

Computer Science at Kent

**Implementation and Application  
of Functional Languages  
19th International Symposium, IFL 2007**

Olaf Chitil (Ed.)

Freiburg, Germany, 27th-29th September 2007

Technical Report No. 12-07  
September 2007

---

Published by the Computing Laboratory,  
University of Kent, Canterbury, Kent, CT2 7NF, UK

## Preface

The 19th International Symposium on Implementation and Application of Functional Languages (IFL 2007) is held at Freiburg, Germany, on the 27th to the 29th September 2007. Local organiser is the Programming Languages Group of the Department of Computer Science of the University of Freiburg.

IFL brings together researchers active in the area of functional programming, with an emphasis on the implementation and application of the same. IFL provides an annual open forum for researchers who wish to present and discuss new ideas and concepts, work in progress, preliminary results, etc. IFL has been held throughout Europe in the Netherlands, United Kingdom, Germany, Sweden, Spain, Ireland and Hungary. This year for the first time IFL is co-located with the International Conference on Functional Programming (ICFP). A record number of 44 papers have been submitted for these draft proceedings. By the time of printing 73 researchers had registered for attendance at the symposium.

Following tradition, two proceedings are to be published: the draft proceedings used at the symposium (this document), released as a technical report of the Computing Laboratory of the University of Kent, and the post-symposium proceedings based on revised papers. The draft proceedings are un-refereed and provide a useful reference to the delegates at the symposium. All participants who give talks at the symposium are invited to submit revised papers for review after the symposium, to normal conference standards. The post-symposium proceedings of selected revised papers will be published by Springer-Verlag in its Lecture Notes in Computer Science (LNCS) series.

Olaf Chitil  
Programme Chair  
University of Kent  
September 2007

## Local Organisers

Markus Degen  
Peter Thiemann  
Stefan Wehr

Supported by Deutsche Forschungsgemeinschaft (DFG)

## Table of Contents

Termination and Complexity Bounds for SAFE programs .....	8
<i>Salvador Lucas, Ricardo Peña</i>	
Graph Parser Combinators .....	24
<i>Steffen Mazanek, Mark Minas</i>	
Encoding Iterators in Interaction Nets .....	40
<i>José Almeida, Ian Mackie, Jorge Sousa Pinto, Miguel Vilaça</i>	
Testing Erlang Refactorings with QuickCheck .....	55
<i>Huiqing Li, Simon Thompson</i>	
Call Graphs, Dominator Trees, and Lambda Lifting .....	71
<i>Marco T. Morazan, Ulrik Schultz</i>	
To Be or Not to Be ... Lazy .....	89
<i>Mercedes Hidalgo-Herrero, Yolanda Ortega-Mallén</i>	
The Structure of the Essential Haskell Compiler, or Coping with Compiler Complexity .....	107
<i>Atze Dijkstra, Jeroen Fokker, Doaitse Swierstra</i>	
XHaskell — Adding Regular Expression Types to Haskell .....	123
<i>Martin Sulzmann, Kenny Zhuo Ming Lu</i>	
Evaluating and Using a Grid-Enabled Parallel Haskell .....	139
<i>Phil Trinder, Abyd Al Zain, Kevin Hammond</i>	
Partial Parsing: Combining Choice with Commitment .....	140
<i>Malcolm Wallace</i>	
Functional Master-Worker Skeletons .....	152
<i>Jost Berthold, Mischa Dieterle, Rita Loogen, Steffen Priebe</i>	
Towards an Implementation of a Computer Algebra System in a Functional Programming Language .....	168
<i>Oleg Lobachev</i>	
Lazy Contract Checking for Immutable Data Structures .....	179
<i>Robert Bruce Findler, Shu-yu Guo, Anne Rogers</i>	
Haskell – Join – Rules .....	195
<i>Martin Sulzmann, Edmund Lam</i>	
Splitting and Merging Program Refactorings .....	211
<i>Christopher Brown, Simon Thompson</i>	
An Interpretation of Temporal Properties in Functional Programs .....	224
<i>Máté Tejfel, Tamás Kozsik, Zoltán Horváth</i>	

Approaches to Subtyping in Functional Languages .....	229
<i>Glenn Strong</i>	
On the Validation of Specifications used in Model-Based Testing .....	230
<i>Pieter Koopman, Peter Achten, Rinus Plasmeijer</i>	
Car Damage Subrogation Workflow — an iTask exercise .....	232
<i>Erik Zuurbier, Rinus Plasmeijer</i>	
Towards Open Type Functions for Haskell .....	233
<i>Tom Schrijvers, Martin Sulzmann, Simon Peyton Jones, Manuel Chakravarty</i>	
Transparent Ajax and Client-Site Evaluation of iTasks .....	252
<i>Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, Peter Achten</i>	
Static Inference of Non-Monotonic Polynomial Sized Types .....	254
<i>Marko van Eekelen, Olha Shkaravska</i>	
Efficient, Modular Tries .....	258
<i>Frank Huch, Sebastian Fischer</i>	
FunSETL — Functional Reporting for ERP Systems .....	268
<i>Michael Nissen, Ken Friis Larsen</i>	
The Reduceron: Widening the von Neumann Bottleneck for Graph Reduction using an FPGA .....	290
<i>Matthew Naylor, Colin Runciman</i>	
Incremental Extension of a Domain Specific Language Interpreter .....	301
<i>Olivier Michel, Jean-Louis Giavitto</i>	
Generic Programming Combinators .....	318
<i>Sebastian Fischer, Frank Huch</i>	
Supero: Making Haskell Faster .....	334
<i>Neil Mitchell, Colin Runciman</i>	
Checking Dependent Types Efficiently .....	350
<i>Dirk Kleebblatt</i>	
HW-Hume in Isabelle .....	366
<i>Chunxu Liu, Greg Michaelson</i>	
Static Contract Checking for Haskell .....	382
<i>Dana Na Xu, Simon Peyton Jones, Koen Claessen</i>	
Debugging Lazy Functional Programs by Asking the Oracle .....	400
<i>Bernd Braßel, Holger Siegel</i>	
Uniqueness Typing Simplified .....	416
<i>Edsko de Vries, Rinus Plasmeijer, David Abrahamson</i>	
Tabular Expressions and Total Functional Programming .....	431
<i>Baltasar Trancón y Widemann, David L. Parnas</i>	

Positive Supercompilation for a Higher Order Call-By-Value Language .....	441
<i>Peter Jonsson, Johan Nordlander</i>	
The Simple Category of Modules .....	457
<i>Mikolaj Konarski</i>	
Polytopes & Polytypes: Generic Isosurfacing & Functional Programming .....	474
<i>Colin Runciman, David Duke, Rita Borgo, Malcolm Wallace</i>	
Meta⟨Fun⟩ — Towards a Functional-Style Interface for C++ Template Metaprograms .....	489
<i>Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, Viktória Zsók</i>	
Speculative Inlining of Predefined Procedures in an R5RS Scheme to C Compiler ..	503
<i>Marc Feeley</i>	
Circuit Parallelism in Haskell Programs .....	519
<i>Andreas Koltès, John O’Donnell</i>	
On Implementing S-Net .....	531
<i>Clemens Grelck, Frank Penczek</i>	
From Contracts Towards Dependent Types: Proofs by Partial Evaluation .....	534
<i>Stephan Herhut, Sven-Bodo Scholz, Robert Bernecky, Clemens Grelck, Kai Trojahn-ner</i>	
A Rational Simplifier for GHC .....	551
<i>Laszlo Nemeth</i>	
Amortizing the Cost of Commuting Conversions when Beta-Reducing Monadic Normal Forms and A-Normal Forms .....	552
<i>Olivier Danvy</i>	

## Index

- Abrahamson, David, 416  
Achten, Peter, 230, 252  
Al Zain, Abyd, 139  
Almeida, Jose, 40
- Bernecky, Robert, 534  
Berthold, Jost, 152  
Borgo, Rita, 474  
Brassel, Bernd, 400  
Brown, Christopher, 211
- Chakravarty, Manuel, 233  
Claessen, Koen, 382
- Danvy, Olivier, 552  
de Vries, Edsko, 416  
Dieterle, Mischa, 152  
Dijkstra, Atze, 107  
Duke, David, 474
- Feeley, Marc, 503  
Findler, Robert Bruce, 179  
Fischer, Sebastian, 258, 318  
Fokker, Jeroen, 107
- Giavitto, Jean-Louis, 301  
Grelck, Clemens, 531, 534  
Guo, Shu-yu, 179
- Hammond, Kevin, 139  
Herhut, Stephan, 534  
Hidalgo-Herrero, Mercedes, 89  
Horvath, Zoltan, 224  
Huch, Frank, 258, 318
- Jansen, Jan Martin, 252  
Jonsson, Peter, 441
- Kleeblatt, Dirk, 350  
Koltès, Andreas, 519  
Konarski, Mikolaj, 457  
Koopman, Pieter, 230, 252  
Kozsik, Tamás, 224
- Lam, Edmund, 195  
Larsen, Ken Friis, 268  
Li, Huiqing, 55  
Liu, Chunxu, 366  
Lobachev, Oleg, 168  
Loogen, Rita, 152  
Lu, Kenny Zhuo Ming, 123  
Lucas, Salvador, 8
- Mackie, Ian, 40  
Mazamek, Steffen, 24  
Michaelson, Greg, 366  
Michel, Olivier, 301  
Minas, Mark, 24  
Mitchell, Neil, 334  
Morazan, Marco T., 71
- Naylor, Matthew, 290  
Nemeth, Lazlo, 551  
Nissen, Michael, 268  
Nordlander, Johan, 441
- O'Donnell, John, 519  
Ortega-Mallen, Yolanda, 89
- Parnas, David L., 431  
Pataki, Norbert, 489  
Pena, Ricardo, 8  
Penczek, Frank, 531  
Peyton Jones, Simon, 233, 382  
Pinto, Jorge Sousa, 40  
Plasmeijer, Rinus, 230, 232, 252, 416  
Porkolab, Zoltan, 489  
Priebe, Steffen, 152
- Rogers, Anne, 179  
Runciman, Colin, 290, 334, 474
- Scholz, Sven-Bodo, 534  
Schrijvers, Tom, 233  
Schultz, Ulrik, 71  
Shkaravska, Olha, 254

Siegel, Holger, 400  
Sipos, Adam, 489  
Strong, Glenn, 229  
Sulzmann, Martin, 123, 195, 233  
Swierstra, Doaitse, 107

Tejfel, Máté, 224  
Thompson, Simon, 55, 211  
Trancón y Widemann, Baltasar, 431  
Trinder, Phil, 139  
Trojahner, Kai, 534

van Eekelen, Marko, 254  
Vilaca, Miguel, 40

Wallace, Malcolm, 140, 474

Xu, Dana Na, 382

Zsok, Viktoria, 489  
Zuurbier, Erik, 232

# On the Validation of Specifications used in Model-Based Testing

Pieter Koopman, Peter Achten, and Rinus Plasmeijer

Software Technology, Nijmegen Institute for Computing and Information Sciences,  
Radboud University Nijmegen, The Netherlands  
{pieter, p.achten, rinus}@cs.ru.nl

**Abstract.** In model-based testing the behavior of a system under test, *sut*, is compared automatically with the behavior of the specification. A significant fraction of issues found in testing appear to be caused by problems with the specification. In order to ensure that the specification prescribes the desired behavior, it has to be *validated* by a human. In this work we introduce a tool to support this validation. In addition to an interactive simulator of the specification, the tool is able to generate transition tables and diagrams of the observed behavior. In order to make simulation and the displaying of the observed behavior finite, we introduce equivalence of states, inputs and outputs.

## Extended Abstract

In model-based testing the behavior of a system under test, *sut*, is compared automatically with the behavior of the specification. The specification is a state transition system that can be nondeterministic. Usually the number of states, inputs and outputs possible is infinite. The *sut* is also assumed to be a state transition system, but its state is hidden. One can only apply input to the system and observe the corresponding output. We have used model-based testing successfully to improve controllers, protocols, javacard applets and more.

For this comparison of behavior, the test system takes a specification and executes a user defined number of traces. For each trace the *sut* and the specification starts in their initial states. The test system selects an input that is covered by the specification, applies this input to the *sut*, and computes the allowed states of the specification. If no states are possible for the specification the *sut* has shown behavior that is not covered by the specification. The testers say that an *issue* is found.

Ideally, each issue indicates an error in the *sut*. However, in practice a significant fraction of issues appear to be caused by problems with the specification: the specification does not correctly capture the intentions of the users and the *sut* does something different. Although the fraction of issues caused by the specification differs with the kind of system and the amount of effort put in the correctness of the specification, we estimate that on average in about 25% of the issues found in model-based testing one has to blame the specification.



Incorrect specifications are a problem for several reasons. First, if an issue is found it is not clear whether we have to blame the specification or the sut. Finding and correcting errors in the specification takes time during the test phase of the project. This is not the right moment to create a correct specification. In many projects there is a significant time pressure during the testing phase of a system. Second, only behavior that is implemented differently by the sut can cause issues. All other errors in the specification are not found at all during model-based testing. Third, any change in the specification during the testing phase can cause significant implementation changes to the sut. Finally, any change in the specification invalidates in principle all previous test results (just like any change in the sut). This implies that errors in the specification can be very expensive and it is worthwhile to invest effort to ensure the quality of the specification.

In our model-based test system `Gvst` we use the functional language `Clean` as specification language. Due to the high abstraction level of this language it is possible to write concise specifications which contributes to their quality. The `Clean` compiler will check quality aspects like type correctness and consistent definition of identifiers used. We have shown that quality aspects such as the reachability of states, determinism and completeness of the specification, and the preservation of constraints can be checked by systematic testing.

However, this does not rule out the possibility that the specification prescribes the wrong behavior in a consistent way. In order to ensure that the specification prescribes the desired behavior, it has to be *validated* by a human. In this work we introduce tools to support this validation. First, a simulator enables the user to execute the specification. Such an interactive execution can be much more illustrative than looking at the code of the specification. Second, it is possible to record the traces of the specification executed in the simulator. The states visited and their transitions can be visualized in a table or a state transition diagram. Since the number of states, inputs and outputs can be infinite and different in each and every specification, this is not straightforward. The key to the solution is an operator to define equivalence of states, inputs and outputs. For instance, values that are handled by the same symbolic transition in the specification (function alternative in the specifying function) are usually considered to be equivalent. All states that are considered equivalent can be mapped to the same entry of the table or the same place in the transition diagram. Since the equivalence of values is problem dependent, some human input is required to define equivalence.