VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kansliets noteringar Kod	Dnr
2013-24716-106139-24	

2013
Project Research Grant

Area of science
Natural and Engineering Sciences
Announced grants
Research grants NT April 11, 2013
Total amount for which applied (kSEK)

2014	2015	2016	2017	2018
922	870	920	931	1004

APPLICANT

Name(Last name, First name)
Lindström Claessen, Koen

Date of birth
750724-8115

Gender
Male

Email address
koen@chalmers.se

Academic title
Professor

Position
Bitr. Professor

Phone
+46-31-7725424

Doctoral degree awarded (yyyy-mm-dd)
2001-05-04

WORKING ADDRESS

University/corresponding, Department, Section/Unit, Address, etc.
Chalmers tekniska högskola
Institutionen för data-och informationsteknik
Programvaruteknik

41296 Göteborg, Sweden

ADMINISTRATING ORGANISATION

Administrating Organisation
Chalmers tekniska högskola

DESCRIPTIVE DATA

Project title, Swedish (max 200 char)
Bevisbaserad Testning -- Bevistekniker som gör det möjligt att testa komplicerade algoritmer automatiskt

Project title, English (max 200 char)
Proof-based Testing -- Using Proof Techniques to Enable Automated Testing of Complex Algorithms

Abstract (max 1500 char)
Automatic testing requires executable specifications. However, these executable specifications very often end up being as complicated as the programs they are meant to test. We propose a new method called ?proof-based testing? that uses mathematical proof techniques to break up a complex specification which is hard to test directly, into smaller properties which together logically imply the original specification, but are easier to test. The proof techniques we have already found useful in this context are contrapositives, induction, and co-induction.

The kind of programs that benefit most from this new approach are those with a high algorithmic content. Examples are optimization software, automated theorem provers, model checkers, and compilers.

The project?s aim is to (1) investigate and classify the kind of programs that benefit from this technique, (2) develop a supporting theoretical framework for reasoning about the choices that must be made when applying this technique, and (3) evaluate experimentally the effectiveness of proof-based testing against more traditional methods.

Abstract language

Kod
2013-24716-106139-24

Name of Applicant
Lindström Claessen, Koen

Date of birth
750724-8115

English

Keywords
testing, property-based testing, induction, software development, verification

Research areas
*Nat-Tek generellt

Review panel
NT-2

Classification codes (SCB) in order of priority
10201, 10205,

Aspects

Continuation grant
Application concerns: New grant
Registration Number:

Application is also submitted to

similar to:

identical to:

ANIMAL STUDIES

Animal studies
No animal experiments

OTHER CO-WORKER

Name(Last name, First name)
,

University/corresponding, Department, Section/Unit, Addressetc.

Date of birth

Gender

Academic title

Doctoral degree awarded (yyyy-mm-dd)

Name(Last name, First name)
,

University/corresponding, Department, Section/Unit, Addressetc.

Date of birth

Gender

Academic title

Doctoral degree awarded (yyyy-mm-dd)

Name(Last name, First name)
,

University/corresponding, Department, Section/Unit, Addressetc.

Date of birth

Gender

Academic title

Doctoral degree awarded (yyyy-mm-dd)

Name(Last name, First name)
,

University/corresponding, Department, Section/Unit, Addressetc.

Date of birth

Gender

Academic title

Doctoral degree awarded (yyyy-mm-dd)

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of Applicant
Lindström Claessen, Koen

Date of birth
750724-8115

ENCLOSED APPENDICES

A, A, B, C, N, S

APPLIED FUNDING: THIS APPLICATION

Funding period (planned start and end date)
2014-01-01 -- 2018-12-31

Staff/ salaries (kSEK)

Main applicant	**% of full time in the project**	**2014**	**2015**	**2016**	**2017**	**2018**
Koen Lindström Claessen	10	142	147	152	158	164
Other staff						
doktorand	80	604	625	648	671	695
Total, salaries (kSEK):		746	772	800	829	859

Other projectrelated costs (kSek)	**2014**	**2015**	**2016**	**2017**	**2018**
Utrustning	80				
Resor	40	40	40	40	40
Lic./Disp.			20		40
Lokaler	48	49	51	53	55
IT	8	9	9	9	10
Total, other costs (kSEK):	176	98	120	102	145

Total amount for which applied (kSEK)

2014	2015	2016	2017	2018
922	870	920	931	1004

ALL FUNDING

Other VR-projects (granted and applied) by the applicant and co-workers, if applic. (kSEK)

Proj.no.(M) or reg.nr.	**Funded 2013**	**Funded 2014**	**Applied 2014**
2012-5746	4441	4441	

Project title
Reliable Multilingual Digital Communication: Methods and Applications

Applicant
Aarne Ranta

Funds received by the applicant from other funding sources, incl ALF-grant (kSEK)

Funding source	**Total**	**Proj.period**	**Applied 2014**
SSF	25000	2011-2016	

Project title
Resource-Aware Functional Programming

Applicant
John Hughes

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of Applicant
Lindström Claessen, Koen

Date of birth
750724-8115

POPULAR SCIENCE DESCRIPTION

Popularscience heading and description (max 4500 char)

Det är ofta problematiskt att testa de allra mest komplexa komponenter i ett datasystem helt automatiskt, eftersom beskrivningen av vad dessa komponenter ska göra är minst lika komplex som komponenterna själva. Testningen blir då inte tillräckligt pålitlig.

I detta projekt undersöker vi möjligheten att använda en teknik från matematisk logik, s.k. "induktion", för att möjliggöra pålitlig testning av dessa komplexa komponenter. I ett matematiskt bevis kan man med hjälp av induktion bryta ner ett komplext påstående till ett flertal enklare påståenden: s.k. "basfall", som täcker de allra enklaste fallen, och s.k. "stegfall", som beskriver hur man kan öka komplexiteten i ett påstående med ett litet snäpp.

Vi har upptäckt att samma teknik kan användas för att bryta ner ett komplext påstående om beteendet hos ett dataprogram till ett flertal enklare påståenden, som i sin tur kan testas separat. Detta möjliggör pålitlig testning av dessa dataprogram, som tidigare inte var möjligt.

Exempel på dataprogram som kan testas med denna metod, och som tidigare inte kunde testas pålitligt, är automatiska teorembevisare och modellcheckare (som används för att hitta fel i mjukvara och hårdvara), kompilatorer (som används dagligen av mjukvarukonstruktörer), och kortaste-vägsökningsprogram (som används i t.ex. GPS).

Projektet kommer att (1) undersöka och kategorisera den typ av dataprogram och datasystem där denna metod kan tillämpas, (2) ta fram datoriserat stöd till mjukvaruutvecklare som vill tillämpa metoden som garanterar att alla fall är täckta, och (3) kvantifiera med hjälp av experiment hur effektiv testning blir med hjälp av den nya metoden, jämfört med befintliga metoder.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of applicant
Lindström Claessen, Koen

Date of birth
750724-8115

Title of research programme
Proof-based Testing -- Using Proof Techniques to Enable Automated Testing of Complex Algorithms

Appendix A

Research programme

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

Research Programme

Purpose and aims

This project is about software verification; the activity of finding out whether or not a piece of software does what it is supposed to do. *Testing* is by far the most widely used software verification method today. There exist other methods too, most notably formal methods, which involve some form of computer-supported mathematical proof. Formal methods have the possibility of being much more effective at finding bugs in programs, but they are also much more expensive to deploy.

The particular way of testing we look at in this project is *property-based testing*. Here, a *test data generator* generates input to the program under test, and a *property* (also called "*oracle*") checks if the behavior of the program is OK or not. Both generators and properties are programs which can be run automatically, giving us the possibility of running very many automated tests once everything is set up. This in contrast to for example traditional unit testing, where test input is constructed by hand on a case by case basis, and where test output is checked in an ad-hoc way.

There has been much -- and indeed much needed -- focus within property-based testing on the generators. In contrast, this project will mainly focus on the *properties*.

Motivational example

Imagine we are testing the software running in a modern GPS-based car navigation system. In particular, let us focus on the component that calculates the shortest way between two points A and B on a given map M, if there is such a path. It is important that this component does the Right Thing, otherwise the navigation system may lead the driver astray.

We are going to use *property-based testing*, which uses a *generator* that creates input data to the component at hand, and a *property* to decide if the produced output is OK or not.
The generator (not the focus of this project) provides maps M and points A and B on that map. The maps could be randomly generated, or taken from a database of existing maps, or both. The points A and B could in principle be randomly chosen on M.

But how about the property? How should it decide if the answer that is given by the component is the correct one?

The typical way this situation is dealt with today is to compare the component under test to a so-called *golden implementation,* which is an alternative implementation of the component's behavior that we trust. This leads up to a big practical problem. The golden implementation needs to

be simple enough to be trustworthy. But it also needs to be efficient enough so that it can be practically used to run many tests. These two goals are very often contradictory; the more algorithmic content a piece of software contains, the harder it is to fulfill this goal.

The path-finding component of the car navigation system contains some clever algorithmic elements that are there to avoid falling into the many loops that a given map M may contain. Any golden implementation needs to do the same thing, otherwise it may not terminate. So any golden implementation needs to use a shortest-path algorithm, which probably makes it as complex as the car navigation component itself. And this makes it likely that the golden implementation contains some of the same mistakes that are in the car navigation component, which means that the testing framework will not discover the bug.

One alternative is to use an existing shortest-path algorithm to test against, perhaps an open-source version found on the web somewhere. Apart from the trust issues, there are also problems here. There might not be any shortest-path algorithm that deals with all the particular features that your maps contain. And, it is much more likely than people think that two completely different people presented with the same programming task will make the same mistakes in their solutions, thus again increasing the risk of mistakes that may be missed by the testing framework.

To sum up, what we need in an automated testing framework are properties that (1) are efficient enough so that they can be run many times on different test data, (2) are significantly less complex than the program under test, and (3) still cover the full functionality that we want to test.

Point (3) alludes to the *completeness* of a test framework; it is understood that we desire a test framework that, for any fault contained in the program under test, there is at least one property and one input that can be generated by the corresponding test data generator, such that the property fails. (This does not mean that the test framework will definitely find every bug; it means that we have not a priori precluded the finding of any bug, and that for any bug there is a possibility that it will be found by the test framework.)

Property-based testing

We can take another approach, and rather than look for a complete monolithic golden implementation, look at different properties we would like the program under test to satisfy.

One interesting property we should definitely check is the following: Property “Safe”: Given a map M, and two points A and B, if the car navigation component produces a path P, it should actually be a valid path from A to B. It is a bit tedious, but doable to write a program that given a map M, two points A and B, and a path P, checks that P actually is a path from A to B on M. The program that checks this is definitely a lot simpler than the component we are testing.

Is the above property “Safe” enough?

No. There are many possible flaws in the car navigation component that cannot be detected by this property alone. For example, the property only says what should happen when the component actually finds a path. It does not say whether it is OK or not if the component does not find a path. Also, the component is supposed to find the *shortest* path, something which the property does not say anything about either.

So, we would also like to formulate and test a property we can call “Complete”: If the component does not find a path, there actually was no path from A to B. And a property “Optimal”: If the component finds a path P between A and B, there exists no shorter path than P. But how? Doesn’t it seem daunting to formulate test oracles that check things like “there is indeed no path from A to B” or “there are no shorter paths between A and B” without making use of complex shortest path algorithms?

This project proposes a new technique to solve the above problem. The problem pattern appears in many other contexts as well, all involving programs with a high algorithmic content. Some such hard-to-test programs solve decidable problems and are inherently finite, such as SAT-solvers, constraint solvers, and model checkers. Others involve non-decidable issues such as theorem provers and compilers. What they have in common is that their original specifications (which were not practically testable) can be broken down into several smaller properties, which (a) together logically imply the original specification, and (b) are much easier to test. **The way we break down the original specification is by following at the structure of a possible mathematical correctness proof of the program**; the essence of our proposed **proof-based testing**. Note that we do not actually perform this correctness proof; this would move us over to the field of formal methods, which involves a mathematical model of the program under test, plus a logical system in which we can reason about that model; a much more expensive operation. Instead, the test framework designer merely gets inspired by a possible mathematical proof and structures the test framework accordingly.

Contrapositive testing

So how then can we test the car navigation component? One way is to slightly shift viewpoint on the property “Complete” we want to test. Instead of checking that “If the program cannot find a path, there exists no path between A and B”, we use contrapositive reasoning (which considers “(not Q) implies (not P)” when proving “P implies Q”, which are logically equivelent statements). The contrapositive of “Complete” is: If there exist a path between A and B, the component will find a path. This is easy to test; just create a generator that generates maps M with points A and B *and a guaranteed path between A and B*, and have the oracle check that the produced answer always is

a path. (It seems so obvious once you see it!) So, using contrapositives we can transform a hard-to-check post-condition after the fact into an easy-to-generate pre-condition!

Testing “Optimal” is now also easy, using contrapositive: Use the generator that generates maps M with points A and B and a guaranteed path between A and B, and have the oracle check that the produced answer always is a path of length not greater than the guaranteed path.

Inductive testing

Contrapositive is a simple yet powerful method when it works. But it is not always the case that it easy to generate inputs for which we have guaranteed outputs, and still have the generator be surjective (meaning it is still possible to generate *any* input with a guaranteed output). In this case, inductive testing may help.

In inductive testing, we look at which prerequisites a proof by induction would need to arrive at the conclusion that the program is correct. In our case, we want to test that the program finds all paths that are there to find. Let us reason to this by induction. First, we test that the program finds all paths of length 0. So, we test: For any map, finding the shortest path from A to A should lead to a path from A to A of length 0. This is the *base case*.

Then we test: For any map M and points A and B, the answer by the car navigation component should be consistent with the answer given when we ask it to find paths from N to B, for all neighbours N of A. So, in this test, we run the component several times, and check that the answers are consistent with each other. Consistent means that: (a) if no path was found from A to B, no path should exist from any N to B either, and (b) if a path was found from A to B, it should not be longer than the shortest path found through any N to B. This is the *step case*.

If both of these tests pass, we can (by testing) assume that the base case and step case of the induction hold, and then by induction conclude that the original component satisfies its specification! The soundness of mathematical induction guarantees that we shave not lost completeness of the test framework.

Survey of the field

Property-based testing, the main testing technique used in this project, lies in between two extremes of software verification: traditional unit testing (in which particular test cases are chosen by a human) and formal verification (in which properties about programs can be proved using a theorem prover). Property-based testing allows for general properties of programs, albeit in a very restrictive logic. Examples of property-based testing systems are Microsoft’s *generalized unit testing* and our own

very successful system QuickCheck.

There have been other combinations of testing and theorem proving before. The most obvious one is *compositional testing*, where properties of a larger system are decomposed into properties for parts of a system. Those properties can be tested for those parts of the system. The final conclusion that the original system satisfies its properties is made by logically combining the tested properties of the parts. The main reason for doing this is to increase test efficacy; it is easier to find bugs in a component of a system by testing that component than by testing the whole system at once.

In some sense, inductive testing can be seen as a generalization of this idea. However, the decomposition we propose does not follow the structure of the program, it follows the structure of an imaginary, possible correctness proof of the program. The reason we do this is also not to increase test efficacy, it is to be able to test the desired properties *at all*.

We have observed a number of specific instances of some of the ideas behind inductive testing in the literature. One notable example is the verification of pipelined microprocessors. The key property one wants to verify is that a pipelined microprocessor behaves “the same” as its non-pipelined counterpart. However, specifying exactly what a modern microprocessor should do, even ignoring pipelining, is a huge task. Instead, the community developed a method where the expected behavior of a pipelined processor is expressed in terms of the behavior of that same processor, where pipelining is somehow disabled (e.g. by flushing the pipeline after each instruction). In some sense, this can be seen as inductive testing but only using the step case of the induction. The base case (actually comparing with a non-pipelined processor) is left out for practical reasons.

Project description

The project will investigate proof-based testing as a technique, and also develop a theoretical framework which can be used to reason about logically equivalent, but testing-wise different properties. Finally, we want to work on implementing these ideas in a tool usable by real software developers.

As for finding complex algorithmic software, we plan to concentrate on two kinds of software:

A. Automated reasoning tools used in formal methods. We choose this domain because (1) we have a lot of experience developing this kind of software, and (2) it is important that programs used to prove other software correct is bug-free! A recent study by Biere et al. has shown, rather embarrassingly, that virtually all solvers of a particular category (SMT solvers) competing in a competition had a number of bugs in them. So, it is important that we develop new ways of testing these pieces of software. Our preliminary results have shown that this seems to be a viable way.

Examples of software we want to deal with here are automated theorem provers, and model checkers.

B. Non-deterministic concurrent software. This is an increasingly important class of software programs, because more and more software runs on servers talking to clients. It is notoriously difficult to specify the expected behavior of concurrent software, especially when non-determinism is involved, which is almost always the case. Model-based testing methods often decide to only concentrate on certain parts of the system that are predictable enough to be modelled correctly. Our hope is that inductive testing can be used here successfully. This software domain is more speculative, but our earlier work on testing concurrent software (ICFP 2010) uses a method rather like inductive testing, so we believe this is possible.

For proof-based testing to be really usable we need to have a way to reason about which way of testing a property is better. Often, there are many possible choices of what element of the input to do induction on. Which is best? A simple example to illustrate is testing the property "Forall x . A(x) => B(x)". The most direct way to test this is to generate some x that satisfies A(x), and then check if B(x) holds. However, the property is logically equivalent to "Forall x . ~B(x) => ~A(x)", which can be tested by generating some x that satify ~B(x) and check that ~A(x) holds. Which is better? It turns out that it depends on the frequency with which A(x) and B(x) are true on the whole domain.

A framework for reasoning about such choices could include an improvement operator >>, which we can use to say things like P >> Q, which means that the formulas P and Q are logically equivalent, but testing P gives more information than testing Q. It is vital that P and Q express the same thing logically! This ensures that the induction technique we applied when testing was sound, which is important to check. We envision a calculus involving >> and logical formulas, relating a logical semantics of formulas to a *testing semantics* of formulas.

Project plan

Year 1: We will investigate in detail how to use proof-based testing effectively for a number of different examples, mostly taken from the two categories above. This will hopefully lead to more general insights in the do's and don'ts behind the method. The expected outcome here is a number of well-documented examples of testing complicated software, some experimental results, together with some general conclusions. We also expect to run into limitations of the applicability of the method, which we hopefully can document as well. One such limitation is having to deal with *semi-decision procedures,* which either terminate with a result or do not terminate at all.

Year 2: We will develop a theoretical framework that can be used to reason about how to choose in which way we should test a given property. The hope is that we can "predict" the outcome of the experimental results gathered in Year 1. We also expect to come up with other testing techniques,

similarly inspired by mathematical logic proof techniques.

Year 3: We will design and implement a prototype tool that programmers can use to apply proof-based testing to their own programs, and reason about which approach is best. For example, it should be possible to express a desired property in a high-level way, and with the help of the tool "refine" the high-level property into a number of testable properties. We probably need to draw on our own expertise on automated theorem proving here.

Year 4, 5: We will push the limits on the limitations we have encountered earlier. For example, what to do for semi-decision procedures? Can time-outs be used as sound approximations? Also, we expect we may run into scalability issues when starting to deal with larger examples that will have to be addressed.

Significance

Proof-based testing has the potential to bring complete testing to a class of software components which were previously thought impossible to specify and test in this way. Opening up the possibility of testing complex algorithmic software without having to compare against a golden implementation is one contribution of the project.

Another contribution of the project will be the theoretical framework used for reasoning about what is the most effective way of testing a certain high-level property, in terms of a number of (more low-level) testable properties. This will have impact beyond the cases where proof-based testing makes sense.

Finally, we really strive to make our findings available to real software developers, and plan to implement a tool that can be used to reason about proof-based testing properties. This will greatly increase the applicability of our methods.

Preliminary results

We have already applied proof-based testing to several different kinds of software components.

We successfully managed to specify and test a number of academically developed software components, such as search in graphs (e.g. shortest path algorithms), optimization problems (e.g. finding the shortest edit distance between two text files), and translators (e.g. some simple compilers).

We have also applied inductive testing to the development of our own automated reasoning tools, notably our SAT-solver MiniSat and our hardware model checker Tip. Both these programs contain many sophisticated algorithms and their complexity is comparable to industrial versions of these programs.

For SAT-solvers, inductive testing works in straightforward by doing induction on the number of variables in a given problem P. For the step case, the SAT-solver under test is run on the problem P as well on the problems P[x/0] and P[x/1]. The three results should satisfy a simple boolean relationship. The base cases check all problems with no variables.

For model checkers, there was no obvious way to express their correctness in terms of testable properties using inductive testing, because there was no obvious element in the input to perform induction over! In the end, we found a couple of ways of expressing the correctness using *co-inductive testing,* which uses co-induction over the infinite streams used in the semantics of hardware circuits. We were amazed that something so theoretical as co-induction could be used to make testing practical!

Because these programs were developed by ourselves, we had a database of about a dozen non-trivial bugs we had made during their development. We evaluated the (co-)inductive testing techniques against traditional testing (which compares the output of the program against a golden implementation).

Our findings in these (rather non-trivial) cases was that we need about 2-5 times more random tests to find these bugs compared to traditional testing. This means that, for this application, if the golden implementation is more than 2-5 times slower than the implementation under test, inductive testing seems to be a good idea! It is rather likely that a golden implementation is much slower than a real implementation for complex algorithms, so this is a very promising result.

One final note. During the development of our model checker Tip, we *only* used traditional testing against a golden implementation. However, it turned out that, because of us being not careful enough, a very subtle bug had creeped into both our release implementation and the golden implementation, making it impossible for the testing we did to discover the bug! This shows the dangers with testing against a golden model.

Research Programme

Purpose and aims

This project is about software verification; the activity of finding out whether or not a piece of software does what it is supposed to do. *Testing* is by far the most widely used software verification method today. There exist other methods too, most notably formal methods, which involve some form of computer-supported mathematical proof. Formal methods have the possibility of being much more effective at finding bugs in programs, but they are also much more expensive to deploy.

The particular way of testing we look at in this project is *property-based testing*. Here, a *test data generator* generates input to the program under test, and a *property* (also called "*oracle*") checks if the behavior of the program is OK or not. Both generators and properties are programs which can be run automatically, giving us the possibility of running very many automated tests once everything is set up. This in contrast to for example traditional unit testing, where test input is constructed by hand on a case by case basis, and where test output is checked in an ad-hoc way.

There has been much -- and indeed much needed -- focus within property-based testing on the generators. In contrast, this project will mainly focus on the *properties*.

Motivational example

Imagine we are testing the software running in a modern GPS-based car navigation system. In particular, let us focus on the component that calculates the shortest way between two points A and B on a given map M, if there is such a path. It is important that this component does the Right Thing, otherwise the navigation system may lead the driver astray.

We are going to use *property-based testing*, which uses a *generator* that creates input data to the component at hand, and a *property* to decide if the produced output is OK or not.
The generator (not the focus of this project) provides maps M and points A and B on that map. The maps could be randomly generated, or taken from a database of existing maps, or both. The points A and B could in principle be randomly chosen on M.

But how about the property? How should it decide if the answer that is given by the component is the correct one?

The typical way this situation is dealt with today is to compare the component under test to a so-called *golden implementation,* which is an alternative implementation of the component's behavior that we trust. This leads up to a big practical problem. The golden implementation needs to

be simple enough to be trustworthy. But it also needs to be efficient enough so that it can be practically used to run many tests. These two goals are very often contradictory; the more algorithmic content a piece of software contains, the harder it is to fulfill this goal.

The path-finding component of the car navigation system contains some clever algorithmic elements that are there to avoid falling into the many loops that a given map M may contain. Any golden implementation needs to do the same thing, otherwise it may not terminate. So any golden implementation needs to use a shortest-path algorithm, which probably makes it as complex as the car navigation component itself. And this makes it likely that the golden implementation contains some of the same mistakes that are in the car navigation component, which means that the testing framework will not discover the bug.

One alternative is to use an existing shortest-path algorithm to test against, perhaps an open-source version found on the web somewhere. Apart from the trust issues, there are also problems here. There might not be any shortest-path algorithm that deals with all the particular features that your maps contain. And, it is much more likely than people think that two completely different people presented with the same programming task will make the same mistakes in their solutions, thus again increasing the risk of mistakes that may be missed by the testing framework.

To sum up, what we need in an automated testing framework are properties that (1) are efficient enough so that they can be run many times on different test data, (2) are significantly less complex than the program under test, and (3) still cover the full functionality that we want to test.

Point (3) alludes to the *completeness* of a test framework; it is understood that we desire a test framework that, for any fault contained in the program under test, there is at least one property and one input that can be generated by the corresponding test data generator, such that the property fails. (This does not mean that the test framework will definitely find every bug; it means that we have not a priori precluded the finding of any bug, and that for any bug there is a possibility that it will be found by the test framework.)

Property-based testing

We can take another approach, and rather than look for a complete monolithic golden implementation, look at different properties we would like the program under test to satisfy.

One interesting property we should definitely check is the following: Property “Safe”: Given a map M, and two points A and B, if the car navigation component produces a path P, it should actually be a valid path from A to B. It is a bit tedious, but doable to write a program that given a map M, two points A and B, and a path P, checks that P actually is a path from A to B on M. The program that checks this is definitely a lot simpler than the component we are testing.

Is the above property “Safe” enough?

No. There are many possible flaws in the car navigation component that cannot be detected by this property alone. For example, the property only says what should happen when the component actually finds a path. It does not say whether it is OK or not if the component does not find a path. Also, the component is supposed to find the *shortest* path, something which the property does not say anything about either.

So, we would also like to formulate and test a property we can call “Complete”: If the component does not find a path, there actually was no path from A to B. And a property “Optimal”: If the component finds a path P between A and B, there exists no shorter path than P. But how? Doesn’t it seem daunting to formulate test oracles that check things like “there is indeed no path from A to B” or “there are no shorter paths between A and B” without making use of complex shortest path algorithms?

This project proposes a new technique to solve the above problem. The problem pattern appears in many other contexts as well, all involving programs with a high algorithmic content. Some such hard-to-test programs solve decidable problems and are inherently finite, such as SAT-solvers, constraint solvers, and model checkers. Others involve non-decidable issues such as theorem provers and compilers. What they have in common is that their original specifications (which were not practically testable) can be broken down into several smaller properties, which (a) together logically imply the original specification, and (b) are much easier to test. **The way we break down the original specification is by following at the structure of a possible mathematical correctness proof of the program**; the essence of our proposed **proof-based testing**. Note that we do not actually perform this correctness proof; this would move us over to the field of formal methods, which involves a mathematical model of the program under test, plus a logical system in which we can reason about that model; a much more expensive operation. Instead, the test framework designer merely gets inspired by a possible mathematical proof and structures the test framework accordingly.

Contrapositive testing

So how then can we test the car navigation component? One way is to slightly shift viewpoint on the property “Complete” we want to test. Instead of checking that “If the program cannot find a path, there exists no path between A and B”, we use contrapositive reasoning (which considers “(not Q) implies (not P)” when proving “P implies Q”, which are logically equivelent statements). The contrapositive of “Complete” is: If there exist a path between A and B, the component will find a path. This is easy to test; just create a generator that generates maps M with points A and B *and a guaranteed path between A and B*, and have the oracle check that the produced answer always is

a path. (It seems so obvious once you see it!) So, using contrapositives we can transform a hard-to-check post-condition after the fact into an easy-to-generate pre-condition!

Testing “Optimal” is now also easy, using contrapositive: Use the generator that generates maps M with points A and B and a guaranteed path between A and B, and have the oracle check that the produced answer always is a path of length not greater than the guaranteed path.

Inductive testing

Contrapositive is a simple yet powerful method when it works. But it is not always the case that it easy to generate inputs for which we have guaranteed outputs, and still have the generator be surjective (meaning it is still possible to generate *any* input with a guaranteed output). In this case, inductive testing may help.

In inductive testing, we look at which prerequisites a proof by induction would need to arrive at the conclusion that the program is correct. In our case, we want to test that the program finds all paths that are there to find. Let us reason to this by induction. First, we test that the program finds all paths of length 0. So, we test: For any map, finding the shortest path from A to A should lead to a path from A to A of length 0. This is the *base case*.

Then we test: For any map M and points A and B, the answer by the car navigation component should be consistent with the answer given when we ask it to find paths from N to B, for all neighbours N of A. So, in this test, we run the component several times, and check that the answers are consistent with each other. Consistent means that: (a) if no path was found from A to B, no path should exist from any N to B either, and (b) if a path was found from A to B, it should not be longer than the shortest path found through any N to B. This is the *step case*.

If both of these tests pass, we can (by testing) assume that the base case and step case of the induction hold, and then by induction conclude that the original component satisfies its specification! The soundness of mathematical induction guarantees that we shave not lost completeness of the test framework.

Survey of the field

Property-based testing, the main testing technique used in this project, lies in between two extremes of software verification: traditional unit testing (in which particular test cases are chosen by a human) and formal verification (in which properties about programs can be proved using a theorem prover). Property-based testing allows for general properties of programs, albeit in a very restrictive logic. Examples of property-based testing systems are Microsoft’s *generalized unit testing* and our own

very successful system QuickCheck.

There have been other combinations of testing and theorem proving before. The most obvious one is *compositional testing*, where properties of a larger system are decomposed into properties for parts of a system. Those properties can be tested for those parts of the system. The final conclusion that the original system satisfies its properties is made by logically combining the tested properties of the parts. The main reason for doing this is to increase test efficacy; it is easier to find bugs in a component of a system by testing that component than by testing the whole system at once.

In some sense, inductive testing can be seen as a generalization of this idea. However, the decomposition we propose does not follow the structure of the program, it follows the structure of an imaginary, possible correctness proof of the program. The reason we do this is also not to increase test efficacy, it is to be able to test the desired properties *at all*.

We have observed a number of specific instances of some of the ideas behind inductive testing in the literature. One notable example is the verification of pipelined microprocessors. The key property one wants to verify is that a pipelined microprocessor behaves “the same” as its non-pipelined counterpart. However, specifying exactly what a modern microprocessor should do, even ignoring pipelining, is a huge task. Instead, the community developed a method where the expected behavior of a pipelined processor is expressed in terms of the behavior of that same processor, where pipelining is somehow disabled (e.g. by flushing the pipeline after each instruction). In some sense, this can be seen as inductive testing but only using the step case of the induction. The base case (actually comparing with a non-pipelined processor) is left out for practical reasons.

Project description

The project will investigate proof-based testing as a technique, and also develop a theoretical framework which can be used to reason about logically equivalent, but testing-wise different properties. Finally, we want to work on implementing these ideas in a tool usable by real software developers.

As for finding complex algorithmic software, we plan to concentrate on two kinds of software:

A. Automated reasoning tools used in formal methods. We choose this domain because (1) we have a lot of experience developing this kind of software, and (2) it is important that programs used to prove other software correct is bug-free! A recent study by Biere et al. has shown, rather embarrassingly, that virtually all solvers of a particular category (SMT solvers) competing in a competition had a number of bugs in them. So, it is important that we develop new ways of testing these pieces of software. Our preliminary results have shown that this seems to be a viable way.

Examples of software we want to deal with here are automated theorem provers, and model checkers.

B. Non-deterministic concurrent software. This is an increasingly important class of software programs, because more and more software runs on servers talking to clients. It is notoriously difficult to specify the expected behavior of concurrent software, especially when non-determinism is involved, which is almost always the case. Model-based testing methods often decide to only concentrate on certain parts of the system that are predictable enough to be modelled correctly. Our hope is that inductive testing can be used here successfully. This software domain is more speculative, but our earlier work on testing concurrent software (ICFP 2010) uses a method rather like inductive testing, so we believe this is possible.

For proof-based testing to be really usable we need to have a way to reason about which way of testing a property is better. Often, there are many possible choices of what element of the input to do induction on. Which is best? A simple example to illustrate is testing the property "Forall x . A(x) => B(x)". The most direct way to test this is to generate some x that satisfies A(x), and then check if B(x) holds. However, the property is logically equivalent to "Forall x . ~B(x) => ~A(x)", which can be tested by generating some x that satify ~B(x) and check that ~A(x) holds. Which is better? It turns out that it depends on the frequency with which A(x) and B(x) are true on the whole domain.

A framework for reasoning about such choices could include an improvement operator >>, which we can use to say things like P >> Q, which means that the formulas P and Q are logically equivalent, but testing P gives more information than testing Q. It is vital that P and Q express the same thing logically! This ensures that the induction technique we applied when testing was sound, which is important to check. We envision a calculus involving >> and logical formulas, relating a logical semantics of formulas to a *testing semantics* of formulas.

Project plan

Year 1: We will investigate in detail how to use proof-based testing effectively for a number of different examples, mostly taken from the two categories above. This will hopefully lead to more general insights in the do's and don'ts behind the method. The expected outcome here is a number of well-documented examples of testing complicated software, some experimental results, together with some general conclusions. We also expect to run into limitations of the applicability of the method, which we hopefully can document as well. One such limitation is having to deal with *semi-decision procedures,* which either terminate with a result or do not terminate at all.

Year 2: We will develop a theoretical framework that can be used to reason about how to choose in which way we should test a given property. The hope is that we can "predict" the outcome of the experimental results gathered in Year 1. We also expect to come up with other testing techniques,

similarly inspired by mathematical logic proof techniques.

Year 3: We will design and implement a prototype tool that programmers can use to apply proof-based testing to their own programs, and reason about which approach is best. For example, it should be possible to express a desired property in a high-level way, and with the help of the tool "refine" the high-level property into a number of testable properties. We probably need to draw on our own expertise on automated theorem proving here.

Year 4, 5: We will push the limits on the limitations we have encountered earlier. For example, what to do for semi-decision procedures? Can time-outs be used as sound approximations? Also, we expect we may run into scalability issues when starting to deal with larger examples that will have to be addressed.

Significance

Proof-based testing has the potential to bring complete testing to a class of software components which were previously thought impossible to specify and test in this way. Opening up the possibility of testing complex algorithmic software without having to compare against a golden implementation is one contribution of the project.

Another contribution of the project will be the theoretical framework used for reasoning about what is the most effective way of testing a certain high-level property, in terms of a number of (more low-level) testable properties. This will have impact beyond the cases where proof-based testing makes sense.

Finally, we really strive to make our findings available to real software developers, and plan to implement a tool that can be used to reason about proof-based testing properties. This will greatly increase the applicability of our methods.

Preliminary results

We have already applied proof-based testing to several different kinds of software components.

We successfully managed to specify and test a number of academically developed software components, such as search in graphs (e.g. shortest path algorithms), optimization problems (e.g. finding the shortest edit distance between two text files), and translators (e.g. some simple compilers).

We have also applied inductive testing to the development of our own automated reasoning tools, notably our SAT-solver MiniSat and our hardware model checker Tip. Both these programs contain many sophisticated algorithms and their complexity is comparable to industrial versions of these programs.

For SAT-solvers, inductive testing works in straightforward by doing induction on the number of variables in a given problem P. For the step case, the SAT-solver under test is run on the problem P as well on the problems P[x/0] and P[x/1]. The three results should satisfy a simple boolean relationship. The base cases check all problems with no variables.

For model checkers, there was no obvious way to express their correctness in terms of testable properties using inductive testing, because there was no obvious element in the input to perform induction over! In the end, we found a couple of ways of expressing the correctness using ***co-inductive testing***, which uses co-induction over the infinite streams used in the semantics of hardware circuits. We were amazed that something so theoretical as co-induction could be used to make testing practical!

Because these programs were developed by ourselves, we had a database of about a dozen non-trivial bugs we had made during their development. We evaluated the (co-)inductive testing techniques against traditional testing (which compares the output of the program against a golden implementation).

Our findings in these (rather non-trivial) cases was that we need about 2-5 times more random tests to find these bugs compared to traditional testing. This means that, for this application, if the golden implementation is more than 2-5 times slower than the implementation under test, inductive testing seems to be a good idea! It is rather likely that a golden implementation is much slower than a real implementation for complex algorithms, so this is a very promising result.

One final note. During the development of our model checker Tip, we *only* used traditional testing against a golden implementation. However, it turned out that, because of us being not careful enough, a very subtle bug had creeped into both our release implementation and the golden implementation, making it impossible for the testing we did to discover the bug! This shows the dangers with testing against a golden model.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of applicant
Lindström Claessen, Koen

Date of birth
750724-8115

Title of research programme
Proof-based Testing -- Using Proof Techniques to Enable Automated Testing of Complex Algorithms

Appendix B

Curriculum vitae

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

CV: Koen Lindström Claessen, 1975-07-24

Academic Degrees

• 2010: Biträdande professor (Professor) in Computer Science, Chalmers University of Technology.

• 2004: Docent (Associate Prof.) in Computer Science, Chalmers University of Technology.

• 2001: Ph.D. degree in Computer Science, *Embedded Languages for Describing and Verifying Hardware*, Chalmers University of Technology. Advisor: Mary Sheeran.

Employment

• 2010–now: Professor at the dept. of Computer Science and Engineering, Chalmers. Research 75%.

• 2011-2013: I was on parental leave for a total of 10 months during 2011-2013.

• 2004–2010: Associate Prof. at the dept. of Computer Science and Engineering, Chalmers. Research 75%.

• 2001–2008: Senior Researcher at Safelogic AB / Jasper AB. Employment 20%.

• 2001–2004: Assistant Prof. at the dept. of Computer Science and Engineering, Chalmers.

Supervision Experience

Finished students

• Jean-Philippe Bernardy, PhD. 2011 (co-supervisor; main supervisor was Patrik Jansson).

• Niklas Sörensson, PhD. 2010 (technical supervisor; main supervisor was Reiner Hähnle).

• Hans Svensson, PhD. 2008 (main supervisor).

• Jan-Willem Roorda, PhD. 2007 (technical supervisor; main supervisor was Mary Sheeran).

Current students

• Joel Svensson, expected PhD. 2012 (co-supervisor; main supervisor is Mary Sheeran).

• Nicholas Smallbone, expected PhD. 2012 (main supervisor).

• Michal Palka, expected PhD. 2013 (main supervisor).

• Ann Lilliestöm, expected PhD. 2014 (main supervisor).

• Ramona Enache, expected PhD. 2015 (co-supervisor; main supervisor is Aarne Ranta).

• Nikita Frolov, expected PhD. 2017 (co-supervisor; main supervisor is John Hughes).

• Dan Rosen, expected PhD. 2017 (main supervisor).

Invited Talks, Lectures, Tutorials

• 2011: I was an invited speaker at the Conference on Automated Deduction (CADE). Title: "Equinox: An Instantiation-based Theorem Prover".

• 2009: I was an invited speaker at the Asian Symposium on Programming Languages and Systems (APLAS). Title: "The Twilight Zone: From formal specification to informal verification and back again".

• 2008: I gave a tutorial at the International Conference on Functional Programming (ICFP). Title: "Debugging Haskell programs using QuickCheck and HPC".

• 2007: I was an invited speaker at the Workshop on Disproving, at the Conference on Automated Deduction (CADE) in Bremen. Title: "A Paradigm Shift in ATP: Towards Model-based Reasoning Systems".

• 2006: I was an invited speaker at the Workshop on Pragmatics of Decision Procedures, at the Federated Logic Conference (FLoC) in Seattle. Title: "New Applications of Finite Model Finding".

• 2006: I was a lecturer at the 6th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Hardware Verification in Bertinoro, Italy. Title: "An Introduction to Symbolic Trajectory Evaluation".

Refereeing Experience

• I have served on the program committees of the following conferences in my research fields: ICFP (04,09), Haskell Symposium (11, as chair), FMCAD (07,08,09), TAP (09), IFL (08), HOPL (07), LPAR (05,06), DATE (06,07,08), ICCAD (05,06,07). Workshops: PAAR (08), LaSh (08), ESFOR/ESCAR/ESCOR (04,05,06), BMC (04,05,06), Haskell Workshop (99,06).

Selected Achievements

• 2003, 2004, 2005, 2006, 2007, 2008, 2009, 2010, 2011, 2012: Paradox (my automatic model finder for first-order logic) placed 1st in the CASC "World Championship on Theorem Proving" in the satisfiability/models category. See http://www.tptp.org/CASC/

• 2008, 2011, 2012: Tip (a hardware model checker developed together with Niklas Sörensson) placed 1st in the international model checking competition HWMCC, in the SAT category (2008) and in the liveness checking category (2011), and in both (2012). See http://fmv.jku.at/hwmcc12/.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of applicant
Lindström Claessen, Koen

Date of birth
750724-8115

Title of research programme
Proof-based Testing -- Using Proof Techniques to Enable Automated Testing of Complex Algorithms

Appendix C

List of publications

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

Publication List 2004-2013: Koen Lindström Claessen

(**Note to non-computer scientists** Conference articles in computer science are peer reviewed full articles — not 1–2 page abstracts, and are the "normal" form of refereed publication. The top conferences in each subfield (like *POPL* below) typically have the highest "impact factor" within that field. All articles listed below have been selected for publication by a competitive peer review process.)

Claessen's Hirsch-index is 20, as calculated using Google Scholar.

1. Journal articles

1. Koen Claessen, Ann Lilliestr\"om. Automated inference of finite unsatisfiability. *Journal of Automated Reasoning* 47(2):111-132, 2010.

2. Koen Claessen, Niklas Een, Mary Sheeran, Niklas Sörensson, Alexey Voronov, and Knut Åkesson. SAT-solving in practice, with a tutorial example from supervisory control. *Discrete Event Dynamic Systems* 19(4):495, 2009.

3. Koen Claessen and Jan-Willem Roorda. A faithful semantics for generalised symbolic trajectory evaluation. *Logical Methods in Computer Science*, 5(2), 2009.

4. Koen Claessen. Parallel parsing processes. *Journal of Functional Programming*, 14(6):741–757, 2004.

2. Articles in refereed collections and conference proceedings

5. (*) D. Rosen, N. Smallbone, M. Johansson, K. Claessen. Automating Inductive Proofs using Theory Exploration. In *Proc. of Conference on Automated Deduction (CADE)*, 2013.

6. D. Vytiniotis, S. Jones, K. Claessen, D. Rosén. HALO: From Haskell to first-order logic through denotational semantics. In *Proc. of Principles of Programming Languages (POPL)*, 2013.

7. Koen Claessen, Niklas Sörensson. A Liveness Checking Algorithm that Counts. In *Proc. of Formal Methods for Computer Aided Design (FMCAD)*, 2012.

8. Koen Claessen. Shrinking and Showing Functions (Functional Pearl). In *Proc. of Haskell Symposium*, 2012.

9. Koen Claessen, Moa Johansson, Dan Rosen, Nick Smallbone. HipSpec: Automating Inductive Proofs of Program Properties. In *Proc. of Automated Theory Exploration (ATX)*, 2012.

10. J. Svensson, K. Claessen, M. Sheeran. Expressive array constructs in an embedded GPU kernel programming language. In *Proc. of 7th workshop on Declarative aspects and applications of multicore programming (DAMP)*, 2012.

11. G. Sutcliffe, S. Schulz, K. Claessen, et. al. The TPTP typed first-order form with arithmetic. In *Proc. of Logics for Programming and Automated Reasoning (LPAR)*, 2012.

12. K. Claessen, A. Lilliestr\"om, N. Smallbone. Sort It Out with Monotonicity. In *Proc. of Conference on Automated Deduction (CADE)*, 2011.

13. Emil Axelsson, Koen Claessen, Mary Sheeran, Josef Svenningsson, David Engdal and Anders Persson. The Design and Implementation of Feldspar. In *Proc. of Implementation of Functional Languages (IFL)*, 2011.

14. J. Svensson, K. Claessen, M. Sheeran. Obsidian: A domain specific embedded language for parallel programming of graphics processors. In *Proc. of Implementation of Functional Languages (IFL)*, 2011.

15. (*) M. Palka, K. Claessen, A. Russo, J. Hughes. Testing an optimising compiler by generating random lambda terms. In *Proc. of 6th International Workshop on Automation of Software Test (AST)*, 2011.

16. (*) K. Claessen, N. Smallbone, J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Proc. of Tests and Proofs (TAP)*, 2010.

17. Jasmin Blanchette, Koen Claessen. Generating counterexamples for structural inductions by exploiting nonstandard models. In *Proc. of Logics for Programming and Automated Reasoning (LPAR)*, 2010.

18. (*) J-P. Bernardy, P.Jansson, K. Claessen. Testing polymorphic properties. In *Proc. of European Symposium on Programming (ESOP)*, 2010.

19. Koen Claessen, John Hughes, Michal Palka, Nick Smallbone, Hans Svensson. Ranking programs using black box testing. In *Proc. of the 5th Workshop on Automation of Software Test*, 2010.

20. E. Axelsson, K. Claessen, G. Devai, Z. Horvath, K. Keijzer, B. Lyckegard, A. Persson, M. Sheeran, J. Svenningsson, A. Vajdax. Feldspar: A domain specific language for digital signal processing algorithms. In *Proc. of Formal Methods and Models for Codesign (MEMOCODE)*, 2010.

21. (*) Koen Claessen, John Hughes, Nick Smallbone, Michal Palka, Thomas Arts, Ulf Wiger. Testing Distributed Programs using Pulse and QuickCheck. In *Proc. of International Conference on Functional Programming*, September 2009.

22. Koen Claessen, Ann Lilliestr\"om. Automated Inference of Finite Unsatisfiability. In *Proc. of Conference on Automated Deduction (CADE)*, August 2009.

23. Koen Claessen, Carl Seger, Mary Sheeran, Emily Shriver, and Wouter Swierstra. High-level architectural modelling for early estimation of power and performance. In *Proc. of Hardware Description using Functional Languages (HFL)*, March 2009.

24. Joel Svensson, Mary Sheeran, and Koen Claessen. Obsidian: A domain specific embedded language for general-purpose parallel programming of graphics processors. In *Proc. of Implementation and Applications of Functional Languages (IFL)*, Lecture Notes in Computer Science. Springer Verlag, March 2009.

25. Dana N. Xu, Simon Peyton Jones, and Koen Claessen. Static contract checking for Haskell. In *Proc. of Symposium on Principles of Programming Languages (POPL)*. ACM SIGPLAN, January 2009.

26. Koen Claessen and Hans Svensson. Finding counter examples to induction proofs. In *Proc. of International Conference on Tests and Proofs (TAP)*, Lecture Notes in Computer Science. Springer Verlag, April 2008.

27. Koen Claessen, Niklas Een, Mary Sheeran, and Niklas Sörensson. SAT-solving in practice. In *Proc. of Workshop on Discrete Event Systems (WODES)*. IEEE, May 2008.

28. Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in Haskell. In *Proc. of Haskell Symposium*. ACM SIGPLAN, September 2008.

29. Koen Claessen and Gordon Pace. Embedded hardware description languages: Exploring the design space. In *Proc. of Hardware Description using Functional Languages (HFL)*, March 2007.

30. Koen Claessen. A coverage analysis for safety property lists. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*. IEEE, November 2007.

31. Geoff Sutcliffe, Stephan Schulz, Koen Claessen, and Allan van Gelder. Using the TPTP language for writing derivations and finite interpretations. In *Proc. of International Joint Conference on Automated Reasoning (IJCAR)*, Lecture Notes in Computer Science. Springer Verlag, August 2006.

32. Jan-Willem Roorda and Koen Claessen. SAT-based assistance in abstraction refinement for Symbolic Trajectory Evaluation. In *Proc. of Conference on Computer-Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer Verlag, August 2006.

33. Jan-Willem Roorda and Koen Claessen. Explaining Symbolic Trajectory Evaluation by giving it a faithful semantics. In *Proc. of International Computer Science Symposium in Russia*, Lecture Notes in Computer Science. Springer Verlag, June 2006.

34. Thomas Arts, Koen Claessen, John Hughes, and Hans Svensson. Testing implementations of formally verified algorithms. In *Proc. of Conference on Software Engineering Research and Practice in Sweden (SERPS)*. Mälardalen University, October 2005.

35. Koen Claessen and Jan-Willem Roorda. A new SAT-based algorithm for Symbolic Trajectory Evaluation. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

36. Emil Axelsson, Koen Claessen, and Mary Sheeran. Wired: Wire-aware circuit design. In *Proc. of Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 3725 of *Lecture Notes in Computer Science*. Springer Verlag, October 2005.

37. Koen Claessen and Hans Svensson. A semantics for distributed Erlang. In *Proc. of Erlang Workshop*. ACM SIGPLAN, September 2005.

38. Thomas Arts, Koen Claessen, and Hans Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Proc. of Workshop on Formal Aspects of Testing (FATES)*, Lecture Notes in Computer Science. Springer Verlag, 2004.

39. Koen Claessen and Johan Mårtensson. An operational semantics for weak PSL. In *Proc. of Conference on Formal Methods for Computer Aided Design (FMCAD)*, Lecture Notes in Computer Science. Springer Verlag, 2004.

4. Book chapters, lecture notes

Number of citations calculated by Google Scholar.

40. Koen Claessen and Jan-Willem Roorda. An introduction to Symbolic Trajectory Evaluation. In *6th International School on Formal Methods for the Design of Computer, Communication and Software Systems: Hardware Verification*, Lecture Notes in Computer Science. Springer Verlag, May 2006.

5. Publicly available implementations

41. Koen Claessen and Niklas Sörensson. Tip. *Tip is an automatic model checker for hardware circuits. It has won the HWMCC "Hardware Model Checking Competition" in 2008, 2011 and 2012.* 2008–2012.
Available from https://github.com/niklasso/tip

42. Koen Claessen and Niklas Sörensson. Paradox. *Paradox is an automatic model finder for first-order logic. It has won the CASC "World-Championship on Theorem Proving" in its category every year since 2003.*

2003–2012.
Available from http://www.cs.chalmers.se/~koen/Folkung/

43. Koen Claessen. Equinox. *Equinox is an experimental theorem prover for first-order logic.* 2007–2012.
Available from http://www.cs.chalmers.se/~koen/Folkung/

44. Ann Lillieström and Koen Claessen. Infinox. *Infinox is automated tool for showing the absence of finite models for first-order logic problems.* 2009.
Available from http://www.cs.chalmers.se/~koen/Folkung/

45. Koen Claessen. Luke. *Luke is an automatic verification system for the Lustre synchronous programming language. It has been mainly developed for teaching purposes.* 2004.
Available from http://www.cs.chalmers.se/Cs/Grundutb/Kurser/form/luke.html

46. Koen Claessen, Nick Smallbone, and John Hughes. QuickCheck. *QuickCheck is a specification-based random testing tool for the functional programming language Haskell.*. 2001–2012.
Available from http://code.haskell.org/QuickCheck/

6. Popular Scientific Presentations

47. Koen Claessen. Datasystem som fungerar – en utopi? *Presentation at the Science Fair ("Vetenskapsfestivalen") in Göteborg.* 2005.

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Name of applicant
Lindström Claessen, Koen

Date of birth
750724-8115

Title of research programme

Appendix N

Budget and research resources

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se

Budget

We plan to hire one doctoral student for 5 years on the project, who will need supervision, which will be 10% of the time of the main applicant. Amounts are in MSEK.

	%	2014	2015	2016	2017	2018
Koen Lindström Claessen	10%	142	147	152	158	164
PhD student	80%	604	625	648	671	695

Other costs are travel, computer equipment, rooms, licentiate and PhD defense costs, and administrative and IT support.

	2014	2015	2016	2017	2018
Travel (conferences, etc.)	40	40	40	40	40
Equipment	80				
Lic/defense			20		20
Rooms	48	49	51	53	55
IT	8	9	9	9	10

The total amounts per year are:

	2014	2015	2016	2017	2018	total
Total	922	871	920	931	1 004	**4 649**

VETENSKAPSRÅDET
THE SWEDISH RESEARCH COUNCIL

Kod
2013-24716-106139-24

Dnr
2013 -

Name of applicant
Lindström Claessen, Koen

Date of birth
750724-8115

Reg date
2013-04-11 23:55:53

Project title
Proof-based Testing -- Using Proof Techniques to Enable Automated Testing of Complex Algorithms

NT

Research grants NT April 11, 2013
Project Research Grant
*Nat-Tek generellt
Chalmers tekniska högskola

Appendix S

Signatures

NT-2

2014	2015	2016	2017	2018	
922	870	920	931	1004	2014-01-01 -- 2018-12-31

Applicant | **Date**

Head of department at host University | **Clarifi cation of signature** | **Telephone**

A signature on the application is required not only from the applicant but also from the authorised representative of the administrating organisation/institution or equivalent (normally the head of the department or establishment where the research is to be conducted). The signature confirms that the department can accommodate the proposed research, position or equipment; that the costing in the application is approved for the department's part, that any proposed experimentation on human or animal subjects has been reported, and that the applicant has reported any secondary occupations and commercial ties (s)he may have, and nothing inconsistent with good research practice has thereby emerged. The applicant must have discussed these conditions with the representative of the administrating organisation/institution or equivalent before the latter approves and signs the application.

NOTE the exceptions!
Please note that a different procedure applies for the following types of grants:

- Infrastructure- Operation Grants
- Infrastructure- Grants for Expensive Equipment
- Infrastructure- Grants for Large Databases
- Framework Grant SIMSAM

For these types of grants the vice-chancellor, instead of the head of department, must sign Appendix S. Thereby (s)he confirms the commitments of the administrative organization. The details are specified in the Call for Proposals, published at www.vr.se.

Vetenskapsrådets noteringar
Kod

2013-24716-106139-24

Vetenskapsrådet, Box 1035, SE-101 38 Stockholm, tel. +46 (0)8 546 44 000, vetenskapsradet@vr.se