UNIVERSITÄT
KOBLENZ · LANDAU
Institut für Informatik

FB 4
Informatik

Tests and Proofs: Papers Presented at the Second International Conference TAP 2008 Prato, Italy, April 2008

Bernhard Beckert
Reiner Hähnle
eds.

No. 5/2008

Reports of the Faculty of Informatics

Reports of the Faculty of Informatics

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Volume editors:
Bernhard Beckert
Faculty of Informatics
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: beckert@uni-koblenz.de
Reiner Hähnle
Dep. of Computer Science
Chalmers University of Technology
Rännvägen 6B,
SE-412 96 Gothenburg,
Email : reiner@cs.chalmers.se

Preface

This volume contains those research papers presented at the Second International Conference on Tests and Proofs (TAP 2008) that were not included in the main conference proceedings[1].

TAP was the second conference devoted to the convergence of proofs and tests. It combines ideas from both areas for the advancement of software quality.

To prove the correctness of a program is to demonstrate, through impeccable mathematical techniques, that it has no bugs; to test a program is to run it with the expectation of discovering bugs. On the surface, the two techniques seem contradictory: if you have proved your program, it is fruitless to comb it for bugs; and if you are testing it, that is surely a sign that you have given up on any hope of proving its correctness. Accordingly, proofs and tests have, since the onset of software engineering research, been pursued by distinct communities using rather different techniques and tools.

And yet the development of both approaches leads to the discovery of common issues and to the realization that each may need the other. The emergence of model checking has been one of the first signs that contradiction may yield to complementarity, but in the past few years an increasing number of research efforts have encountered the need for combining proofs and tests, dropping earlier dogmatic views of their incompatibility and taking instead the best of what each of these software engineering domains has to offer.

The first TAP conference (held at ETH Zurich in February 2007) was an attempt to provide a forum for the cross-fertilization of ideas and approaches from the testing and proving communities. For the 2008 edition we found the Monash University Prato Centre near Florence to be an ideal place providing a stimulating environment.

We wish to sincerely thank all the authors who submitted their work for consideration. And we would like to thank the Program Committee members as well as additional referees for their great effort and professional work in the review and selection process. Their names are listed on the following pages.

In addition to the contributed papers, the program included three excellent keynote talks. We are grateful to Michael Hennell (LDRA Ltd., Cheshire, UK), Orna Kupferman (Hebrew University, Israel), and Elaine Weyuker (AT&T Labs Inc., USA) for accepting the invitation to address the conference.

Two very interesting tutorials were part of TAP 2008: "Parameterized Unit Testing with Pex" (J. de Halleux, N. Tillmann) and "Integrating Verification and Testing of Object-Oriented Software" (C. Engel, C. Gladisch, V. Klebanov,

[1] Bernhard Beckert and Reiner Hähnle (eds.). *Tests and Proofs: Second International Conference, TAP 2008, Prato, Italy, April 2008, Proceedings*, LNCS 4966. Springer-Verlag, 2008.

and P. Rümmer). We would like to express our thanks to the tutorial presenters for their contribution.

It was a team effort that made the conference so successful. We are grateful to the Conference Chair and the Steering Committee members for their support. And we particularly thank Christoph Gladisch, Beate Körner, and Philipp Rümmer for their hard work and help in making the conference a success. In addition, we gratefully acknowledge the generous support of Microsoft Research Redmond, who financed an invited speaker.

April 2008

Bernhard Beckert
Reiner Hähnle

Conference Chair

Bertrand Meyer	ETH Zurich, Switzerland

Program Committee Chairs

Bernhard Beckert	University of Koblenz, Germany
Reiner Hähnle	Chalmers University, Gothenburg, Sweden

Program Committee

Bernhard Aichernig	TU Graz, Austria
Michael Butler	University of Southampton, UK
Patrice Chalin	Concordia University Montreal, Canada
T. Y. Chen	Swinburne University of Technology, Australia
Yuri Gurevich	Microsoft Research, USA
Dick Hamlet	Portland State University, USA
William Howden	University of California at San Diego, USA
Daniel Jackson	MIT, USA
Karl Meinke	KTH Stockholm, Sweden
Peter Müller	Microsoft Research, USA
Tobias Nipkow	TU München, Germany
Andrea Polini	University of Camerino, Italy
Robby	Kansas State University, USA
David Rosenblum	University College London, UK
Wolfram Schulte	Microsoft Research, USA
Natasha Sharygina	CMU & University of Lugano, Switzerland
Betti Venneri	University of Florence, Italy
Burkhart Wolff	ETH Zurich, Switzerland

Additional Referees

Michele Boreale	Mads Dam	Karol Ostrovsky
Roberto Bruttomesso	Andrew Edmunds	Edgar Pek
Myra Cohen	Viktor Kuncak	Rosario Pugliese
John Colley	Rupak Majumdar	Steffen Schlager

Steering Committee

Yuri Gurevich	Microsoft Research, USA
Bertrand Meyer	ETH Zurich, Switzerland

Organising Committee

Christoph Gladisch	University of Koblenz, Germany
Philipp Rümmer	Chalmers University, Gothenburg, Sweden

Sponsoring Institutions

Microsoft Research Redmond, USA
Chalmers University of Technology, Gothenburg, Sweden
University of Koblenz-Landau, Germany
ETH Zurich, Switzerland

Table of Contents

Experiences from Testing a Radiotherapy Support System with QuickCheck

Aiko Fallas Yamashita[1], Andreas Bergqvist[2], and Thomas Arts[2]

[1] Simula Research Laboratory, Box 134, 1325 Lysaker, Norway
aiko@simula.no
[2] IT University of Göteborg, Box 8718, 402 75 Göteborg, Sweden
{bergqvia, thomas.arts}@ituniv.se

Abstract. We present a case study on the use of lightweight formal methods for testing part of a real-time organ position tracking system used in radiotherapy. Several properties were modeled and verified through automated test cases generated by QuickCheck. QuickCheck was found useful in reducing the complexity inherent to testing medical devices by detecting faults at system level, and assisting in the exploration of atypical errors that could later be analyzed and fixed in the system. We suggest that a combination of lightweight formal methods and random test generation, supported by automated simplification of test cases may represent a feasible option in the medical domain; particularly for those projects with high-pace development, a need for proof-based techniques/tools for certification processes, and when the non-deterministic nature of real-time devices demands the exploration/identification of heterogeneous fault sources.

Keywords: Lightweight formal methods, model-based testing, medical software, software verification, software testing.

1 Introduction

With the increased use of software in medical devices, high demands on software verification and analysis in the medical domain are inevitable. In many cases formal methods are used to model medical devices [1], medical protocols [2], or even entire systems [3]. Proofs are becoming an important aspect in medical device certifications as organizations like the Food and Drugs Administration (FDA) [4] and its European counterpart, Medical Device Directive (MDD) [5] are moving from process-centered towards proof-based certification [6].

Nevertheless, full formalization of systems implies potentially high costs [7] and, in some industrial contexts, it may constitute an unrealistic task. Yet after the correctness of a model has been formally proven, its implementation still needs to be tested. A combination comprising of lightweight formal methods and testing has been proposed as a means for connecting the actual implementation and the formal properties of the system in a feasible and more direct way, avoiding errors in implementation details [8]. Studies addressing the usage of lightweight formal methods within different industrial contexts can be found in current literature [9–12].

In this article we present a case study on the use of lightweight formal methods for testing an implementation of a medical device. The device (constructed

by Micropos Medical [13]) is a real-time organ position tracking system used in radiotherapy, i.e., a tumor is positioned in real-time in order to be able to accurately deliver a dose of radiation. Part of this system was tested with QuickCheck [14] as a master thesis project.

The software to determine the position of the organ is critical. Only the intended area should be exposed to radiation, and any damage to surrounding healthy tissue should be avoided. Proving correctness of the software is extremely hard; radio signals are combined in order to determine a position. Imagine a radio antenna of which the top determines the position. Even if the top is fixed, the antenna can be in many different positions, all resulting in different radio signals. Even worst, the same antenna position not necessarily results in exactly the same signals each time one measures. Thus, we have deterministic software that provides a lot of computations on radio signals and returns a measured position. However, the software reacts very non-deterministic in a test setting, since placing the antenna in a certain position, may result in different signals and therefore slightly different measured positions. The algorithms for computing the measured position are under constant improvement, still we want to be able to ensure accuracy in any product that is released.

QuickCheck is a testing tool that randomly generates test cases from a given model and supports automated simplification of failed test cases. Thus, in theory we are able to generate test cases from the same specification that is used in to formulate and prove correctness of the system. The study aims to explore the potential contributions as well as the challenges of this specific set of techniques (i.e. lightweight formal methods and random testing supported by automated test case simplification) in testing a safety critical medical device, in order to assess the viability of lightweight methods within the medical domain.

The paper is subsequently organized as follows: Sect. 2 introduces briefly the context for medical device verification, proposing the necessity of integrating test tools and formal methods in the medical industry. Sect. 3 details related work to the approach used in the case study. Sect. 4 provides the motivations for our approach, and a description of the case study; indicating the testing setup, and the properties tested in the SUT. Sect. 5 describes the results and analysis derived from this study. Finally, Sect. 6 specifies the conclusions reached based on this study.

2 Verifying medical devices

Two primary approaches to the process of medical devices delivery are utilized: process centered verification and artifact-centered verification [15]. Process centered verification is often described by standards that suggest a series of practices for the medical practitioners to base the development of the safety-critical software products on [16, 17]. However, there is a need for more artifact-centered

approaches as medical devices turn more and more sophisticated, complex and wide-ranging; and general guidelines are proving to be insufficient for delivering a safe product [18, 15].

The application of formal methods in medical devices supports this line of action and could add significant confidence in the system by revealing errors in both the system's model and its implementation [19]. There are many success stories regarding the use of formal methods in the medical domain, which range from medical protocols [2, 20] to medical equipment controllers [21, 3] and medical devices [1]. Studies elucidate the need of well-established processes that include formal methods and ensure safe systems [22]. Despite the numerous advantages of formal methods, the actual implementation still needs to be tested given the differences that could exist between the model and the implementation. Furthermore, a testing process is a "must" in current certification processes regulated by medical authorities [23–25]. In our case study we had a mathematical formula to compute the difference between a position and a measured position. Given such a clear formal model, we liked to investigate how we could automatically generate tests from that model. What is needed is a connection between mathematics and the software in the implementation. We looked for a general approach instead of a specific solution for this case, in order to be able to apply the method to other parts of the system later on.

3 Related work

Our approach is based on a tool called QuickCheck, which is a property-based testing tool that automatically generates tests from a specification and has the ability to simplify failing test cases (or counter examples) automatically. Properties are modeled and used as input in QuickCheck in order to generate random test cases. Although we use the term property-based, it is clear that QuickCheck can be seen as a *model-based testing* tool, since it relies on a formal abstraction of a system property in order to generate the test cases. Another definition that may be applicable is *specification-based testing*, as we specify several aspects of the system in the form of properties. In the subsequent text, we describe related work in the area of testing (such as model-based testing, specification-based testing, boundary-based testing, on-the-fly testing) as well as other associated approaches in the field of formal methods (e.g. model checking).

Model-based testing (MBT) [27] is an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the system. Examples of MBT can be found in [28–30]. Specification-based testing on the other hand, tries to demonstrate that an implementation conforms to a certain specification of a system. Some examples are [31–38]. Both approaches can be considered orthogonal and most of the time well complemented with formal methods. An attempt to establish a taxonomy for MBT can be found in [39].

Boundary-driven testing as well as coverage-oriented testing are approaches that can be found together with MBT and Formal Methods. Boundary-driven testing selects values that are directly on, above, and beneath the edges of the legal input and output values. In contrast to random testing, boundary testing may require some expertise in order to select effective boundary cases [40]. Examples of boundary-driven approaches can be found in [41, 42]. A study addressing a combination of MBT and the coverage-oriented approach can be found in [43].

Model-based testing and Specification-based testing differ as white-box testing and black-box testing. Given a model, one has insight in border cases that occur during runtime (or simulation-time), whereas specifications only give borders that can be specified and do not consider borders that occur during runtime. For example, a specification could state that input values should be in a certain range, a model of the computation of could reveal that a buffer is overflow for specific input values, not necessarily the border cases of the given range. We used QuickCheck in a Specification-based approach, trusting that random generation of sufficiently many values would reveal errors. We are not actively looking for border cases, since many of these cases are caused by how well the radio signals can be interpreted, which cannot be read from the source code of an algorithm or its model. Moreover, the algorithm for computing the values is constantly improving and changing, therefore, the model would have to be re-designed after each such change.

The properties we checked with QuickCheck were somehow side-effect free. We send an antenna to a certain position, measure the position and compare with the actual position. In that sense, we differ from a model checking approach, where one may check that certain properties hold in a state, no matter which path one chooses to get to that state. In theory, no matter from which direction we reach a point in space, the computation should always return the same value for the same position. This is, though, not true, but that is not because of non-deterministic software, but because of radio signals not being deterministic.

Some may classify QuickCheck's approach under the rubric on-the-fly test generation, since it generates random test cases and verifies properties on the fly. On-the-fly test generation has been used before in the verification of real-time communication protocols [44]. This approach is considered suitable for real-time systems, specially when the test case generation can react to the actual outputs of the SUT while running under the operation environment (Further on we will explain how QuickCheck does this through the simplification of failed test cases). Since real-time systems are characterized as being non-deterministic, offline testing (the opposite approach to on-the-fly testing) is limited in its capacity to react to changes in the environment and identify faults that are linked to such changes.

4 Case study

This section provides the details of the study. We start by describing the medical SUT. Secondly, we present a description of QuickCheck and motivate our approach for testing. Subsequently, we present the properties tested along with a description of the testing set-up.

4.1 Position tracking device

The SUT is called 4DRT (Four Dimension Radio Therapy) and is a real-time organ position tracking system intended for supporting radiotherapy. It is able to locate the position of an organ in four dimensions, three-dimensional space and time. This enables one to monitor the position of tumors in prostate cancer patients and thereby helps to improve the accuracy of the radiation during radiotherapy treatments.

The SUT is based on radio frequency transmission. The measurement of the position is done through an implantable device in the organ (or nearby), which acts as a transmitter. The transmitter emits a radio frequency, which is captured by multiple receivers, typically arranged in a plate on the treatment table under the patient (See Fig. 1). The software uses the signal captured by the receivers as input to calculate the position of the organ. A set of floating-point values (representing the measured signals) is continuously sent to the software. The software maps the floating-point values retrieved from the Receivers to a specific coordinate position in the real world. This coordinate position is given in a coordinate system specific to the SUT, which has a predetermined range for each axis (X, Y, Z) and two angles (Vy, Vz), i.e., rotation over y and z axes. The "mapping process" or the algorithm for calculating the position uses a mathematical model not discussed here due to disclosure agreements. The non-functional requirement described in Table 1 explicates the accuracy required, and it is expressed in terms of confidence intervals; i.e., positions calculated by the SUT should be within a radial distance (in Euclidean space) of 2±1mm from the actual position to ensure that the tumor receives radiation and not the healthy tissue around it. That is, the calculated position should not divert from the actual position with more than 2mm with a standard deviation of 1mm.

Table 1. Description of the functional requirement (and its corresponding non-functional requirement) tested in the SUT

Functional requirement:	The software component should calculate the 5D positioning of the transmitter (X, Y, Z, Vy, and Vz, where Vy is rotation around Y axis and Vz is rotation around Z axis)
Non-functional requirement:	The system should achieve 3D difference or radial accuracy of 2 ±1 mm

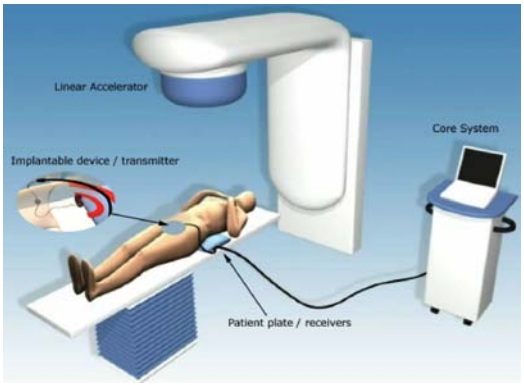

Fig. 1. View of the 4DRT system in a treatment environment. Elements such as the Linear Accelerator, the implantable device, and the patient plate are depicted.

The software of the SUT is the result of migrating a prototype from LabView [47] to a commercial platform language i.e., Microsoft .NET C#. Pseudo-code describing the underlying algorithm for position estimation and the LabView code were used for performing the migration. Even if the equivalence of the algorithm implementation (between LabView and C#) can be reviewed through code inspection, it is still a challenge to ensure correct behavior during its execution. Micropos Medical was mainly interested in a solution that may enable testing in real life conditions and identify problems at system level. Due to signal fluctuations that are dependent on the environment, on-site calibration is also required. Micropos needs a cost-effective solution for performing system level testing on a regular basis during development and after deployment.

4.2 QuickCheck and the proposed approach

QuickCheck is a tool that combines random test generation, with a flexible language for specifying generators and the use of properties to verify test results [8]. The properties can be written in a restricted logic, and then QuickCheck can be invoked to test the property in a large number of cases. Properties are specified in Erlang [48]. Among other things, one can quantify over sets and express preconditions. For example, the property

```
prop_positive() ->
   ?FORALL({Pos1,Pos2},{coordinate(),coordinate(),
      ?IMPLIES(Pos1 =/= Pos2,
               radial_distance(Pos1,Pos2) > 0)).

radial_distance({XP,YP,ZP},{XC,YC,ZC}) ->
   math:sqrt(
```

```
        math:pow(XP-XC,2)+math:pow(YP-YC,2)+math:pow(ZP-ZC,2)).

coordinate() ->
   {int(),int(),int()}.
```

checks whether for two generated coordinates `Pos1` and `Pos2` that if they are not equal, then there distance is positive. Here, `FORALL` and `IMPLIES` are examples of logic operators provided by the QuickCheck library. QuickCheck generates test cases according to the provided generators, in this case coordinates, which only uses randomly generated integers in three dimensions. QuickCheck allows focusing on the properties that code should satisfy, rather than on the selection of individual test cases. As mentioned before, QuickCheck also performs the automated simplification of failing test cases. Details concerning this last feature can be found in [26].

Property-based testing. From a Risk-Based Analysis outset, verifying the accurate position calculation is key in assuring a safe treatment delivery. A QuickCheck property was formulated and corroborated through execution (See Fig. 2). The property should hold if the radial distance (See Formula 1) between the position estimated by the software and the actual position is less or equal to 2mm. The advantage of QuickCheck over other tools in this context is that the input to QuickCheck (the property modeled) is very close to the mathematical specification that one would expect. Hence, it is easy to inspect that the right aspect of the SUT has been tested) (cf. Fig 2).

$$\sqrt{((X_p - X_c)^2 + (Y_p - Y_c)^2 + (Z_p - Z_c)^2)} \tag{1}$$

Random testing. In terms of coverage in the underlying test domain, it is clear that due to the nature of the software we are testing, the process of requesting only one single position calculation will cover the critical path of the modules. Thus, if the transmitter is located in {0,0,0,0,0} and then we request the position, we would have full code coverage without revealing any failure in the SUT. Therefore, we need to test many data points. The test set-up is such that we mounted an antenna to a kind of three-dimensional plotter. We can instruct this plotter to move the antenna to a certain position, specified in millimeter precision and we could then turn the antenna in a certain angle. The parameters needed to place the antenna in a given position are given in natural numbers, which correspond to the number of millimeters the antenna is moved away from the origin. Typically an area much larger than a potential tumor is tested, say a cube of 200 millimeters on each side. Together with a potential angle of the antenna, this results in about 720 million positions to test. Moving the antenna in a certain position takes up to 7 seconds, thus testing all positions is rather impractical. Moreover, the same position may return different values at different points in time, thus testing positions more than once is not superfluous.

System level testing. Given the simplicity of the property (the accuracy) that we want to test and the dependency of the whole SUT to correctly pass a large number of tests; we estimate that we can catch all failures that otherwise would be caught by unit testing. Thus, it seems that starting with system level testing and leaving out unit testing is cheaper in this case than designing dedicated tests for each unit. Because of the simplicity of the underlying formula for correctness (Formula 1), the ease with which this formula can be expressed in QuickCheck, and the kind of errors we can expect (typical for floating point handling), we decided to use system level testing as the only way of testing.

4.3 Testing environment and tested properties

The testing set-up. The lab setting used during testing consisted of a transmitter, a receiver, software and an additional mechanical device called Auto Setup[3] to which the transmitter is attached. QuickCheck generated coordinates as integer values within a range supported by the SUT. The coordinates were then used to control the Auto Setup, which, in turn, moved the transmitter to a corresponding position. The software of the SUT calculated the position of the transmitter and "sent" the calculated X, Y, Z, Vy, Vz coordinates back to QuickCheck. These calculated values are floats. QuickCheck then determined the radial distance between the initially generated position and the position calculated by the SUT. A test fails if this distance is more than 2mm. The property is depicted in Fig. 2. QuickCheck communicated via TCP/IP with a sort of request broker that we implemented in C#. This broker receives commands from QuickCheck and requests the Auto Setup to move the transmitter to a specified coordinate and then calls the software component of the SUT to request the position estimation. Details of the testing set-up are provided in [49].

```
prop_within_margin(Margin) ->
   ?FORALL(Coordinate, antenna_coordinate(),
           begin
              move_antenna_to(Coordinate),
              Position = read_position(),
              radial_distance(Position, Coordinate) =< Margin
           end).
```

Fig. 2. Accuracy Property tested in QuickCheck

Accuracy Property. In Fig. 2 `antenna_coordinate()` is a function that generates a random triplet of x, y and z coordinates and a pair representing the

[3] A simplified version of a Coordinate Measurement Machine (CMM)[52], referred to here as Auto Setup is used. A CMM consists of a workspace where parts (a sensor and a mechanical assembly for moving the sensor around the workspace) are fixed. In our case, the sensor consists of the transmitter and the mechanical assembly situates the transmitter at specific coordinates indicated through an external software interface.

angles of the transmitter relative to the antenna's y and z axes. The generated value is bound to the variable `Coordinate`. First the transmitter is moved to a certain position. The function called `move_antenna_to(Coordinate)` returns a value when the transmitter has reached the desired point. After that the most recently estimated position is fetched from the SUT, it is then compared to the actual coordinates. Whenever a test fails, i.e., any of the actions fails or the result of the last inequality is `false`, then QuickCheck will automatically search for simplified failing test cases. An example of a generated simplification of failing test case constituted one of the border cases i.e., {0,0,0,0,0}. QuickCheck randomly generated each test from a QuickCheck property such as the one presented in Fig. 2. Typically, integer values specifying millimeters were used to move the transmitter to a given position (the Auto setup can be moved in steps of 1mm). QuickCheck could for instance generate a test from the property in which the transmitter is steered to position: X=58, Y=127, Z=94, Vy=0, Vz=0. The estimated position: X=58.15106462, Y=126.9147189, Z=94.82734652, Vy=-2.582979671, Vz=-3.070729491 is then registered. The distance to the real value is computed: 0.84533759 and since it is less than 2mm, the test passes successfully[4].

QuickCheck uses a uniform distribution in its random generation of coordinates. For the purpose of testing the software, we are satisfied by that. The non-functional requirement in Table 1 does indicate, however, to use a normally distributed set of sample points and to generate a normally distributed sample from them. Since patient data is unavailable at this point, we decided to be stricter than that and use a uniform distribution, requiring an accuracy of 2mm, without leaving space for points in one standard deviation. We analyzed the few failing tests (i.e., those with a distance larger than 2mm) to see by how much they deviated.

Symmetry Property. The SUT works under the assumption that the underlying mathematical model used by the position calculation algorithm is symmetric. This means that given a coordinate, a similar accuracy on the corresponding extrapolated coordinate is attained with SUT (i.e. if the transmitter position is {36,41,73}, the SUT will give similar results in accuracy as if the transmitter was located in the extrapolated value {134,139,171}). This is presented in property `prop_symmetric()`which is depicted in Fig. 3. We verify that the accuracy distance between a given coordinate value and its corresponding extrapolated coordinate is less than 1mm. We used 1mm as the delimitation value for practical reasons. The value was experimentally determined by a test simplification that helped us to determine that the major difference between results of extrapolated coordinates in the SUT didn't exceed 1mm.

[4] It is important to point out that the Vy and Vz are only considered for performing the position calculation and not for computing the radial distance. Hence the radial distance shown in the example only contemplates X, Y and Z.

```
prop_symmetric(Margin)->
   ?FORALL(Coordinate,antenna_coordinate(),
           begin
              Extrapolated = extrapolate(Coordinate),
              move_antenna_to(Coordinate),
              Pos1 = read_position(),
              move_antenna_to(Extrapolated),
              Pos2 = read_position(),
              Distance1 = radial_distance(Coordinate, Pos1),
              Distance2 = radial_distance(Extrapolated, Pos2),
              abs(Distance1 - Distance2) =< 1
           end).

extrapolate({X,Y,Z})->
      {?upper_x - abs(X-?lower_x),
       ?upper_y - abs(Y-?lower_y),
       ?upper_z - abs(Z-?lower_z)}.
```

Fig. 3. Symmetry property tested in QuickCheck

5 Results and Analysis

In this section, results from the testing process, perceived benefits from our approach, and possible areas for improvement are presented and discussed.

5.1 Test results

Within the given coverage range, the SUT provided even better accuracy than that specified by the non-functional requirement. One large sample of generated tests had a mean of 1.528505mm for the radial distance, with a standard deviation of ± 0.477921, where 87% of test cases passed and 13% failed; from the failed test cases, 11% had between 2mm and 2.4mm for radial distance and 2% between 2.4mm and 3.4mm. Others were even more accurate and only 2% of the test cases failed, displaying a radial distance of 2.02mm to 2.04mm. The set of test cases proved that the SUT had better accuracy than the requirement, and we felt very satisfied considering the results above.

Most of the failures were detected in the first test cases QuickCheck produced from the main property described in Sect. 4.3. In all cases, it was possible to trace the failures back to the code. So, adequate corrections could be performed. Typical issues involved floating point operations, type conversion, and the use of erroneous types in the drivers' interfaces. For instance, we found out that the hardware driver for the Auto Setup did not accept decimal points as parameters in one of the interfaces. This problem was identified when using QuickCheck for sending the coordinates to the Auto Setup and it was observed that the latter did not move the transmitter as expected. We could trace this problem back to a division operation in the Auto Setup interface, which was performed prior to sending the coordinates to the actual Auto Setup controller. This division

produced decimal values occasionally instead of just integers. Consequently, the Auto Setup only moved when the resulting division was a whole number.

Another problem came about because of the use of incorrect casting operations (i.e. truncating decimals instead of rounding), which was detected while observing a set of failed test cases showing a very similar radial distance. We found that conversion in LabView is implicitly managed, in contrast to C#, which requires a specific conversion method.

In addition, errors due to misinterpretations of the pseudo-code (i.e. declaration of global variables and static values interpreted as local and dynamic variables) could be detected by observing failed test cases that showed a very big radial distance. A similar error was found in the same test cases, where an incorrect constant value for one of the algorithms was used (due to the mistakes during the migration process where an outdated version of LabView code was used for a specific module).

The aforementioned issues are typical when performing migration from two different platforms (in this case from LabView to C#), where some assumptions (such as typing and management of decimal values) in the old platform are not longer valid in the new platform. They are also related to a typical situation in the medical domain when specifications regarding interfaces for hardware drivers as well as software COTS (components off the shelf) are not so clear [50]. QuickCheck facilitates code refinement and simplifies the task of detecting those errors (mostly within a couple of property executions).

Note that we found all these errors by specifying just one property and generating random test cases from it. Therewith, the work of creating test cases is dramatically simplified in contrast to more traditional testing approaches. Also note that most of these errors are implementation errors and no matter how well the model is formally verified, such errors can appear.

It was moreover possible to determine which of the underlying mathematical models (See Section 4.1) for calculating the position support a given radial accuracy. Whenever a new model is introduced, it is possible to test it with QuickCheck and its adequacy being visible almost immediately. For instance, one day we had introduced a model, which was stated to provide better accuracy than the previous one; we ran QuickCheck on it and found a coordinate with an unacceptable accuracy. Hence, the model was further improved before being introduced again.

Issues in the communication protocol between QuickCheck and the C# request broker were also detected with QuickCheck. Incidentally, a problem due to the overwriting of instructions from the C# request broker into the Auto Setup driver was detected while executing tests (this overwriting issue resulted in a series of incomplete test executions). We found that the Auto Setup driver demands a lapse of 40ms in order to process one instruction and read the next one.

5.2 Perceived contributions from the approach

Improved coverage in regression testing. We perceived that it was possible to introduce changes in other parts of the SUT (e.g. hardware, since this product is evolving constantly; making devices smaller and faster) and afterwards use QuickCheck to perform high-level testing. This enabled us to detect any incongruence or errors that might result as a consequence from those changes. The same situation applies to code enhancements performed in order to improve performance. Some data processing in LabView could be implemented in C# in a more efficient way. We run QuickCheck to make sure that these enhancements in the code gave the same results as the original algorithms written in LabView. Furthermore, the coverage of the side effects resulting from changes introduced in the software (or hardware) is more comprehensive with QuickCheck since it generates new random test cases each time. In that sense, QuickCheck constitutes a good asset for a product that is constantly evolving (a scenario very typical in medical device development [51]) in contrast to regression testing which will run the same tests-suites every time.

Improving the system quality. An example of how QuickCheck helped in improving the quality of the software occurred when we utilized various mathematical models to see which ones gave better results (as explained previously in Sect. 5.1). Furthermore, having a formal specification of the SUT that can actually be run and corroborated constitutes a significant advantage for certification processes (as pointed out by [6] and mentioned in Sect. 2).

Cost effectiveness. Faults related to testing the mathematical model (accuracy checking) were detected after on average 12.85 test cases, and abnormal cases were detected after approximately 78 tests. It is very unlikely that one would manually write test cases with the same results, but it shows that several cases would have to be written for a good test suite, whereas here we only write one property once.

Each time a test case is run, the transmitter must be positioned before performing the measurement, and this is a rather expensive task if an automated tool does not support it. In our case, it took in around 5-7 seconds per each test case; depending on to which position the Auto Setup was moved. QuickCheck requires relatively little amount of effort, and supports repetitiveness and generation of new values every time.

Support for detection of atypical faults. Sometimes you want to run the property for a longer period and use extreme values (including boundary cases) on the test parameters in order to find atypical results. By extending the margin tolerance (increasing the radial accuracy limit), we could detect atypical cases related to the transmitter angles (angles very close to the negative or positive borders brought about significant radial distances). For instance, when we modified the accuracy property and set the accuracy tolerance up to 6mm; an apparently normal position (in the sense that it was within the coverage-range

of the SUT) resulted in a radial distance of almost 6mm. Following some more tests, we found out that one version of the underlying mathematical model used in the SUT was sensitive to strongly angled positions (in terms of Vy and Vz). This finding lead to adjustments in the mathematical model in order to improve its robustness against angling. It must be mentioned that the parameters used for performing this type of testing exceeded the limits of what could be called a normal scenario (e.g. test parameters derived from real patient data).

5.3 Areas for improvement

We have identified a number of limitations in our case study. This study does not cover the necessity of having a given distribution (in this case a normal distribution) and usage of sample data from patients.

When a test fails, we want to obtain the coordinates that give the highest possible measurement fault, in other words, for which the distance to the position is greatest. This is not possible to perform automatically with the current version of QuickCheck. QuickCheck provides simplification of input data but cannot yet generate data depending on the outcome of particular tests.

It is worth mentioning that one of the limitations of working in a lab is the presence of sporadic radio transmission noise due to research activities taking place at nearby companies. This also enforces a sufficient number of tests in order to assure the robustness of the SUT in less than ideal situations. We can store the test case sequences in QuickCheck and redo the property execution in order to see any behavior that can be influenced by the environment and signal fluctuations. This would be particularly good if we want to improve the robustness of the system to external noise factors, which is very common in an environment like a hospital.

Throughout this study, we have observed a potential for QuickCheck to support statistical functionalities (e.g. to test confidence intervals). Some planned features for future releases include control or specification of the number of test cases, and generation of test cases by sampling from a defined set of data (i.e. real patient data). Also, improving the logging capabilities for QuickCheck could notably expand the potential of using QuickCheck for test results analysis. Logging not only failed test cases but also the asserted test cases would potentially upgrade the tool.

6 Conclusion

We have described a case study on testing a medical device by using a formal model as a basis for the automatic generation of test cases with the tool QuickCheck. We found a number of errors in the code we developed and were able to spot inaccuracies in prototype models. Early detection and correction of

these errors has lead to a high quality product being developed by the medical company at which the case study was performed.

This case study assembles adequate conditions for using formal models. The model is simple, clear and based on a mathematical formula. Verifying medical devices may not always be like this case, and there may be a need for more complex modeling for system behavior. Nevertheless we believe that is it worthwhile to try the technology on more medical equipment. We intend to continue this work involving more complex properties than the ones presented here.

The most remarkable aspects of this study focus on several positive results: First, property-based testing proved to be feasible and cost-effective within this domain in contrast to the normal tendency of using test suites. This is of great value particularly for those projects with high-pace development, typically involving continuous modifications in the code in order to improve performance and constant incorporation of new features. QuickCheck's approach on lightweight formal specification has great potential to be used in proof-based certifications for medical devices as recognized by several medical practitioners involved with the project.

References

1. J. L. Cyrus, J. Daren and P. D. Harry: Formal Specification and Structured Design in Software Development. Hewlett-Packard Journal, 1991
2. J.W. Brakel: Formal Verification of a Medical Protocol. ISIS Technical Report, University of Twente, 2005
3. V. Kasurinen and K. Sere: Integrating action systems and Z in a medical system specification. Industrial Benefit and Advances in Formal Methods, LNCS **1051**, (1996) 105–19
4. Food and Drug Administration (FDA). Online: *http://www.fda.gov*
5. Medical device directive (MDD). Online: *http://www.mdss.com/MDD/mddtoc.htm*
6. I. Lee, G. Pappas, R. Cleaveland, J. Hatcliff, B. Krogh, P. Lee, H. Rubin and L. Sha: High-Confidence Medical Device Software and Systems. Computer, **39(4)**, (2006) 33–38
7. D. Jackson and J. Wing: Lightweight Formal Methods. IEEE Computer, **29(4)**, (1996) 22–23
8. T. Arts, K. Claessen, J. Hughes, and H. Svensson: Testing implementations of formally verified algorithms. *Proc. 5th Conf. on Soft. Eng. Research and Practice in Sweden*, (2005) 20–21
9. M. Kim, S. Kannan, I. Lee and O. Sokolsky: Java-MaC: a Run-time Assurance Tool for Java. *Proc. Runtime Verification*, Electronic Notes in Theoretical Computer Science, **55**, Elsevier Science, 2001.
10. S. Nelson and C. Pecheur: V&V of advanced systems at NASA, Produced for the Space Launch Initiative 2nd Generation RLV TA-5 IVHM Project, 2002
11. M. Taghdiri and D. Jackson: A lightweight formal analysis of a multicast key management scheme. *Proc. 23rd IFIP Int. Conf. on FTNDS*, (2003) 240–256
12. S. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, D. Hamilton: Experiences using lightweight formal methods for requirements modeling. IEEE Soft. Eng. **24(1)**, (1998) 4–14
13. Micropos Medical AB. Official web site: *http://www.micropos.se*
14. K. Claessen and J. Hughes: QuickCheck: a lightweight tool for random testing of Haskell programs. *Proc. 5th ICFP'00*, (2000) 268–279
15. P. Jones: Assurance and Certification of Software Artifacts for High-Confidence Medical Devices. High Confidence Medical Device Software and Systems Workshop, Philadelphia, USA, 2005.
16. J. Bowen and V. Stavridou: Safety-critical systems, formal methods and standards. Soft. Eng. Journal **8(4)**, (1993) 189–209

17. D.R. Wallace, D.R. Kuhn and L.M. Ippolito: An analysis of selected software safety standards. *Proc. 7th Conf. on Computer Assurance*, (1992) 123–136
18. R. Jetley and S. P. Iyer: Enabling Certification through an Integrated Comprehension Approach. In [15].
19. J.P. Bowen and V. Stavridou: Formal methods and software safety. Safety of Computer Control Systems, Pergamon Press, (1992) 93–98
20. M. Marcos, M. Balser, A. ten Teije and F. van Harmelen: From informal knowledge to formal logic: a realistic case study in medical protocols. *Proc. 13th Int. Conf. EKAW*, LNCS **2473**, (2002) 49–64
21. J. Jacky, J. Unger, M. Patrick, D. Reid and R. Risler: Experience with Z developing a control program for a radiation therapy machine. *Proc. of 10th Int. Conf. of Z Users*, LNCS **1212**, (1996) 317–328
22. J. Rushby. Formal Methods and the Certification of Critical Systems. Computer Science Laboratory, SRI International, Menlo Park, CA. Number SRI-CSL-93-7, December 1993.
23. R. Jetley, P. Iyer and P. Jones: A Formal Methods Approach to Medical Device Review. IEEE Computer, **39(4)**, (2006) 61–67
24. Food and Drug Administration (FDA). "General Principles of Software Validation: Final Guidance for Industry and FDA Staff", FDA, 2002.
25. Medical Device Directory (MDD). "Council directive 93/42/EEC of 14 June 1993 concerning medical devices", Medical Device Directory, 2003.
26. T. Arts, J. Hughes, J. Johansson, and U. Wiger: Testing telecoms software with Quviq QuickCheck. *Proc. ACM SIGPLAN Workshop*, (2006) 2–10
27. Encyclopedia on Software Engineering (edited by J.J. Marciniak), Wiley, 2001. Ibrahim K. El-Far and James A. Whittaker: "Model-Based Software Testing"
28. H.S. Hong, I. Lee, O. Sokolsky and H. Ural: A temporal logic based coverage theory of test coverage and generation. *Proc. 8th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems*, LNCS **2280**, (2002) 327–339
29. I. Gronau, A. Hartman, A. Kirshin, K. Nagin and S. Olvovsky: A methodology and architecture for automated software testing. IBM Research Laboratory in Haifa Technical Report, MATAM Advanced Technology Center, Haifa 31905, Israel
30. J. Dick and A. Faivre: Automating the generation and sequencing of test cases from model-based specifications. *Proc. of Industrial-Strength Formal Methods*, LNCS **670**, (1993) 268–284
31. P. Ammann and A. J. Offutt: Using formal methods to derive test frames in category-partition testing *Proc. 9th Conf. on Computer Assurance*, IEEE Computer Society Press, (1994) 69–80
32. R. A. Kemmerer: Testing formal specifications to detect design errors. IEEE Trans. on Soft. Eng., **11(1)**, (1985) 32–43
33. G. Laycock: Formal specification and testing: A case study. The Journal of Software Testing, specification, and reliability. **2(1)**, (1992) 7–23
34. P. Stocks and D. Carrington: A framework for specification-based testing. IEEE Trans. on Soft. Eng., **22(11)**, (1996) 777–793
35. W. T. Tsai, D. Volovik, and T. F. Keefe: Automated test case generation for programs specified by relational algebra queries. IEEE Trans. on Soft. Eng., **16(3)**, (1990) 316–324
36. A. J. Offutt, Yiwei Xiong, Shaoying Liu: Criteria for generating specification-based tests. *5th IEEE Int. Conf. ICECCS*, (1999) 119–129
37. E. Weyuker, T. Goradia, and A. Singh: Automatically generating test data from a boolean specification. IEEE Trans. on Soft. Eng., **20(5)**, (1994) 353–363
38. B. Nielsen and A. Skou: Automated Test Generation from Timed Automata. *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, (2001)
39. M. Utting, A. Pretschner and B. Legeard: A Taxonomy of model-based testing. University of Waikato, Department of Computer Science. No. 04/2006.
40. S. Butler, S. Chalasani, S. Jha, O. Raz and M. Shaw: The Potential of Portfolio Analysis in Guiding Software Decisions. *EDSER-1 as part of ICSE'99*, (1999)
41. B. Legeard, F. Peureux, and M. Utting: Automated Boundary Testing from Z and B. *Proc. Int. Symposium of Formal Methods*. LNCS **2391**, (2002) 21–40
42. N. Kosmatov, B. Legeard, F. Peureux, M. Utting: Boundary Coverage Criteria for Test Generation from Formal Models. *ISSRE*, (2004) 139–150

43. F. Belli, M. Eminov and N. Gokce: Prioritizing Coverage-Oriented Testing Process – An Adaptive Learning Based Approach and Case Study. *Proc. 31st Int. Conf. Computer Soft. and Applications*. IEEE Computer Society, 197–203
44. R. Castanet, O. Koné, P. Laurençot: On-the-fly test generation for real-time protocols. *Proc. 7th Int. Conf. in Comp. Comm. and Networks*, IEEE Computer, 1998.
45. R. Alur, C. Courcoubetis and D. Dill: Model-checking for real-time systems. Logic in Computer Science, (1990) 414–425
46. T. Jéron, P. Morel: Test generation derived from model-checking. *CAV'99, Italy*, LNCS **1633**, (1999) 108–122
47. National Instruments:"NI LabVIEW" Online: *http://www.ni.com/labview/whatis/*
48. J. L. Armstrong, M. Williams, R. Virding, and C. Wilkström: ERLANG for Concurrent Programming. Prentice-Hall, 1993.
49. A. Fallas Yamashita and A. Bergqvist: Testing a radiotherapy support system with QuickCheck. Masters thesis, IT University of Göteborg, Sweden, 2007.
50. G. Sharp and N. Kandasamy: A Dependable System Architecture for Safety-Critical Respiratory-Gated Radiation Therapy. *Proc. Int. Conf. on Dependable Systems and Networks*, **00**, DSN, IEEE Computer Society, June 2006.
51. M. Poonawala, S. Subramanian, W. Tsai, R. Mojdehbakhsh and L. Elliott: Testing Safety-Critical Systems – A Reuse-Oriented Approach. *Proc. 9th Int. Conf. Software Eng. and Knowledge Eng.*, Knowledge Systems Institute, (1997) 271–278
52. W. Singhose, N. Singer and W. Seering: Improving repeatability of coordinate measuring machines with shaped command signals. Precision Engineering **18**, (1996) 138–146

Model Validation through CooML Snapshot Generation

Camillo Fiorentini and Mario Ornaghi

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti, ornaghi}@dsi.unimi.it *

Abstract. The object of this paper is model validation, namely the study of the "correctness" of formal specifications (or models) with respect to their requirements. Since requirements are informal, validation can be only experimental. In the UML, snapshot generation has been proposed as a tool for validating class diagrams and contracts. Snapshots are object diagrams representing the possible system states, where one compares the snapshots generated from the UML model with the expected ones. In this paper we present a DLV implementation of snapshot generation for CooML (Constructive Object Oriented Modeling Language), a modeling language based on a constructive semantics we are developing. We firstly explain CooML snapshot semantics and we show by an example how UML class diagrams with OCL constraints can be represented in CooML. Then we explain our implementation of CooML snapshot generation in DLV and discuss its potential application for model validation.

1 Introduction

The object of this paper is model validation, namely the study of the "correctness" of formal specifications (or models) with respect to their requirements. More specifically, we consider OO models, where the system behavior arises from the interaction of its objects. Furthermore, an OO model should represent an abstraction of the problem domain, that is, especially in the early design phases, classes should have an "immediate" counterpart in the problem domain; the objects populating a system state should represent a "snapshot" of a corresponding counterpart in the modeled world. Here, by *snapshot* we mean a user oriented representation of a possible internal system state, at the level of abstraction of the OO model considered, starting from the models. In this context, tools for "snapshot validation" through automatic snapshot generation become an important part of model validation. For example, USE [12] is a snapshot generator for the UML, where snapshots are object diagrams. They represent possible system states at the level of details of the class diagram considered. Since requirements are informal, validation can be only empirical, i.e., it is performed by comparing the formal model with the user's expectations. Validation through snapshot generation should (typically) support the following kinds of experiments:

* Work partly supported by the MIUR Project "Potenziamento e Applicazioni della Programmazione Logica Disgiuntiva".

- Check that unexpected populations are not generated.
- Generate all the snapshots that should correspond to specific expected states.
- Generate histories, i.e., sequences of snapshots showing how the system evolves as effected by external events (such as, for example, a withdraw from a bank account).

In the first part of the paper, we present CooML (Constructive Object Oriented Modeling Language) [9, 20], a object oriented modeling language we are developing. The novelty of CooML is its semantics, based on *Fcl*, an intermediate constructive logic introduced by Miglioli at al. [18] and similar to Medvedev's Logic of finite problems [17]. In this logic the information content of objects is defined through information terms. Informally, an information term τ for a formula F, indicated by $\tau : F$, represents an explanation of F according to the BHK interpretation [22]. We consider the pair $\tau : F$ as a piece of information that can be queried and we formally define its "answer set" as a set of formulas representing its information content. A special operator T (standing for classical truth) is used to explicitly indicate the subformulas for which we do not want a constructive analysis. For example, $\exists x A(x)$ means that a witness for x is required in the answer set, while $T(\exists x A(x))$ only requires that an x exists, without asking for a witness. The following features of CooML semantics help with respect to the above motivations.

- The information term semantics allows us to specify consistent snapshots in a formal and clear way. Furthermore, UML class diagrams with OCL constraints can be represented in CooML.
- At the class specification level, the operator T allows us to selectively define the information required for the objects of a class.
- At the system specification level, it gives a clear-cut distinction between the information content of objects, given by their information terms, and the general constraints coming from the problem domain, axiomatized by T-formulas. The use of T allows us to hide details in the generated snapshots, as well as considering different levels of abstraction.

In the second part we explain our implementation of CooML snapshot generation in DLV and discuss its potential applications for model validation. DLV [14] is a system implementing stable model semantics for disjunctive Datalog programs (DDP), namely Datalog programs (logic programs without function symbols) allowing disjunction in the head. The expressive power of DLV is strictly higher than the one of Datalog [6]. In our implementation, we want to exploit the expressivity of DLV to represent CooML snapshot generation problems by DDP's and its capability of generating in an efficient way the stable models of a DDP to effectively generate CooML snapshots.

Finally, in the conclusion, we briefly mention some related work and we list some of our future goals.

2 CooML specifications

In this section, we explain the basic features of CooML. Instead of introducing the original Java-like syntax, we base our explanation directly on many-sorted first order classical logic. Signatures Σ, Σ-formulas F, Σ-interpretations i and the truth relation $\mathrm{i} \models F$ are defined as usual. CooML specifications are special first order formulas, that we call *properties*. Properties have a constructive semantics based on information terms and I-answer sets. They may contain the "classical truth" operator T, where by $T(P)$ we indicate that we are not interested in a constructive analysis of P, but only in its classical truth. The truth relation $\models$ is extended to T just by ignoring it: $\mathrm{i} \models T(F)$ iff $\mathrm{i} \models F$. The syntax of CooML properties is introduced in Section 2.1, while information terms, their I-answer sets and the constructive semantics of CooML are explained in Section 2.2. In Section 2.3, we introduce generation problems.

2.1 The syntax of CooML properties

Let Σ be a first order signature. In CooML Σ-properties (or "properties" for short), universal quantifiers are bounded by *generators*, i.e., by formulas that are true over a finite domain. For conciseness, we do not specify the syntax of generators, but it will be clear from the examples. The syntax of Σ-properties is shown in Table 1; a CooML specification is a set of properties.

Table 1. CooML Σ properties

Simple S	::=	*true*	| literal	| $T(F)$	where F is any Σ-formula
Existential E	::=	S	| $E \wedge E$	| $E \vee E$	| $\exists \underline{x}\, E$
Property P	::=	E	| $\forall \underline{x}.\, G(\underline{x}) \rightarrow E$	where $G(\underline{x})$ is a Σ-generator binding $\underline{x}$		

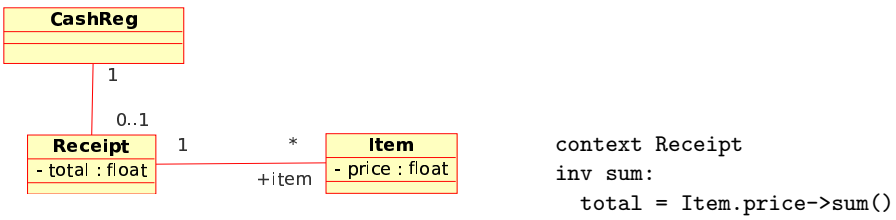

Fig. 1. UML class diagram

Example 1. In this example, we show how UML+OCL specifications can be represented in CooML. Let us consider the UML diagram in Fig. 1. It models a

problem domain consisting of cash registers, receipts and items. A cash register contains 0 or 1 receipts and each receipt is contained in one cash register. A receipt may have 0 or more items, and each item is contained in one receipt. A receipt computes a grand total and each item has a price. The OCL constraint indicates that the grand total of a receipt is the sum of the prices of all its items. To translate the UML model into CooML, we firstly introduce the following signature.

$$\begin{array}{ll} \text{class}: & cashReg(c : \text{OBJ}),\ \ receipt(r : \text{OBJ}, c : cashReg),\ \ item(i : \text{OBJ}) \\ \text{assoc}: & itemOf(i : item, r : receipt) \\ \text{funct}: & total(r : receipt) : float,\ \ price(i : item) : float \end{array}$$

The sorts OBJ and $float$ are *pre-defined* and are implicitly imported. The pre-interpretation of $float$ depends on floating point arithmetic, while OBJ provides the names to be used to identify the objects. The declared symbols are *internal*. In our example, they are used to represent the UML diagram, as follows:

- We associate with each UML class C a CooML class predicate $C(\_)$, where $C(o)$ means that o is a live object of class C. A CooML class predicate may depend on parameters, we call "environment parameters". For example, the class predicate $receipt(r : \text{OBJ}, c : cashReg)$ depends on $c : cashReg$, indicating that the receipt r is associated with cash register c.
- To represent an association, we can use a parametric class predicate, such as $receipt(r : \text{OBJ}, c : cashReg)$ in our example. We can also introduce *association predicates*, such as $itemOf(i : item, r : receipt)$.
- Finally, attributes are represented by functions. In our example, we use the functions $total(r : receipt) : float$ and $price(i : item) : float$.

We represent the UML diagram of Fig. 1 with the following CooML specification $CashSpec$:

$$\begin{array}{ll} \text{classax1}: & \forall c.\, cashReg(c) \ \rightarrow\ T(\neg\exists r.\ receipt(r,c)) \vee \exists r.\ receipt(r,c) \\ \text{classax2}: & \forall r, c.\, receipt(r,c) \ \rightarrow\ \exists t.\ t = total(r) \\ \text{classax3}: & \forall i.\, item(i) \ \rightarrow\ \exists p.\ p = price(i) \\ \text{assocax1}: & \forall i, r.\, itemOf(i,r) \ \rightarrow\ true \\ \text{constr1}: & T(\forall r_1, r_2, c.\, receipt(r_1,c) \wedge receipt(r_2,c) \ \rightarrow\ r_1 = r_2) \\ \text{constr2}: & T(\forall r, c_1, c_2.\, receipt(r,c_1) \wedge receipt(r,c_2) \ \rightarrow\ c_1 = c_2) \\ \text{constr3}: & T(\forall i \exists r.\, item(i) \ \rightarrow\ itemOf(i,r)) \\ \text{constr4}: & T(\forall i, r_1, r_2.\, itemOf(i,r_1) \wedge itemOf(i,r_2) \ \rightarrow\ r_1 = r_2) \\ \text{constr5}: & T(\forall r, c.\, receipt(r,c) \ \rightarrow\ total(r) \ =\ sum(p : \exists i.(p = price(i) \wedge itemOf(i,r)))) \end{array}$$

The above properties represent the structure of the class diagram, its multiplicity constraints and the OCL constraint. The class axiom classax1 says that a cash register is empty or has one receipt, classax2 that a receipt contains a witness for its total, classax3 that an item contains a witness for its price. The association axiom assocax1 links an item i with its receipt r by means of the $itemOf(i,r)$ predicate. We remark that the class predicates $cashReg(c)$, $receipt(r,c)$, $item(i)$ and the association predicate $itemOf(i,r)$ are generators and fix the domain of live objects and links among them. On the other hand, T-formulas are used

to represent constraints. For instance, constr1 establishes that a cash has at most one receipt, while constr5 corresponds to the OCL constraint of Fig. 1. The external operator $sum(L)$ represents the (pre-defined) sum over lists L of numbers, where L is the list of the prices p. Formally, $s = sum(p : G(p))$ is an abbreviation of $T((\exists L.p \in L \leftrightarrow G(p)) \wedge s = sum(L))$. Class predicates are not types, but they can be used as types in declarations and implicitly introduce "declaration constraints". For example, $itemOf(i : item, r : receipt)$ has declaration constraint $T(\forall i, r.\, itemOf(i, r) \rightarrow item(i) \wedge \exists c.\, receipt(r, c))$ indicating that i must be an item and r a receipt.

2.2 Constructive semantics

We introduce the *constructive* interpretation of a CooML specification (see also [9,20] for more details). Each property P specifies a set $\textsc{it}(P)$ (*information type*) of possible *information terms*, corresponding to the possible explanations of the truth of P. Formally, the information type $\textsc{it}(P)$ is recursively defined as follows:

$$
\begin{array}{lcl}
\textsc{it}(S) & = & \{\mathtt{tt}\}, \text{ where } S \text{ is a simple } \Sigma\text{-formula} \\
\textsc{it}(P_1 \wedge P_2) & = & \{\, [\tau_1, \tau_2] \mid \tau_1 \in \textsc{it}(P_1) \text{ and } \tau_2 \in \textsc{it}(P_2) \,\} \\
\textsc{it}(P_1 \vee P_2) & = & \{\, [k, \tau] \mid k \in \{1,2\} \text{ and } \tau \in \textsc{it}(P_k) \,\} \\
\textsc{it}(\exists \underline{x}.P) & = & \{\, [\underline{t}, \tau] \mid \underline{t} \text{ is a ground } \Sigma\text{-term and } \tau \in \textsc{it}(P) \,\} \\
\textsc{it}(\forall \underline{x}.G(\underline{x}) \rightarrow P) & = & \{\, [\,[\underline{t}_1, \tau_1], \ldots, [\underline{t}_n, \tau_n]\,] \mid \text{for all } 1 \le j \le n, \\
 & & \quad \underline{t}_j \text{ is a ground } \Sigma\text{-term and } \tau_j \in \textsc{it}(P) \,\}
\end{array}
$$

We write $\tau : P$ for $\tau \in \textsc{it}(P)$. The meaning of $\tau : P$ is given by the I-answer set $\textsc{ians}(\tau : P)$, recursively defined as follows:

$$
\begin{array}{lcl}
\textsc{ians}(\mathtt{tt} : S) & = & \{S\} \\
\textsc{ians}([\tau_1, \tau_2] : P_1 \wedge P_2) & = & \textsc{ians}(\tau_1 : P_1) \cup \textsc{ians}(\tau_2 : P_2) \\
\textsc{ians}([k, \tau] : P_1 \vee P_2) & = & \textsc{ians}(\tau : P_k) \\
\textsc{ians}([\underline{t}, \tau] : \exists \underline{x}.P(\underline{x})) & = & \textsc{ians}(\tau : P(\underline{t})) \\
\textsc{ians}(\,[\,[\underline{t}_1, \tau_1], \ldots, [\underline{t}_n, \tau_n]\,] : \forall \underline{x}.G(\underline{x}) \rightarrow P(\underline{x})\,) & = & \bigcup_{1 \le j \le n} \textsc{ians}(\tau_j : P(\underline{t}_j)) \cup \\
 & & \{\, \textsc{cc}(G(\underline{t}_1), \ldots, G(\underline{t}_n)) \,\}
\end{array}
$$

where $\textsc{cc}(G)$ is the "only if" part of Clark's Completion of G.

A *snapshot* for a specification *Spec* is a list of information terms, one for each formula of *Spec*. Formally:

Definition 1 (Snapshot). *Let $Spec = \{P_1, \ldots, P_n\}$ be a CooML specification. A snapshot $\underline{\tau}$ for Spec, written $\underline{\tau} : Spec$, is a list $\underline{\tau} = [\tau_1, \ldots, \tau_n]$ such that, for all $1 \le j \le n$, $\tau_j : P_j$. The I-answer set $\textsc{ians}(\underline{\tau} : Spec)$ is defined as $\textsc{ians}(\underline{\tau} : Spec) = \bigcup_{j=1}^{n} \textsc{ians}(\tau_j : P_j)$.*

The intuitive meaning of the above definitions is explained next.

Example 2. Let $CashSpec$ be the specification of Example 1. A possible snapshot $\underline{\tau} : CashSpec$, corresponding to the object class diagram of Fig. 2, contains the following information terms (we put a tuple of ground terms between round brackets):

$$[[\mathtt{c1}, [1, \mathtt{tt}]], [\mathtt{c2}, [2, [\mathtt{r(c2)}, \mathtt{tt}]]]] \quad : \forall c.\, cashReg(c) \rightarrow T(\neg\exists r.\, receipt(r, c)) \vee \exists r.\, receipt(r, c)$$

$$[[(\mathtt{r(c2)}, \mathtt{c2}), [12, \mathtt{tt}]]] \quad : \forall r, c.\, receipt(r, c) \rightarrow \exists t.\, t = total(r)$$

$$[[\mathtt{it1}, [5, \mathtt{tt}]], [\mathtt{it2}, [7, \mathtt{tt}]]] \quad : \forall i.\, item(i) \rightarrow \exists p.\, p = price(i)$$

$$[[(\mathtt{it1}, \mathtt{r(c2)}), \mathtt{tt}], [(\mathtt{it2}, \mathtt{r(c2)}), \mathtt{tt}]] \; : \forall i, r.\, itemOf(i, r) \rightarrow true$$

$\mathtt{tt}$: constr1 $\mathtt{tt}$: constr2 $\mathtt{tt}$: constr3 $\mathtt{tt}$: constr4 $\mathtt{tt}$: constr5

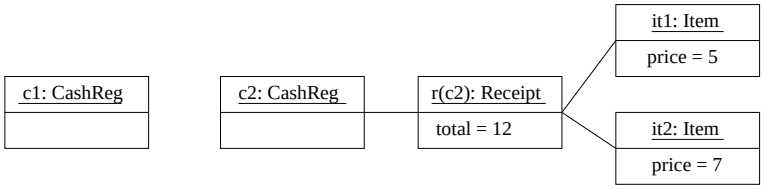

Fig. 2. An object class diagram (snapshot)

The first information term has the form $[[\mathtt{c1}, \tau_1], [\mathtt{c2}, \tau_2]] : \forall c.\, cashReg(c) \rightarrow D$ and the corresponding I-answer set is shown in row 1 of Fig. 3. It indicates that the generator $cashReg(c)$ holds for $c = \mathtt{c1}$, $c = \mathtt{c2}$ and nothing else. Note that the I-answer set contains the completion axiom $T(\forall c.\, cashReg(c) \rightarrow\ c = \mathtt{c1} \vee c = \mathtt{c2})$. Let us consider the subterm $\tau_1 : D$ associated with $\mathtt{c1}$. Since $\tau_1 = [1, \mathtt{tt}]$, the first disjunct of D holds, hence $\mathtt{c1}$ is empty. Let $\tau_2 = [2, [\mathtt{r(c2)}, \mathtt{tt}]] : D$ be the information term for $\mathtt{c2}$. In this case, the information term $[\mathtt{r(c2)}, \mathtt{tt}] : \exists r.\, receipt(r, \mathtt{c2})$ (the second disjunct of D) holds. It follows that $r = \mathtt{r(c2)}$ is a receipt of $\mathtt{c2}$ and, by constr1, the unique receipt of $\mathtt{c2}$. The second information term of the snapshot states that the domain of the $receipt(r, c)$ generator only contains $(\mathtt{r(c2)}, \mathtt{c2})$ and that the total of $\mathtt{r(c2)}$ is 12 (see the corresponding I-answer set in row 2 of Fig. 3). Finally, the third information term establishes the items and their prices (row 3 of Fig. 3), whereas the fourth only fixes the domain of the $itemOf(i, r)$ generator (row 4 of Fig. 3). We remark that the constraints have $\mathtt{tt}$ as unique information term, since we only require that they are classically true, but we are not interested in a "constructive" explanation of them.

2.3 Snapshots generation modulo theories

In the previous examples we have shown that formulas of a CooML specification play different roles. For instance, in Example 1 classax1-3 and assocax1 generate the relevant part of the snapshot (i.e, the information terms different from $\mathtt{tt}$),

1) $cashReg(\texttt{c1})$, $cashReg(\texttt{c2})$, $T(\forall c.\, cashReg(c) \rightarrow c = \texttt{c1} \vee c = \texttt{c2})$,
   $T(\neg\exists r.\, receipt(r, \texttt{c1}))$, $receipt(\texttt{r(c2)}, \texttt{c2})$,
2) $receipt(\texttt{r(c2)}, \texttt{c2})$, $T(\forall r, c.\, receipt(r, c) \rightarrow (r, c) = (\texttt{r(c2)}, \texttt{c2}))$, $12 = total(\texttt{r(c2)})$,
3) $item(\texttt{it1})$, $item(\texttt{it2})$, $T(\forall i.\, item(i) \rightarrow i = \texttt{it1} \vee i = \texttt{it2})$, $5 = price(\texttt{it1})$, $7 = price(\texttt{it2})$,
4) $itemOf(\texttt{it1}, \texttt{r(c2)})$, $itemOf(\texttt{it2}, \texttt{r(c2)})$,
   $T(\forall i, r.\, itemOf(i, r) \rightarrow (i, r) = (\texttt{it1}, \texttt{r(c2)}) \vee (i, r) = (\texttt{it2}, \texttt{r(c2)}))$
5) constr1, constr2, constr3, constr4, constr5.

Fig. 3. The I-answer set IANS($\underline{\tau} : CashSpec$)

whereas the constraints are needed to cut-off undesired models. Moreover, we have left understood the formalization of the external $float$ theory. To better point out the meaning of the sentences in a Σ-specification $Spec$, we distinguish *internal* and *external symbols*. The meaning of the former is defined by $Spec$ and can rely on an external theory. To point out this dependency, we indicate the signature by $\Sigma(\Pi)$, where Π is the sub-signature of the external symbols, and we use the notation $Spec = \langle \mathcal{SN} \cup \mathcal{K} \cup \mathcal{P} \rangle$, where:

- $\mathcal{SN}$ is a finite set of universally bounded quantified sentences, we call the *snapshot axioms*.
- $\mathcal{K}$ is a (possibly infinite) set of T-sentences representing the *constraints*.
- $\mathcal{P}$ is a (possibly infinite) set of T-sentences over the signature Π (*external axioms*), axiomatizing the external symbols.

We remark that a snapshot $\underline{\tau} : Spec$ is completely determined by the information terms associated with the sentences in $\mathcal{SN}$, since $\mathcal{K} \cup \mathcal{P}$ only contains T-sentences with information term `tt`. Thus, we can identify a snapshot for $Spec$ with an information term $\underline{\tau} : \mathcal{SN}$. The axioms of $\mathcal{P}$ characterize a class of Π-interpretations corresponding to the intended meaning of the external symbols. The Π-interpretations that satisfy $\mathcal{P}$ will be called pre-interpretations, since they are fixed externally.

Example 3. Let $CashSpec'$ be the CooML specification obtained obtained by replacing in $CashSpec$ the axiom classax1 with:

$$\text{classax1'}: \ \forall c.\, cashReg(c) \rightarrow (\exists p.\, p = cashier(c)) \wedge (T(\neg\exists r.\, receipt(r, c)) \vee \exists r.\, receipt(r, c)))$$

where the function $cashier(c : cashReg) : person$ depends on the external type $person$. The external symbols are those for $float$, having a fixed pre-defined meaning, and $person$, which is generic. A possible information term τ_1' : classax1' is

$$[\,[\,\texttt{c1},\, [\,[\texttt{john}, \texttt{tt}],\, [1, \texttt{tt}]]\,]\,,\ [\,\texttt{c2},\, [\,[\texttt{mary}, \texttt{tt}],\, [2, [\texttt{r(c2)}, \texttt{tt}]]]\,]\,]$$

Let us consider the snapshot $\underline{\tau}' : CashSpec'$ obtained by replacing τ_1 with τ_1' in the information term $\underline{\tau} : CashSpec$ of Example 2. Then, we get IANS($\underline{\tau}' : CashSpec'$) by adding to Fig. 3 the formulas $\texttt{john} = cashier(\texttt{c1})$ and $\texttt{mary} = cashier(\texttt{c2})$. Since $person$ is generic, we consider as possible all the pre-interpretations where $person$ is a finite set containing `john` and `mary`.

The classical consistency of a specification can be defined as follows:

Definition 2 (Classically consistent snapshots). *Let $Spec = \langle \mathcal{SN} \cup \mathcal{K} \cup \mathcal{P} \rangle$ be a CooML specification with signature $\Sigma(\Pi)$. A snapshot $\underline{\tau} : \mathcal{SN}$ is (classically) consistent iff there is a $\Sigma(\Pi)$-interpretation i such that $i \models \text{IANS}(\underline{\tau} : \mathcal{SN}) \cup \mathcal{K} \cup \mathcal{P}$.*

Snapshot consistency is related to consistency in classical logic. The relationship with classical logic is shown by the following theorem, where we recall that a $\Sigma(\Pi)$-interpretation i is *reachable* iff for every sort s of $\Sigma(\Pi)$, every element of the domain s^{i} can be denoted by a ground $\Sigma(\Pi)$-term.

Theorem 1 (Relationship with classical logic). *Let i be a reachable $\Sigma(\Pi)$-interpretation. Then, $i \models P$ iff there is $\tau : P$ such that $i \models \text{IANS}(\tau : P)$.*

As we discuss in Section 4, it is useful to allow the user to specify additional constraints, in order to obtain a finite (and usually small) number of snapshots exhibiting some specific features of the model. To this aim, in CooML we have further constraints $\mathcal{H}$, that we call *choice axioms*.

Definition 3 (Generation Problems). *Let $Spec = \langle \mathcal{SN} \cup \mathcal{K} \cup \mathcal{P} \rangle$ be a specification. A generation problem for Spec with choice axioms $\mathcal{H}$ is the specification $Spec(\mathcal{H}) = \langle \mathcal{SN} \cup (\mathcal{K} \cup \mathcal{H}) \cup \mathcal{P} \rangle$. A solution of $Spec(\mathcal{H})$ is a consistent snapshot of it.*

We may have different forms of choice axioms. In our DLV implementation:

- A choice axiom for a generator predicate $G(\underline{x})$ has the form $T(\forall \underline{x}.G(\underline{x}) \rightarrow a\_G(\underline{x}))$, where $a\_G(\underline{x})$ is a new predicate symbol, called the *domain predicate* for G. If the user specifies $a\_G(\underline{x})$ as a finite set D, the domain of $G(\underline{x})$ is constrained to be a subset of D.
- A choice axiom for a simple formula $S(\underline{x}.\underline{y})$, with $\underline{y}$ occurring in the scope of an existential quantifier, is of the form $T(\forall \underline{x}, \underline{y}.S(\underline{x}, \underline{y}) \rightarrow elg(\underline{y}, S(\underline{x}, \underline{y}))$ (elg stands for "eligible"). By $elg(\underline{y}, S(\underline{x}, \underline{y}))$ the user can specify, for every $\underline{\mathbf{x}}$, a finite set of values of $\underline{y}$, to be considered as the unique possible witnesses for $\exists \underline{y}.\ S(\underline{\mathbf{x}}, \underline{y})$.

The above choice axioms guarantee that there is a finite set of solutions. Furthermore, we may be interested in the *minimal solutions*, where a solution $\tau : Spec(\mathcal{H})$ is minimal iff there is no solution $\tau' : Spec(\mathcal{H})$ such that $\text{IANS}(\tau' : Spec(\mathcal{H})) \subset \text{IANS}(\tau : Spec(\mathcal{H}))$.

Example 4. The snapshot in Example 2 is a minimal solution of the generation problem for the specification $CashSpec$ of Example 1 with the following choice axioms $(c1),\ldots,(c5)$, domain axioms $(d1),\ldots,(d3)$ and elg-axioms $(d4),(d5)$:

$$
\begin{array}{ll}
c1 & T(\forall c.\ cashReg(c) \ \rightarrow\ a\_cashReg(c)) \\
c2 & T(\forall r, c.\ receipt(r,c) \ \rightarrow\ a\_receipt(r,c)) \\
c3 & T(\forall i.\ item(i) \ \rightarrow\ a\_item(i)) \\
c4 & T(\forall p, i.\ p = price(i) \ \rightarrow\ elg(p, p = price(i))) \\
c5 & T(\forall t, r.\ t = total(r) \ \rightarrow\ elg(t, t = total(r))) \\
d1 & T(\forall c.\ a\_cashReg(c) \ \leftrightarrow\ c \in [\mathtt{c1}, \mathtt{c2}]) \\
d2 & T(\forall r, c.\ a\_receipt(r,c) \ \leftrightarrow\ a\_cashReg(c) \wedge r = \mathtt{r}(c)) \\
d3 & T(\forall i.\ a\_item(i) \ \leftrightarrow\ i \in [\mathtt{it1}, \mathtt{it2}]) \\
d4 & T(\forall p, i.\ elg(p, p = price(i)) \ \leftrightarrow\ item(i) \wedge p \in [5, 7, 20]) \\
d5 & T(\forall t, r, c.\ elg(t, t = total(r)) \ \leftrightarrow\ receipt(r,c) \wedge t \in [12])
\end{array}
$$

By replacing $(d1),\ldots,(d5)$, we change the possible choices of possible populations. For example, to obtain snapshots with different combinations of prices and totals we have only to modify $(d4)$ and $(d5)$.

3 Solving generation problems with DLV

In this section we briefly describe our DLV implementation of CooML snapshot generation. The overall process is illustrated in Fig. 4 The DLV translation of

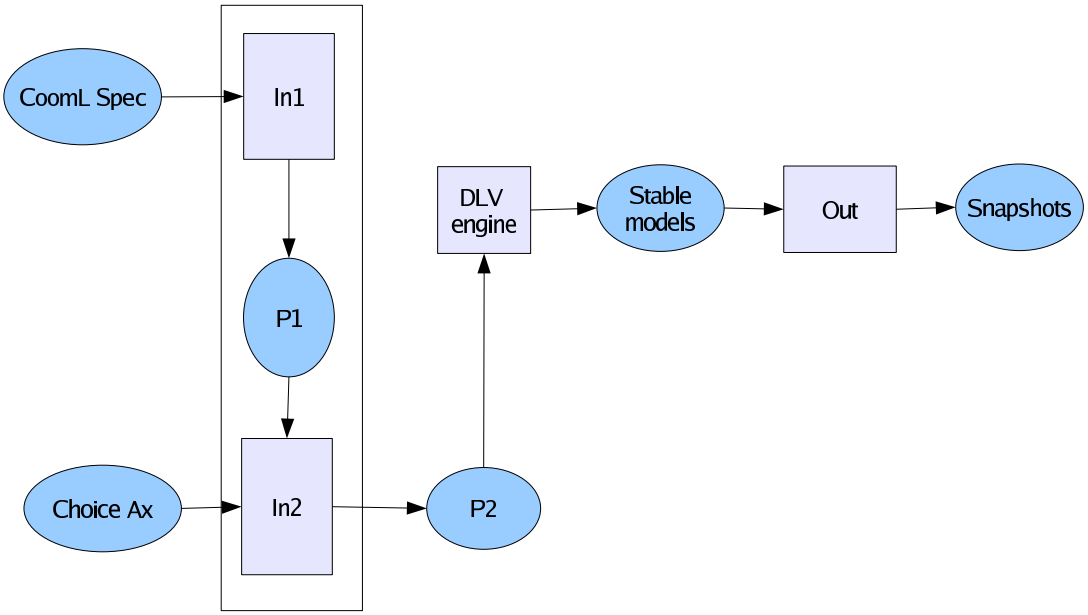

Fig. 4. DLV Implementation of Snapshot Generation

a snapshot generation problem is accomplished in two steps In1 and In2. The detailed exposition is rather cumbersome and here we give only an idea.

In step In1, the CooML specification *Spec* is translated into a DLV program P_1, by the following transformations.

- Application of the Morgan rules, introduction of new atoms as abbreviations of simple formulas, elimination of the existentials by introducing suitable *elg*-atoms, in order to translate CooML class properties into DLV clauses.
- Translation of the CooML constraints into DLV constraints.

Example 5. Let us consider the specification $CashSpec'$ of Example 3. Some clauses of the corresponding translation are shown below. The `cax1` clauses translate `classax1'` and the `c1` constraint corresponds to `constr1`.

```
%cax1.
  cEmpty(C,P) v cReceipt(C,P,R) :-
        cashReg(C), elg_cashier(C,P), a_receipt(R,C).
  empty(C) :-
        cEmpty(C,P).
  cashier(C,P) :-
        cReceipt(C,P,R).
....
%c1
:- receipt(R1,C), receipt(R2,C), R1 != R2.
....
```

In general, the program P_1 has infinitely many models, but a unique empty stable model (or no stable model at all). To generate non-empty snapshots, we have to impose that the population of live objects is not empty and that *elg*-atoms are satisfied by the suitable witness values. This is obtained by the translation In2 of the choice and domain axioms and *elg*-axioms, giving rise to the DLV program P_2.

Example 6. Let us consider the choice and domain axioms in Example 4. They are implemented in DLV by suitable *guess* axioms (as defined in [14]). For instance, the following domain axioms for `item` guess 4 domains of `item` (`[]`, `[it1]`, `[it2]`, `[it1,it2]`), while the elg-axioms select one of the prices 5 and 7.

```
%%  domain axioms
a_item(it1).
a_item(it2).
item(I) v -item(I) :- a_item(I).
....
%%%%  elg-axioms
elg_price(I,5) v elg_price(I,7):- item(I).
..........
```

One can prove that:

Theorem 2. *Let Spec be a CooML specification and P_2 be the DLV program obtained by translating Spec. Then:*

– *For every snapshot τ : Spec there is a stable model M of P_2 such that* IANS$(\tau : Spec) \subseteq M$, *and*

– *for every model M of P_2 there is a snapshot $\tau : Spec$ such that*
$\text{IANS}(\tau : Spec) \subseteq M$.

As a consequence, from stable models of program P_2 one can reconstruct the solution of the snapshot generation problem given as input. The last transformation is performed by the `Out` translation.

4 Snapshot generation for model validation

In this section we discuss snapshot generation for model validation and the use of answer set programming [3]in this context. The discussion use the previous cash register example and, in addition, the example of a circular queue, as defined by the following Java classes.

```
class Queue{
   Element first = null;
   Element last = null;
   public void insert(Elem e){
      last.next = e;
      e.next = first;
      first = e;}
}
```

```
class Elem{
   Content content;
   Element next;
}
```

The insert method contains a mistake, since it does not consider the null case. The above Java classes have the following CooML representation where the class axioms correspond to the Java classes and the constraints impose the structure of circular queues. We omit the signature declaration for conciseness; we only remark that the function symbols $first(q), last(q)$ and $next(e, q)$ have type $elem(q)$. Furthermore, we do not model the attribute `content`.

$$\forall q.\, queue(q) \;\rightarrow\; T(\neg\exists e.\, elem(e,q)) \;\vee\; \exists f, l.\, f = first(q) \wedge l = last(q) \wedge f = next(l,q)$$
$$\forall e, q.\, elem(e,q) \;\rightarrow\; \exists c, e'.\, c = content(e,q) \wedge e' = next(e,q)$$
$$T(\forall e, q.\, elem(e,q) \;\rightarrow\; \exists! e'.e = next(e', q))$$

Starting from the above examples, we illustrate how we can carry out the validation experiments considered in the introduction. An experiment tests the formal model with respect to the informal requirements expressing the user's expectation. Each experiment is obtained by choosing the domain predicates in a suitable way.

a) Check that the expected populations are generated. In this case, we choose the set $\mathcal{H}$ of domain and elg predicates in such a way that, according to the informal requirements, there is a finite and small number of snapshots. Eventually, we know their information content. Then we generate all the snapshots that satisfy $\mathcal{H}$ and we compare them with the expected ones. If the expected snapshots are generated exactly, the experiment is successful. Otherwise we may get no snapshot (a kind of inconsistency result), fewer snapshots, indicating that our constraints are too strong, or unexpected snapshots, indicating that our model is wrong.

Example 7. Let P be the DLV program obtained by translating the $CashSpec'$ specification of Ex. 3 with the choice axioms $\mathcal{H}$ of Ex. 6. Examples of stable models of P are:

```
1)  cashReg(c1), cashier(c1,john), empty(c1).
2)  cashReg(c1), cashier(c1,john), receipt(r_c1,c1), total(r_c1, 12),
    itemOf(it1,r_c1), itemOf(it2,r_c1), price(it1,7), price(it2,5).
   ....
```

They correspond to realistic snapshots for the $CashSpec'$ specification with choice axioms $\mathcal{H}$. Now, let $CashSpec'_1$ be the specification obtained by deleting constr5 (corresponding to the OCL constraint in Fig. 1) from $CashSpec'$ and let P_1 be the DLV program obtained by translating $CashSpec'_1$ with choice axioms $\mathcal{H}$. Then, among the stable models of P_1 computed by DLV we find:

```
cashReg(c1), cashier(c1,john), receipt(r_c1,c1), total(r_c1, 12),
itemOf(it1,r_c1), price(it1,5).
```

Clearly, this does not correspond to a feasible snapshot, since the receipt `r_c1` has only an item `it1` with price 5, whereas the total of `r_c1` is 12. By inspecting the solution, one realizes that $CashSpec'_1$ is too loose and the constraint constr5 must be added. A similar discussion applies if any of the other constraints is dropped or if we have constraints cutting expected solutions.

b) Check that unexpected snapshots are not generated. The domain and *gen*-predicates are chosen as in a). Furthermore, we add as new constraint a property that, according to the informal requirements, should be excluded. If no snapshot is generated, then the experiment succeeds, otherwise it fails. Failure exhibits counterexamples, which are admitted by the formal model but should be excluded. This approach can be considered as a kind of "partial model checking". Indeed, the fact that no snapshot is generated entails inconsistency with respect to $Spec_{ch}$, which does not necessarily entail the inconsistency of $Spec$.

Example 8. Each element of a queue should be reachable starting from the first one. To check that there are no unreachable elements, we assume by absurd that there exists one. Our experiment considers the queues with at most 2 elements, and we impose that they are not linked. The domain and elg axioms are

$T(\forall x.\ a\_queue(x) \ \leftrightarrow\ member(x, [\mathtt{q}]))$
$T(\forall e, q.\ a\_elem(e, q), \leftrightarrow\ a\_queue(q) \wedge member(e, [\mathtt{e1}, \mathtt{e2}]))$
$T(\forall n, e, q.\ elg(n, n = next(e, q)) \ \leftrightarrow\ a\_elem(e, q) \wedge a\_elem(n, q))$
$T(\neg \mathtt{e1} = next(\mathtt{e2}) \wedge \neg \mathtt{e2} = next(\mathtt{e1}))$

We get a snapshot where `e1=next(e1,q)` is the first and last element of `q`, while the element `e2=next(e2,q)` is isolated. Thus, we discover that we have to impose reachability as a new constraint.

c) Checking Methods Specifications and generating Test Cases. In OO models, specifications are typically modeled through contracts, in the pre-post condition style: the precondition Pre must be satisfied at call time, while the post condition relates the call-time and exit-time states. Thus we have to generate pair of snapshots $\langle \tau : Spec, \tau' : Spec \rangle$ such that

- $\tau : Spec$, representing the call state, is consistent in $Spec(Prec)$, i.e., $Spec$ with additional constraints $Prec$, and
- $\tau' : Spec$, representing the exit state, is consistent in $Spec$ and the post condition is satisfied by $\langle \tau : Spec, \tau' : Spec \rangle$.

To represent constraints on pairs $\langle \tau : Spec, \tau' : Spec \rangle$, we introduce an extra-argument of type *time* for predicates that may be updated by methods, where $time = \{pre, now\}$ (intuitively, pre is the call time of the specified method, while now is the exit time). At the CooML specification level, we formalize states by the meta-predicate $holds(t, S)$, indicating that the simple formula S holds at state (or "time") t. Using $holds$, the fact that non-assigned atoms are preserved (also known as *inertia principle* [7]) is enforced by the axiom schema:

$$holds(now, A) \texttt{ :- } holds(pre, A),\ \neg assignable(A).$$

Methods are represented with assignable simple formulas and by specifying their effect as shown in the following example. Contracts are represented in a similar way.

Example 9. Here consider the `insert` method of the class `Queue`. The predicate $insert(Q, E)$ means that the call `insert(E)` has been performed by Q. For conciseness, we give only some EDF axioms. The CooML axiom a corresponds to the fact that `last.next` is assigned, while h corresponds to the first assignment of the `insert` method.

$a.$ $assignable(next(Last, E, Q))$:- $queue(Q)$, $holds(pre, last(Last, Q))$.
$h.$ $holds(now, next(Last, E, Q))$:- $holds(pre, last(Last, Q))$, $insert(Q, E)$.

The DLV translation contains, for example, the clauses:

```
i1.   next(now, E1, E2, Q) :-  next(pre, E1, E2, Q), not last(pre, E1, Q).
i2.   elem(now, E, Q) :- elem(pre, E, Q).
...
h.    next(now, Last, E, Q) :- last(pre, Last, Q), insert(Q, E).
...
```

where `i1`, `i2` are examples of the inertia principle, while `h` comes from h. Now we rewrite the DLV translation of the queue specification considered in Example 8, just by adding the extra-argument pre. The result of snapshot generation on the DLV program corresponding to the above method specification and on the CooML queue specification contains the empty queue both at time `pre` and at time `now`; this shows that the case of the empty queue was not considered.

d) Generating Test Cases. Here, we give only some hints, since this part has not been studied yet. Once we have validated the specification of a method m, we can use a similar approach to generate test cases and test oracles for an implementation of m, say in Java. We need a representation method, to build Java states from snapshots. Essentially, this method accepts information terms as input parameters and creates the corresponding Java objects. The test cases are obtained by calling the representation method with the snapshots for the

generated *pre*-states. The oracle is implemented by comparing the Java states obtained after the execution of m with the *now*-states obtained from the method specification. We believe also that our snapshot generation algorithm could be employed in Bounded Exhaustive Testing [16]. We plan to experiment our system with meaningful examples and to compare it with the existing tools, such as Alloy (`http://alloy.mit.edu/`) and TestEra [16].

5 Related work and conclusion

We have used the modeling language CooML and we have shown how OO models, their snapshots and the related information content can be specified. Then we have presented a DLV implementation of CooML Snapshot Generation (SG) and we have discussed its potential application in model validation.

The relevance of SG for validation and testing in OO software development is widely known. The USE tool [12] for validation of UML+OCL models has been recently extended with a SG mechanism; differently from us, this is achieved via a procedural language. Other animation tools include [5] w.r.t. JML specification. In [4] the specification of features models are translated into SAT problems; tentative solutions are then propagated with a Truth Maintenance System. Related work is also [19], where design space specs are seen as trees whose nodes are constrained by OCL statements and BDD's used to find solutions, and [15], proposing Armstrong data bases [2] as a way of extracting small examples from existing data bases, to show their functional and inclusion dependencies.

As far as testing is concerned, [1] studies Z and B specifications. In [8] the authors propose a testing method extracting test case and oracles from formal correctness proofs. One of the advantages is that one can test both code and specifications and that a complete specification is not needed. Our approach has some similarities, since we can deal with incomplete specifications and we can use the T operator to hide details. Further, we can test code by testing the effect of method calls against contracts, but we cannot test the internal parts of a method unless in simple cases, such the one considered in Example 9.

Our implementation is still at an initial stage, and we have considered only a limited set of small examples. Nevertheless, we believe that SG is a promising approach, the more once the following points are addressed.

- We have to complete the DLV implementation. So far, we have implemented the translation from CooML into DLV, yet constraints are to be translated in the DLV format manually. We have to implement the map reconstructing snapshots from DLV models.
- We will study the possibility of integrating our DLV snapshot generator and the DLV front-end for planning [7]. The idea is to apply AI planning, in a way similar to the one proposed in [13].

– We will develop tools supporting the interoperability with UML + OCL and OO languages. This will allow us to apply our approach with known meaningful examples and compare it to other architectures.
– From a more theoretical point of view, it is interesting to compare the CooML snapshot semantics with the stable model semantics [10]. We argue that, under suitable minimality requirements, they are related and information terms could be proposed as a tool for the justification and debugging of answer set programs [21]. Some preliminary results are presented in [11].

References

1. F. Ambert and et al. BZ-TT: A Tool-Set for Test Generation from Z and B using Constraint Logic Programming. In H. Hierons and T. Jerron, editors, *FATES 2002*, pages 105–120, 2002.
2. W. Armstrong and C. Delobel. Decompositions and functional dependencies in relations. *ACM Transactions on Database Systems*, 5(4):404–430, 1980.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2003.
4. D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
5. F. Bouquet, F. Dadeau, B. Legeard, and M. Utting. JML-testing-tools: A symbolic animator for JML specifications using CLP. In N. Halbwachs and L. D. Zuck, editors, *TACAS*, volume 3440 of *LNCS*, pages 551–556. Springer, 2005.
6. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CCC '97: Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, page 82, Washington, DC, USA, 1997. IEEE Computer Society.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In *Computational logic—CL 2000 (London)*, volume 1861 of *Lecture Notes in Comput. Sci.*, pages 807–821. Springer, Berlin, 2000.
8. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In *TAP*, pages 169–188, 2007.
9. M. Ferrari, C. Fiorentini, A. Momigliano, and M. Ornaghi. Snapshot generation in a constructive object-oriented modeling language. In *LOPSTR'07 Proceedings*, pages 145–159, 2007.
10. P. Ferraris, J. Lee, and V. Lifschitz. A new perspective on stable models. In *IJCAI*, pages 372–379, 2007.
11. C. Fiorentini and M. Ornagh. Answer Set Semantics vs. Information Term Semantics. In S. Costantini and R. Watson, editors, *Proceedings of the 4th Workshop on Answer Set Programming: Advances in Theory and Implementation*, pages 241–253, 2007.
12. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
13. A. E. Howe, A. von Mayrhauser, and R. T. Mraz. Test case generation as an AI planning problem. *Autom. Softw. Eng.*, 4(1):77–106, 1997.
14. N. Leone and et al. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.
15. F. D. Marchi and J.-M. Petit. Semantic sampling of existing databases through informative Armstrong databases. *Information Systems*, 32:446–457, 2007.
16. D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *ASE*, pages 22–34, 2001.
17. J. Medvedev. Finite problems. *Soviet Mathematics Doklady*, 3:227–230, 1962.
18. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
19. S. Neema and et al. Constraint-based design-space exploration and model synthesis. In R. Alur and et al, editors, *EMSOFT*, volume 2855 of *LNCS*, pages 290–305. Springer, 2003.
20. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.

21. E. Pontelli and T. C. Son. Justifications for logic programs under answer set semantics. In S. Etalle and M. Truszczynski, editors, *ICLP*, volume 4079 of *LNCS*, pages 196–210. Springer, 2006.
22. A. Troelstra. Aspects of constructive mathematics. In J. Barwise, editor, *Handbook of Mathematical Logic*. North-Holland, 1977.

Verification-based Test Case Generation with Loop Invariants and Method Specifications

Christoph Gladisch*

University of Koblenz-Landau
Department of Computer Science
Germany

Abstract. The goal of this work is to improve the testing of programs that contain loops and complex methods. We achieve this goal with verification-based testing, which is a technique that can generate test cases not only from source code but also from loop invariants and method specifications provided by the user. These test cases ensure the execution of interesting program paths that are likely to be missed by existing testing techniques that are based on symbolic program execution. These techniques would require an exhaustive inspection of all execution paths, which is hard to achieve in presence of complex methods and impossible if loops are involved. Verification-based testing takes a different approach.

1 Introduction

The goal of the presented approach is to improve existing software testing techniques that use symbolic program execution for test case generation and constraint solving for computing concrete test data. The improvement is the generation of test cases that are likely to be missed by the existing testing techniques. These would require an exhaustive inspection of all execution paths which is infeasible in the presence of complex methods and impossible in the presence of loops because loops represent infinitely many paths.

An example is given in listing 1.1 of Figure 1. In order to execute `A()` the loop body has to be executed at least 10 times and in order to execute `C()` it has to be executed exactly 20 times. In similar programs these numbers could be much larger or be the result of complex expressions requiring an exhaustive inspection of all paths in order to find the case where the branch conditions are satisfied. The situation is similar in listing 1.2 where an exhaustive inspection of `D()` may be required in order to find a path such that after the execution of `D()` the branch condition $i \doteq 20$ holds. Since exhaustive symbolic execution is not possible existing testing techniques are likely to miss these cases because they have a bound on the amount of inspected execution paths. For loops and recursive methods the typical approach is to symbolically execute the first k loop iterations or recursion steps, called k-bounded unwinding, where k is a limiting constant.

* gladisch@uni-koblenz.de

JAVA (1.1)

```
void foo(int n){
 int i=0;
 while(i < n){
  if(i==10){ A();}
  B(); i++;
 }
 if(i==20){ C(); }
}
```

JAVA

JAVA + JML (1.2)

```
/*@ requires i<n;
  @ assignable i;
  @ ensures i==n; */
void D(int n){while(i<n)...}
void foo(int n){
 D(n);
 if(i==20){ C(); }
}
```

JAVA + JML

Fig. 1. Motivating examples

1.1 Verification-based Testing

Our solution, which is an extension of *verification-based testing* [9], takes a different approach. It incorporates verification technology into test case generation by replacing the symbolic execution of a complex method or loop by the application of a method specification or loop invariant rule. The purpose of both approaches is the same, namely to compute a precondition for a given branch condition (the condition that has to be fulfilled later on for that branch to be taken): the *branch precondition*.

The challenge in test case generation for programs with loops and complex methods like in Figure 1 is to compute the precondition of the loop or the method for a given *branch condition* which occurs within the loop (e.g. `i==10`) or after the loop or method (e.g. `i==20`). The precondition is important because it describes test data for establishing a program state *before* the loop or method that guarantees the execution of program branch or path. If the precondition is known and represented as a first-order logic formula, then the desired test data can be computed using a constraint solver.

For example the desired preconditions of the loop in listing 1.1 that are computed with our approach by using the loop invariant $i^{old} \leqslant i \wedge i \leqslant n$ are: $i \leqslant 10 \wedge 10 < n$ to execute `A()` and $i \leqslant n \wedge 20 = n$ to execute `C()` , where i^{old} refers to the value of i before the loop. The loop is replaced in listing 1.2 with the method `D()` which has the specification: if $i < n$ is satisfied before the execution of `D()`, then $i \doteq n$ is satisfied after its execution. Instead of searching for a program path in `D()` with a path-condition that ensures the execution of `C()` we use the given specification of `D()`. Our approach yields in this case the desired constraint $i \leqslant n \wedge 20 = n$. These preconditions are not generated by symbolic execution if we choose for instance the bound $k = 3$.

Precondition computation based on specifications or loop invariants is applicable in situations where the inspection of all execution paths with symbolic execution is infeasible or impossible. Computing the precondition for a given branch condition based on a loop invariant resp. a method specification are

however essentially the same problem. The loop invariant is a pre- and postcondition of the loop's body and of the loop itself. We will therefore refer to the loop invariant or method specification just by specification. The presented technique is similar to the well known verification technique called weakest precondition computation except that for the purpose of test data generation the constraint solver does not require weak but rather *strong* constraints.

Typical use-cases of our approach are those where specifications of methods and loops are provided. This is for instance the case in specification driven software development methodologies (e.g. the B method [6] , SOCOS [2]) or in test cases generation based on verification proofs, e.g. [9].

1.2 Plan of the Paper

In the next section we describe the logic and the calculus on which we build our approach and give some definitions. The approach can however easily be adapted to other formalisms. The main part is Section 3 where we introduce two formulae that are built from a specification and a branch condition: the *disjunctive branch precondition* (DBPC) and the *conjunctive branch precondition* (CBPC). The CBPC is the desired precondition (constraint) for test data generation using constraint solving as described in Section 1.1. The DBPC is needed to show how the CBPC can be used depending on the properties of the involved specification. In Section 4 we give examples of how to generate the desired preconditions of methods and loops for a given branch condition by constructing the CBPC. Finally in Section 5 we describe how this contribution relates to existing work and we draw conclusions in Section 6.

2 Dynamic Logic and the Verification Calculus

2.1 Overview of Java Card DL and KeY's Sequent Calculus

The work presented here is based a Dynamic Logic [10] for Java Card(Java Card DL [3]) but it can be adapted to similar logics like Hoare Logic and for different programming languages. Java Card DL is the program logic of the KeY-System [5, 1], which is a combined interactive and automatic verification and test generation system with symbolic execution rules for a subset of Java.

Dynamic Logic is an extension of first-order logic where a formula φ can be *prepended* by the modal operators $\langle \mathtt{p} \rangle$ and $[\mathtt{p}]$ for every program $\mathtt{p}$. The formula $[\mathtt{p}]\varphi$ means that if $\mathtt{p}$ terminates, then φ holds in the state after the execution of $\mathtt{p}$. The formula $\langle \mathtt{p} \rangle \varphi$ means that after all possible executions of $\mathtt{p}$ the formula φ is true but since we consider only sequential and deterministic Java programs the meaning of $\langle \mathtt{p} \rangle \varphi$ is that the program terminates and that $[\mathtt{p}]\varphi$ is true. Thus $[\mathtt{p}]\varphi \wedge \langle \mathtt{p} \rangle \mathit{true}$ is equivalent to $\langle \mathtt{p} \rangle \varphi$. In the following we denote the set of programs

by π, the set of DL-formulae by Fml, and the set of first-order logic formulae by Fml_{FOL}. All variables in formulae are bound with quantifiers.

An implication of the form $pre \rightarrow [\mathtt{p}]post \in Fml$ with $pre, post \in Fml_{FOL}$ corresponds to the Hoare triple $\{pre\}\mathtt{p}\{post\}$ in Hoare logic. If the precondition pre is true in the state before the execution of the program and the program terminates, then the postcondition $post$ holds after the execution of the program; if the precondition does not hold before the execution of the program, then no statement is made about the post-state. The implication $pre \rightarrow \langle\mathtt{p}\rangle post$ states additionally that `p` terminates. Dynamic logic allows pre and $post$ to *contain* programs in contrast to Hoare logic: if $pre, post \in Fml$ then $pre \rightarrow [\mathtt{p}]post \in Fml$.

Program variables are modelled in JAVA CARD DL as *non-rigid* function symbols $f \in \Sigma_{nr} \subset \Sigma$ of the signature Σ. Different program states are therefore realized as different first-order interpretations of the non-rigid function symbols. For instance let $a, o, i, acc_{[]} \in \Sigma_{nr}$. In this case a program variable `a` is represented by a logical non-rigid constant a, an expression like `o.a`, that accesses an object attribute, is modeled as the term $a(o)$, and in case of an array access `o.a[i]` the corresponding term is $acc_{[]}(a(o), i)$. We use constant domain semantics which means that in all states terms are evaluated to values of the same universe. In contrast to interpretations a variable assignment β cannot be modified by a program so that logical variables are always *rigid*, i.e., they have the same value in all program states.

To express that a formula φ is true in a state $s \in S$ (S is the set of all states) under a variables assignment β we write $s, \beta \vDash \varphi$. Furthermore $s \vDash \varphi$ means that for all variable assignments β: $s, \beta \vDash \varphi$; and $\vDash \varphi$ means that φ is valid, i.e., for all $s \in S$ and all variable assignments β the statement $s, \beta \vDash \varphi$ holds. Non-standard, but important in this paper, is the case where $s_{\Sigma\setminus A}$ is only a partial interpretation $s_{\Sigma\setminus A} \in S_{\Sigma\setminus A}$, i.e., it gives a meaning to symbols from some subset $(\Sigma\setminus A) \subset \Sigma$ of the signature Σ. In this case each partial interpretation $s_A \in S_A$ of the unspecified symbols $A \subset \Sigma$ is combined with the given partial interpretation $s_{\Sigma\setminus A} \in S_{\Sigma\setminus A}$ resulting in a total interpretation $(s_A \cup s_{\Sigma\setminus A}) \in (S_A \cup_\times S_{\Sigma\setminus A}) = S_\Sigma$. Thus $s_{\Sigma\setminus A}, \beta \vDash \varphi$ means that for all $s_A \in S_A$ the statement $(s_A \cup s_{\Sigma\setminus A}), \beta \vDash \varphi$ holds.

In this paper we understand *test data* as a partial state, i.e. a mapping from program variables Σ_{nr} to values.

Definition 1. *Let $\Phi = \phi_1, \ldots, \phi_n$ be a set of formulae and let $\Phi_\wedge$ be the conjunction $\phi_1 \wedge \ldots \wedge \phi_n$. Let γ be a formula, S the set of all states (interpretations), and B the set of all variables assignments, then*

- *γ is a local consequence of Φ, written as $\Phi \vDash_l \gamma$, iff*

$$\begin{array}{c} \textit{for all } \beta \in B\textit{: for all } s \in S\textit{:} \\ s, \beta \vDash \Phi_\wedge \textit{ implies } s, \beta \vDash \gamma \end{array}$$

– *Let $SK \subset \Sigma$ and let $S_{\Sigma\setminus SK}$ be the set of all partial states that are defined only for the symbols $\Sigma\setminus SK$. γ is a semi-local consequence of Φ, or alternatively, γ is a local consequence of Φ modulo the interpretation of $SK \subset \Sigma$, written as $\Phi \models_{SK} \gamma$, iff*

$$\begin{array}{c} \textit{for all } \beta \in B\textit{: for all } s \in S_{\Sigma\setminus SK}\textit{:} \\ s, \beta \models \Phi_\wedge \textit{ implies } s, \beta \models \gamma \end{array}$$

(note that $s, \beta \models \Phi_\wedge$ and $s, b \models \gamma$ quantify locally over the interpretations of the symbols SK).

For example $a \doteq sk \wedge sk \doteq b \models_l a \doteq b$ but $a \doteq sk \rightarrow sk \doteq b \models_{\{sk\}} a \doteq b$. A state $s \in S$ where local consequence fails in the latter case is, e.g., $s = \{a \mapsto 0, sk \mapsto 1, b \mapsto 1\}$.

For the sake of a shorter notation we use the sequent calculus notation. A formula of the form

$$((\gamma_1) \wedge \ldots \wedge (\gamma_k)) \rightarrow ((\delta_1) \vee \ldots \vee (\delta_l))$$

with $\gamma_1, \ldots, \gamma_k, \delta_1, \ldots, \delta_l \in Fml$ is equivalent to the sequent

$$\gamma_1, \ldots, \gamma_k \Longrightarrow \delta_1, \ldots, \delta_l$$

A sequent rule is of the form

$$\frac{\Gamma_1 \Longrightarrow \Delta_1 \;\; \ldots \;\; \Gamma_n \Longrightarrow \Delta_n}{\Gamma_0 \Longrightarrow \Delta_0}$$

and it is locally correct iff $\Gamma_1 \Longrightarrow \Delta_1 \;\; \ldots \;\; \Gamma_n \Longrightarrow \Delta_n \models_l \Gamma_0 \Longrightarrow \Delta_0$ and locally correct modulo $SK \subset \Sigma$ (or semi-locally correct) iff $\Gamma_1 \Longrightarrow \Delta_1 \;\; \ldots \;\; \Gamma_n \Longrightarrow \Delta_n \models_{SK} \Gamma_0 \Longrightarrow \Delta_0$. In sequent calculus, proofs are constructed by applying sequent rules bottom-up, i.e., in order to prove $\Gamma_0 \Longrightarrow \Delta_0$ the new proof obligations $\Gamma_1 \Longrightarrow \Delta_1 \;\; \ldots \;\; \Gamma_n \Longrightarrow \Delta_n$ are generated that have to be proved instead.

2.2 State Updates

In order to write that $f \in \Sigma$ is replaced by $g \in \Sigma$ in $\varphi \in Fml$ we could use the substitution $[g/f]\varphi$. This is a rather technical notion of our intention to express that φ is evaluated in the state after assigning $f(x_1, \ldots, x_n)$ to $g(x_1, \ldots, x_n)$ for all argument values. A more intuitive notation allowing us to refer to a pre-state and a post-state of an assignment is $\{f := g\}\varphi$ which abbreviates a *quantified update* [12]. Updates allow also an extension of our approach, e.g. semantic selection of memory locations, that is, however, not discussed here due to space limitation. JAVA CARD DL extends classical Dynamic Logic [10] with updates [3]. These are assignments between terms (not between JAVA expressions) and are

therefore free of side-effects allowing an efficient way of handling aliasing. An update has the form $\{t_1 := t_2\}$ and means that in the post-state of the update the term t_1 has the value of the term t_2 which is evaluated in the pre-state.

For instance in order to prove $i \doteq 0 \rightarrow \langle \texttt{i++;} \rangle i \doteq 1$ the diamond operator $\langle \texttt{i++;} \rangle$ is replaced with an update yielding $i \doteq 0 \rightarrow \{i := i + 1\} i \doteq 1$. Update application finally gives $i \doteq 0 \rightarrow i + 1 \doteq 1$.

Furthermore the notation $\{A := B\}\varphi$ with $A, B \subset \Sigma$ is equivalent to the substitution $[b_1/a_1, \ldots, b_n/a_n]\varphi$ where the function symbols $a_1, \ldots, a_n \in A$ are replaced by the function symbols $b_1, \ldots, b_n \in B$ respectively.

2.3 Modifier Sets and Anonymous Updates

A modifier set for a program is a set of function symbols that model program variables. The purpose of using a modifier set as part of a specification is to specify which program variables are modified by the program.

Definition 2. *The* minimal *modifier set of a program* p *is denoted by* $Mod(\texttt{p}) \subset \Sigma$ *and it consists exactly of those function symbols that can be modified by* p*.*

A correct *modifier set* $M \subset \Sigma_{nr}$ *contains* at least *the function symbols that are modifiable by* p*, i.e.* $M \supseteq Mod(\texttt{p})$*.*

A modifier set $M \subset \Sigma_{nr}$ can be used to create an *anonymous update* of the form $\{M := M_{sk}\}$ which replaces each function symbol $f \in M$ by a fresh function symbol $f_{sk} \in M_{sk}$. Anonymous updates enable us to transform a post-condition into a precondition preserving only the common information of the pre-state and the post-state. For instance the formula $\langle \texttt{o.a=x;} \rangle (o.a \doteq c \wedge o.a \doteq d)$ specifies a pre-state such that after the execution of `o.a=x` the postcondition $o.a \doteq c \wedge o.a \doteq d$ holds. The anonymous update $\{a := a_{sk}\}$ replaces the modal operator $\langle \texttt{o.a=x;} \rangle$ resulting in the formula $\{a := a_{sk}\}(o.a = c \wedge o.a \doteq d)$ which can be reduced to $o.a_{sk} = c \wedge o.a_{sk} \doteq d$. Thus we have gained the additional information about the pre-state that $c \doteq d$.

2.4 Specifications

Definition 3. *A specification is a triple* $(pre, post, M)$ *where* $pre \in Fml_{FOL}$ *is the precondition,* $post \in Fml_{FOL}$ *is the postcondition and* $M \in \Sigma$ *is a modifier set.*

A specification typically describes the behavior of a method but it can specify the behavior of any statement or sequence of statements. For instance a loop invariant $I \in Fml$ is the pre- and postcondition of a loop's body and the loop itself. A stronger postcondition of the loop is $I \wedge \neg lc$ where $lc \in Fml$ is the loop condition, i.e. the loop iterates while lc is true. The specification of a loop is

therefore the triple $(I, I \wedge \neg lc, M)$ and the specification of a loop body before loop termination is the triple $(I, I \wedge lc, M)$.

In the next section we describe how to compute preconditions from a given specification and a branch condition in the program. The preconditions have different semantic properties depending on the semantic properties of the involved specifications that are described next.

Satisfied Specification An important verification technique is to use a specification in a proof instead of a program which is allowed if the program is correct wrt. the specification. A program p is correct wrt. a specification $(pre, post, M)$ iff $M \supseteq Mod(\mathtt{p})$ and $pre \rightarrow \langle \mathtt{p} \rangle post$ is valid. That a program p is correct wrt. a specification is equivalent to the statement, the specification is *satisfied* by p.

Strong Specification In the following sections we will abbreviate the formula

$$\begin{aligned} &\forall x_1 \ldots \forall x_{m_1}.f_1(x_1, \ldots, x_{m_1}) \doteq f_1^{sk}(x_1, \ldots, x_{m_1}) \\ \wedge \ldots \wedge &\forall x_1 \ldots \forall x_{m_n}.f_n(x_1, \ldots, x_{m_n}) \doteq f_n^{sk}(x_1, \ldots, x_{m_n}) \end{aligned}$$

where $\{f_1, \ldots, f_n\} = M \subset \Sigma$ and $\{f_1^{sk}, \ldots, f_n^{sk}\} = M_{sk} \subset \Sigma$ are disjunct, with the notation

$$M \doteq_\forall M_{sk}$$

The definition below defines a property of a specification that is important when we construct a precondition based on the specification.

Definition 4. *Let* p *be a program and* $\sigma = (pre, post, M)$ *a specification with* $M \supseteq (Mod(\mathtt{p}) \cap \Sigma(pre, post))$*, where* $\Sigma(pre, post)$ *denotes the set of symbols that are in pre or in post. We say that*

- σ *is strong wrt.* p *iff the following formula is valid*

$$(pre \wedge \{M := M_{sk}\}post) \rightarrow \langle \mathtt{p} \rangle M \doteq_\forall M_{sk} \tag{1}$$

- σ *is strong wrt.* p *in state* s *iff* $s \vDash (1)$.

For example the specification $(y \doteq y', y \doteq y' + 1, \{y\})$ is strong for the program `y=y+1` but the specification $(y \doteq y', y' > 0 \rightarrow y \doteq y' + 1, \{y\})$ is weak, e.g., it is not strong in the state $s = \{y \mapsto 0, y' \mapsto 0, y_{sk} \mapsto 7\}$, where y_{sk} is the new symbol introduced by the anonymous update $\{M := M_{sk}\}$. If a specification is strong and $pre = true$, then it is the strongest specification.

3 Branch Preconditions

The goal of the presented approach is to improve existing software testing techniques which use symbolic program execution and constraint solving. As described in the first section the purpose of the symbolic execution is to compute preconditions for a given branch condition from which test data can be computed by using constraint solving.

Our approach is to replace the symbolic execution of a complex method or loop by the computation of a precondition based on a *specification* and a *branch condition* (see Section 1.1). We define two kinds of such preconditions: the *disjunctive branch precondition* (DBPC) and the *conjunctive branch precondition* (CBPC). The DBPC is what many weakest precondition calculi compute in practice but it is not necessarily *the* weakest precondition for a given branch condition. The CBPC is stronger than the DBPC. Depending on the properties of the involved specification the DBPC is suitable to detect infeasible paths or unsatisfiable test data constraints and the CBPC is suitable for test data generation. The CBPC is therefore more important for test case generation but we explain the DBPC first because it is known from verification.

3.1 Disjunctive Branch Precondition

The disjunctive branch precondition is a formula that is suitable for detecting infeasible paths or unsatisfiable test data constraints. It fulfills this purpose however only if it is constructed from a satisfied specification giving rise to the full disjunctive branch precondition as explained below.

Definition 5. *Let $\sigma = (pre, post, M)$, $\varphi \in Fml$ a branch condition, and $M_{sk} \subset \Sigma$ new symbols for the symbols in M.*

The disjunctive branch precondition *(DBPC) for φ is the formula:*

$$pre \rightarrow \{M := M_{sk}\}(post \rightarrow \varphi)$$

Definition 6. *Let σ be a specification of a program $\mathtt{p} \in \pi$, and $\varphi \in Fml$ a branch condition. The* full disjunctive branch precondition *(F-DBPC) is the conjunction of:*

- *the condition that σ is satisfied by* `p` *(see Section 2.4)*
- *the* disjunctive branch precondition *(DBPC) for φ*

For example for the specification $(x' \doteq x \wedge y > 0, x' < x, \{x\})$, the program `x=x+y;` and the branch condition $x \doteq y$ the DBPC is:

$$x' \doteq x \wedge y > 0 \rightarrow \{x := x_{sk}\}(x' < x \rightarrow x \doteq y)$$
$$x' \doteq x \wedge y > 0 \rightarrow (x' < x_{sk} \rightarrow x_{sk} \doteq y) \quad \text{(simplified)} \tag{2}$$

Lemma 1. *Let $M_{sk} \subset \Sigma$ be new symbols for $M \supseteq Mod(\mathtt{p})$ (see Section 2.3). The following rule is locally correct modulo M_{sk}.*

$$\frac{pre \Longrightarrow \langle \mathtt{p} \rangle post \quad pre \Longrightarrow \{M := M_{sk}\}(post \rightarrow \varphi)}{pre \Longrightarrow \langle \mathtt{p} \rangle \varphi} \tag{3}$$

Given a correct specification and branch condition φ the rule says that the DBPC_φ is a precondition of $pre \rightarrow \langle \mathtt{p} \rangle post$. Note that the two premisses of the rule constitute the F-DBPC. The sub-formula $\{M := M_{sk}\}(post \rightarrow \varphi)$ is a precondition of $\langle \mathtt{p} \rangle \varphi$ which is typically the precondition generated by weakest precondition calculi for the purpose of verification. For the purpose of test data generation using a constraint solver it is not suitable because not every model of this formula (solution of the constraint solver) ensures that after executing $\mathtt{p}$ the branch condition φ holds. For instance the model $s = \{x \mapsto 0, x_{sk} \mapsto 0, y \mapsto 2\}$ satisfies the DBPC (2) but not $\langle \mathtt{p} \rangle \varphi$, i.e. $s \not\models \langle \mathtt{p} \rangle \varphi$.

The DBPC is however useful for solving the problem

"no test data exists such that after the execution of $\mathtt{p}$, φ is satisfied"

for identifying infeasible execution paths before the attempt to generate test data. To do so let's assume we want to generate test data which represents a state s such that it satisfies the precondition, i.e. $s \vDash pre$, and $s \vDash \langle \mathtt{p} \rangle \varphi$. We can determine whether $pre \wedge \langle \mathtt{p} \rangle \varphi$ is unsatisfiable by proving $\neg pre \vee [\mathtt{p}] \neg \varphi$. This means, if $(pre, post, M)$ is a specification satisfied by $\mathtt{p}$ and we prove the validity of $pre \rightarrow \{M := M_{sk}\}(post \rightarrow \neg\varphi)$, then we know, without the need to inspect all paths of $\mathtt{p}$ with symbolic execution, that there is no state that satisfies the precondition and $\langle \mathtt{p} \rangle \varphi$.

Note that there may exist states satisfying $\langle \mathtt{p} \rangle \varphi$ but not pre. In this case, however, the specification is useless because we can make no assumption about the postcondition.

In order to prove lemma 1 we need the following rule.

Lemma 2. *The following rule is locally correct modulo the new symbols $M_{sk} \subset \Sigma$ for the symbols $M \supseteq Mod(\mathtt{p})$*

$$\frac{pre \Longrightarrow \{M := M_{sk}\} post}{pre \Longrightarrow \langle \mathtt{p} \rangle post} \tag{4}$$

We omit a detailed proof of rule (4) but its *semi-local correctness* (see Section 2.1) is obvious: if for all possible assignments of values (see Section. 2.3) to the modifiable program variables the postcondition is true, i.e. $\{M := M_{sk}\} post$, then for any program variable assignments in $\mathtt{p}$ the postcondition must be true, i.e. $\langle \mathtt{p} \rangle post$.

We start the proof of lemma 1 with the tautology $pre \rightarrow \langle \mathtt{p} \rangle(((post \wedge (post \rightarrow \varphi))) \rightarrow \varphi)$ and obtain through equivalence transformations the tautology: $(pre \rightarrow (\langle \mathtt{p} \rangle post \wedge \langle \mathtt{p} \rangle(post \rightarrow \varphi)) \rightarrow (pre \rightarrow \langle \mathtt{p} \rangle \varphi)$. This is again equivalent to the local correctness of the rule:

$$\frac{pre \Longrightarrow (\langle \mathtt{p} \rangle post) \wedge \langle \mathtt{p} \rangle(post \rightarrow \varphi)}{pre \Longrightarrow \langle \mathtt{p} \rangle \varphi}$$

The open branches of the following proof tree are the ones of rule (3).

$$\frac{\dfrac{pre \Longrightarrow \langle \mathtt{p} \rangle post \qquad \dfrac{pre \Longrightarrow \{M := M_{sk}\}(post \rightarrow \varphi)}{pre \Longrightarrow \langle \mathtt{p} \rangle(post \rightarrow \varphi)}(\text{using (4)})}{pre \Longrightarrow (\langle \mathtt{p} \rangle post) \wedge \langle \mathtt{p} \rangle(post \rightarrow \varphi)}}{pre \Longrightarrow \langle \mathtt{p} \rangle \varphi}$$

■

3.2 Conjunctive Branch Precondition

The *conjunctive branch precondition* is the precondition of branch conditions in the program that we suggest for test data generation using a constraint solver (see Section 1.1). To save space we define it within the definition of the *full conjunctive branch precondition* which adds a constraint on the involved specification.

Definition 7. *Let $\sigma = (pre, post, M)$ with $M \supseteq (Mod(\mathtt{p}) \cap \Sigma(pre, post))$ be a specification for $\mathtt{p} \in \pi$, where $\Sigma(pre, post)$ denotes the symbols occuring in pre and post. The* full conjunctive branch precondition *(F-CBPC) for a formula φ is the conjunction of:*

- *σ is strong for $\mathtt{p}$: $(pre \wedge \{M := M_{sk}\}post) \rightarrow \langle \mathtt{p} \rangle M \doteq_{\forall} M_{sk}$*
- *the* conjunctive branch precondition *for φ ($CBPC_{\varphi}$):*

$$pre \wedge \{M := M_{sk}\}(post \wedge \varphi)$$

Theorem 1. *Each state satisfying the F-CBPC for φ also satisfies $\langle \mathtt{p} \rangle \varphi$.*

The CBPC_{φ} is a precondition for $\langle \mathtt{p} \rangle \varphi$ that is suitable for test data generation using constraint solvers: (1) if $pre, post$, and φ are first-order logic formulae, then CBPC_{φ} can be trivially simplified to a first-order formula and (2) every model (test data) of the CBPC_{φ} guarantees that after executing $\mathtt{p}$ the condition φ is satisfied if σ is strong in this state. For instance, for $\sigma = (x' \doteq x \wedge y > 0, x' < x, \{x\})$, the program `x=x+y;` and the branch condition $x \doteq y$ the CBPC is:

$$\begin{aligned} &x' \doteq x \wedge y > 0 \wedge \{x := x_{sk}\}(x' < x \wedge x \doteq y) \\ &x' \doteq x \wedge y > 0 \wedge (x' < x_{sk} \wedge x_{sk} \doteq y) \quad \text{(simplified)} \end{aligned} \tag{5}$$

A model of this condition is the state $s = \{x \mapsto 0, x' \mapsto 0, x_{sk} \mapsto 1, y \mapsto 1\}$. It satisfies also the strength condition, i.e. $s \vDash (x' \doteq x \land y > 0 \rightarrow \{x := x_{sk}\}x' < x) \rightarrow \langle \mathtt{p} \rangle x \doteq x_{sk}$, and therefore the formula $\langle \mathtt{p} \rangle x \doteq y$ as well.

The CBPC differs from preconditions generated with weakest precondition calculi because it is stronger, which is our objective. Interesting is that in contrast to the F-DBPC, the F-CBPC does not require a satisfied specification. Instead the specification must be strong. For example consider the specification $\sigma_2 = (x' \doteq x, x \doteq 2z \land x' \doteq z \land y \doteq z, M)$ with $M = \{x\}$. It is not satisfied by $\mathtt{p}$ but it is strong and therefore $\mathrm{CBPC}_{x \doteq y}$ implies $\langle \mathtt{p} \rangle x \doteq y$.

Even though the anonymous update $\{M := M_{sk}\}$ destroys information about the pre-state that is encoded in *post* and φ the strength condition ensures that just enough information is preserved to ensure the satisfaction of $\langle \mathtt{p} \rangle \varphi$. In contrast to the F-DBPC the F-CBPC is more sensitive to the size of the modifier set. The specification σ is not strong if, e.g., $M = \{x, y\}$.

We prove theorem 1 by proving the validity of the sequent

$$(pre \land \{M := M_{sk}\}post) \rightarrow \langle \mathtt{p} \rangle M \doteq_\forall M_{sk}, pre \land \{M := M_{sk}\}(post \land \varphi) \Rightarrow \langle \mathtt{p} \rangle \varphi$$

This sequent can be simplified to (let $\varphi' = \{M := M_{sk}\}\varphi$):

$$\langle \mathtt{p} \rangle M \doteq_\forall M_{sk}, \varphi' \Longrightarrow \langle \mathtt{p} \rangle \varphi$$

Since φ' is rigid for $\mathtt{p}$ we can use the equivalence $\varphi' \equiv \langle p \rangle \varphi'$ and obtain

$$\langle \mathtt{p} \rangle M \doteq_\forall M_{sk}, \langle \mathtt{p} \rangle \varphi' \Longrightarrow \langle \mathtt{p} \rangle \varphi$$
$$M \doteq_\forall M_{sk}, \varphi' \Longrightarrow \varphi$$

Applying $M \doteq_\forall M_{sk}$ on φ yields φ' resulting in $\varphi' \Longrightarrow \varphi'$. ■

3.3 Using the CBPC if the Specification is satisfied but weak

According to lemma 1 the DBPC requires the involved specification to be satisfied and allows then the detection of infeasible execution paths using theorem provers. According to theorem 1 the CBPC requires the involved specification to be strong to provide a constraint for test data generation using a constraint solver that ensures the satisfaction of a desired branch condition.

We show that the CBPC can be used for test data generation even in use-cases (e.g. it may stem from a verification proof) where it is known that the specification constituting the CBPC is satisfied but not whether it is strong. Assume that the specification is partially correct but we do not know whether it is strong. Generating test data, i.e. a partial state that gives meaning to program variables, for which the CBPC is unsatisfiable is undesired. The reason is that if the specification is correct for this state, then it is guaranteed that either the branch condition is unsatisfied or the precondition is unsatisfied. In the latter case we can make no assumptions about the postcondition.

Theorem 2. *Let $\sigma = (pre, post, M)$ be a specification, $\mathtt{p} \in \pi$, and $\varphi \in Fml$. If σ is satisfied by $\mathtt{p}$ and $s \not\models CBPC^{\sigma}_{\varphi}$, then either $s \not\models pre$ or $s \not\models \langle \mathtt{p} \rangle \varphi$*

The theorem states that if the specification is satisfied but eventually weak, then the CBPC does not guarantee that every model $s \in S$ of the CBPC ensures $s \models \langle \mathtt{p} \rangle \varphi$ but it reduces the search space of the constraint solver appropriately.

Proof. Let $s_{\Sigma \setminus M_{sk}} \in S_{\Sigma \setminus M_{sk}}$ be a partial state where CBPC is unsatisfiable. This means that for each partial states $s_{M_{sk}} \in S_{M_{sk}}$ the CBPC is not satisfied in the total state $s = (s_{\Sigma \setminus M_{sk}} \cup s_{M_{sk}}) \in S$ (see Section 2.1):

$$s_{\Sigma \setminus M_{sk}} \not\models pre \wedge \{M := M_{sk}\}(post \wedge \varphi)$$
$$s_{\Sigma \setminus M_{sk}} \models \neg pre \vee \{M := M_{sk}\}\neg(post \wedge \varphi)$$
$$s_{\Sigma \setminus M_{sk}} \models pre \rightarrow \{M := M_{sk}\}(post \rightarrow \neg\varphi)$$

Since $pre \rightarrow \{M := M_{sk}\}post \rightarrow \neg\varphi$ is the DBPC$_{\neg\varphi}$ and since we assume that the specification is satisfied lemma 1 implies $s_{\Sigma \setminus M_{sk}} \models \neg pre \vee \langle \mathtt{p} \rangle \neg\varphi$. Note that $\langle \mathtt{p} \rangle \neg\varphi$ implies $\neg\langle \mathtt{p} \rangle \varphi$ (see Section 2.1). ■

4 Applications of the CBPC

4.1 Branch Precondition of Loops and Methods

In this section we show how to compute a precondition for a branch condition that follows a method call or loop by constructing the CBPC. Given is a pre- and postcondition $pre_{pq}, post_{pq} \in Fml_{FOL}$ for a sequence of statements `p; q;`, and a specification of `p` : $\sigma = \{pre_p, post_p, M\}$. Testing the program `p; q;` wrt. σ is equivalent to testing the DL-formula: $pre_{pq} \rightarrow \langle \mathtt{p;q;} \rangle post_{pq}$.

The goal is to test the formula such that the test cases ensure the execution of each branch in `q`. Symbolic execution of `q` yields branches that using Dynamic Logic can be represented in form of formulae as follows

$$pre_{pq} \rightarrow \langle \mathtt{p} \rangle(\varphi_1 \rightarrow \langle \mathtt{q1} \rangle post_{pq}) \ldots pre_{pq} \rightarrow \langle \mathtt{p} \rangle(\varphi_n \rightarrow \langle \mathtt{qN} \rangle post_{pq})$$

where $\varphi_1, \ldots, \varphi_n \in Fml$ are the *branch conditions* for executing the sub-programs `q1` ,...,`qN` $\in \pi$ in `q`. Using the specification σ and a branch condition φ_i with $i \in 1 \ldots n$ the CBPC$_{\varphi_i}$ can be constructed according to Def 7. If σ is strong, then using theorem 1 the precondition of a branch $pre_{pq} \rightarrow \langle \mathtt{p} \rangle(\varphi_i \rightarrow \langle \mathtt{qi} \rangle post_{pq})$ is obviously the CBPC$_{\varphi_i}$:

$$pre_{pq} \wedge pre_p \wedge \{M := M_{sk}\}(post_p \wedge \varphi_i)$$

because it ensures in states where it is true that also $\langle \mathtt{p} \rangle \varphi_i$ is true.

As an example we construct the branch precondition that ensures the execution of `C()` in listing 1.2 of Figure 1. The initial formula to test is:

$$\underbrace{true}_{pre_{pq}} \rightarrow \underbrace{\langle \mathtt{i=0;D() ;} \rangle}_{\mathtt{p}} \underbrace{\langle \mathtt{if(i==20)\{C();\}} \rangle}_{\mathtt{q}} \underbrace{true}_{post_{pq}}$$

We make the trivial choices for pre_{pq} and $post_{pq}$ in order to simplify this example. Symbolic execution of the *if*-statement yields two branches according to the case distinction of the *if*-statement.

$$\underbrace{\langle \texttt{i=0;D();} \rangle}_{\texttt{p}} (\underbrace{i \doteq 20}_{\varphi_1} \rightarrow \underbrace{\langle \texttt{C();} \rangle}_{\texttt{q1}} \underbrace{true}_{post_{pq}}) \text{ and } \underbrace{\langle \texttt{i=0;D() ;} \rangle}_{\texttt{p}} (\underbrace{\neg(i \doteq 20)}_{\varphi_2} \rightarrow \underbrace{\langle\rangle}_{\texttt{q2}} \underbrace{true}_{post_{pq}})$$

The interesting branch is where $i \doteq 20$ is true after the execution of `D()`. In order to compute the precondition we use the specification of `D()` consisting of the pre- and postconditions: $i < n$ and $i \doteq n$. Note that the postcondition of `D()` and the branch condition $i \doteq 20$ are unrelated formulae, i.e. neither $i \doteq n$ implies $i \doteq 20$ nor does $i \doteq 20$ imply $i \doteq n$. The CBPC is

$$i < n \wedge \{i := i_{sk}\} i \doteq n \wedge i \doteq 20$$

Applying the update results in the formula $i < n \wedge i_{sk} \doteq n \wedge i_{sk} \doteq 20$ which in fact implies the desired branch precondition $i < 20 \wedge n \doteq 20$. This example is almost identical for listing 1.1 using a loop invariant.

4.2 Loop Branch Precondition

The loop branch precondition (LBPC) is a precondition of a loop that ensures that *during* the execution of the loop a certain branch is taken, e.g. to execute `A()` in listing 1.1 (Fig 1). In the following example we derive a loop branch precondition by constructing a CBPC from a loop invariant and branch condition.

— Java (1.3) —

```
1  MyHashMap incHM(IDObj[] a){
2   int i = 0;  MyHashMap map = new MyHashMap();
3   while(i < a.length ){
4    if(map.count > (map.size*3)/4){
5      tmp = new MyHashMap(map.size*2);
6      tmp.copyFrom(map);  map = tmp;
7    }
8    map.put(a[i].id, a[i]);   i++;
9   }
10 }
```

— Java —

We assume the following behavior of the program in listing 1.3. The constructor `MyHashMap()` creates a hash map object with the initial capacity `map.size` $\doteq 8$. The field `map.count` tracks the current number of elements in the map. If during loop iteration the branch condition `map.count > (map.size*3)/4` becomes true, then a new hash map of size `map.size*2` is created. The statement

`tmp.copyFrom(map);` copies all elements from the old hash map to the new hash map so that `tmp.count` $\doteq$`map.count`. The method `put(x,y)` stores `y` in the hash map under key `x` and overwrites any previously store entry under the same key.

The goal is to compute the CBPC for the specification of the loop and branch condition $map.size * 3/4 < map.count$, which we abbreviate with φ. The specification of loop body chains before loop termination is $\sigma = \{I, I \wedge lc, M\}$ (see Section 2.4), where $M = \{map, size, count, i\}$, the loop condition lc is $i < a.length$, and I consists of the invariants

$$\underbrace{map^{old}.size^{old} \leqslant map.size}_{inv_1} \wedge \underbrace{map.count \leqslant N}_{inv_2} \wedge \underbrace{map.count \leqslant i}_{inv_3}$$

For the precondition of `incHM()` we assume that `a` $\neq$`null` and that $N \in \mathbb{N}$ is the total number of distinct elements in the array `a` formalized by

$$a \neq null \wedge \forall x.0 \leqslant x < N \leqslant a.length \rightarrow \exists y.0 < y < a.length \wedge a[y].id = x$$

For this setting the resulting CBPC is:

$$\begin{aligned} &\{map^{old}, size^{old}, count^{old}, i^{old}\} \doteq_{\forall} \{map, size, count, i\} \wedge I \\ &\wedge \{\{map, size, count, i\} := \{map_{sk}, size_{sk}, count_{sk}, i_{sk}\}\}(I \wedge lc \wedge \varphi) \end{aligned} \tag{6}$$

Applying the update on $(I \wedge lc \wedge \varphi)$ yields the conjunction

$$\begin{aligned} &\underbrace{map^{old}.size^{old} \leqslant map_{sk}.size_{sk}}_{inv'_1} \wedge \underbrace{map_{sk}.count_{sk} \leqslant N}_{inv'_2} \wedge \underbrace{map_{sk}.count_{sk} \leqslant i_{sk}}_{inv'_3} \\ \wedge\ &\underbrace{i_{sk} < a.length}_{lc'} \wedge \underbrace{(map_{sk}.size_{sk} * 3)/4 < map_{sk}.count_{sk}}_{\varphi'} \end{aligned}$$

that can be simplified as follows. From inv'_1, φ', and inv'_2 follows $(map^{old}.size^{old} * 3)/4 < N$. From phi', inv'_3, and lc follows $(map^{old}.size^{old} * 3)/4 + 1 < a.length$. Using the equations in (6) we can replace $map^{old}.size^{old}$ with $map.size$ deriving the important constraint from the CBPC:

$$(map.size * 3)/4 < N \wedge (map.size * 3)/4 + 1 < a.length \tag{7}$$

This precondition guarantees, when satisfied just before the execution of the loop of listing 1.3, that during the iteration of the loop the *then*-branch of the *if*-statement is entered. In order to transfer the CBPC into the pre-sate of `incHM()` the effect of line 2 has to be taken into account. Symbolic execution of line 2 results in a sequence of updates. One of them is $\{map.size := 8\}$ according to our description of the listing. Applying the update on (7) results in the desired precondition of `incHM()` : $(8 * 3)/4 < N \wedge (8 * 3)/4 + 1 < a.length$.

5 Related Work

This work is an extension of [9] and [4] —both developed within the KeY-project [5]. In [9] verification-based testing is introduced as a method for deriving test cases from verification proofs. In [4] we present a white-box testing approach which combines verification-based specification inference and black-box testing allowing to combine different coverage criteria. Both approaches consider the derivation of test cases based on loop invariants by example but not in the depth as it is done in this work.

Other approaches using symbolic execution to derive test cases are, e.g. [16, 15, 7]. The tools Symstra [16] and Unit Meister [15] use a bound on the number of analysed loop iterations and [7] uses a bound on the size of the analysed data structures. The latter approach yields an implicit bound on set of symbolically executed paths.

How to compute a precondition for a given condition after the execution of a program has been a research topic especially in the context of weakest precondition calculi for the purpose of software verification [8, 14, 11]. The *disjunctive branch precondition* (DBPC) that we use to detect unsatisfiable constraints for test data generation turns out to be the precondition that most weakest precondition calculi generate for verification. This precondition is however too weak for the purpose of test data generation because not every model of the precondition yields a test case that executes the program in the desired way.

The generation of test cases based on specifications has been suggested, e.g., in [2] and [13]. In [2] a software development methodology is described whose main artifacts are invariants. In [13] the generated test cases are based on *effect predicates* derived from executable specifications. *Effect predicates* are related to strong specifications in this paper. We use however first-order logic based specifications and executable Java source code.

6 Conclusion

We have shown how test cases can be generated for testing program branches that occur within loops and after the execution of loops or complex methods. The challenge is to generate a precondition for a branch condition of a program branch. This may be infeasible or impossible with precondition computation based on symbolic execution. Our approach is to compute the *conjunctive branch precondition* (CBPC), which is a precondition generated from a branch condition and a specification. The CBPC is stronger than the *disjunctive branch precondition* (DBPC) that is usually generated by weakest-precondition calculi.

Our contributions are the described approach and the analysis of the two kinds of preconditions, i.e. CBPC and DBPC, as well as of the required properties of the involved specifications. We proved the correctness of (1) a theorem for

test data generation, (2) a rule for detecting infeasible execution paths, and (3) another theorem showing the relation between the CBPC and DBPC. The CBPC can improve test case generation even if the F-CBPC, which additionally requires the involved specification to be strong, is not satisfied. A suitable use-case for our approach is, e.g., the generation of test cases from verification proofs because the CBPC occurs, except for some details, as a sub-formula in proof branches. The approach is implemented in the KeY-system.

References

[1] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] R.-J. Back, J. Eriksson, and M. Myreen. Testing and verifying invariant based programs in the SOCOS environment. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 61–78. Springer, 2007.

[3] B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.

[4] B. Beckert and C. Gladisch. White-box Testing by Combining Deduction-based Specification Extraction and Black-box Testing. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.

[5] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Springer, 2007.

[6] D. Cansell and D. Méry. Foundations of the B method. *Computers and Artificial Intelligence*, 22(3), 2003.

[7] X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems. In *ASE*, pages 157–166, 2006.

[8] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[9] C. Engel and R. Hähnle. Generating Unit Tests from Formal Proofs. In Y. Gurevich, editor, *Proceedings, Testing and Proofs, Zürich, Switzerland*, LNCS. Springer, 2007.

[10] D. Harel. Dynamic logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, volume II: Extensions of Classical Logic, chapter 10, pages 497–604. Reidel, Dordrecht, 1984.

[11] B. Jacobs. Weakest pre-condition reasoning for Java programs with JML annotations. *J. Log. Algebr. Program.*, 58(1-2):61–88, 2004.

[12] P. Rümmer. A language for sequential, parallel and quantified updates of first-order structures. March 2005.

[13] M. Satpathy, M. Butler, M. Leuschel, and S. Ramesh. Automatic testing from formal specifications. In Y. Gurevich and B. Meyer, editors, *TAP*, volume 4454 of *Lecture Notes in Computer Science*, pages 95–113. Springer, 2007.

[14] K.-D. Schewe and B. Thalheim. A generalization of dijkstra's calculus to typed program specifications. In *FCT*, pages 463–474, 1999.

[15] N. Tillmann and W. Schulte. Parameterized unit tests with unit meister. In *ESEC/SIGSOFT FSE*, pages 241–244, 2005.

[16] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings, Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK*, LNCS 3440, pages 365–381. Springer, 2005.

Extracting Bugs from the Failed Proofs in Verification via Supercompilation

Alexei Lisitsa[1] and Andrei P. Nemytykh[2⋆]

[1] Department of Computer Science, The University of Liverpool
A.Lisitsa@csc.liv.ac.uk
[2] Program Systems Institute of Russian Academy of Sciences
nemytykh@math.botik.ru

Abstract. If a program q produces result satisfying property P then its composition with a program implementing the test for P will never return the negative result of the test. In the verification via supercompilation approach a proof of correctness of a program is done by transformation of the above composition into an equivalent form from which the required property of composition ("never returns the negative result") can be easily established by simple syntactical check. A powerful program transformation technique known as supercompilation is used. In the previous work we have applied this approach to the automated verification of various parameterized protocols. In this paper we show how to extract bugs (execution traces violating correctness condition) from the failed proofs of correctness via testing and supercompilation. As a case studies we consider verification of Dragon cache coherence protocol and a Client-Server protocol, in both cases starting with incorrect specifications.

1 Introduction

V. Turchin [34] has proposed the following scheme combining the *testing* and *program transformation* for proving properties of the (functional) programs:... *if we want to check that the output of a function $F(x)$ has the property $P(x)$ we can try to transform the function $P(F(x))$ into an identical T*. In principle, any program transformation technique preserving equivalence of the programs can be used within the scheme and Turchin in [34–36] has illustrated the approach with simple examples using *supercompilation* (supervised compilation) for program transformation. Despite being very natural the approach has not attracted much attention and as far as we know has not been used in verification until recently. Having said that we should notice that a range of other verification methods based on program transformation have been developed and/or proposed [12, 16, 17, 31, 13]. In [18–22] we extended the Turchin's proposal to the verification of parameterized protocols using particular scheme *parameterised testing* and reported about successful automated verification experiments using SCP4 supercompiler [29, 30]. The web page [23] is a reference point for the

⋆ The second author is supported by the Program for Basic Research of the Presidium of Russian Academy of Sciences (as a part of "Development of the basis of scientific distributed informational-computing environment on the base of GRID technologies"), and the Russian Ministry of Sciences and Education (contract 200777-4-1.4-18-02-064).

growing collection of protocols, programs and algorithms successfully verified by *verification via supercompilation* method. In [21] we presented supercompilation process within verification scenario as an inductive theorem proving for term rewriting systems. Notice that supercompilation is a program transformation technique based on a kind of symbolic execution, so the overall approach can be defined also as the *program proving* via *testing and symbolic execution*.

It has turned out that the approach is not only powerful in proving the properties but also *informative* in the situations when the proofs fail. When syntactical form of residual (transformed) program does not allow to derive the correctness property it may often lead to the extraction of explicit execution traces violating the property, or guide the further tests on the program eventually leading to the bugs discovery. In the present paper we discuss these aspects of the method and illustrate them by two case studies.

The paper is organised as follows. In the next section we give a general overview of our verification via parameterised testing and supercompilation approach. Then in Section 3 we briefly introduce the functional programming language Refal used in experiments. In Section 4 we outline supercompilation program transformation technique and provides some details of particular supercompiler SCP4. Sections 5 and 6 present case studies. Section 7 presents discussion and outlines further directions.

2 Parameterised testing and verification

In this section we describe briefly our general technique for the verification of parameterised systems. It is an adaptation of the Turchin's proposal to the case of non-deterministic, in general, protocols.

The first important assumption is that the protocol to be verified has to be encoded into a functional program. The paper [11] discusses in details different aspects of such encoding and as we found afterwards we use the same principle of encoding of non-deterministic protocols as discussed in that paper: "we view a non-deterministic agent as an incompletely specified deterministic one" [11].

The scheme works as follows. Let S be a parameterised system (a protocol) and we would like to establish some safety property P of S. We write a functional program φ_S encoding a system S meaning φ_S simulates execution of S for n steps, where n is an input parameter. If the system is non-deterministic, then, following to the above-mentioned principle, an additional parameter $\bar{x}$ is provided, whose value is assumed to be a sequence of choices at the branching points of execution, e.g. it may be a string of characters labelling the choices.

Thus, we assume that given the values of input parameters n and $\bar{x}$, the program φ_S returns the state of the system S after the execution of n steps of the system, following the choices provided by the value of $\bar{x}$. Let $T_P(\_)$ be a testing

program, which given a state s of S returns the result of testing the property P on s (*True* or *False*). Consider a composition $T_P(\varphi_S(n, \bar{x}))$. This program first simulates the execution of the system and then tests the property required. Now the statement "the safety property P holds in any possible state reachable by the execution of the system S" is equivalent to the statement "the program $T(\varphi_P(n, \bar{x}))$ never returns the value *False*, no matter what values are given to the input parameters". Here we assume additionally that both programs φ_P and T terminate for all possible inputs, but still may terminate with an abnormal stop. The syntactical property of the residual program sufficient to conclude that the program never returns the value *False* is simple: it does not contain operator *return False*. Of course, it is not in general, necessary condition – there might be some branches in the program with *return False* operators of which are never executed. Obviously, precise form of the condition depends on the language of implementation.

In practical implementation of the scheme we use a functional programming language Refal [37, 38] to implement a program $T_P(\varphi_S(n, \bar{x}))$ and a supercompiler SCP4 (an optimizer) [26, 29, 30] to transform a program to a form, from which one can easily establish the required property.

In this paper we focus mainly on the situations when the form of a supercompiled version of the program $T_P(\varphi_S(n, \bar{x}))$ does not allow to derive the property. It may be the case the program transformer has proved to be not powerful enough to eliminate all unreachable occurrences of *return False* operator while the program *does* satisfy correctness condition. On the other hand, it may indicate that, indeed, the program does not satisfy the property.

As we have discovered during experiments, the form of residual program may suggest tests for the original program, which may lead to the discovery of the execution traces violating correctness condition. We illustrate this point with the case study of a parameterised cache coherence Xerox PARC Dragon protocol. Its specification given in [9] has turned out to be incorrect and we show how supercompilation guided analysis led to the discovery of incorrect traces. To highlight the power of the approach we also briefly discuss the successful verification of the correct version of the same protocol.

It may also be the case, that despite formally successful verification, the form of the residual program has easily recognizable property, witnessing the problems with initial specification. Our second case study (verification of Client-server protocol) demonstrates such a case.

3 Refal programming language

The Refal programming language [37] (Recursive Functions Algorithmic Language) is a first-order strict functional language. Unlike LISP the language is based on the model of computation known as Markov's algorithms.

```
program    ::= $ENTRY definition+
definition ::= function-name { sentence;+ }
sentence   ::= left-side = expression
left-side  ::= pattern
expression ::= empty | term expression | function-call expression
function-call ::= <function-name argument>
argument ::= expression
pattern  ::= empty | term pattern
term     ::= SYMBOL | variable | (expression)
variable ::= e.variable-name | s.variable-name | t.variable-name
empty    ::=                        /* nihil */
```

Refal data are defined by the grammar:

d ::= (d_1) | d_1 d_2 | SYMBOL | empty

Roughly speaking, a program in Refal is a term rewriting system. The semantics of the language is based on pattern matching. As usually, the rewriting rules are ordered to match from the top to the bottom. The terms are generated using two constructors. The first is concatenation. It is binary, *associative* and is used in infix notation, which allows us to drop its parentheses. The blank is used to denote concatenation. The second constructor is unary. It is syntactically denoted by its parentheses only (that is without a name). The unary constructor is used for constructing tree structures. Formally, every function is unary. However, n-ary functions can be easily represented using second term constructor (parentheses) for separation of the function arguments. Empty sequence is a special basic ground term. This term is denoted with nothing and called "empty expression". It is the neutral element (both left and right) of concatenation. All other basic ground terms are named as "symbols". There exist three types of variables - *e.name*, *s.name* and *t.name*. An e-variable can take any expression as its value, an s-variable can take any symbol as its value and t-variable can take any term as its value (a term is either a symbol or an expression in the structure brackets). For every sentence its set of variables from the left-hand side includes the set of variables from the right-hand side; there are no other restrictions on the variables.

Given a current active function call, the step of Refal machine is defined as the following sequence of actions: (1) pattern matching, (2) replacement of the right-hand side variables with their values obtained as the result of the pattern matching, (3) replacement of the active function call (in the function stack) with the updated right-hand side and labelling of a new function call on the top of the changed stack as active.

Example 1. The following program replaces every occurrence of the identifier `Lisp` with the identifier `Refal` in an arbitrary Refal datum.

```
$ENTRY Go { e.inp = <Repl (Lisp Refal) e.inp>; }
Repl {
(s.x e.v) = ;
```

```
(s.x e.v) s.x e.inp = e.v <Repl (s.x e.v) e.inp>;
(s.x e.v) s.y e.inp = s.y <Repl (s.x e.v) e.inp>;
(s.x e.v) (e.y) e.inp = (<Repl (s.x e.v) e.y>) <Repl (s.x e.v) e.inp>;
}
```

Consider a trace of a Refal computation for the program given above. Let the computation start with the call

```
<Go (A Lisp)>.
```

Refal datum `(A Lisp)` represents a binary tree with the leaves `A`, `Lisp`. The computation proceeds with the following steps:

```
2: <Repl (Lisp Refal)  (A Lisp)>
3: (<Repl (Lisp Refal) A Lisp>) <Repl (Lisp Refal)>
4: (A <Repl (Lisp Refal)  Lisp>) <Repl (Lisp Refal)>
5: (A Refal <Repl (Lisp Refal)>) <Repl (Lisp Refal)>
6: (A Refal) <Repl (Lisp Refal)>
7: (A Refal)
```

Example 2. Another example is the function *append*, which can be defined in Refal in one line:

```
$ENTRY append { (e.xs) (e.ys) = e.xs e.ys; }
```

4 Supercompilation and the supercompiler SCP4

In this section we present some details of supercompilation process (which are relevant to the subject of this paper), as it is implemented in the supercompiler SCP4. More details can be found in [26, 30, 28, 29].

Consider a program written in some programming language together with a partially defined (parameterised) input entry of the program. Such a pair defines a partial input-output mapping `f:` $D \mapsto D$, where D is the data set of the language. By definition, a supercompiler is a transformer of such pairs.

The supercompiler SCP4 iterates an *extension* of the interpretation of Refal steps (pattern matching plus constructing a right side), called *driving* [33], on parameterised sets of the input entries. Driving constructs a directed tree of all possible computations for the given parameterised input entry and a given Refal step. The edges of this tree are labelled with predicates over values of the parameters. The predicates specify concrete computation branches and describe the narrowing of the parameters (unknown data) along the chosen branches[3].

Iteration of the driving unfolds a potentially infinite tree of all possible computations. The computations can depend on the values of the parameters that can be unknown during transformation. The supercompiler reduces in the process the redundancy that could be present in the original program. It folds the

[3] In this sense the driving works similarly to a PROLOG interpreter. Both tools accept parameters (free variables) as their input data and narrow the parameters.

tree into a finite graph of states and transformations between possible configurations of the computing system. To make a folding possible a *generalization* procedure is used. Sometimes it may lead to the lost of some information on the structure of arguments of configurations.

If it is not possible to reduce a current configuration (to be developed in the meta-tree) to a previous configuration (on the path from the tree root to the latter) then generalization looks for a previous configuration, which is *similar* to the current. A variant of *homeomorphic embedding pre-order* specifies the *similarity* relation on the configurations [15, 32, 26]. Only similar configurations are generalized.

In a *weak* strategy of supercompilation *all* configurations appeared in the meta-tree are analyzed by the generalization and this ensures the termination of the whole supercompilation process.

In order to perform as many actions of the input program as possible in supercompile time the *strong* strategy of supercompilation excludes from the generalization those parameterised configurations which appear in the meta-tree nodes with *one* outgoing branch. In general, it may allow more powerful transformations to be achieved, but the termination of supercompilation is not guaranteed in that case.

Thus, we emphasize that the output of the supercompiler is defined in terms of the parameters (*semantic objects*). The resulting definition is constructed solely based on the meta-interpretation of the source program rather than by a step-by-step transformation of the program. The crucial property of the supercompilation procedure, that we rely upon in our verification methodology, is

Property 1. The output pair (the residual program and its input entry) defines an extension of the partial mapping defined by the corresponding input pair.

5 A case study: verification of the Xerox PARC Dragon protocol

Cache coherence protocols play an important role in models of shared memory multiprocessor systems. Usually, in such systems, every individual processor has its own private cache memory, which is used to hold local copies of main memory blocks [14]. While reducing the access time, this approach poses the problem of cache consistency, whereby one has to ensure that the copies of the *same* memory block in the caches of *different* processors are consistent. Such data consistency is supported by cache coherence protocols, which typically operate as follows: every processor is equipped with a finite state control, which reacts to the *read* and *write* requests. Abstracting from the low-level implementation details of read, write and synchronisation primitives, one may model cache coherence

protocols as *families of identical finite state machines*, with the number of the machines being a parameter, together with a primitive form of communication: if one automaton makes a transition (an action) a, then it is required that *all* other automata make a complementary transition (reaction) $\bar{a}$ [7, 9]. The computation is assumed to be non-deterministic, i.e. at every step one automaton is chosen to make one of the available actions. Because in the model automata are assumed identical, there is a lot of symmetry in their behaviour, so for the analysis one may apply a *counting abstraction* and keep track only of the *number of automata* in every possible (local) states. That gives Extended Finite State Machines formulation of the parameterized protocols. Extended Finite State Machines (EFSM) [3] are essentially transition systems with data variables ranging over non-negative numbers. EFSM-transitions are linear transformations with the guards expressed as linear constraints.

5.1 An EFSM model of the Xerox PARC Dragon protocol

As a case study we consider in this section the verification of the parameterised Xerox PARC Dragon cache coherence protocol. We start with its specification in terms of Extended Finite State Machines (EFSM) model taken from [6]. Here $dirty$, $shared\_clean$, $exclusive$, $shared\_dirty$, $invalid$ are non-negative integer variables of EFSM model, which represent *counting abstraction* of an original parameterised automata model. That means that during the run of EFSM, the value of, for example, $shared\_clean$ variable indicates the number of automata in the state $shared\_clean$. The rules below describe the dynamic of an EFSM model of the Dragon protocol (as described in [6]). Starting with some initial valuation of variables, the model may apply non-deterministically any of the applicable rules. Applicability of the rules defined by left-hand sides of rules (guards). For example, rule $rm1$ is applicable to a state, only if $invalid \geq 1$ and $dirty = shared\_clean = shared\_dirty = exclusive = 0$. The application of a rule is an execution of the update (assignment) expressed by the right-hand side of the rule. For example, for the rule $rm1$ execution of the update amounts to decreasing the value of variable $invalid$ by 1 and increasing the value of $exclusive$ by 1.

(rh) $dirty + shared\_clean + exclusive + shared\_dirty \geq 1$.
$(rm1)$ $invalid \geq 1, dirty = shared\_clean = shared\_dirty = exclusive = 0 \rightarrow invalid' = invalid - 1,$ $exclusive' = 1 + exclusive.$
$(rm2)$ $invalid \geq 1, shared\_clean + shared\_dirty + exclusive + dirty \geq 1 \rightarrow invalid' = invalid - 1,$ $dirty' = 0, exclusive' = 0, shared\_dirty' = dirty + shared\_dirty,$ $shared\_clean' = 1 + exclusive + shared\_clean.$
$(wm1)$ $invalid \geq 1, dirty = shared\_clean = shared\_dirty = exclusive = 0 \rightarrow invalid' = invalid - 1,$ $dirty' = 1 + dirty.$
$(wm2)$ $invalid \geq 1, shared\_clean + shared\_dirty + exclusive + dirty >= 1 \rightarrow invalid' = invalid -$ $1, exclusive' = 0, shared\_clean' = shared\_dirty + exclusive + shared\_clean, shared\_dirty' = 1.$
$(wh1)$ $dirty \geq 1 \rightarrow$.
$(wh2)$ $exclusive \geq 1 \rightarrow exclusive' = exclusive - 1, dirty' = 1 + dirty$.

($wh3$) $shared\_dirty = 1$, $shared\_clean = 0 \rightarrow shared\_clean' = 0$, $dirty' = 1 + dirty$.
($wh4$) $shared\_dirty = 0$, $shared\_clean = 1$, $\rightarrow shared\_clean' = 0$, $dirty' = 1 + dirty$.
($wh5$) $shared\_dirty + shared\_clean \geq 2 \rightarrow shared\_clean' = shared\_clean + shared\_dirty - 1$, $shared\_dirty' = 1$.

Correctness of the parameterized Xerox Parc Dragon protocol according to [6] can be expressed in terms of the EFSM model as follows: the system started in *any* initial state (an assignment of the variables) satisfying $invalid \geq 1$ and $shared\_clean = shared\_dirty = dirty = exclusive = 0$ and evolving according to the rules above can never get into the state satisfying one of the conditions below:

C_1 $dirty \geq 2$;
C_2 $exclusive \geq 2$;
C_3 $exclusive + shared\_clean + shared\_dirty \geq 1$, $dirty \geq 1$;
C_4 $exclusive \geq 1$, $shared\_clean + shared\_dirty \geq 1$.

It has turned out though that the specification given in [6] and shown above is incorrect. We have found the following trace of the protocol execution which violates the correctness condition. Starting with the initial configuration $invalid = 3$, $shared\_clean = shared\_dirty = dirty = exclusive = 0$, the sequence $wm1$, $wm2$, $wh3$ of the rule application leads to the configuration with $dirty = 2$ (i.e. satisfying condition C_1 above). This trace has been obtained by the analysis of a failed attempt of verification via supercompilation. We decsribe now this attempt.

5.2 Refal encoding of the EFSM model

We apply the technique described in Section 2 and encode the EFSM model and the correctness test in a Refal program. The full text of the program is in Appendix A [24]. Here we explain some crucial details only. The entry point of the program is the function *Go* defined as follows:

```
$ENTRY Go {e.time (e.x1) = <Loop (time e.time) (invalid I e.x1)(shared_clean)
                                 (shared_dirty)(dirty)(exclusive)
                          >;
          }
```

It has two arguments $e.time$ and $e.x1$ which according to the Refal conventions may take arbitrary Refal expressions as the values. The function will be defined though only for $e.time$ taking the sequence of the names of the EFSM rules as the value (see definition of *Loop* and *RandomAction* functions below). The length of a string (value of) e.x1 is a value of the variable *invalid* in the initial configuration of EFSM decreased by 1. In general, we use the following representation of the global state of the system by Refal data:

```
(e.time)
(invalid e.x1)(shared_clean e.x2)(shared_dirty e.x3)(dirty e.x4)(exclusive e.x5)
```

Here the value of $e.time$ represents the sequence of remaining rules to be executed, and the length of the value of each $e.xi$ ($i = 1, \ldots 5$) represents the value of the corresponding variable of the EFSM model, which are given in the unary system.

As to the function *Go* it takes two arguments and calls the function *Loop*. The syntactical form of this call reflects the constraints on the initial configuration of EFSM - $invalid \geq 1$ and all other variables have the value 0.

The definition of the function *Loop* contains two sentences: one for quitting the loop and calling the $Test$ function, and another for making recursive call of *Loop* with decremented value of the first argument and updated state of the encoded EFSM. Update is done by the call to the *RandomAction* function. The sentences in the definition of the *RandomAction* function correspond to the rules of EFSM.

Since the application of the rules rh and $wh1$ do not change the global state of the system and we intend to verify the safety property, we can safely omit these rules from the Refal specification. Further, notice that the rules $rm2$, $wm2$ and $wh5$ are specified by several sentences in the *RandomAction* definition. This is because they have more complex guards which can not be specified by a single sentence.

Consider the following fragment of the *RandomAction* definition, specifying the $wh5$ rule:

```
...
wh5A (invalid e.x1)(shared_clean I I e.x2)(shared_dirty e.x3)(dirty e.x4)
                                                          (exclusive e.x5) =
     (invalid e.x1)(shared_clean e.x3 I e.x2)(shared_dirty I)(dirty e.x4)
                                                          (exclusive e.x5);
wh5B (invalid e.x1)(shared_clean I e.x2)(shared_dirty I e.x3)(dirty e.x4)
                                                          (exclusive e.x5) =
     (inavlid e.x1)(shared_clean e.x3 I e.x2)(shared_dirty I)(dirty e.x4)
                                                          (exclusive e.x5);
wh5C (invalid e.x1)(shared_clean e.x2)(shared_dirty I I e.x3)(dirty e.x4)
                                                          (exclusive e.x5) =
     (inavlid e.x1)(shared_clean I e.x3 e.x2)(shared_dirty I)(dirty e.x4)
                                                          (exclusive e.x5);
   ...
```

The left-hand side of the first shown sentence

```
wh5A...(shared_clean I I e.x2)...
```

can be matched with the argument of *RandomAction* if and only if this argument encodes the state with $shared\_clean \geq 2$. In the same way, left-hand sides of remaining two sentences correspond to the conditions $shared\_clean \geq 1$, $shared\_dirty \geq 1$ and $shared\_dirty \geq 2$, respectively. Then any state satisfying the guard of the rule $wh5$, that is $shared\_clean + shared\_dirty \geq 2$ would satisfy *at least one* of the conditions expressed by the left-hand sides of the sentences

above. The right-hand sides of the sentences all express the same update of the rule $wh5$ that is

$$shared\_clean' = shared\_clean + shared\_dirty - 1, shared\_dirty' = 1 \ .$$

The function *Test* (see Appendix A in [24] for the definition) performs the correctness condition testing.

Taking all definitions together we get a Refal program which simulates execution of the Xerox PARC Dragon protocol for k automata (= the length of the value of the input parameter $e.x1$ plus 1) following the sequence of actions presented by $e.time$ and then tests the correctness condition.

5.3 Failed attempt to verify and search for the error

If we apply the supercompiler SCP4 to the program above we get a residual program presented in Appendix B [24]. As we see its syntax does not allow to conclude the correctness of the encoded protocol - both constants *True* and *False* are presented in the right-hand sides of the sentences. The question we address now is whether the program (an executable specification of the Dragon protocol) is indeed, incorrect, or the program is correct and the supercompiler failed to prove it (i.e. to remove all occurences of *False*).

To trace the origin of *False* constants in the residual program we modify the *Test* function of the original program and index all *False* constants in its definition by the numbers of configurations in which they appear. For this variant, after supercompilation the residual program contains only *False1* and *False3*.

Now we focus on the analysis of occurences of *False1* only. To this end we simplify the definition of the *Test* function to just two sentences, the first and the last sentences of the original *Test* definition. The result of supercompilation for such a variant still contains the constant *False*.

For further analysis we consider the execution of the protocol for the small fixed number of steps. For this purpose we first modify the definition of the *Go* function and set $e.time = s.t1 \ s.t2$ (two steps execution). Supercompiler returns in that case a program without *False*, indicating that there are no two steps traces violating the correctness. Consider the case of 3 steps execution and set $e.time = s.t1 \ s.t2 \ s.t3$. As the result of supercompilation we get a residual program of the following form:

```
$ENTRY Go {
....
wm1 wm2C wh3 (s.153 e.104) = False
...
}
```

which immediately suggests a trace of the protocol execution violating the correctness condition. To test it we run original program with the start configuration

```
<Loop (time wm1 wm2C wh3)(invalid I I I)(shared_clean)(shared_dirty)(dirty)
        (exclusive)
>
```

and get the result *False*. The protocol as specified above is indeed incorrect despite the correctness statement given in [6].

5.4 Successful verification of the Xerox PARC Dragon protocol

Following the references given in [6, 9] we have found another (*program*) specification of the parameterized Dragon protocol in terms of EFSM [5]. We present here only the clauses where this specification is different from one given in [6]

$(wm2)$ $invalid \geq 1$, $shared\_clean + shared\_dirty + exclusive + dirty \geq 1 \rightarrow invalid' = invalid - 1$, $exclusive' = 0$, $dirty' = 0$, $shared\_clean' = shared\_dirty + dirty + exclusive + shared\_clean$.
$(wh1)$ $dirty \geq 1 \rightarrow invalid' = 1 + invalid$, $dirty' = dirty - 1$.
$(wh2$ $shared\_clean \geq 1 \rightarrow invalid' = 1 + invalid$, $shared\_clean' = shared\_clean - 1$.
$(wh3)$ $shared\_dirty \geq 1 \rightarrow$, $invalid' = 1 + invalid$, $shared\_dirty' = shared\_dirty - 1$.
$(wh4)$ $exclusive \geq 1 \rightarrow invalid' = 1 + invalid$, $exclusive' = exclusive - 1$.
$(wh5)$ *this clause is absent now.*

The correctness condition for this version of the protocol remains the same. We encoded this specification together with the correctness condition in a Refal program. Result if its supercompilation can be found in the Appendix C [24]. The residual program does not contain the constant *False* and therefore the correctness of the (program specifying the) Dragon protocol is proved.

As we argued in [21] the successful verification of safety properties via supercompilation can be seen as a proof by mutual induction and the process of supercompilation as a proof search. Not going into detailed discussion and referring an interested reader to [21] for more details, we would like to illustrate here the complexity of the proof involved. The figure 1 shows a proof diagram which we derived manually from the proof of the correct version of the Dragon protocol via supercompilation. The vertices of the graph represent *terms* describing parameterized configurations (sets of configurations) of the Refal program and therefore, parameterized configurations of the encoded Dragon protocol. The following properties are supported.

- Any possible configuration reachable during actual execution of the program (protocol) belongs to a set of configurations represented by some vertex. In that case we say that the configuration is *covered* by the vertex.
- Arcs in the graph represent one-step reachability between parameterised configurations, that is if some configuration covered by a vertex a, then its successor in the execution is covered by a vertex b such that $a \rightarrow b$ (there is an arc going from a to b).
- Vertices labelled by $t1, t2, \ldots$ represent sets of final configurations where execution of the program (development of the protocol) stops and the correctness condition is successfully tested.

With every vertex one can also associate a correctness statement: every final configuration reachable from any configuration covered by the vertex satisfies the

correctness condition. All vertices of the rectangular shape represent inductive hypotheses. The hypotheses labelled by Ti were created by automatic generalization of the inductive hypotheses encountered in the process of the proof (or being given as the initial problem definition). Taking this point the proof diagram represents the proof of *all* correctness hypotheses by simultaneous induction. Statements associated with the vertices labelled by $t1, t2, \ldots$ form the base of induction.

We notice also that although we have derived the proof diagram manually, we do not see any problems in automation of this process.

6 Verification of the Client-Server Protocol

As the second case study we consider verification of the parameterized abstract Client-Server protocol as specified in [10], again in terms of EFSM, but extended in this case by the finite state control and boolean variables.

First, we translated a specification given in [10] into executable specification + testing function in Refal terms as shown in Appendix D [24]. To represent integer variables we followed the same conventions as with the Dragon protocol. The states of the finite state control and a (single) boolean variable of the EFSM are represented in this case by terms of the form $(Server \ldots)$. For example, $(Server\ ServS\ No)$ indicates the $ServS$ is being the state of the finite state control and a boolean variable has a value No. As before the rules of the protocol are encoded by sentences of the *RandomAction* definition, the initial condition is encoded in the *Go* function definition and the *Test* function embodies the correctness condition.

When we applied SCP4 for this specification we have got a small residual program:

```
* InputFormat: <Go e.41>
$ENTRY Go {
 (e.101)  = True ;
 reqS (e.101)  = True ;
 reqS nonex1 (e.101)  = True ;
 reqE1 (e.101)  = True ;
 reqE1 invS (e.101) = True ;
 reqE1 invS nonex2 (e.101)  = True ;
}
```

Formally, it does not contain the *False* constant and the verification seemed successful and it was, in fact successful for the *given* executable specification. But ... it was immediately transparent that this residual program allows only a few different *finite* executions of the protocol. That was in the contradiction with the assumption that the protocol is designed to work indefinitely long.

That suggested a test for the original executable specification: to execute it for the sequence of steps extending the cases presented in the residual program. An attempt to execute the sequence $reqS, nonex1, grantS$ resulted in the

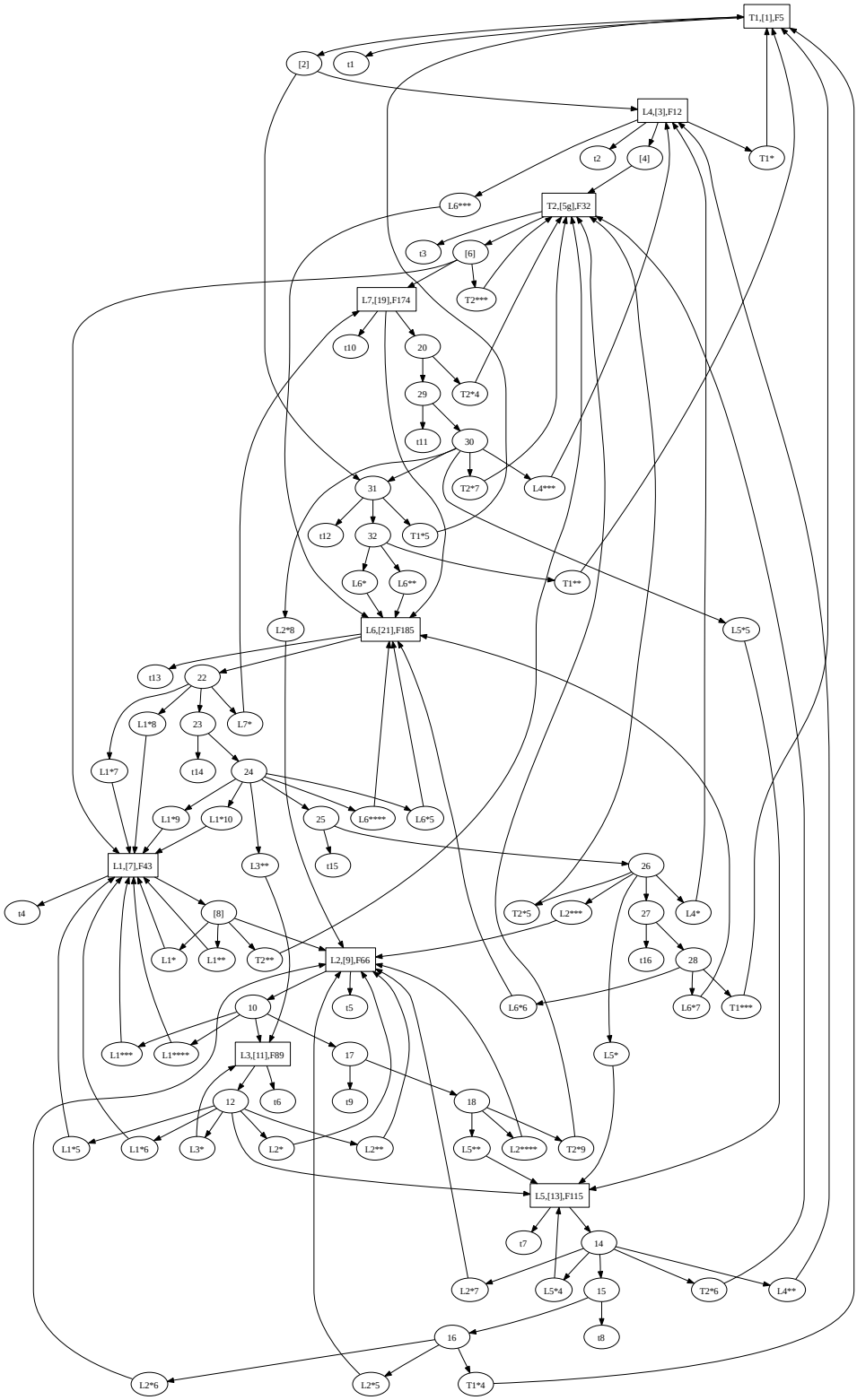

Fig. 1. Proof diagram for the parameterised Xerox PARC Dragon protocol

halted execution with the Refal interpreter returning the message `RECOGNITION IMPOSSIBLE`, meaning a call to the *RandomAction* function failed, that is an argument of the call could not be matched with any left-hand side of the sentences in the definition of *RandomAction*. Simple analysis then has shown that the reason for that was *two typos*[4] made in the definition of *RandomAction*. To correct the specification one needs to replace in the rule 9 the term *(Server grantS s.bool)* with *(Server GrantS s.bool)* and in the rule 10 the term *(Server grantE s.bool)* with *(Server GrantE s.bool)*. After such a correction the Client-Server protocol has been successfully verified [23].

7 Discussion and Further Work

Two case studies presented here highlighted the role of supercompilation it may play both in proving the programs and disproving them by guided tests.

If the main proof procedure fails to prove the correctness the form of the residual program may lead to the tests and the bug discovery. As the second case study has demonstrated it may even lead to discovery of inadequacy of a specification, revealing *implicit* assumptions which are violated.

In the case studies we used the supercompiler SCP4 interactively. An important further development would be to automate the test generation as far as possible. The aim of using the supercompiler in this context would be to narrow the search space of potential tests. For this purpose, control of the *folding strategy*, available in SCP4, can be used. To get a description of *potential* test cases in the syntax of the residual program one may forbid folding applying to n consecutive steps. Increasing parameter n would initiate a search for longer potential test cases.

On a practical side, we have tested this approach on the incorrect specification of the Xerox PARC Dragon protocol and, indeed, the test witnessing incorrectness was generated (without any assumption on the length of the trace!). It is clear though that, in general, one can not avoid exponential blow up in the search for incorrect traces of increasing length, so interactive mode may be a viable choice here.

The class of the programs to which we have applied our verification and testing method admittedly quite restricted. Likewise, the properties which can be proved or disproved by the method (as described above) belong to the class of *safety* properties. Another important direction for further work is to extend the approach to wider classes of programs and properties.

Notice that the verification of the parameterized Xerox PARC Dragon protocol as well as of others cache coherence protocols has already been done by using counting abstraction, constraint-based methods and symbolic model checking e.g. in [7, 9]. The applicability of parameterized testing and supercompilation

[4] Mea culpa!

scheme presented here potentially is much wider, but it remains to be convincingly demonstrated.

The applications of the discussed methods are restricted at the moment to programs and correctness conditions expressed in the supercompiler's input language Refal. Yet another interesting development here would be to use the remarkable features of supercompilation as a specializing procedure, capable in some cases of eliminating a level of interpretation. If one would like to apply supercompilation based verification and testing methods to programs written in a different language L one may proceed as follows. First, implement an interpreter of L in Refal and then apply the verification procedure based on supercompilation to the Refal program executing the interpreter on a particular L program to be verified. We anticipate that, given enough effort in developing a suitable interpreter, the supercompiler would be able to eliminate (to some extent, at least) the overhead of interpretation, effectively reducing verification of L programs to the verification of Refal programs.

References

1. P. A. Abdulla, B. Jonsson, M. Nilsson, J. d'Orso, and M. Saksena. Regular Model Checking for LTL(MSO). In *Proc. 16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 348–360. Springer, 2004.
2. P. A. Abdulla, B. Jonsson, A. Rezine, and M. Saksena. Proving Liveness by Backwards Reachability. In *Proc. CONCUR*, volume 4137 of *LNCS*, pages 95–109. Springer, 2006.
3. K.-T. Cheng and A.S. Krishnakumar, Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. ACM Transactions on Design Automation of Electronic Systems 1(1):57–79,1996
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Dec. 1999.
5. G. Delzanno, *Automatic Verification of Parameterized Synchronous Systems.* `http://www.disi.unige.it/person/DelzannoG/protocol.html`
6. G. Delzanno. Automatic Verification of Parameterized Cache Coherence Protocols, 2000, 27pp, available online at `ftp://ftp.disi.unige.it/person/DelzannoG/papers/ccp.ps.gz`
7. G. Delzanno. Automatic Verification of Parameterised Cache Coherence Protocols. *Proc. of 13th Int. Conference on Computer Aided Verification* LNCS 1855, Springer Verlag, 2000, pp 53–68.
8. G. Delzanno. Verification of Consistency Protocols via Infinite-state Symbolic Model Checking, A Case Study. In *Proc. of FORTE/PSTV*, 2000, pp 171-188
9. G. Delzanno. Contsraint-based Verification of Paremeterized Cache Coherence Protocols. *Formal Methods in System Design* 23(3):257-301, 2003.
10. G. Delzanno and T. Bultan. Constraint-based Verification of Client-Server Protocols. In Proc. the 7th International Conference on Principles and Practice of Constraint Programming, LNCS, Vol. 2239, pp. 286-301, Springer-Verlag, 2001.
11. P. Dybier and H. P. Sander, A Functional Programming Approach to the Specification and Verification of Concurrent Systems. *Formal Aspects of Computing*, 1:303–319
12. R. Glück and M. Leuschel, Abstraction-based partial deduction for solving inverse problems – a transformational approach to software verification. In *Proceedings of Systems Informatics*, LNCS 1755, pp:93-100, Novosibirsk, Russia, 1999. Springer-Verlag.
13. G. W. Hamilton. Distilling Programs for Verification. In *Proceedings of the International Conference on Compiler Optimization Meets Compiler Verification*, Braga, Portugal, March 2007. To Appear in Electronic Notes in Theoretical Computer Science.
14. J. Handy. The Cache Memory Book. Academic Press, 1993.

15. M. Leuschel. On the Power of Homeomorphic Embedding for Online Termination. *Proceeding of the SAS'98*, LNCS 1503, Springer-Verlag, 1998.
16. M. Leuschel, H. Lehmann, Solving coverability problems of Petri nets by partial deduction. In *Proc. 2nd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming (PPDP'2000)*, Montreal, Canada, pp 268-279, 2000.
17. M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In A. Bossi, editor, *Logic-Based Program Synthesis and Transformation. Proceedings of LOPSTR'99*, LNCS 1817, pages 63-82, Venice, Italy, 2000.
18. A. Lisitsa, A. Nemytykh. Towards Verification via Supercompilation. In *Proc. of COMPSAC 05, the 29th Annual International Computer Software and Applications Conference, Workshop Papers and Fast Abstracts*, pages 9-10, IEEE, 2005.
19. A. Lisitsa, A. Nemytykh: Verification of parameterized systems using supercompilation. A case study, in *Proc. of the Third Workshop on Applied Semantics (APPSEM05)*, M. Hofmann, H.W. Loidl (Eds.) , Fraunchiemsee, Germany. Ludwig Maximillians Universitat Munchen. (2005), Accessible via: `ftp://www.botik.ru/pub/local/scp/refal5/appsem_verification2005.ps`
20. A. Lisitsa, A. Nemytykh, Verification as a Parameterized Testing (Experiments with the SCP4 Supercompiler). Programmirovanie. No.**1** (2007) (In Russian), pp: 22-34. English translation in J. Programming and Computer Software, Vol. **33**, No.1 (2007), pp: 14–23.
21. A. Lisitsa, A. Nemytykh, Reachability Analisys in Verification via Supercompilation. In *Proceedings of the Satellite Workshops of DTL'2007* , TUCS General Publication, No.**45**, Part 2, June 2007, pp: 53-67.
22. A. Lisitsa, A. Nemytykh, A Note on Specialization of Interpreters. In *Proceedings of The 2nd International Symposium on Computer Science in Russia (CSR-2007)*, LNCS 4649, pp 237-248.
23. A. Lisitsa, A. Nemytykh, Experiments on verification via supercompilation. `http://refal.botik.ru/protocols/`, 2007.
24. A. Lisitsa, A. Nemytykh, *Appendices to the paper:* Extracting bugs from the failed proofs in verification via supercompilation. `http://refal.botik.ru/protocols/TAP08_Appendices.zip`, 2008.
25. A. A. Markov and N. M. Nagorny, The Theory of Algorithms, Kluwer Academic Publishers, Dordrecht, (Translated from the Russian by M. Greendlinger), 1988.
26. A. Nemytykh: The Supercompiler SCP4: General Structure (extended abstract). In *Proc. of the Perspectives of System Informatics*, LNCS, **2890** (2003) 162–170, Springer-Verlag.
27. A. Nemytykh: Playing on REFAL. In *Proc. of the International Workshop on Program Understanding*, (2003) 29–39, A.P. Ershov Institute of Informatics Systems, Syberian Branch of Russian Academy of Sciences. Accessible via: `ftp://www.botik.ru/pub/local/scp/refal5/nemytykh_PU03.ps.gz`
28. A. P. Nemytykh: The Supercompiler SCP4: General Structure (in English), *Programmnye sistemy: teoriya i prilozheniya*, Vol. 1, pp.448-485. Moscow, Fizmatlit, 2004. (`ftp://ftp.botik.ru/pub/local/scp/refal5/GenStruct.ps.gz`).
29. A. Nemytykh: The Supercompiler SCP4: General Structure. Moscow, URSS, 2007. (A book in Russian).
30. A. P. Nemytykh, V. F. Turchin: The Supercompiler SCP4: sources, on-line demonstration, `http://www.botik.ru/pub/local/scp/refal5/`, (2000).
31. A. Roychoudhury, I.V. Ramakrishnan, Inductively Verifying Invariant Properties of Parameterized Systems, *Automated Software Engineering*, 11, pp. 101–139, 2004.
32. M. H. Sørensen, R. Glück, An algorithm of generalization in positive supercompilation. In *Logic Programming: Proceedings of the 1995 International Symposium*, pages 486–479. MIT Press, 1995.
33. V.F. Turchin, The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8:292–325, 1986.
34. V. F. Turchin: The use of metasystem transition in theorem proving and program optimization. In *Proc. the 7th Colloquium on Automata, Languages and Programming*, LNCS, Vol. **85** (1980) 645–657, Springer-Verlag
35. V. F. Turchin: Metacomputation: Metasystem transition plus supercompilation LNCS, Vol. 1110, pp.481-509 The Proc. of the PEPM'96. Springer-Verlag, 1996.
36. V. F. Turchin: Supercompilation: Techniques and results LNCS, Vol. 1181, pp.227-248 The *Proc. of Perspectives of System Informatics*. Springer-Verlag, 1996.

37. V. F. Turchin: Refal-5, Programming Guide and Reference Manual. Holyoke, Massachusetts. (1989) New England Publishing Co. Electronic version: `http://www.botik.ru/pub/local/scp/refal5/`, 2000).
38. V. F. Turchin, D. V. Turchin, A. P. Konyshev, A. P. Nemytykh: Refal-5: sources, executable modules. `http://www.botik.ru/pub/local/scp/refal5/`, 2000.

Bisher erschienen

Arbeitsberichte aus dem Fachbereich Informatik

(http://www.uni-koblenz.de/fb4/publikationen/arbeitsberichte)

Bernhard Beckert, Reiner Hähnle, Tests and Proofs: Papers Presented at the Second International Conference, TAP 2008, Prato, Italy, April 2008, Arbeitsberichte aus dem Fachbereich Informatik 5/2008

Klaas Dellschaft, Steffen Staab, Unterstützung und Dokumentation kollaborativer Entwurfs- und Entscheidungsprozesse, Arbeitsberichte aus dem Fachbereich Informatik 4/2008

Rüdiger Grimm: IT-Sicherheitsmodelle, Arbeitsberichte aus dem Fachbereich Informatik 3/2008

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik 2/2008

Markus Maron, Kevin Read, Michael Schulze: CAMPUS NEWS – Artificial Intelligence Methods Combined for an Intelligent Information Network, Arbeitsberichte aus dem Fachbereich Informatik 1/2008

Lutz Priese,Frank Schmitt, Patrick Sturm, Haojun Wang: BMBF-Verbundprojekt 3D-RETISEG Abschlussbericht des Labors Bilderkennen der Universität Koblenz-Landau, Arbeitsberichte aus dem Fachbereich Informatik 26/2007

Stephan Philippi, Alexander Pinl: Proceedings 14. Workshop 20.-21. September 2007 Algorithmen und Werkzeuge für Petrinetze, Arbeitsberichte aus dem Fachbereich Informatik 25/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS – an Intelligent Bluetooth-based Mobile Information Network, Arbeitsberichte aus dem Fachbereich Informatik 24/2007

Ulrich Furbach, Markus Maron, Kevin Read: CAMPUS NEWS - an Information Network for Pervasive Universities, Arbeitsberichte aus dem Fachbereich Informatik 23/2007

Lutz Priese: Finite Automata on Unranked and Unordered DAGs Extented Version, Arbeitsberichte aus dem Fachbereich Informatik 22/2007

Mario Schaarschmidt, Harald F.O. von Kortzfleisch: Modularität als alternative Technologie- und Innovationsstrategie, Arbeitsberichte aus dem Fachbereich Informatik 21/2007

Kurt Lautenbach, Alexander Pinl: Probability Propagation Nets, Arbeitsberichte aus dem Fachbereich Informatik 20/2007

Rüdiger Grimm, Farid Mehr, Anastasia Meletiadou, Daniel Pähler, Ilka Uerz: SOA-Security, Arbeitsberichte aus dem Fachbereich Informatik 19/2007

Christoph Wernhard: Tableaux Between Proving, Projection and Compilation, Arbeitsberichte aus dem Fachbereich Informatik 18/2007

Ulrich Furbach, Claudia Obermaier: Knowledge Compilation for Description Logics, Arbeitsberichte aus dem Fachbereich Informatik 17/2007

Fernando Silva Parreiras, Steffen Staab, Andreas Winter: TwoUse: Integrating UML Models and OWL Ontologies, Arbeitsberichte aus dem Fachbereich Informatik 16/2007

Rüdiger Grimm, Anastasia Meletiadou: Rollenbasierte Zugriffskontrolle (RBAC) im Gesundheitswesen, Arbeitsberichte aud dem Fachbereich Informatik 15/2007

Ulrich Furbach, Jan Murray, Falk Schmidsberger, Frieder Stolzenburg: Hybrid Multiagent Systems with Timed Synchronization-Specification and Model Checking, Arbeitsberichte aus dem Fachbereich Informatik 14/2007

Björn Pelzer, Christoph Wernhard: System Description:“E-KRHyper“, Arbeitsberichte aus dem Fachbereich Informatik, 13/2007

Ulrich Furbach, Peter Baumgartner, Björn Pelzer: Hyper Tableaux with Equality, Arbeitsberichte aus dem Fachbereich Informatik, 12/2007

Ulrich Furbach, Markus Maron, Kevin Read: Location based Informationsystems, Arbeitsberichte aus dem Fachbereich Informatik, 11/2007

Philipp Schaer, Marco Thum: State-of-the-Art: Interaktion in erweiterten Realitäten, Arbeitsberichte aus dem Fachbereich Informatik, 10/2007

Ulrich Furbach, Claudia Obermaier: Applications of Automated Reasoning, Arbeitsberichte aus dem Fachbereich Informatik, 9/2007

Jürgen Ebert, Kerstin Falkowski: A First Proposal for an Overall Structure of an Enhanced Reality Framework, Arbeitsberichte aus dem Fachbereich Informatik, 8/2007

Lutz Priese, Frank Schmitt, Paul Lemke: Automatische See-Through Kalibrierung, Arbeitsberichte aus dem Fachbereich Informatik, 7/2007

Rüdiger Grimm, Robert Krimmer, Nils Meißner, Kai Reinhard, Melanie Volkamer, Marcel Weinand, Jörg Helbach: Security Requirements for Non-political Internet Voting, Arbeitsberichte aus dem Fachbereich Informatik, 6/2007

Daniel Bildhauer, Volker Riediger, Hannes Schwarz, Sascha Strauß, „grUML – Eine UML-basierte Modellierungssprache für T-Graphen“, Arbeitsberichte aus dem Fachbereich Informatik, 5/2007

Richard Arndt, Steffen Staab, Raphaël Troncy, Lynda Hardman: Adding Formal Semantics to MPEG-7: Designing a Well Founded Multimedia Ontology for the Web, Arbeitsberichte aus dem Fachbereich Informatik, 4/2007

Simon Schenk, Steffen Staab: Networked RDF Graphs, Arbeitsberichte aus dem Fachbereich Informatik, 3/2007

Rüdiger Grimm, Helge Hundacker, Anastasia Meletiadou: Anwendungsbeispiele für Kryptographie, Arbeitsberichte aus dem Fachbereich Informatik, 2/2007

Anastasia Meletiadou, J. Felix Hampe: Begriffsbestimmung und erwartete Trends im IT-Risk-Management, Arbeitsberichte aus dem Fachbereich Informatik, 1/2007

„Gelbe Reihe“

(http://www.uni-koblenz.de/fb4/publikationen/gelbereihe)

Lutz Priese: Some Examples of Semi-rational and Non-semi-rational DAG Languages. Extended Version, Fachberichte Informatik 3-2006

Kurt Lautenbach, Stephan Philippi, and Alexander Pinl: Bayesian Networks and Petri Nets, Fachberichte Informatik 2-2006

Rainer Gimnich and Andreas Winter: Workshop Software-Reengineering und Services, Fachberichte Informatik 1-2006

Kurt Lautenbach and Alexander Pinl: Probability Propagation in Petri Nets, Fachberichte Informatik 16-2005

Rainer Gimnich, Uwe Kaiser, and Andreas Winter: 2. Workshop "Reengineering Prozesse" – Software Migration, Fachberichte Informatik 15-2005

Jan Murray, Frieder Stolzenburg, and Toshiaki Arai: Hybrid State Machines with Timed Synchronization for Multi-Robot System Specification, Fachberichte Informatik 14-2005

Reinhold Letz: FTP 2005 – Fifth International Workshop on First-Order Theorem Proving, Fachberichte Informatik 13-2005

Bernhard Beckert: TABLEAUX 2005 – Position Papers and Tutorial Descriptions, Fachberichte Informatik 12-2005

Dietrich Paulus and Detlev Droege: Mixed-reality as a challenge to image understanding and artificial intelligence, Fachberichte Informatik 11-2005

Jürgen Sauer: 19. Workshop Planen, Scheduling und Konfigurieren / Entwerfen, Fachberichte Informatik 10-2005

Pascal Hitzler, Carsten Lutz, and Gerd Stumme: Foundational Aspects of Ontologies, Fachberichte Informatik 9-2005

Joachim Baumeister and Dietmar Seipel: Knowledge Engineering and Software Engineering, Fachberichte Informatik 8-2005

Benno Stein and Sven Meier zu Eißen: Proceedings of the Second International Workshop on Text-Based Information Retrieval, Fachberichte Informatik 7-2005

Andreas Winter and Jürgen Ebert: Metamodel-driven Service Interoperability, Fachberichte Informatik 6-2005

Joschka Boedecker, Norbert Michael Mayer, Masaki Ogino, Rodrigo da Silva Guerra, Masaaki Kikuchi, and Minoru Asada: Getting closer: How Simulation and Humanoid League can benefit from each other, Fachberichte Informatik 5-2005

Torsten Gipp and Jürgen Ebert: Web Engineering does profit from a Functional Approach, Fachberichte Informatik 4-2005

Oliver Obst, Anita Maas, and Joschka Boedecker: HTN Planning for Flexible Coordination Of Multiagent Team Behavior, Fachberichte Informatik 3-2005

Andreas von Hessling, Thomas Kleemann, and Alex Sinner: Semantic User Profiles and their Applications in a Mobile Environment, Fachberichte Informatik 2-2005

Heni Ben Amor and Achim Rettinger: Intelligent Exploration for Genetic Algorithms – Using Self-Organizing Maps in Evolutionary Computation, Fachberichte Informatik 1-2005