A Safe Regression Test Selection Technique for Modelica

Niklas Fors[1] Jon Sten[2] Markus Olsson[2] Filip Stenström[2]

[1]Department of Computer Science, Lund University, Sweden, niklas.fors@cs.lth.se
[2]Modelon AB, Sweden, jon.sten@gmail.com, {filip.stenstrom|markus.olsson}@modelon.com

Abstract

Running regression tests for Modelica models usually takes a long time. This paper presents a safe regression test selection technique for Modelica based on static analysis. The technique tracks dependencies between classes to compute which tests that need to be run given a change. The dependency rules have been verified using mutation testing. The technique has been evaluated on the Modelica Standard Library and another library with promising results.

Keywords: regression test selection, mutation testing

1 Introduction

Regression testing is an important activity when developing software today, preventing software changes from introducing bugs that break previous functionality. Regression tests allow the developer to run tests while developing for spotting errors early on. However, running all tests may take a long time, which makes the developer less likely to run the tests very often. *Regression test selection* tries to solve this problem by running a subset of all tests. There are different test selection techniques (Rothermel and Harrold, 1996; Yoo and Harman, 2012), some of them are *safe*, meaning that all tests that may be affected by the change are selected. Safe techniques are usually approximate and include tests that are not affected by the change. These techniques may also have a higher runtime for computing the selection, which should be lower than the time saved using the selection compared to running all tests.

This paper presents a safe regression test selection technique for the modeling language Modelica (2018), which is based on *extraction-based test selection* introduced by Öqvist et al. (2016) for Java. Tests in Modelica usually require relatively long compilation and simulation time, which makes test selection especially suitable for Modelica. For example, we have evaluated the technique on the Modelica Standard Library (MSL), where running all tests takes about 2-3 hours. If a random class is changed in MSL, then on average 95.5% time can be saved by running the selected tests compared to running all tests. The runtime for the test selection algorithm is only 0.14% of running all tests.

Our technique analyzes the source code and finds dependencies between classes, which form a dependency graph. The test selection algorithm takes a set of changed classes and gives back the tests that depend on the changed classes, according to the dependency graph.

The contributions of this paper are the following:

- Dependency rules that describe when a class depends on another class (Section 3). These rules include name bindings, nested classes, `redeclare` clauses, and implicitly called functions such as `equalityConstraint`.
- Empirical verification of the dependency rules using mutation testing (Section 4). The mutation testing was performed on MSL, where each class was mutated to detect which tests that actually depend on the mutated class. The actual dependencies were then compared with the output of the test selection algorithm.
- Evaluation of how much time is saved using test selection on two libraries, the Modelica Standard Library and the Heat Exchanger Library (Section 5).
- An open source test suite for dependency analysis algorithms for Modelica[1].

The work in this paper has been carried out as a master's thesis project by Olsson and Stenström (2018) at Modelon, and is a continuation of the master's thesis project by Hedblom and Rundquist (2017)[2] with higher precision.

2 Safe Test Selection

The test selection algorithm takes a set of changed classes and returns a set of test cases that need to be run. Consider the set of all classes C in a system, and the test cases T that is a subset of C, and which are considered as entry points. For a change in a subset of C, the test selection algorithm gives back a subset $T_{ts} \subseteq T$ that need to be run due to the test results may have changed. The algorithm is *safe*, meaning that the test cases T_f that actually fail due to the changes is a subset of T_{ts} (hence, $T_f \subseteq T_{ts} \subseteq T \subseteq C$).

The test selection is implemented by tracking dependencies between classes using static analysis. A class `X` *depends* on another class `Y` if a change in `Y` may affect the meaning of `X`. The dependencies form a dependency graph, which is used by the test selection algorithm to

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

[2]Hedblom and Rundquist were also supervised by Niklas Fors.

```
model A
  ...
end A;

model B
  A a;
end B;

model T1
  A a;
end T1;

model T2
  B b;
end T2;
```

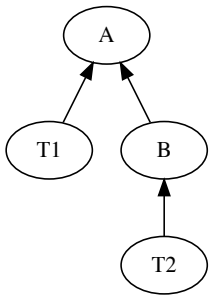

Figure 1. Two test cases `T1` and `T2` that use the models `A` and `B`, respectively. The right part of the figure shows the dependencies between the models.

find dependent test cases given a set of changes. We use the term *class* in accordance with the Modelica specification, which includes *specialized classes* such as `model`, `record`, `type`, `function`, etc.

For example, consider the Modelica source code in Figure 1. In this example, we have two test cases `T1` and `T2` that use the models `A` and `B`, respectively. The figure also shows the dependency graph between the classes. If a change is made in model `A`, then the dependency graph tells us that test case `T1` and `T2` need to be rerun. However, if a change is made in model `B` instead, then only `T2` needs to be rerun. As illustrated by the example, the dependencies are transitive, for instance, test case `T2` depends transitively on `A`.

2.1 Detecting Changes

Detecting changes can be done in different ways. One approach is to detect file changes, for instance, using version control systems, and then map files to classes. For example, if a file has changed, then all classes in that file can be considered as changed. This is the approach we currently use and which was implemented during the previous master's thesis. It was chosen because it was the easiest approach to implement. The effect of the approach depends on how the Modelica code is organized. For example, MSL usually has quite many classes per file (on average 30 per file) and there are other libraries that have fewer classes per file. A more fine-grained approach would analyze what part of the file that has changed and which classes that corresponds to.

2.2 External Code

Modelica supports interfacing with other languages, such as C and FORTRAN. This can be challenging as it is very hard to calculate the dependencies within the external code. It is also common that the external code, based on input from the Modelica code, will read and access other resources, e.g. files or network resources.

In the current implementation, changes to non-Modelica files will mark all Modelica classes with external dependencies as changed. This, in turn, will force a rerun of all test cases which directly or indirectly depend on external code. This is not ideal as, to the user, seemingly harmless changes may mark some test cases for rerun. Possible improvement is to allow the user to provide a list of files to monitor or not to monitor for changes. For example, by allowing the user to list a set of files which won't cause rerun of tests, or provide a list of files which will force a rerun of tests.

3 Dependency Rules

The following dependency rules are used for building the dependency graph between classes:

Rule 1. A class has a dependency on an accessed class, including all parts of the qualified name. This includes component declarations, extending clauses, function calls, import statements, functions in annotations (derivative, inverse) and overloaded operators.

Rule 2. A class has a dependency on its enclosing class.

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 4. A class has a dependency on implicitly called classes. This includes a record or type enclosing a function named `equalityConstraint`, and a class extending the class `ExternalObject` has dependency on enclosed function `destructor`.

3.1 Motivation

The rules will now be motivated with examples.

Rule 1 was illustrated in Section 2, namely that a class depends on classes it references. Additionally, the rule also handles qualified accesses. For example, for the access `A.B.C`, the rule creates dependencies to `A`, `A.B` and `A.B.C`. This is needed because changes in `A` or `A.B` may change what `C` refers to. For instance, class `C` may be declared in a supertype of `A.B`, and changing the supertype of `A.B` may then change what `C` refers to.

Rule 2 is illustrated in Figure 2. If model `B` is changed to extend `A2` instead of `A1`, then the type of `m` in `B.C` is changed from `A1.M` to `A2.M`. Thus, Rule 2 is needed to create a dependency from `B.C` to `B` to handle when `B` changes.

Rule 3 is illustrated in Figure 3. The dependency analysis needs to be careful with redeclare clauses. In the figure, the class `C` needs a dependency to all nested classes in `A2` due to the redeclare modifier in component declaration for `b`. This because the replaceable package `P` in package `B` is redeclared to `A2` in the context of component `b` in

DOI
10.3384/ECP18154131

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

package A2
  model M
  end M;
end A2;

package B
  extends A1;

  model C
    M m;
  end C;
end P;
```

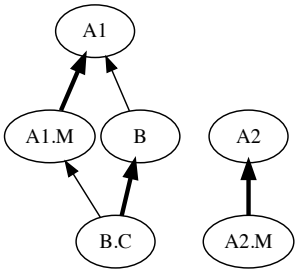

Figure 2. Example for Rule 2. Dependencies stipulated by Rule 2 are illustrated with thicker edges. The rule is needed to handle if `B` changes its `extends` clause to `A2`, which changes the reference in class `B.C`.

model `C`. This broad addition of dependencies is needed in order to capture the actual use of `A2.f` in package `B` when in context of model `C`. Alternatively, a complex and time consuming analysis of the usages of the package `P` in context of model `C` could be done. Note that the rule is recursive, meaning that if `A2` would have a nested class `NC1` which in turn would have another nested class `NC2`, then `C` would depend on both `NC1` and `NC2`.

Rule 3 also handles classes prefixed with `redeclare`, which is illustrated in Figure 4. In this example, we want a dependency from `B` to `B.f`, since `B` redeclares the function `f` that is declared in `A`.

Rule 4 is needed due to specific dependencies that the language specification dictates based on class context. See the test suite for examples of Rule 4.

3.2 Implementation

We implemented the dependency analysis based on the rules in this section in the OPTIMICA Compiler Toolkit[3], which is based on the JModelica.org compiler (Åkesson et al., 2010a).

4 Verification

We want the test selection algorithm to be safe, but it is hard to know if the dependency rules (Section 3) cover all cases since Modelica is a complicated language. We have verified the dependency rules empirically by using tests from the previous master's thesis project (Hedblom and Rundquist, 2017), coming up with new tests manually and performed verification based on mutation testing.

Mutation testing is a technique for evaluating how efficient a test suite is (DeMillo et al., 1978). This is done by automatically introducing small faults in the program

```
package A1
  function f
  end f;
end A1;

package A2
  function f
  end f;
end A2;

package B
  replaceable
    package P = A1;
  Real x = P.f();
end B;

model C
  B b(redeclare
      package P = A2);
end P;
```

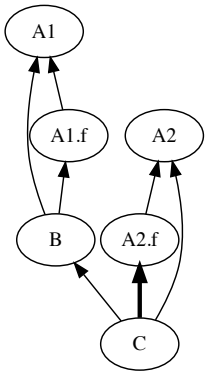

Figure 3. Example for Rule 3, which creates a dependency from `C` to `A2.f`. This dependency is needed to handle changes in `A2.f`.

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

model B
  extends A;

  redeclare
    function f
  end f;
end B;

model C
  B b;
end C;
```

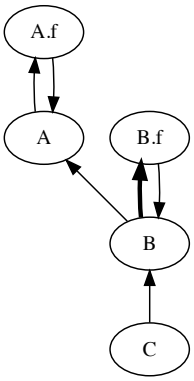

Figure 4. Another example for Rule 3 that illustrates classes with the redeclare prefix, which creates a dependency from `B` to `B.f`.

[3] http://www.modelon.com/products/modelon-creator-suite/optimica-compiler-toolkit/

```
model A;
  Real x;
equation
  x = 7;
end A;

model B
  A a;
  Real y;
equation
  y = a.x + 3;
end B;

model T
  B b;
end T;
```

(a) Source system

```
fclass T;
  Real b.a.x;
  Real b.y;
equation
  b.a.x = 7;
  b.y = b.a.x + 3;
end T;
```

(b) Flat class

Figure 5. Model `T` is selected and the compiler instantiates it to a flat class.

and then checking if these faults are detected by any test case. If a fault is not detected, then a new test case can be added that covers the fault. A change may be, for example, switching the branches in an if statement or replacing arithmetic operators, for instance, replacing a minus (−) with a plus (+). However, we use this technique in a little different way since we are interested in verifying the test selection algorithm, and thus are interested in finding out the actual dependencies between classes. We do this by introducing changes in a class and then computing which tests that are actually affected by the changed class. For the affected tests, there is an actual dependency from the test to the changed class, and we want this dependency to be in the dependency graph. If this is not the case, then the algorithm contains a bug or a dependency rule is missing.

We only change one class at a time and then check which tests are actually affected by the change. We detect that a test is affected by the change by computing and comparing the string representation of the flat (unoptimized) class for the test. This flat class contains all the variables, equations, function and other parts of the model which is needed in order to solve the equation system. If the flat class has changed in any way, we consider that the test is affected by the change. Note that running the test may actually give the same results as before since a test involves comparing simulation results. However, we are only interested in the dependencies to the changed class from the test class, and if the flat test class changes in any way, then there is a dependency (direct or indirect).

When a test is selected to run, the compiler creates a flat class representing the equation system for the test, which is then optimized and later solved using a numerical solver. The flat class is illustrated in Figure 5, where the test `T` has been selected and instantiated by the compiler. As can be seen, the object-oriented and hierarchical structure are removed in the flat class (but not optimized in the figure). For this class, we could change the value of `x` in model `A` to 5, and the flat class would change, and we can then infer that there is an indirect dependency from `T` to `A`. To verify that the test selection algorithm works, we check that this dependency is in the dependency graph. If we instead would change the value of `x` to $4+3$, then the flat class would still change, even if the meaning would remain (since $4+3=7$). This is as earlier described not an issue, since we are interested in computing actual dependencies between classes.

Table 1. Mutation testing on MSL. The unique column is how many classes the mutation was unique for. Mutations is the total number of mutations carried out. The total number of classes in MSL is 5946.

Mutation	Mutated classes	Unique	Mutations
LocalVar	1772 (29.5%)	485	1772
Arit	2026 (33.8%)	16	9647
Lit	3306 (52.3%)	805	25563
Bool	913 (15.2%)	17	3028
Redecl	68 (1.1%)	17	68
Comment	1650 (27.3%)	205	10142

4.1 Mutation Testing on MSL

We performed mutation testing on MSL with the following types of mutations.

LocalVar. Add local variable to function.

Arit. Switching arithmetic operands, e.g., `1-2` ⇒ `2-1`.

Lit. Changing literals, e.g., `2` ⇒ `3`.

Bool. Changing boolean operators, e.g., `a<b` ⇒ `a>b`.

Redecl. Redeclaring a replaceable function.

Comment. Changing string comments.

Changing string comments will not change the meaning of the program, but they are carried over to the flat class, which is what we are interested in.

The mutation testing was performed for one class in MSL at the time and one mutation type at the time. However, several mutations of the same type could be applied for one class, for example, by changing several literals. The results are shown in Table 1. As can be seen in the table, changing literals was the most applicable mutation type and could be applied for more than half of the classes in MSL. Performing one mutation test took around 18 minutes, since it requires all tests to be individually instantiated to a flat class. We performed the mutation tests on a cluster of Jenkins machines, and it would take 280 days if they were carried out in a sequence (non-parallel).

From the result of mutation testing, we needed to generalize Rule 3, discovered Rule 4, found derivative/inverse in annotations, and found six bugs in the implementation. A test case was added to the test suite for dependency analysis algorithms for each fault found.

DOI
10.3384/ECP18154131

4.2 Threat to Validity

Note that we have only verified the test selection algorithm empirically and not formally. Thus, we cannot know with certainty that the rules and implementation are safe. A complete formal verification would require a formal model of Modelica, a language that is very complicated. It would also be possible to use a simplified formal model that does not cover the complete language, but some aspects of it. However, using a simplified formal model would probably miss, for example, equality constraints (Rule 4) because they are not a central part of the language, which the mutation testing did find. Also, the mutation testing was performed on MSL, which might not use all language features. Thus, it would be desirable with more mutation testing on other libraries to make the verification more complete.

4.3 Partial Dependency Graph

We also used mutation testing to compute a partial dependency graph between classes in MSL. For each mutated class, we get actual dependencies to the mutated class from all test classes that changed because of the mutation. Combining all actual dependencies from all mutations yield a partial dependency graph from test classes to classes. This partial dependency graph can be used to partly verify test selection algorithms. Thus, in addition to the manually created test cases for dependency analysis algorithms, the open test suite also contains the partial dependency graph from the mutation testing as an XML file.

4.4 Instrumenting the Compiler

We have used mutation testing to compute actual dependencies between test classes and classes. Another way would be to instrument the compiler to compute all actual dependencies when instantiating a test class, which would require less resources. The reason why we chose mutation testing is because we believe that instrumenting the compiler might contain the same bugs as the test selection implementation. Mutation testing is more like black-box testing in this context. However, it would be useful to complement the mutation testing with compiler instrumentation to improve the verification, which is something we would like to do.

4.5 Previous Technique Unsafe

During the mutation testing we found that the implementation by Hedblom and Rundquist (H&R) was not safe since it ignored classes with the `redeclare` prefixes. We also found another bug in their implementation that included too many dependencies. We fixed these two issues in the evaluation (Section 5) for H&R's technique when comparing it to our technique.

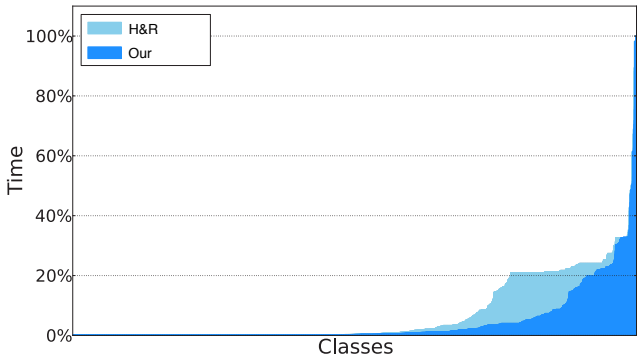

Figure 6. Tests runtime for each class changed in MSL. The number of classes is 5946, of which 366 are tests.

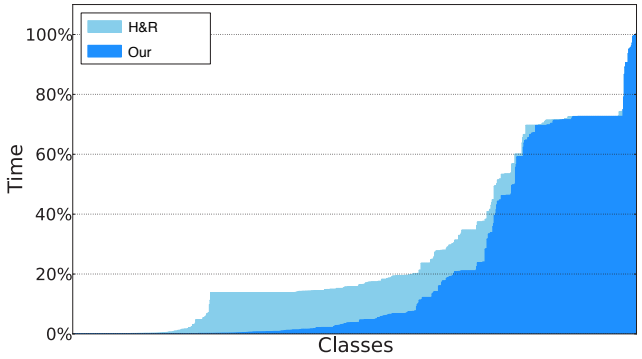

Figure 7. Tests runtime for each class changed in HXL. The number of classes is 871, of which 227 are tests.

5 Evaluation

We have evaluated the time saved using test selection compared to running all tests on two libraries, the Modelica Standard Library (MSL) and the Heat Exchanger Library (HXL) by Modelon[4]. The time saved is compared to the test selection defined by Hedblom and Rundquist (2017), which will be denoted H&R. Their technique is more coarse-grained than ours and operates in an earlier step in the compilation process, leading to more tests selected, but with a lower analysis running time (see Section 6 for comparison).

We computed the test selection for each class in the library and then the runtime for the selected tests, which includes compilation and simulation time. The runtime for the selected tests was computed as a percentage of the runtime for all tests. The results are shown in Figures 6-7, where classes are sorted by tests runtime in ascending order. For both techniques, the average savings is above 90% for MSL and above 70% for HXL. The average savings, the average test runtime, the median test runtime and the analysis time are shown in Tables 2-3 under *Class*. The complement of the average savings in the tables correspond to the blue areas in the graphs.

We also computed the test selection for each file in MSL and HXL, where all classes in a file were consid-

[4]https://www.modelon.com/library/heat-exchanger-library/

Table 2. Performance results for MSL. All values are in percentage of the time it takes to run all tests.

	Class		File	
	Our	H&R	Our	H&R
Average savings	95.5%	93.1%	88.9%	87.9%
Average test time	4.3%	6.9%	11.0%	12.1%
Median test time	0.22%	0.33%	1.19%	1.7%
Analysis time	0.14%	0.04%	0.14%	0.04%

Table 3. Performance results for HXL. All values are in percentage of the time it takes to run all tests.

	Class		File	
	Our	H&R	Our	H&R
Average savings	78.9%	72.0%	80.5%	74.8%
Average test time	20.9%	27.9%	19.4%	25.1%
Median test time	3.61%	15.7%	3.77%	15.7%
Analysis time	0.12%	0.10%	0.12%	0.10%

ered changed. The results can be seen in the tables under *File*. As expected, the time saved for files is less than for classes.

As we can see, the new test selection technique has higher precision than H&R's technique leading to improved savings. However, our dependency analysis is a bit more advanced and the implementation is 201 source lines of code[5] (SLOC) specified in JastAdd (Hedin and Magnusson, 2003), whereas H&R's implementation is 134 SLOC.

5.1 MSL Commit History

We have also used the MSL commit history to get more realistic sets of changed files. Thus, we use each commit as a set of changed files and compute the time saved for that set. The results are shown in Figure 8, where commits are sorted by tests runtime in ascending order. The average saving is 68.9% for our technique and 57.0% for the old technique.

6 Related Work

As described earlier, the dependency rules presented in this paper is a continuation of the master's thesis by Hedblom and Rundquist (2017), but with higher precision. H&R implemented their test selection algorithm in an earlier step in the compilation process[6] with less static information available about name bindings etc., leading to more coarse-grained rules. One major difference is that the previous implementation could not resolve all parts of a qualified access like `a.b.c`. This lead to a dependency rule for class accesses where an access to a class `A` created dependencies to all nested classes enclosed by `A`

[5]Measured with *cloc*, see https://github.com/AlDanial/cloc

[6]H&R implemented their algorithm in the *source tree*, whereas our algorithm is implemented in *instance tree*, according to the different compilation steps defined by Åkesson et al. (2010b).

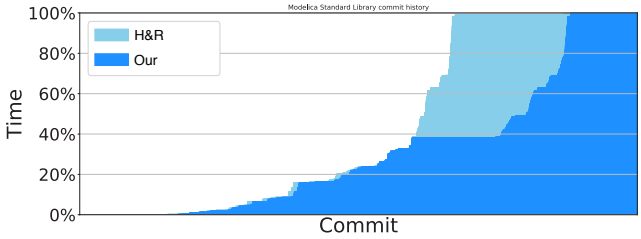

Figure 8. Test runtime for MSL commit history.

(recursively). Another difference is that H&R's rules are implementation-specific, where the rules use the term *resolvable* to mean names that are resolved in that compilation step in the OPTIMICA compiler. In contrast, our rules are defined in terms of the language and are not dependent on the implementation. However, it would be possible to generalize the H&R's rules to be implementation-neutral.

There is previous work on safe test selection for other languages (Chen et al., 1994; Rothermel and Harrold, 1997), such as Java (Öqvist et al., 2016; Gligoric et al., 2015), where both dynamic- and static analyses have been investigated. Our technique is based on *extraction-based test selection* by Öqvist et al. (2016), who applied it for Java. This kind of technique uses only static analysis and is more coarse-grained (for instance, dependencies between files or classes) than other techniques. Modelica is quite different from Java, leading to other dependency rules, for example, covering the `redeclare` mechanism. One interesting property from a safe test selection perspective is that compilation and simulation usually takes rather long time for Modelica models, especially when comparing to compilation and test time for Java. This makes test selection especially useful for Modelica. Also, since Modelica compilers generate flat equation systems for test classes, you cannot use dynamic analysis, but only static analysis.

7 Conclusions

We have in this paper presented a regression test selection technique using static analysis for Modelica with very promising results. In the evaluation, we found that changing a class in MSL and only running the tests selected by the algorithm saved on average 95.5% tests runtime compared to running all tests. Using MSL commit history as basis for changed files, then the average saving is 68.9%. Since Modelica is a complicated language, we performed mutation testing on MSL to verify that our dependency rules are correct and safe. However, with mutation testing, we cannot prove that the rules or the implementation of the rules are complete.

In the future, we would like to do more mutation testing on other libraries than MSL, and also add more mutation types. It would also be interesting to update the dependency graph incrementally. The current implementation computes the dependency from scratch for each change that the test selection is run on. However, computing the

DOI
10.3384/ECP18154131

dependency graph is relatively fast. For example, computing it for MSL only takes 0.14% of the time it takes to run all tests. We would also like to detect changes on a more fine-grained level and identify which classes in a file that are actually changed.

Acknowledgements

We thank Jesper Öqvist for comments on an earlier draft of this paper. This research was partly supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA), within the strategic innovation program Process Industrial IT and Automation, under contract number (2017-02371).

References

Johan Åkesson, Karl-Erik Årzén, Magnus Gäfvert, Tove Bergdahl, and Hubertus Tummescheit. Modeling and optimization with Optimica and JModelica.org—languages and tools for solving large-scale dynamic optimization problem. *Computers and Chemical Engineering*, 34(11):1737–1749, November 2010a.

Johan Åkesson, Torbjörn Ekman, and Görel Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1-2):21–38, January 2010b.

Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, pages 211–220, 1994.

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.

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.

Erik Hedblom and Kasper Rundquist. Safe test selection for modelica using static analysis. Master's thesis, Lund University, 2017. LU-CS-EX 2017-26.

Görel Hedin and Eva Magnusson. JastAdd: an aspect-oriented compiler construction system. *Science of Computer Programming*, 47(1):37–58, 2003. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/S0167-6423(02)00109-0.

Modelica. *The Modelica Association*, 2018. `http://www.modelica.org`.

Markus Olsson and Filip Stenström. Improved precision and verification for test selection in Modelica. Master's thesis, Lund University, 2018. LU-CS-EX 2018-08.

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.

Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Trans. Software Eng.*, 22(8): 529–551, 1996.

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

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test., Verif. Reliab.*, 22(2):67–120, 2012.