative.

2.2.1 Example

The bouncing ball model is good example to demonstrate the basics of Modelica.

```
model BouncingBall
    parameter Real e=0.8;
    Real h;
    Real v;
initial equation
    h = 1.0;
    v = 0;
equation
    v = der(h);
    der(v) = -9.81;
    when h<0 then
        reinit(v, -e*pre(v));
    end when;
end BouncingBall;
```

Listing 2.1: Modelica code describing a bouncing ball

The example – shown in Listing 2.1 – consists of a model named `BouncingBall`. The model takes one parameter, the coefficient of restitution `e`. It has two variable components, height `h` and velocity `v`. The height is initialized as one and the velocity as zero. Equations define the velocity as the time derivative of the height, and the time derivative of the velocity as the gravitational acceleration. There is also an equation that defines what happens when the ball hits the ground; when the height becomes less than zero, the velocity is negated and multiplied with `e`. A simulation of the height over time can be seen in Figure 2.1.

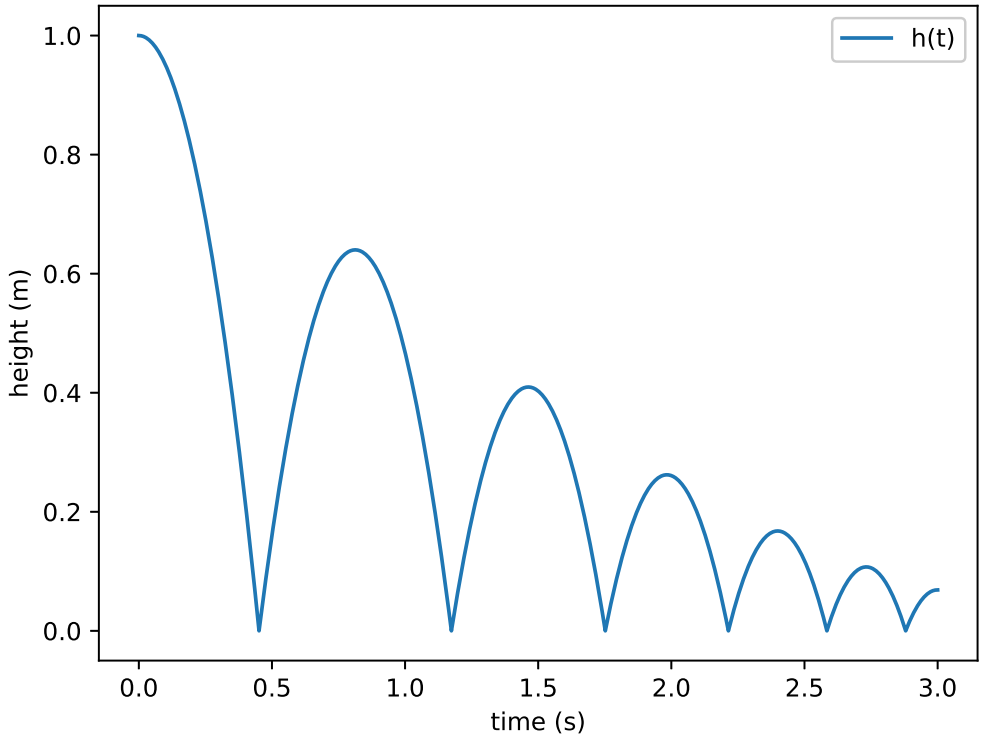

Figure 2.1: Simulation results of how the height of the bouncing ball changes over time.

2.2.2 Modelica classes

Modelica consists of different kinds of classes. The default class is simply called `class` and defines all class functionality. All other classes are based on `class` but have some restrictions, and they are called *specialized classes*. Examples of specialized classes are `model`, `package` and `function`. For instance, `package` may not contain any non-constant *component*. A component is an instantiation of a class or a primitive type, similar to variables in imperative programming languages.

Throughout this report we use a number of specialized classes. The ones we use are therefore in the following clauses given a short description.

The `model` specialization is used for classes that describe simulatable systems or parts of such a system. This specialization is not very restricted; it can for instance contain class declarations, equations, algorithms and components.

The `package` specialization is used to group other classes together. Packages can contain declarations of classes and constant components.

The `function` specialization is used for functions. Functions have zero or more input arguments, one or more outputs and an algorithm section. The input arguments and the output are declared as components with the keywords `input` and `output` respectively. Functions can have the `external` keyword instead of an algorithm section, which means that the algorithm is defined in an external file as C or Fortran code.

The `record` and `operator record` specializations are used to make data types that are more complicated than the primitive types. Records are similar to models but do not have equations. Operator records can also have operator functions that overload the behavior of operators.

The `connector` specialization is used for connecting components. It can be seen as a way of linking the output of one component to another.

The `type` specialization is used to extend the built-in primitive types.

2.2.3 Dot notation

Modelica supports having multiple classes with the same name, though they can not be located in the same class. To uniquely identify a class, dot notation is used.

Dot notation is more generally used to access members of classes. The dot notation consists of an access to the class (note that this access can use dot notation), a dot and the name of the member. If the first identifier in a dot notation is a top level package, the dotted name is called a *fully qualified name*.

Listing 2.2 features nested classes. `C` can be identified with dot notation as `A.B.C`.

2.2.4 Name lookup and class access

There are two types of name lookup in Modelica that are relevant for this thesis: simple name lookup and composite name lookup.

Simple name lookup is used if the access is a single identifier (i.e. not dot notation). The name lookup algorithm can be described as the following:

1. If inside a for-loop, check if the identifier matches an iteration variable.

```
package A
   model B
      model C
      end C;
   end B;
end A;
```

Listing 2.2: Modelica code showing nested classes. `C` can be identified with dot notation as `A.B.C`.

2. Look for the identifier in the scope of the current class. This includes imported classes and the scope of the super classes.
3. If the identifier is not found in the current scope, change current scope to the scope of the enclosing class and look again.
4. Repeat steps 2 and 3 until either the identifier is found or the current scope is the top package.
5. If the identifier is still not found, check the global scope.

Composite name lookup is used if the access uses dot notation. The first identifier in the dot notation will be found using simple name lookup. The following identifiers are each looked for in the scope of the class found by the previous identifier.

Examples of accesses are seen in Listing 2.3. To access `B` from `C`, a member `B` in `C` is looked for, when it is not found it will look in `B`, where it is not found either. Finally it will look in `A` where class `B` is found. The access to `C` from `A` will not find the class `C` because it is not found in `A` or the global scope. The access to `B.C` will find `B` in `A` and `C` in `B`.

```
package A
   constant C c1; // Class C can not be found
   constant B.C c2; // Class B.C can be found
   model B
      model C
         B b; // Class B can be found
      end C;
   end B;
end A;
```

Listing 2.3: Modelica code demonstrating name lookup.

2.2.5 Inheritance

A class can inherit from another class with the same class specialization. When a class extends another class, it inherits all elements except import statements. The inheriting class can also access enclosed classes in the extended class. An examples of inheritance can be seen in Listing 2.4.

```
model A
  model M
    Real x;
  end M;
  Real x = 0;
end A;

model B
  extends A;
  M m;
equation
  m.x = x;
end B;
```

Listing 2.4: An example of inheritance. The class `B` inherits the component `x` from `A`, and can access `M` without using the composite access `A.M`.

```
model C = A;
```

Listing 2.5: An example of a short class declaration.

In Modelica, there is a construct called a short class declaration. It is a class declaration where the new class extends a target class. For example in Listing 2.5 the model `C` extends `A` without adding anything new.

2.2.6 Modifications and redeclare

In Modelica, the behavior of classes can be altered by using modifications. They can be applied to component declarations, extends clauses and short class declarations. There are value modifications and redeclare modifications. Value modifications modify the value of a component, see example in Listing 2.6. There are two types of redeclare modification. Component redeclare modifications replace the declaration of a component. Class redeclare modifications replace the declaration of a class. Examples of component and class redeclare modifications can be seen is Listings 2.7 and 2.8 respectively.

Classes and components can also be redeclared without modifications, by instead using the `redeclare` keyword as a prefix for the declarations. In Listing 2.9 the class `M` is redeclared using the redeclare keyword as a prefix on the declaration. This is equivalent to using a redeclare modification on the extends statement, as in class `F` (Listing 2.8), but more convenient when many classes are redeclared.

2.2.7 Modelica tests

Classes in Modelica can be annotated, which is a way to include metadata. A test in Modelica is by convention a class that is annotated as an “experiment”. A test is generally

```
model A
  parameter Real x = 0;
end A;

model B
  extends A(x = 1);
end B;
```

Listing 2.6: An example of a value modification. In the model `B`, the value of the parameter `x` in the extends clause is modified.

```
model C
  replaceable A a;
end C;

model D
  C c(redeclare B a);
end D;
```

Listing 2.7: An example of a component redeclare modification. The component `c` in the class `D` has a redeclare modification, replacing the type declaration of component `a` to the class `B`. The `replaceable` keyword signals that the component can be redeclared.

```
model E
  replaceable model M
  end M;
end E;

model F
  extends E(redeclare model M = M);
  model M
  end M;
end F;
```

Listing 2.8: An example of a class redeclare modification. The class `M` is redeclared in the class `F`. Note that the expression "`M = M`" means that `M` in `E` should be replaced with the `M` defined in `F`.

```
model G
  extends E;
  redeclare model M
  end M;
end G;
```

Listing 2.9: An example of how the redeclare prefix is used. The class `M` is redeclared using the redeclare prefix on the declaration. This is equivalent to using a redeclare modification on the extends clause (see Listing 2.8).

executed by simulating the test class and comparing the results of the simulation with predefined expected results.

2.3 OPTIMICA Compiler Toolkit

JModelica.org is a Modelica based open-source software platform used to model and solve dynamical systems [7]. JModelica.org includes a Modelica compiler which is built with the meta-compilation tool JastAdd [8]. *OPTIMICA Compiler Toolkit (OCT)* is a commercial product by Modelon AB that is based on JModelica.org [9]. OCT includes a Modelica compiler which extends the JModelica.org Modelica compiler, and it is this compiler that we have used to implement our test selection technique.

The abstract grammar for the OCT compiler is based on building three *abstract syntax trees (ASTs)* instead of only a single tree. These trees are called the *source tree*, *instance tree* and *flat tree*, and they will be further described in the next sections. Figure 2.2 also further illustrates the difference between the trees.

2.3.1 Source tree

The source tree is an object representation of the source code. What you find in the source code is what you find in the source tree. There is a one-to-one mapping, so if there is a class declaration in the source code, there will be exactly one node for this in the AST.

In the source tree it is sometimes not possible to lookup the class from a qualified access; we then say that the access is not resolvable. There are other things that are not possible in the source tree, such as determining the type of expressions.

2.3.2 Instance tree

The instance tree represents a class instantiation, and is an intermediate tree where modifications have been resolved [10]. While the source tree contains AST nodes for all the parsed classes, the instance tree represents a model instantiation and therefore mainly contains the information necessary to simulate this model. One way the instance tree differs from the source tree is that components in the instance tree contain all information

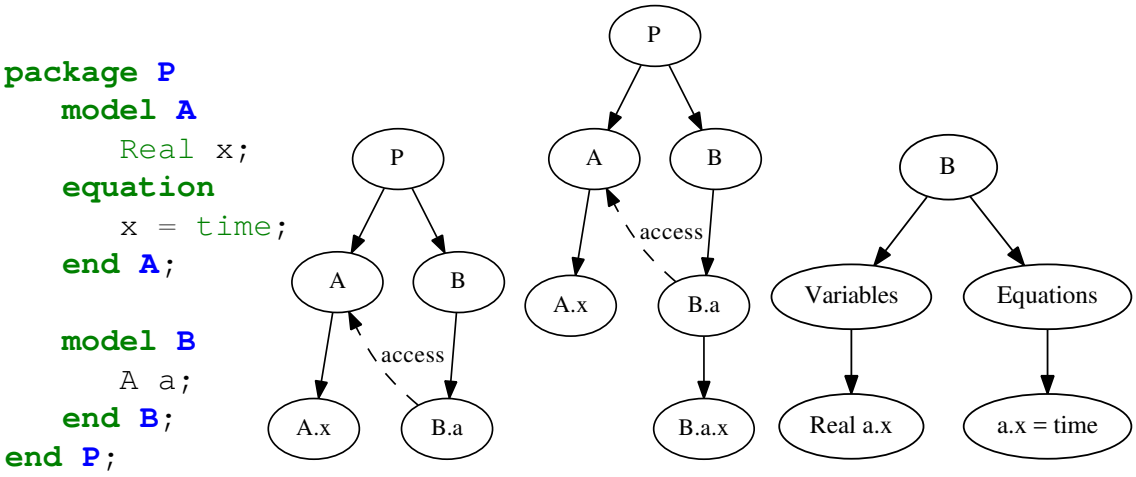

Figure 2.2: An example of the different ASTs in OCT Modelica compiler for the same source code. From left to right we have, source code, source tree, instance tree and flat tree. In this example only `B` was flattened.

about their corresponding classes. Building the whole instance tree will use a considerable amount of memory, so in the OCT compiler the instance tree is dynamically expanded.

2.3.3 Flat tree

The flat tree is an AST where the hierarchy of the instance tree has been removed. The flat tree is essentially what the Modelica specification refers to as a "flat Modelica structure" [11].

The flat tree is generated by *flattening* the instance tree, and can be described as an intermediate model that only contains variables, equations and functions necessary to build an equation system for simulation of the original model [10].

At least for the OCT Modelica compiler, the generation of the flat tree and its textual representation, *flat-code*, is deterministic. This means that as long as the class or any of the classes it depends on is not semantically changed or refactored, recompiling a class to flat-code will always yield the same result. An exception to this is compile time evaluation of external functions, as external functions may return random values.

2.4 Mutation testing

In this report we will not use any classic *mutation testing*, however we will use program mutations in a similar way to detect class dependencies. We therefore include a description of classic mutation testing.

Mutation testing [12] is the process of evaluating test suites. It is in general performed by inserting semantic changes one by one into the original program, thereby producing new programs. If all tests pass for a new program, although the change resulted in a bug, then this means that the tests that cover the changed code can be improved.

Chapter 3
Rules for dependencies

To perform the class dependency analysis it is necessary to have a set of rules for dependencies between classes. We defined rules for Modelica Version 3.2 Revision 2 based on the rules by Hedblom and Rundquist [3]. The following are the rules we have defined for direct class dependencies:

1. (a) A class has a dependency on an accessed class.
   (b) A class has dependencies on all classes in a composite access.
   (c) A class using an overloaded operator has a dependency to the operator function.
2. A class has a dependency on the enclosing class.
3. A class that contains a redeclaration depends on all super classes and enclosed classes of the replacing class (and all their enclosed classes and super classes recursively).
4. A class has dependencies on all implicitly called functions.
   (a) If a record or type encloses a function named `equalityConstraint`, it has a dependency on that function.
   (b) If a class extends `ExternalObject`, it has a dependency on the enclosed function `destructor`.

Rules 1a and 1b were kept from the previous rules, though we did some minor changes since our rules assume that the implementation can resolve all accesses. Rule 2 was also kept from the previous rules, but it was not changed. The other rules were defined by us.

3.1 Descriptions and motivations

This section describes the rules in more detail and motivates why each rule is necessary. The motivations use figures which consist of some Modelica code and a graph with the class dependencies our rules finds for the code. The nodes of the graph correspond to Modelica classes and the edges to dependencies. Each edge is labeled with the rule from which it is created. An edge is dashed if it was added by the change or dotted if it was removed. A rectangular node marks the changed class, and a diamond shaped node marks the affected class. There should be a path in the graph from the diamond to the rectangle.

3.1.1 Rule 1

This rule handles dependencies caused by accesses to other classes. Note that this can include accesses to derivative and inverse functions in annotations.

Rule 1a. *A class has a dependency on an accessed class.*

If a class has a simple access to another class, it will depend on it. If it has a composite access, the dependency will only be to the final element in the composite name. If for example a class has an access to `A.B.C`, this rule will only give a dependency to `C`. This rule is intuitive enough to not need a motivation.

Rule 1b. *A class has dependencies on all classes in a composite access.*

If a class has a composite access to a class or component, it will depend on all classes in the access except the final element. For example a class has an access to `A.B.C`, this rule will create dependencies to `A` and `B` but not `C`. Figure 3.1 shows an example of why this rule is necessary.

Rule 1c. *A class using an overloaded operator has a dependency to the operator function.*

If a class uses an overloaded operator for an operator record the class will depend on the operator function. Figure 3.2 shows an example of why rule this rule is necessary.

3.1.2 Rule 2

A class has a dependency on the enclosing class.

Rule 2 is a complement to rule 1b. In Modelica – because of how the name lookup works – a class enclosed in another class can access its content with a simple access. Figure 3.3 shows an example of why rule 2 is necessary. It is very similar to the example for rule 1b in Figure 3.1. Unlike that example, `C` is enclosed in `B`, so the access to `M` is not qualified, since the class `M` can be accessed directly. `C` must still depend on `B`, so rule 2 is used instead.

```
package A1
   model M
   end M;
end A1;

package A2
   model M
   end M;
end A2;

package B
   extends A1; // Removed
   extends A2; // Added
end B;

model C
   B.M m;
end C;
```

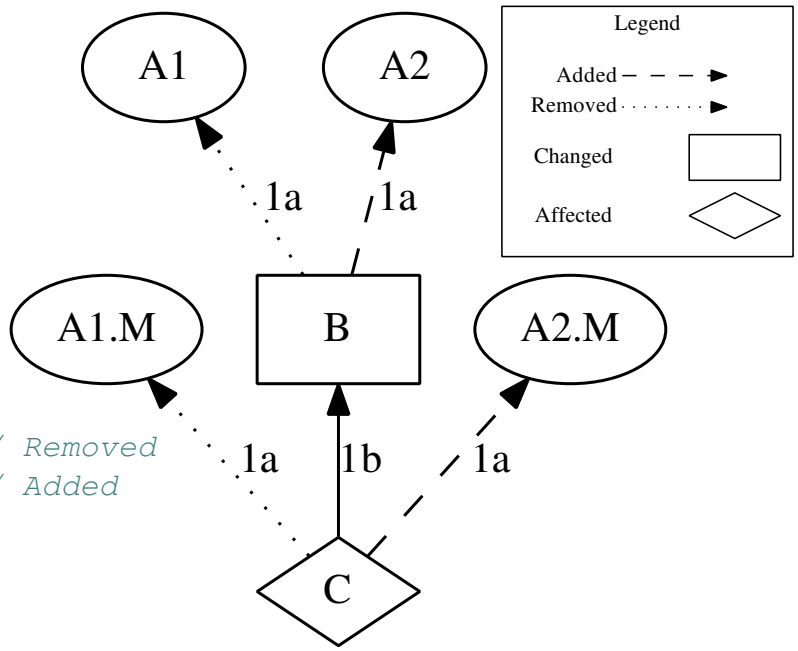

Figure 3.1: An example of why rule 1b is necessary. When the class `B` is changed to extend `A2` instead of `A1`, the class of the component `m` in `C` is changed, therefore `C` must depend on `B`.

```
operator record A
   Real x;

   operator function '+'
      input A a1;
      input A a2;
      output A a3;
   algorithm
      a3 := A(a1.x + a2.x);
   end '+';
end A;

model B
   A a1 = A(1);
   A a2 = a1 + a1;
end B;
```

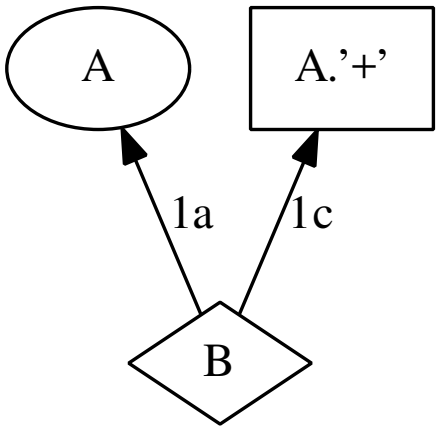

Figure 3.2: An example of why rule 1c is necessary. Changes in operator function '+' will propagate to `a2` in `B`.

```
package A1
   model M
   end M;
end A1;

package A2
   model M
   end M;
end A2;

package B
   extends A1; // Removed
   extends A2; // Added

   model C
      M m;
   end C;
end B;
```

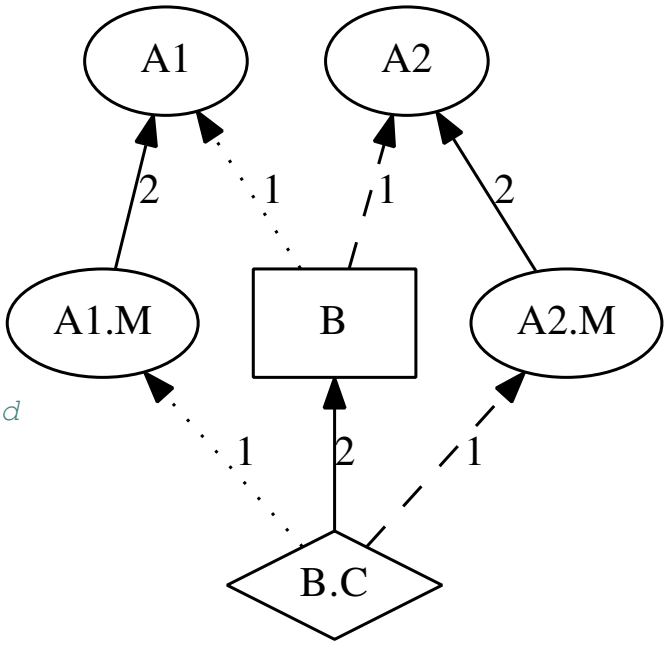

Figure 3.3: An example showing why rule 2 is necessary. When the class `B` is changed to extend `A2` instead of `A1`, the class of the component `m` in `C` is changed, therefore `C` must depend on `B`.

3.1.3 Rule 3

A class that contains a redeclaration depends on all super classes and enclosed classes of the replacing class (and all their enclosed classes and super classes recursively).

Rule 3 describes all dependencies involving redeclarations. There are four types of redeclarations: component redeclare modification, component declaration with the `redeclare` prefix, class redeclare modification, class declarations with the `redeclare` prefix. Figure 3.4 shows an example of why rule 3 is necessary for redeclare modifications. This example is also sufficient to motivate why we need it for the redeclare prefix, as the `redeclare` prefix can be replaced with modifications on the extends statements (see Section 2.2.6).

Figure 3.5 shows an example of why rule 3 must recursively depend on all enclosed classes and super classes. The example features a class declaration with the `redeclare` prefix, but can be constructed for each redeclaration type.

```
model A1
   function f
   end f;
end A1;

model A2
   function f
   end f;
end A2;

model B
   replaceable model A = A1;
   replaceable A1 a;
   Real x = A.f();
   Real y = a.f();
end B;

model C
   extends B(redeclare model A = A2,
             redeclare A2 a);
end C;
```

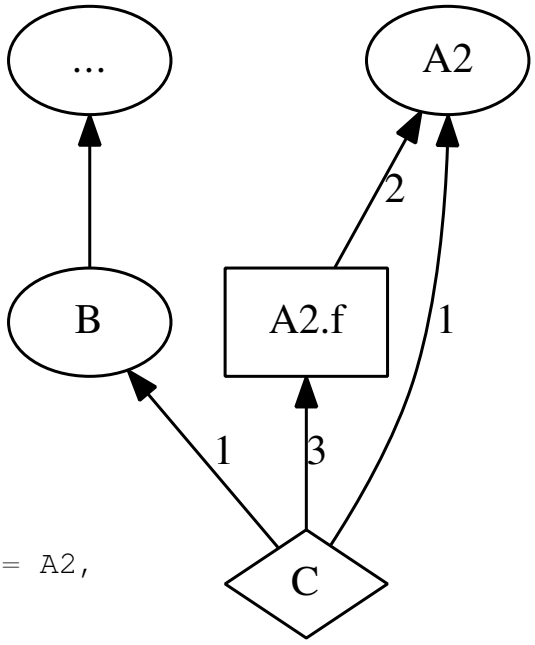

Figure 3.4: An example showing why rule 3 is necessary. When the function `f` in `A2` is changed, the value of `C.x` and `C.y` may also change, hence `C` must depend on `A2.f`.

```
package A
   model M
      Real x = P.f();
   end M;

   replaceable package P
      function f
      end f;
   end P;
end A;

package B
   function f
   end f;
end B;

package C
   extends A;
   redeclare package P
      extends B;
   end P;
end C;

model D
   C.M m;
end D;
```

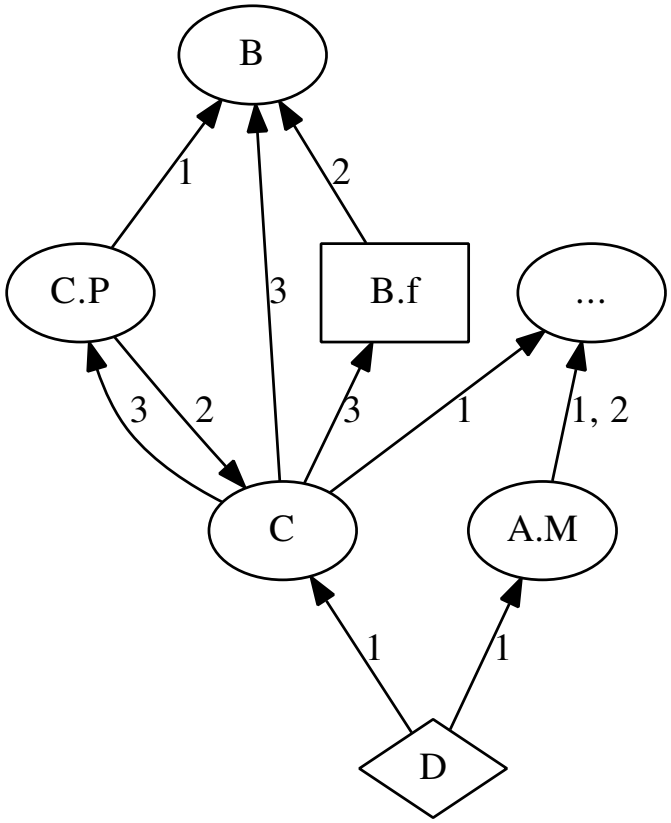

Figure 3.5: An example showing why rule 3 must depend on all enclosed classes and super classes recursively. In the model `D`, when the component `m.x` is set to the results of calling `P.f`, from the context of `C`, `P` is replaced with a new package that extends `B`. Therefore the function called will be `B.f` and `D` must depend on it.

3.1.4 Rule 4

In Modelica simulations, there are sometimes function calls that are not visible in the source code. These dependencies are caught by rule 4.

Rule 4a. *If a record or type encloses a function named* `equalityConstraint`*, it has a dependency on that function.*

If a class with the record or type specializations contains a function `equality-Constraint`, the class has a dependency to the function. Figure 3.6 shows an example of why rule 4a is necessary. The reason `equalityConstraint` needs to be included is because it will be called implicitly in the flattened class according to the Modelica specification.

Rule 4b. *If a class extends* `ExternalObject`*, it has a dependency on the enclosed function* `destructor`*.*

A class extending from `ExternalObject` must enclose functions named `constructor` and `destructor`. The `constructor` function must be called when a new instance of the class is made, this is caught by rule 1. The `destructor` will be implicitly called at some point, according to the specification. Figure 3.7 shows an example of this.

```
record A
   function equalityConstraint
   end equalityConstraint;
end A;

connector B
   A a;
end B;

model C
   B b1;
   B b2;
   B b3;
   Real x;
equation
   b1.a.value = time;
   Connections.root(b1.a);
   connect(b2, b3);
   Connections.branch(b1.a, b2.a);
   b1.a = b2.a;
   Connections.branch(b1.a, b3.a);
   b1.a = b3.a;
   x = b3.a.value;
end C;
```

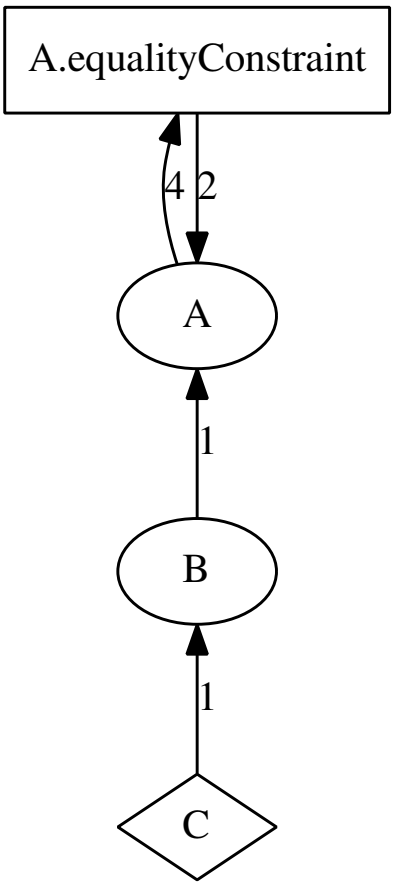

Figure 3.6: An example showing why rule 4a is necessary. The `connect` statement in `C` will be replaced with a function call to `A.equalityConstraint` during compilation, so `C` must depend on `A.equalityConstraint`.

```
class A
   extends ExternalObject;

   function constructor
      output A a;
      external;
   end constructor;

   function destructor
      input A a;
      external;
   end destructor;
end A;

model B
   A a = A();
end B;
```

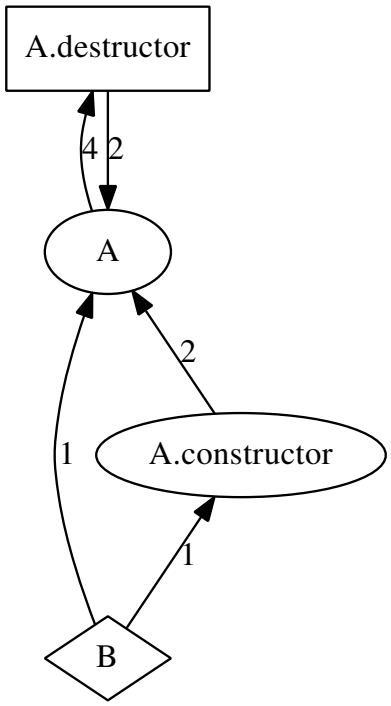

Figure 3.7: An example showing why rule 4b is necessary. The the destructor will be called implicitly, so `B` must depend on `A.destructor`.

Chapter 4
Method

In this section we describe our test selection algorithm and our approach to verify that the RTS technique is safe.:

4.1 Dependency analysis

To be considered safe, it is essential for the test selection algorithm to select any tests that might have changed, and to do this we perform a class dependency analysis to find class dependencies for test classes. In Chapter 3 we defined rules for direct dependencies from a class. The total set of dependencies, can be calculated as a dependency graph, where classes are represented as nodes and dependencies are directed edges.

If a class that is marked as changed is reachable from a test class, that test class is selected for testing. By calculating the dependency graph for all test classes it is therefore possible to select all tests that might be affected by changes.

4.1.1 External files

Modelica has functions that access external files, both as resources and as runnable C or Fortran code. This means that there are dependencies to external files that can not be found with a class analysis. It is therefore necessary to consider additions, removals and changes to existing files. For example, it is possible for an external C function to use any other file as input data. It is also possible for external functions to generate random values.

One alternative is therefore to always select classes that depend on external resources. To keep it reasonable though, another alternative is to flag all Modelica functions that call external functions as changed if any resource file (non-directory and non-*.mo) within the Modelica project directory is changed. Another alternative is to let the user decide if any external file has been changed, which would result in the selection of all tests that depends on any external file.

4.2 Verification

In this section we describe the method we used to verify that the RTS technique is safe. We also describe our test selection test suite for Modelica.

4.2.1 Mutation dependency analysis

Since Modelica is a complex language, we found it impossible to formally prove that our technique was safe. Instead we decided to test our implementation and verify that we selected all changed tests. We therefore had to show that for any change to the Modelica project, our technique would find all changed tests. If possible, we would have used code instrumentation to find all classes which a test class depends on. However, to our knowledge there exists no tool that can log accessed classes for Modelica.

Instead, we decided to use a process we call *mutation dependency analysis*. It is quite similar to "mutation testing" which is described in Section 2.4.

This process works by changing the program P to P' by altering some class, and then comparing which test classes that were identical in P and P'. The changes we made were similar to the ones used in mutation testing, which is why we call a single change to a class a *mutation*. To determine if a test class was different in program P', we compared it to its counterpart in P. The comparison for the test classes was between their textual flat-code representations. The important part to note about why we did this comparison is that if the test class in P' was changed, then there was a dependency from the test class to the mutated class, which is what we were looking for.

4.2.2 Test suite

We built a test suite for regression testing of the dependency analysis. It contains tests based on dependencies that we have manually determined from the Modelica specification [11]. It also contains tests for language constructs that we discovered from the mutation dependency analysis. That is, when we fixed bugs related to missing dependencies in the test selection algorithm, we also added new test cases.

This test suite has been made public and is available online [1] for anyone who wants to test their Modelica class dependency analysis.

[1] https://github.com/modelon/MCDTS

Chapter 5
Implementation

In this section we will describe the implementation of both the test selection technique and the verification system. Both implementations are done in the OCT compiler, which is generated with the meta-compilation tool JastAdd [8].

5.1 Dependency analysis

We used JastAdd's collection attribute feature to add a method to Modelica classes to statically calculate their direct dependencies according to Chapter 3. A dependency to a class is uniquely identified as the fully qualified name of the class. The method was declared for AST class `InstClassDecl`, which represents a class declaration in the instance tree.

To calculate all the dependencies (direct and transitive), we added another method to `InstClassDecl` that recursively added all direct dependencies from the dependent classes. The calculations of direct dependencies are cached, so if multiple classes have dependencies to the same class `c`, the direct dependencies of `c` will only be computed once.

The test selection technique runs the method to calculate all dependencies for each test class. If the set of dependencies for a test class intersects with the set of changed classes, the test is selected for testing.

An exception where we did not implement exactly according to the rules was for rule 1c, which is for operator functions. It was hard to find accesses to operator functions with our implementation, so instead we just let `operator record`s have a dependency on all its enclosed operator functions. This should have a small impact on the number of dependencies.

External file dependencies are handled separately, such that the class is given an extra dependency to external files. This dependency is represented as the string "external" instead of a class name. It is currently not further handled. See Section 4.1.1 for more

details.

5.2 Verification

To implement what we describe as mutation dependency analysis, we used JastAdd to modify the source tree in order to create mutations. This was easier than modifying the source code because we wanted the mutations to be found and applied automatically. The mutations were done to classes by mutating their AST nodes in the source tree. We performed our mutation dependency analysis for MSL.

We will now describe the algorithm for the mutation dependency analysis. For each test class we first created a reference value of its flat-code when there were no mutations to the program. We then mutated all classes one by one and recompiled the flat-code for each test class. The new flat-code was then compared with its reference value. We could then determine that the test was affected by the mutation if the flat-code was changed. This is also described in Algorithm 1.

```
for t in tests do
    ref_t ← flatcode(t);
end
for c in classes do
    mutate(c);
    for t in tests do
        fcode_t ← flatcode(t);
        if ref_t ≠ fcode_t then
            add_dependency(t, c);
        end
    end
end
```

Algorithm 1: Mutation dependency analysis algorithm.

We always mutated only one class for each iteration. If the flat-codes did not match, we could therefore be certain that the change introduced in the mutated class resulted in a change in the test class. We could then determine that there existed a dependency from the test class to the mutated class.

Due to MSL having an indeterministic external function that was evaluated at compile time, flattening some test models would result in different flat-code every time. We solved this by changing the function to be deterministic.

Given a class to mutate, we only performed a single type of mutation at once. We could then get more data about this kind of mutation, and it also reduced the risk of compilation errors. For each mutation type, we performed mutations to as many AST nodes as possible to increase the likelihood of changing the flat-code. When we looked for possible nodes to mutate, we stopped looking further into subtrees with a binary expression as root though. The reason for this is that there would be no difference in mutating one or several binary expressions if they were in the same expression.

Some mutations could result in compilation errors, that did not give any information. If this happened, we decided to try two times with just a single mutation to the class, in

hope that this mutation would not lead to compilation errors. If both attempts lead to compilation errors, we gave up and continued with the next mutation type to save time. If no mutation resulted in any found dependencies, this could either be because there actually was no dependency, or that the changes made by our mutations were insufficient to change the flat-code.

If there were compilation errors, there was no way to determine whether there existed a dependency from the compiled test class to the mutated class. We came to this conclusion by experimenting with a class that caused compilation errors for a test class. We first asserted that mutating the class caused a compilation error for the test. We then removed the mutation and instead manually changed the source code for the class that was previously mutated in such a way that the flat-code should change if there was a dependency from the test class. We then recompiled the test class and found that the flat-code was not changed, which means that the compilation error was not related to a dependency. Instead, it was the compiler that did extra error checking, and this was not something that we could easily modify.

The different type of mutations we performed, and examples of each type, are listed in Table 5.1. To perform the mutation we first used static analysis to determine what AST nodes could be mutated, and by what type of mutation. We then tried to apply all the mutations of the same type, and if this resulted in compilation errors, we instead performed them one at a time, as described in Section 4.2.1.

Table 5.1: Examples for mutation types.

Mutation type	Example before	Example after	Specialized
Add component in function	See Listing 5.1	-	yes
Arithmetic binary expression	1 + 2	2 + 1	no
Literal expressions	39	40	no
Logical binary expression	f() > 0	f() <= 0	no
Redeclare functions	See Listing 5.2	-	yes
String comment	M m; “comment”	M m; “mutated”	no

In addition to using mutations which are used in classical mutation testing (such as changing the value of literal expressions) we also used *specialized* mutations that targeted a certain Modelica language construct. One of the reasons we introduced specialized mutations was because we wanted mutations that could target common language constructs such as functions. Another was that we wanted to target language constructs we thought could result in unexpected dependencies. Since we knew the environment of the mutation, we could invest effort in making the mutations compile and produce changes to flat-code more often. This is compared to for instance mutations to literal expressions, since they can appear about anywhere.

We will now describe the different mutations in more detail. First of all, we had a mutation type that targeted *String comment* nodes and changed their content. String comment is a language construct that has no semantic meaning but can be included in the flat-code by the OCT compiler.

We also mutated *arithmetic binary expressions* and *logical binary expressions*. Both types were mutated by switching to another operator, with two exceptions. Addition is the only binary operation that can be done on strings, so it can not be replaced with another

operator without potentially causing errors. For additions, the mutation instead switched left and right operands. Multiplication on matrices can not be replaced with another binary operation, nor can the operands be switched. To account for this, multiplications instead were changed to have an additional multiplication with 2. As an example, $a \cdot b$ becomes $a \cdot b \cdot 2$.

Literal expressions were also mutated. String literals were appended with a character, while integers were incremented by 1, and floats were multiplied with 2 (or were changed to 1 if their value was 0).

Add component in function is a specialized mutation that targets function declaration nodes. It adds a protected component to the function and adds an annotation which indicates for the compiler to not inline the function. If the function had been inlined, the protected variable would be removed and thereby make the mutation useless. An example is given in Listing 5.1. The changes in function `f` will be reflected in the flattened version of `A` since it accesses `f`. One of the main reasons for this mutation was to find dependencies to implicitly called functions, as described by rule 4.

Redeclare function is also a specialized mutation. This mutation targets classes that have a base class that declares a replaceable function. When applied, this mutation adds modified function declarations to the targeted class, such that the functions redeclares the base class's functions. An example is given in Listing 5.2, which illustrates that `B.f2()` accesses `B.f` as a result of the mutation.

```
function f
  output Real y;
protected                          // Added
  Real MUTATION_VARIABLE;          // Added
algorithm
  y := 0;
  annotation(Inline = false); // Added
end f;

model A
  Real x = f();
end A;
```

Listing 5.1: Example of the "Add component to function" mutation. The mutation adds the local variable `MUTATION_VARIABLE`, which will show up in the flat-code for `A`.

```
model A
  replaceable function f
    output Real y;
  algorithm
    y := 0;
  end f;

  function f2 = f;

end A;

model B
  extends A;

  redeclare function extends f   // Added
  protected                      // Added
    Real MUTATION_VARIABLE;      // Added
    annotation(Inline = false); // Added
  end f;                         // Added
end B;

model C
  Real x = B.f2();
end C;
```

Listing 5.2: Example of the "Redeclare function" mutation. The mutation redeclares `f` in `B` and adds the local variable `MUTATION_VARIABLE` at the same time. This changes the behavior of `B.f2` and will therefore also change the flat-code for `C`.

Chapter 6
Evaluation

In this chapter we will evaluate our technique by comparing its performance to the technique defined by Hedblom and Rundquist [3]. In this section we denote our technique as the "instance tree technique" and Hedblom and Rundquist's as the "source tree technique". We will also evaluate our verification. To do this we will for each mutation type compare the results in terms of found dependencies. To provide a basis for improvements for specialized mutations we will also show the total number and distribution of specialized classes that we found dependencies to.

6.1 Test selection performance

In this section we evaluate the precision and time savings of our test selection technique. We do this by showing that we have increased the average precision and increased the average time savings for small changes in MSL and the *Heat Exchange Library (HXL)*. MSL consists of 5946 classes, where 366 are tests. The corresponding number for HXL is 871 classes where 227 are tests.

All measurements are done with both our instance tree technique and with the source tree technique. To make the comparison fair, the source tree analysis was modified to make it safe (see Section 7.1).

6.1.1 Precision

One of the goals with this thesis was to define a test selection technique with high precision. In Table 6.1 we show the average number of found dependencies in MSL for both our instance tree analysis and the source tree analysis. As can be seen, the instance tree analysis has a lower average amount of dependencies. We therefore reached our aim to increase the precision.

Table 6.1: Average number of transitive dependencies per class and per test class in MSL.

Classes	Avg. num. deps.	% change
Source tree any class	128.8	-
Instance tree any class	84.0	-34.8%
Source tree test class	194.0	-
Instance tree test class	164.4	-15.2%

We discovered that the dependencies from the instance tree analysis is a subset of the ones from the source tree analysis, for all test classes in MSL. It is not quite a subset when used on *all* classes in MSL though. Our implementation found 73 dependencies that were not found by the source tree implementation. The total number of dependencies found by our analysis was 498240. By excluding tests not found by the source tree analysis we could reduce the number of dependencies found by 0.015%. The reason for the extra dependencies is due to our implementation for dependencies to operator functions (see Section 5.1). We think this amount of extra dependencies is negligible.

6.1.2 Savings when one class or file changed

We compared the savings for our instance tree technique with the source tree technique. The comparison is based on the assumption that a single class or file (which contains a set of classes) is changed. According to the techniques, each test class has a set of dependencies, and if there exists a dependency to any changed class, then the test is selected.

To calculate the savings we first mapped each test to a typical simulation time for that test. The total testing time was then calculated as the simulation time for all selected tests plus the execution time for the test selection analysis. By doing this for all classes in MSL we could calculate the average and median savings by comparing the total testing time with the time it takes to run all tests without any test selection.

The results for the comparison with the source tree technique was that our instance tree technique had a better performance for both average savings and for average and median testing times. Although the analysis execution time was longer, the reduced test execution time was worth it since it had better savings. These results are shown in Table 6.2. The data for the calculations for changed classes is also shown in Figure 6.1. The y-axis corresponds to the percent of time it would take to run all tests, and the x-axis corresponds to a changed class. The classes have been sorted in ascending order of test time.

Table 6.2 also shows that the analysis execution time is very low compared to the total execution time for running all tests. The precision therefore did not have to increase much in order for its benefits to outweigh the cost of the increased analysis execution time. Relative to the source tree analysis, our instance tree technique provides on average 8.0% shorter testing time if a file is changed, and 35.1% shorter testing time if only a single class is changed. These values are also shown in Table 6.3.

We also took time measurements for HXL and performed the same comparison as for MSL. The reason for this is that HXL has a different structure compared to MSL. In Table 6.4 we show that the instance tree technique provides higher savings for both changes to one class and to one file. Figure 6.2 show the data used to calculate the savings. As

Table 6.2: Performance results for MSL. All units are in % of the time it takes to run all tests.

Analysis	Avg. savings	Avg. test time	Median test time	Analysis time
File: Source	87.880	12.083	1.737	0.036
File: Instance	88.845	11.018	1.186	0.136
Class: Source	93.100	6.864	0.325	0.036
Class: Instance	95.519	4.345	0.215	0.136

Table 6.3: How much shorter our testing times are for MSL, compared to the source tree technique.

Change type	Percent shorter (%)
File	8.0
Class	35.1

shown in Table 6.5, for HXL our technique provides on average 22.7% shorter testing time when only one file has changed, and 24.7% when only one class has changed.

Table 6.4: Savings compared to not using test selection. Also test times and analysis execution time in % of running all tests for HXL.

Analysis	Avg. savings	Avg. test time	Median test time	Analysis time
File: Source	74.765	25.134	15.693	0.101
File: Instance	80.482	19.399	3.768	0.119
Class: Source	72.011	27.888	15.693	0.101
Class: Instance	78.934	20.947	3.613	0.119

Table 6.5: How much shorter our testing times are for HXL, compared to the source tree technique.

Change type	Percent shorter (%)
File	22.7
Class	24.7

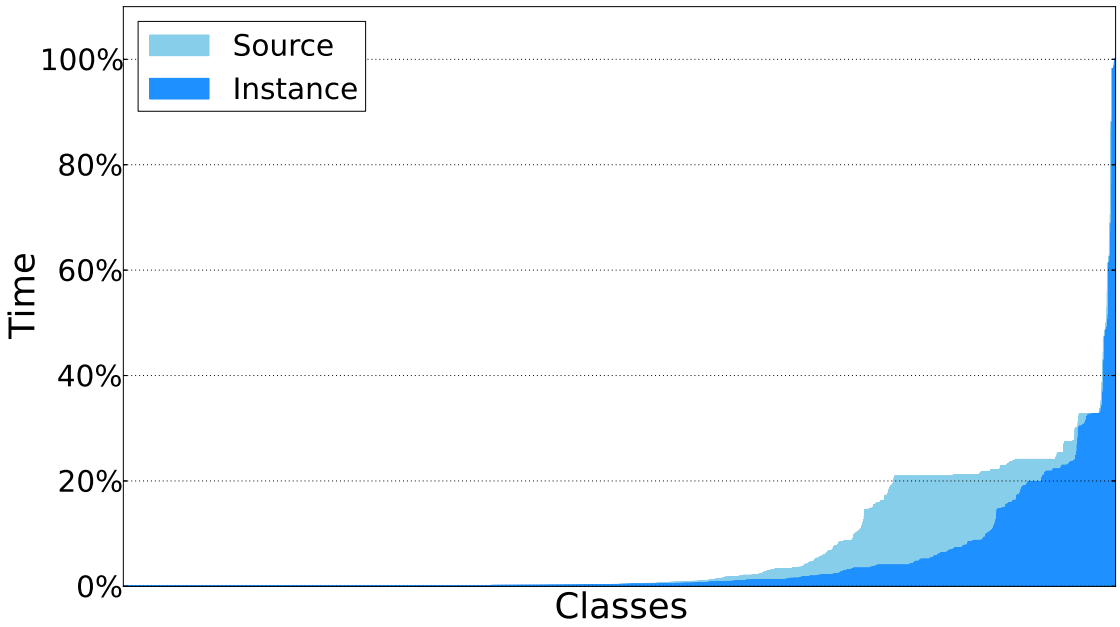

Figure 6.1: Expected testing times when one class changed in MSL, for both instance and source tree techniques. The figure can be viewed as a bar chart where each x-value represents a class, and the corresponding y-value is the total expected testing time if the class is changed. The classes on the x-axis have been sorted in ascending order.

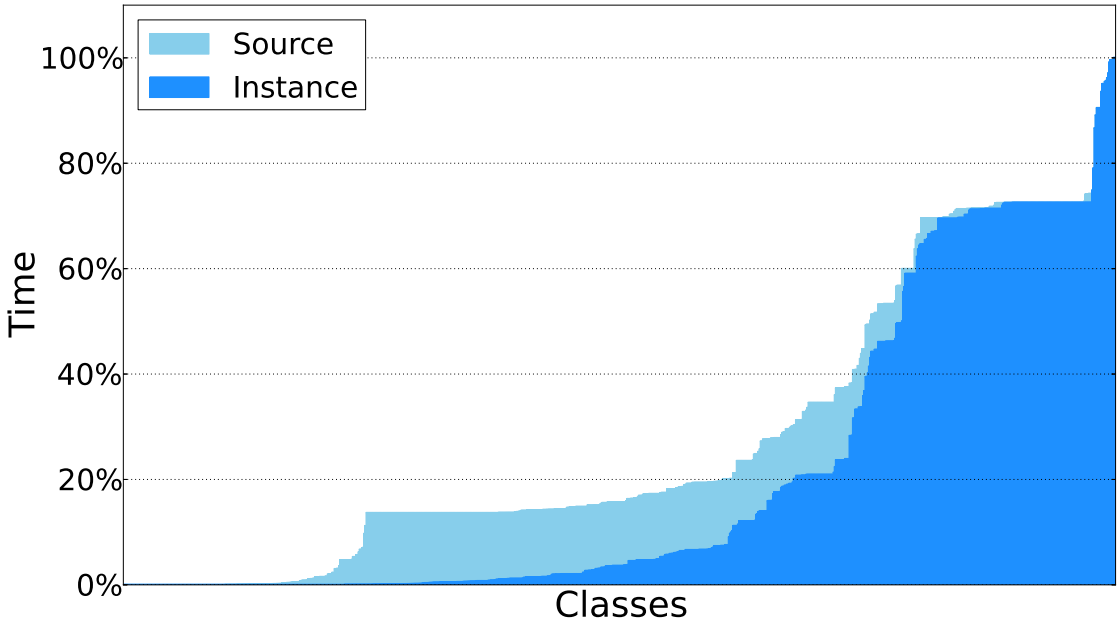

Figure 6.2: Expected testing times when one class changed in HXL, for both instance and source tree techniques.

6.1.3 MSL commit history

In order to get more realistic measurements we wanted to analyze the time savings of a real example. We chose to analyze how much time would have been saved if our analysis was used to select tests for every commit in MSL. In our measurements we assume that all the tests exist for all commits and that their simulation times do not change.

In Figure 6.3 you can see the results of our instance tree analysis and the source tree analysis. On the y-axis is the testing time of the selected tests in percent (100% means all tests selected). On the x-axis are the commits sorted by testing time in ascending order. The average saving is 68.9% for the instance tree technique and 57.0% for the source tree technique. This indicates an improvement in precision.

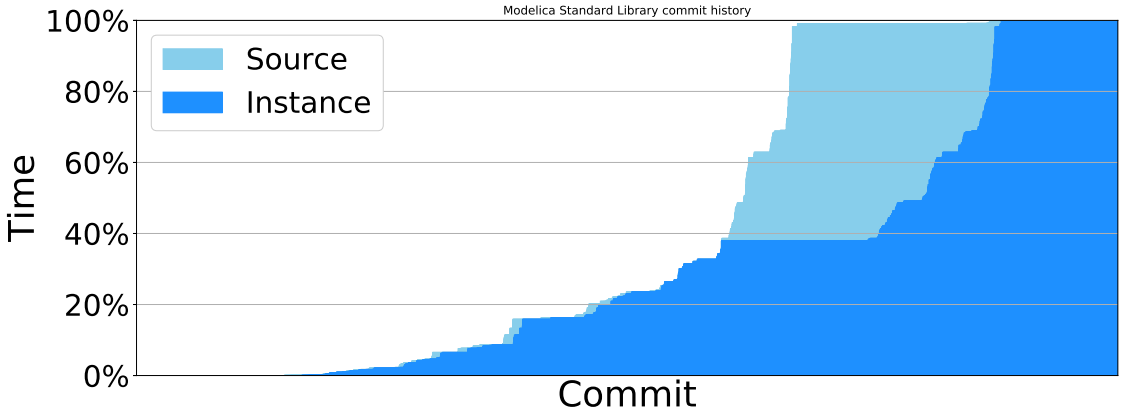

Figure 6.3: Testing time for the instance tree analysis on the MSL commit history.

6.2 Verification results

The mutation dependency analysis resulted in a set of class dependencies for each test class in MSL. These sets of dependencies are available online [1] for anyone who wants to define a Modelica test selection technique. From our mutation dependency analysis we discovered rules 1c and 4. We also discovered that we had to make adjustments to rule 3. Furthermore we found six implementation bugs.

Another important discovery from the mutation dependency analysis was that the source tree technique was not safe. In this report, all measurements are based on a version of the source tree analysis where these problems have been fixed, such that all comparisons are between safe techniques. This means that the results for the source tree technique will be different in this report compared to the report by Hedblom and Rundquist [3]. More details about the problems are given in Section 7.1.

Although we did not fully verify that our technique is safe, we have verified that it is safe for large parts of MSL based on the results of our mutation dependency analysis. The verification only includes test classes though. It would be possible to extend the verification to find dependencies from all classes that can be flattened, however this would be very

[1] https://github.com/modelon/MCDTS

time consuming. Just running the mutation dependency analysis for MSL's 366 tests took about 280 days of execution time, and the total amount of classes that can be flattened is much larger.

6.3 Mutation type results

In this section we will compare the results for the different kind of mutations we used. The results from the mutation dependency analysis are comparable since each type of mutation was executed independently of the other types. We have also abbreviated the names of the mutation types in this section. The mappings for the abbreviations are shown in Table 6.6.

Table 6.6: Abbreviations for mutation types.

Short name	Full name
AddComp	Add component in function
Arit	Arithmetic binary expression
Lit	Literal expression
Bool	Logical binary expression
Redecl	Redeclare function
Comment	String comment

For each mutation type, we counted the number of classes that we applied the mutation to. For each mutated class, we also counted how many dependencies from test classes to this mutated class we found. We denote the efficiency for the mutation type as the average number of dependencies found per mutation. The results for these metrics and the total mutation dependency analysis execution times are shown in Table 6.7.

As can be seen from Table 6.7, the mutation with the highest number of found dependencies was `Lit`. This is most likely related to the fact that it was the most pervasive mutation with about 52% of the classes mutated. `Lit` could have had even better results, however as mentioned in Section 5.2 we stopped looking for mutations in binary expressions, which means that we did not include all literal expressions. Both `AddComp` and `Comment` had about half the performance of `Lit`, while the rest were worse. Since the execution time is very long (total of about 280 days) it is important to choose mutations that have a high efficiency. In this regard, all mutations types were about equal except `Redecl` that was much worse, and `Comment` that was slightly better.

To improve the verification with mutation dependency analysis, it is important to find as many new dependencies as possible. To see which mutations that found "new" dependencies, we also counted the number of unique dependencies and unique mutations for each mutation type, and this result is shown in Table 6.8. We define a mutation as unique when it is the only type that can mutate a class (without regard to compilation errors). We define a dependency as unique if only one mutation type discovered it. As can be seen in this table, the number of unique mutations is somewhat proportional to the number of unique dependencies. A good strategy would therefore be to find mutations types that can be applied as unique mutations. This was the plan with `AddComp` and `Redecl`, however as seen in the table and figures, the result were very different. As the result was positive for `AddComp`, it would appear that it is beneficial to create specialized mutations for common

Table 6.7: Number of mutated classes and data related to number of found dependencies.

Mutation Type	Mutated classes	Dep. found	Efficiency	Exec. time, days
AddComp	30.1% (1792)	7205	4.02	50.8
Arit	34.2% (2035)	8742	4.30	56.4
Lit	53.4% (3173)	14404	4.54	80.6
Bool	15.6% (925)	3984	4.31	26.5
Redecl	1.1% (68)	16	0.24	2.2
Comment	27.7% (1650)	10037	6.08	43.0

language constructs. Considering the more general mutations, `Lit` and `Comment` found many more unique dependencies than `Arit` and `Bool`. The reason for this is most likely because binary expressions appear where other mutations also are applicable.

Table 6.8: Unique mutations and dependencies for the mutation types.

Mutation Type	Unique mutations	Unique dep.
AddComp	425	2725
Arit	16	225
Lit	805	6698
Bool	17	228
Redecl	17	5
Comment	206	3540

We also counted the total number of times a mutation type got compilation errors for all attempts to mutate a class. This result is shown in Table 6.9. It was to be expected that `Arit`, `Lit` and `Bool` would result in compilation errors, however it was not for `AddComp`. Although `AddComp` is a more complex mutation, our plan was for it to never cause compilation errors. Most likely there are some language constructs that we did not consider, and to make it always compile, some updates will have to be done.

Table 6.9: Compilation errors for each type of mutation.

Mutation Type	Classes compile err.
AddComp	6.9% (132)
Arit	0.4% (9)
Lit	4.3% (142)
Bool	1.6% (15)
Redecl	0.0% (0)
Comment	0.0% (0)

6.4 Missing dependencies

In our mutation dependency analysis we could not find dependencies to all classes. We believe that one of the reasons was that not all classes are used in tests. Another reason that we know of was that we could not find mutations for all classes. Furthermore, in some cases the mutations did not result in any change that propagated to the flat-code, which was also a problem. Table 6.10 shows that we managed to find at least one dependency from any test class to about 40% of the classes in MSL. The table also shows that we attempted to mutate about 77% of the classes, which indicates that about half of the mutated classes did not result in any found dependencies. To fully verify the safety of our technique for MSL, we would have had to find all dependencies for each test class. To achieve this we would most likely have to use code instrumentation instead of our approach with mutations.

Table 6.10: The number of classes we tried to mutate and the number of classes which we found dependencies to, compared to the total number of classes.

Total number of ...	Num. (%)
mutated classes with deps.	2345 (39,4)
classes attempted to mutate	4587 (77,1)
MSL classes	5946 (100.0)

In Table 6.11 we show the distribution of classes (sorted by specialization) which we did not find any dependencies to. As can be seen in the table, the most common class type to which there are no dependencies is function. We think that one of the most important things to do in order to improve the mutation dependency analysis is to improve the specialized mutations and create new ones. This includes to investigate why the specialized mutation for functions "Add component in function" sometimes resulted in compilation errors (see Table 6.9) and why it many times did not result in a changed flat-code (see Table 6.11).

Two notable results are that we did not find dependencies to any unspecialized class or operator. We have investigated some unspecialized classes and we did not find anything that indicated that they could not be mutated like any other class. To check if any unspecialized class actually was used in any tests, we ran our dependency analysis on the tests and found dependencies to 8 unspecialized classes. We investigated those 8 classes, and our conclusion was that none of them could be mutated. For operators, we did not find any mutations that we could apply. The reason for this is that the operator specialization is similar to the package specialization, except that it has even more restrictions. There is no point in mutating this class though, since there will always be a dependency to it according to rule 2 in combination with rule 1c, if it contains any function declarations.

From the mutation dependency analysis we also found a dependency to only one operator record of 22 possible. The reason for this is that the only operator record with a full definition is the class `Complex`. All other operator records in MSL extend `Complex` (and sometimes also redeclare the type of the input). It would be possible to mutate these operator records by adding modifications, however it would most likely only yield the same result as mutating models. Since the models are more in number, they should

have higher priority for implementation of better mutations.

Our conclusion is that most classes needs more testing, and although we have used general mutations such as "Literal expression" mutations that were successful in our verification, we think that the next step to improve the mutation dependency analysis is by introducing more specialized mutations. This makes it easier to target specific types of classes and reduces the risk of compilation errors.

Table 6.11: Number of classes which we both did and did not find any dependencies to for MSL. The table is sorted by the number of classes with no found dependency to. These numbers include both classes that we tried to mutate, but did not find any dependencies to, and classes that we did not find any mutations for.

Class specialization	Num. no deps (%)	Num. deps (%)	Total
function	1094 (58,2)	786 (41,8)	1880
package	686 (88,6)	88 (11,4)	774
model	576 (36,9)	987 (63,1)	1563
type	533 (77,7)	153 (22,3)	686
class	220 (100,0)	0 (0,0)	220
record	219 (66,6)	110 (33,4)	329
block	189 (53,2)	166 (46,8)	355
connector	52 (51,5)	49 (48,5)	101
operator record	21 (95,5)	1 (4,5)	22
expandable connector	4 (66,7)	2 (33,3)	6
operator function	4 (57,1)	3 (42,9)	7
operator	3 (100,0)	0 (0,0)	3
sum	3601	2345 (39,4)	5946

Chapter 7
Discussion

In this section we will discuss the difference between our instance tree technique and the source tree technique in more detail. We will also discuss an alternative approach for the test selection technique that we considered.

7.1 Comparison with source tree analysis

We will begin with the comparison of our instance tree technique, and Hedblom and Rundquist's source tree technique.

One of the major problems with the source tree analysis was that it could not resolve the name of an accessed class in some cases when using a composite access. That is, the lookup for the class from a name failed. It therefore had to select all classes that could possibly be associated with the class that could not be resolved. This problem does not exist in the instance tree, which made it possible to define implementation independent rules.

On the other hand, the problem with the instance tree is that it takes longer to instantiate and requires more memory. It is also technically more complex than the source tree, which makes the test selection algorithm harder to implement.

If we just compare the set of dependency rules for our technique and for Hedblom and Rundquist's, we can summarize it as the following:

- Ours can resolve all accesses.
- Ours checks for special language constructs and operators.
- Ours creates dependencies to enclosed classes recursively for classes with any re-declarations.
- Theirs creates dependencies to enclosed classes recursively for accesses to the last resolvable access in a qualified access.

In many cases, the fact that theirs includes enclosed classes from accesses saves it from having to consider most special language constructs. The same rule also finds most of the dependencies for redeclarations, however as we have shown in Section 6.2, it was not enough and had to be fixed. It is also this rule which makes their technique coarse, and by reducing it to only redeclares we have managed to increase the precision.

According to the rules, we expected our technique to always find a subset of the dependencies which theirs did. We checked if this was the case, and the result was that we found some dependencies which they did not. That is, we did not find a subset of dependencies. There are two reasons for this. The first is that the instance tree analysis includes a dependency to the `equalityConstraint` function, as per rule 4a. The source tree analysis only finds a dependency to `equalityConstraint` if it has an access to the enclosing class. The second is that our implementation of rule 1c will create dependencies from one operator function to all other operator functions in the same operator record. Since both techniques are safe, the extra dependencies are redundant and can be removed by improving our implementation. However a reduction of transitive dependencies by 0.015% is not much of an improvement.

```
model A
   model M
      Real x = f()
   end M;

   replaceable function f
   end f;
end A;

model B
   extends A;
   redeclare function f
   end f;
end B;

model C
   B.M m;
end C;
```

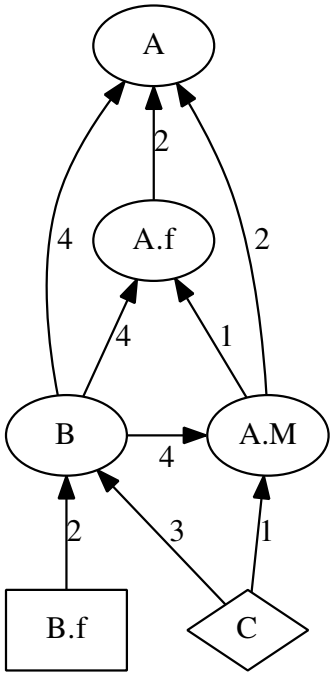

Figure 7.1: Dependency graph showing that there is no dependency from `C` to `B.f`. Note that the rules applied are for the source tree analysis.

As mentioned before, the source tree technique was not safe. The problem was that the analysis does not account for classes with the `redeclare` prefix. The technique will fail to find all dependencies since it has no equivalent to our rule 3 for redeclares. An example of where it fails can be seen in Figure 7.1. Note that the figure uses the rules and rule numbering defined in Hedblom and Rundquist's report [3]. The technique does not find the dependency from `C` to `B.f` because their rule 4 does not apply to `B` in the access to `B.M`. This could be fixed by adding dependencies to enclosed classes with redeclare prefix

(and their enclosed classes recursively).

There was also an implementation bug which caused the analysis to include too many dependencies in some situations, which we have fixed. In addition to our fix to make the source technique safe (see Section 6.2), this is another reason why the results for the source tree technique will be different in this report.

Regarding the size of the implementation, the size of our technique's class dependency analysis is 201 lines of source code, while theirs was 134. This number was measured with the tool *cloc*[1] and the input source files were treated as Java files.

7.2 Alternative technique

An alternative to defining a small amount of direct dependencies and then using transitivity to find the other dependencies is to find all dependencies at once as direct dependencies. Since expanding the instance tree will resolve all modifications, it is possible to find all dependencies, including the ones in modified classes. Our rule 3 can then be skipped, since an access to a redeclared class will find the correct class.

If applied to the example in Figure 3.4, the analysis would expand the extends clause node and the contained component declaration nodes and find a function call node which accesses `A2.f`.

We also implemented this algorithm, and as can be seen in Table 7.1 savings are worse compared to the transitive dependencies approach, although the precision is higher as shown in Table 7.1 (the average test execution time is lower). The reason for this is that the analysis execution time is too high relative the increased precision. In Figure 7.2 the top layer represents time spent on test selection, and the bottom layer represents test execution time. As can be seen, a lot of time is wasted on analysis execution time even if no tests have to be executed. Another problem was also that this algorithm required a lot more memory compared to using transitive dependencies.

Table 7.1: Performance results for MSL for algorithms based on source tree and instance tree. All units are in % of the time it takes to run all tests.

Analysis	Avg. savings	Avg. test time	Median test time	Analysis time
File: Source	87.880	12.083	1.737	0.036
File: Inst. Transitive	88.845	11.018	1.186	0.136
File: Inst. Alternate	86.458	11.267	1.408	2.274
Class: Source	93.100	6.864	0.325	0.036
Class: Inst. Transitive	95.519	4.345	0.215	0.136
Class: Inst. Alternate	94.444	3.282	0.086	2.274

[1] https://github.com/AlDanial/cloc

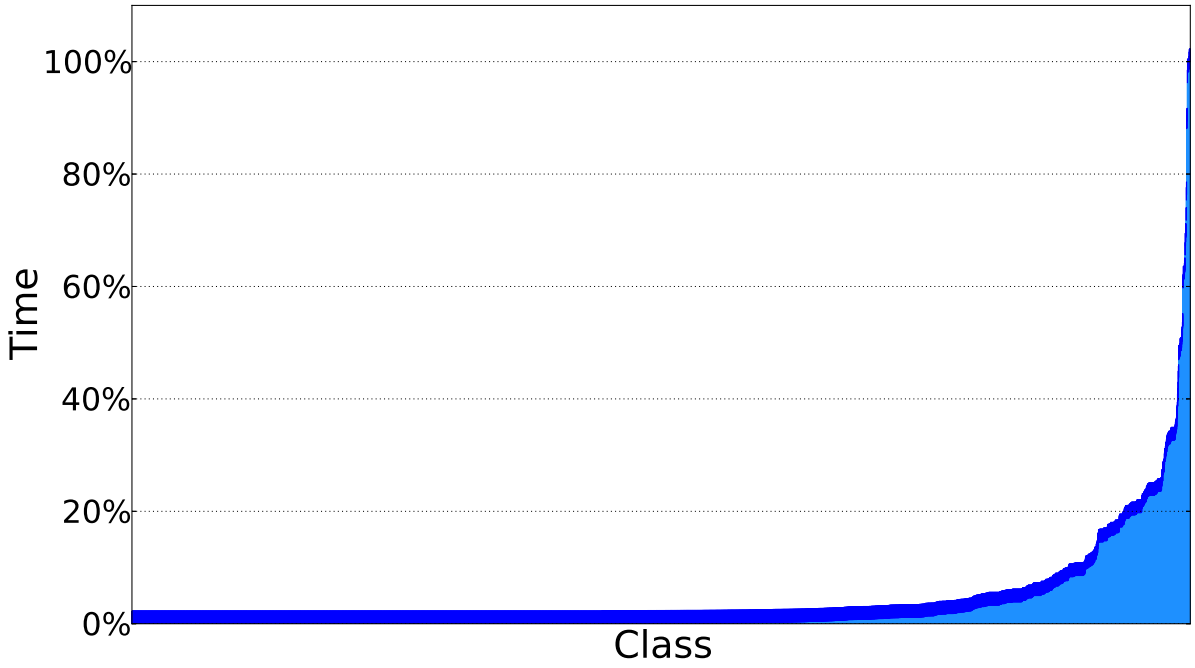

Figure 7.2: Expected execution times when one class changed in MSL, for test selection with instance tree indirect algorithm. Top layer is test selection time and bottom layer is test execution time. 100% is the time for running all tests. The total expected testing time can be larger than 100%, since the total time also includes the time for test selection.

7.3 Replace functions mutation

The “Replace functions” mutation was originally meant to be a mutation that shadowed functions from the base classes, as can be done in Java for example. This is not valid syntax in Modelica though, so we instead changed it to redeclare replaceable functions.

At that point in time we were still learning the Modelica language (and we still are), and the consequences of redeclaring replaceable functions regarding the dependencies were not obvious. From the result of the mutation dependency analysis, we can see that the performance of the mutation was very low. This was related to the fact that the mutation rarely could be applied. The mutation was also overly complex in the regard that it was hard to get it to compile. Based on the results and on the effort to create the mutation, we think that creating simpler mutations that target a specific class specialization will provide the best results for Modelica-specific mutations.

7.4 Improvements to general mutations

Since it is very time consuming to perform the mutation dependency analysis, it is important to choose efficient mutations from the beginning. After we did the mutation dependency analysis we came up with some improvements for the general mutations, however we did not implement them.

Our `Comment` mutation only attempted to mutate existing comments. One way to make it more pervasive is to add comments whenever possible.

Another general mutation would be to add simple value modifications on accesses. Many types of classes does in some way contain a value of a primitive type, for instance `Real`. Those classes can be mutated by modifying the start value of this `Real`. This mutation might not be trivial to implement, but it will most likely provide good results since the modification will show up in the flat-code if there is a dependency.

7.5 Threat to validity

Although we have verified the dependency rules for large parts of MSL, it is still possible that our dependency rules are not complete. For instance, there might be language structures that we have missed if they were not present in MSL. Another problem was that we could not determine the completeness for the set of actual dependencies for test classes that we found from our mutation dependency analysis. The verification of the technique’s safety is therefore not complete.

Chapter 8
Related work

There exists RTS techniques which are based on static analysis, runtime analysis or both. In this section we briefly describe some techniques of each type and compare our technique with the different approaches.

8.1 Runtime and static analysis

One of the most basic RTS techniques that use both static and runtime analysis is called the *class firewall* technique and was first introduced by Hsia et al. [13]. The technique requires calculating one set of classes for each test, and another set of classes for each changed class. To calculate the set for the tests, the idea is to instrument the test cases and add all classes used to a set called the *touch set*. The set for the changed class `C` is instead statically computed as the set of classes that transitively have a dependency on `C`. This set is denoted as the class firewall of `C`. If the touch set for a test intersects the class firewall for any changed class `C`, then the test is selected.

Another approach called *TwoPhase* performs the test selection in two phases to keep the analysis time low while increasing the precision. In the first phase it partitions the program such that only parts of the program that might be affected by the changes remains. This is done with static analysis. In the second phase it uses runtime coverage information for the partition of the program that remains. If the changes might lead to new execution paths for a test, then the test is selected [14]. Harold et al. implemented this technique in the tool `DejaVOO` for Java.

8.2 Runtime analysis

A technique called *Change-based* test selection was proposed by Skoglund and Runeson [15] for Java. To form this technique, they removed the class firewall set from the

original class firewall technique, and instead only select tests if their *touch set* include a changed class. This way, their technique only requires runtime analysis. They also provide a proof that *Changed-based* test selection is safe [15].

Another approach on how to perform test selection is implemented by the tool `Ekstazi` [5] for Java. The tool follows the concept of *Change-based* test selection, but it creates a dependency to all accessed files instead of classes. This is done by instrumenting the test and saving a dependency to each file it accesses. One of the major benefits with this approach is that dependencies to local external files are also collected. The verification of the tool's safety was based on the proof provided for *Changed-based* test selection [15].

8.3 Static analysis

An example of a technique that only uses static analysis is the technique defined by Hedin et al. called *Extraction-Based* RTS for Java [4]. They developed a tool called AutoRTS which statically calculates the dependencies for each test, and saves the dependencies as a dependency graph. Tests that depend on changed classes are selected for testing. By incrementally updating the graph instead of recomputing it from scratch, the test selection time is proportional to the size of the update instead of the project size.

8.4 Discussion

Our RTS technique for Modelica share similarities with the Extraction-Based RTS for Java in the sense that both techniques use static analysis to build a dependency graph and selects the tests that depend on changed classes. One difference is that the Extraction-Based technique performs incremental updates to the graph to save time, while our technique recomputes the graph after each change. The runtime of Java regression testing can be comparable to the time it takes to perform the test selection, why it is important to keep the test selection time short. In Modelica, the testing time is much longer compared to the test selection time, why reducing the test selection time (at least for our technique) won't provide a considerable increase in time savings.

A test selection similar to how Ekstazi for Java works would be possible for Modelica. Instead of using runtime information to get file accesses, it's possible to add a dependency to all files that the compiler loads during flattening of a test. This technique will probably be inefficient though, since the process of flattening is time costly, and it's possible that the compiler loads more files than necessary.

Using runtime analysis to get class dependencies like in imperative languages is not applicable for Modelica. Since tests in Modelica are transformed to equation systems, it's not possible to get a code coverage like in imperative languages. Because of this, techniques such as TwoPhase won't work. However, in the same way as dependencies to files can be added during the compilation process, it should be possible to add dependencies to classes necessary for compiling a test. This requires in-depth knowledge of the compiler though, and is at least for the OCT Modelica compiler that we have used not something easily achieved.

Chapter 9
Conclusions

We have defined a safe test selection technique for Modelica that is based on a class dependency analysis. To perform the analysis we have defined rules for Modelica class dependencies. We have shown that our technique has a higher precision and increased average time saving for MSL and HXL compared to the previous test selection technique by Hedblom and Rundquist. For MSL we managed to reduce the testing time by 88.8% when one file changed, and by 95.5% when one class changed, as compared to running all test classes. In comparison to Hedblom and Rundquist's technique, this corresponds to our technique having a 8.0% shorter test time for one file change, and 35.1% shorter for one class change. Similar results were attained for measurements on HXL.

We have worked with verifying the safety of our test selection technique by making certain that a subset of the actual dependencies that we found with mutation dependency analysis were found by our dependency analysis. Although we did not find dependencies to all classes with the mutation dependency analysis, we believe that this was partially because not all classes were tested, and partially because we could improve the choice of mutations and their implementation. More specifically, to further improve the verification we recommend further use of mutations that are specialized for specific types of classes. If possible, we also recommend the use of code instrumentation instead of mutation dependency analysis to find dependencies if possible.

9.1 Future work

For future work we think that the most important thing to do is to improve the verification of the test selection technique's safety. The verification we performed showed that the test selection algorithm implemented by Hedblom and Rundquist [3] was not safe, and although we have performed systematic verification, it is possible that the same might happen to our technique. We think that future work should strive to further verify our test selection technique so that it can be applied without any worry about it's safety.

One way to improve the verification is to continue with the mutation dependency analysis we performed by testing more libraries and improving the mutations. Another path which we did not try but did consider is to try to use code instrumentation to collect accessed classes during model flattening, and thereby find actual dependencies.

Bibliography

[1] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[2] Modelica testing toolkit. `http://www.modelon.com/products/model-testing-toolkit/`. [Online; accessed 8-May-2018].

[3] Erik Hedblom and Kasper Rundquist. Safe test selection for modelica using static analysis. Master's thesis, Lunds Tekniska Högskola, Department of Computer Science, Faculty of Engineering, 2017.

[4] Jesper Öqvist, Görel Hedin, and Boris Magnusson. Extraction-based regression test selection. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools, Lugano, Switzerland, August 29 - September 2, 2016*, pages 5:1–5:10, 2016.

[5] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pages 211–222, 2015.

[6] Source code instrumentation overview. `https://www.ibm.com/support/knowledgecenter/SSSHUF_8.0.0/com.ibm.rational.testrt.doc/topics/cinstruovw.html`. [Online; accessed 8-May-2018].

[7] Jmodelica.org. `https://jmodelica.org/`. [Online; accessed 7-May-2018].

[8] Görel Hedin and Eva Magnusson. Jastadd–an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, 2003.

[9] Optimica compiler toolkit. `http://www.modelon.com/products/modelon-creator-suite/optimica-compiler-toolkit/`. [Online; accessed 7-May-2018].

[10] Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a modelica compiler using jastadd attribute grammars. *Sci. Comput. Program.*, 75(1-2):21–38, 2010.

[11] Modelica Association et al. Modelica - a unified object-oriented language for physical systems modeling - language specification version 3.2 revision 2. `https://modelica.org/documents/ModelicaSpec32Revision2.pdf`, 2013. [Online; accessed 7-May-2018].

[12] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[13] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin, and Yasufumi Toyoshima. Class firewall, test order, and regression testing of object-oriented programs. *JOOP*, 8(2):51–65, 1995.

[14] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 241–251, 2004.

[15] Mats Skoglund and Per Runeson. Improving class firewall regression test selection by removing the class firewall. *International Journal of Software Engineering and Knowledge Engineering*, 17(3):359–378, 2007.

EXAMENSARBETE Improved precision and verification for test selection in Modelica
STUDENT Markus Olsson, Filip Stenström
HANDLEDARE Niklas Fors (LTH), Jon Sten (Modelon AB)
EXAMINATOR Görel Hedin (LTH)

Förbättrat testurval för Modelica

POPULÄRVETENSKAPLIG SAMMANFATTNING **Markus Olsson, Filip Stenström**

Testurvalstekniker för mjukvara kan spara mycket tid vid testning. I detta arbete har en ny testurvalsteknik för modelleringsspråket Modelica definierats, med högre precision än tidigare tekniker och med verifierad säkerhet.

I dagens mjukvaruindustri släpps det kontinuerligt nya uppdateringar. För att säkerställa att dessa uppdateringarna inte inför nya buggar, kan man använda så kallade regressionstester. Det innebär att man testar programmets funktionalitet för att upptäcka om något oförutsett ändras. Ett problem är att det kan ta lång tid att köra alla tester. Men det är onödigt att köra tester som inte har påverkats av en ändring. Därför kan man använda en testurvalsteknik för att välja ut och köra de tester som man tror kan ha påverkats av uppdateringen. Om urvalstekniken garanterat väljer alla tester som har påverkats så kallas den *säker*.

I vårt examensarbete har vi definierat en säker testurvalsteknik för modelleringsspråket Modelica. Vårt mål var att förbättra precisionen – det vill säga att minska antalet oförändrade tester som valdes ut – jämfört med en tidigare teknik och det lyckades vi med. Den genomsnittliga testtiden minskade med ca 96% för Modelicas standardbibliotek vid små ändringar jämfört med att köra alla tester. Vår teknik minskar testkörtiden med ca 35% jämfört med den enda tidigare testurvalstekniken för Modelica som vi känner till.

För att utvecklare ska våga använda en testurvalsteknik är det viktigt att visa att den är säker. I vårt arbete analyserade vi först Modelicas standardbibliotek för att hitta beroenden från tester till vanliga klasser. Med hjälp av denna informationen kunde vi sen skapa en uppsättning tester som verifierar att en urvalsteknik för Modelica är säker. En intressant upptäckt under vår verifiering av teknikens säkerhet var att den tidigare urvalstekniken faktiskt inte var säker. För att göra den säker lade vi därför till det som saknades i dess implementation. Det var den säkra versionen som vi använde till tidsmätningarna.

Vår teknik är baserad på att analysera beroenden mellan Modelica-klasser. En stor del av vårt examensarbete var att definiera reglerna för dessa klassberoenden. Tekniken fungerar på så sätt att den först hittar alla beroenden för testklasserna i ett Modelica-projekt, och väljer sedan ut de tester som har beroenden på klasser som användaren har ändrat på.

För att tekniken ska spara tid är det viktigt att beroendeanalysen inte tar längre tid än vad det hade tagit att köra alla testerna. För t.ex. Java-program kan detta vara ett problem, men i Modelica tar testerna så lång tid att vår analystid är under 0,3% av den totala testtiden.

En del av verifieringen var att utföra automatiska ändringar i varje klass, en i taget, och undersöka vilka tester som påverkas. Om ett test påverkades, så innebär det att testklassen beror på den ändrade klassen. Om beroendeanalysen inte hittar det beroendet så är testurvalstekniken inte säker.