dco/c++ – Derivative Code by Overloading in C++

Johannes Lotz, Klaus Leppkes, and Uwe Naumann
The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/
**Abstract.** Algorithmic Differentiation is a widely used technique for computing derivatives of implemented mathematical functions. In this technical report we want to introduce the tool dco/c++ written by the authors implementing algorithmic differentiation by overloading for C/C++. After a general picture of the capabilities dco/c++ provides, we show its performance with emphasis on adjoint computations in comparison to four other freely available overloading tools.

1 Introduction

dco/c++ is a development at the institute *Software and Tools for Computational Engineering* implementing Algorithmic Differentiation (AD) [4] by overloading in C++. The range of capabilities covered by dco/c++ is driven by different applications and research subjects. Current projects are in the area of financial engineering, atmospheric physics, or fluid mechanics. dco/c++ is also used in research on the generation of discrete adjoints using parallel environments, in particular OpenMP and MPI. The objective is to provide an efficient and robust tool for the computation of projections of derivatives of arbitrary order of a function given as an implementation in C/C++. Additionally, the capability of coupling the robust overloading technique with optimized computer generated or hand-written external computations of adjoint projections is provided.

During various collaborative research and development projects, we were able to compute fast adjoints for real world applications. For example, we provided an adjoint version of the Juelich Rapid Spectral Simulation Code Version 2 [9] written at the Institute of Energy and Climate Research – Stratosphere at Research Center Jülich, which solves a large-scale inverse problem with gradient-based methods. Here we were able to achieve a factor of roughly 3.5 for the ratio

\[ R = \frac{\text{Run time of one adjoint computation}}{\text{Run time of one function evaluation}}. \]

In Section 4 we will compare \( R \) for different, freely available AD tools.

2 Algorithmic Differentiation

For a given implementation of the function \( F : \mathbb{R}^n \to \mathbb{R}^m \) as a computer program AD enables us to compute derivatives of arbitrary order with an accuracy up to machine precision. Derivatives are meant as the sensitivities \( \frac{\partial y}{\partial x} \) of the output \( y \in \mathbb{R}^m \) with respect to the inputs \( x \in \mathbb{R}^n \), \( y = F(x) \), yielding the Jacobian \( \nabla F(x) \). This is done either via *source code transformation* or *operator overloading* techniques - or via a coupling of both. AD offers two fundamental modes, the
tangent-linear or forward mode and the adjoint or reverse mode. The tangent-linear mode computes tangent-linear projections\(^1\)

\[
y^{(1)} = \nabla F(x) \cdot x^{(1)}
\]  \hspace{1cm} (1)

during one augmented (forward) function evaluation. The adjoint mode computes adjoint projections

\[
x^{(1)} = \nabla F(x)^T \cdot y^{(1)}
\]  \hspace{1cm} (2)

and requires a data-flow reversal. Therefore required data is saved with the forward section and used by the reverse section of the adjoint code. \texttt{dco/c++} introduces new data types and an additional data structure – the \textit{tape} – that records data and dependency information in the forward section used in the reverse section (tape interpretation). Most overloading tools follow a similar strategy.

For more detailed information about Algorithmic Differentiation refer to the text books [4, 3].

3 Capabilities of \texttt{dco/c++}

In this section we present (from the user’s perspective) the capabilities of \texttt{dco/c++}. This part is intentionally kept brief. Refer to the \texttt{dco/c++} user documentation or feel free to contact the authors for further information.

3.1 Basic Data Types for First- and Higher-Order Derivatives

\texttt{dco/c++} provides data types for computing first derivatives in forward and reverse modes. All types and functions are defined in the namespace \texttt{dco}. Additionally, every mode is defined as nested namespaces. The data type is written as a template and can be instatiated with arbitrary base data types (default is \texttt{double}).

For computing the tangent-linear projection of first order, the data type

\[
\texttt{dco::t1s::type}
\]

can be used. \texttt{t1s} corresponds to tangent-linear projection of \texttt{1st-order scalar type}. Furthermore, a vector mode can be used with a vector size defined by the user at compile time. The data type to be used is

\[
\texttt{dco::t1v::type},
\]

where \texttt{t1v} corresponds to tangent-linear projection of \texttt{1st-order vector type}.

For computing the adjoint projection of first order, the data type

\[
\texttt{dco::a1s::type}
\]

can be used. \texttt{a1s} corresponds to adjoint projection of \texttt{1st-order scalar type}. A tape is used. The memory allocation strategy can be chosen to be static or dynamic (\textit{chunk tape}), which results in one large allocation at the beginning or dynamic reallocation during tape recording, respectively.

\(^1\) The following notation is taken from [4].
**dco/c++** supports derivatives of arbitrary order by nesting of the basic data types. For second-order types, we have predefined namespaces

\[
dco::t2s_als::\text{type}, \quad dco::t2v_als::\text{type} \quad \text{and} \quad dco::t2s_tls::\text{type}
\]
for forward-over-reverse mode in scalar or vector fashion and forward-over-forward mode, respectively.

### 3.2 Expression Templates and Statement-Level Preaccumulation

*Expression templates* are used to implement *statement-level preaccumulation*. Expression templates are a C++ template metaprogramming technique in which templates are used to represent part of an expression. Expression templates are heavily used in **dco/c++**. Internally **dco/c++** holds two kinds of data types. First, the active type, which represents a program variable occurring on the left- or right-hand sides of an expression (instantiations are the data types introduced in Section 3.1). Second, the intermediate variables, which are compiler generated from all operations like +, −, sin, pow, ... occurring only on the right-hand side. This yields advantages in the forward vector mode as well as in reverse mode AD. In both cases the gains are achieved by statement-level preaccumulation [3], i.e., a preaccumulation of the local gradients of each assignment (an occurrence of '='). The gradient entries are the partial derivatives of the left-hand side program variable w. r. t. all program variables occurring on the right-hand side.

In forward vector mode AD the naive approach generates intermediate tangent vectors for each operation. This requires a huge amount of allocations and deallocations as well as many operations. With statement-level preaccumulation, we achieve remarkable improvements in the computational efficiency.

In reverse mode AD we can omit the generation of tape entries for each intermediate variable. From the local statement gradients we only save the partial derivatives (as *edge weights* in the linearized computational graph – see [3]). Thereby the forward section becomes more expensive. However, the tape size can be reduced significantly yielding less memory consumption and a more efficient reverse section (tape interpretation).

### 3.3 Thread Safety and OpenMP

Tangent-linear types of any order have only local data dependencies, so they are thread safe by default. In adjoint mode, there are two ways to guarantee thread safety in **dco/c++**. First of all, one can switch to multiple tape mode, which means that each thread has its own tape. This implementation requires no global data and also has only local data dependency. Therefore no synchronisation of the threads is needed, which results in good speed-ups.\(^2\) Nevertheless, for the multiple tape approach, manual effort is needed as well as a deeper understanding of adjoint dependencies is required.

There is also a black-box thread/OpenMP-safe mode implemented, where all threads share one global tape. **dco/c++** internally takes care about multiple write accesses in order to generate one consistent global tape. If multiple threads need to access the global tape at the same time, then this normally results in

\(^2\) Note that the overall memory performance still can be a limiting factor.
a slow-down. Nevertheless, due to the statement-level preaccumulation `dco/c++` needs only one update for each statement as opposed to every operation. So for expensive right-hand sides a speed-up is possible. A small case study can be found in Appendix A.

`dco/c++` has also an interface to the adjoint MPI library (AMPI) [8] (see Appendix B).
4 Case Study – Run time and Memory Consumption

In this section we analyse run time and memory for computing adjoint projections for a case study. The latter is done indirectly by limiting the available memory the tools are allowed to allocate. The results for dco/c++ are shown in comparison to different AD-tools listed on the webpage www.autodiff.org, namely ADOL-C\(^3\), CppAD\(^4\), FADBAD++\(^5\) and Sacado\(^6\). To the best of our knowledge, no significant improvement can be achieved in terms of performance unless major manual intervention by highly educated users of the respective AD-tools is performed. We invite developers/users of these tools to verify this claim in order to ensure best possible comparability. In the current setup we assume users with basic knowledge in AD following the available user documentations of the respective tools.

As the case study we use an in-house implemented code for the simulation of a 3D unsteady incompressible flow. The original Fortran code was written by Johannes Lotz at the Institute of Meteorology and Climatology, University Hanover, numerically identical to PALM \(^6\) and reimplemented in C++ by the authors. The equations are given by the Boussinesq-approximation coupling momentum, energy, and mass conservation.

We use a rectangular domain with equidistant Arakawa-C staggered grid \(^1\). A SIMPLE (Semi-Implicit Method for Pressure-Linked Equations) algorithm is

---

\(^3\) http://www.coin-or.org/projects/ADOL-C.xml
\(^4\) http://www.coin-or.org/CppAD/
\(^5\) http://www.fadbad.com/fadbad.html
\(^6\) http://trilinos.sandia.gov/packages/sacado/
used with an explicit Euler time stepping scheme, the Piascek-Williams scheme [5] for the spacial discretization of the advection term and central finite differences for spacial discretization of the diffusion term. The linear equation system is solved by a matrix-free SOR (successive over relaxation) solver, which is the computationally most expensive part [2].

The computational domain of our test case is a cube (3D) or square (2D) with identical side lengths ($1 \times 1 \times 1$ m$^3$ or $1 \times 1$ m$^2$, respectively). The test case is shown in Fig. 1 after 1.5 s of simulation time from zero initial condition. We have a fixed inflow (Dirichlet boundary for velocity) at the lower left and the outflow (Dirichlet boundary for pressure) at the lower right yielding a large eddy in the upper part. The 3D test case setup is obtained by inducing cyclic boundary conditions in the third dimension.

4.1 Run time and Memory Analysis

The run time is measured for a computation of two time steps in the simulation. The passive run time corresponds to the computation of function values only, while the tools have the challenge to compute the gradient of a single output with respect to the starting values (setup as an inverse problem). We show measurements for the 2D as well as the 3D test case.

![Fig. 2. Run time – comparison for the 2D test case.](image-url)

All measurements are made on a Linux system with gcc 4.6.1 and maximum optimization (-O4). The system is an Intel(R) Core(TM)2 Extreme CPU Q9300.
running at 2.53 GHz and 16 GB of RAM. The executables are limited to the use of maximum 10 GB of RAM defined by `ulimit -v 10000000`.

We plot the run times of one passive execution as well as for all the tools for the 2D and the 3D test case, respectively. We divide the measurements into the time used for the recording step (forward section), the interpretation step (reverse section) and the sum of both. `dco/c++` is used with a static tape and a chunk tape (see Section 3.1). The timing figures show the data in double logarithmic scale, which yields linear curves for a polynomial dependence of the run time $t$ on the problem size $n$ ($t(n) = a \cdot n^e$), where $n$ corresponds to the degrees of freedom of the problem and the length of the gradient. We observe the original problem as well as the gradient computation of all tools to be roughly polynomial with the same exponent $e$ (same slope in double logarithmic scale – see Fig. 2 and Fig. 4). The coefficient $a$ on the other hand is different for all tools (corresponds to different y-intercepts in double logarithmic scale). This yields the ratio of one adjoint projections to one passive function evaluation $R$ (see also Section 1) to be roughly a constant in the problem size $n$. This is shown in Fig. 3 and Fig. 5 in linear scale for the 2D and the 3D test cases, respectively. For `dco/c++` with chunk tape, a kink can be observed for $n \approx 10^{3.6}$. This can be explained by the reallocation of a new chunk for the tape. This, of course, has no effect on the reverse section.

The mean ratio $\bar{R} = \frac{1}{n} \cdot \sum_{\text{all problem sizes}} R(n)$ for the different tools is shown in Table 1. In light of the theoretical minimum of $R \approx 3$, `dco/c++` yields
Fig. 4. Run time – comparison for the 3D test case.

<table>
<thead>
<tr>
<th></th>
<th>dco/c++</th>
<th>dco/c++ chunk</th>
<th>ADOL-C</th>
<th>CppAD</th>
<th>FADBAD++</th>
<th>Sacado</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>8.0</td>
<td>12.3</td>
<td>38.1</td>
<td>30.0</td>
<td>51.7</td>
<td>67.7</td>
</tr>
<tr>
<td>3D</td>
<td>8.3</td>
<td>12.9</td>
<td>39.3</td>
<td>33.3</td>
<td>56.9</td>
<td>72.0</td>
</tr>
</tbody>
</table>

Table 1. The mean ratios $\overline{R}$ of one adjoint projection w.r.t. one passive function evaluation.
very good results. Of course, source code transformation code could achieve a better ratio, but in consideration of the wide range of the C++ language, the overloading technique is the only alternative. The benefits in the sense of robustness, applicability and maintainance over source code transformation approaches are beyond all question. Nevertheless, turning away from the black-box approach and coupling overloading techniques and source code transformation code can lead to the possibility of good applicability and even better efficiency.

Whenever a tool exceeds the memory limit of 10 GB, the corresponding curve ends. The corresponding problem sizes are shown in Table 2.

<table>
<thead>
<tr>
<th></th>
<th>dco/c++</th>
<th>ADOL-C</th>
<th>CppAD</th>
<th>FADBAD++</th>
<th>Sacado</th>
</tr>
</thead>
<tbody>
<tr>
<td>2D</td>
<td>29584</td>
<td>15376</td>
<td>29584</td>
<td>15376</td>
<td>13456</td>
</tr>
<tr>
<td>3D</td>
<td>37044</td>
<td>19652</td>
<td>37044</td>
<td>19652</td>
<td>13500</td>
</tr>
</tbody>
</table>

Table 2. Problem size \( n \) possible with 10 GB of RAM.

References


A Using OpenMP

dco/c++ offers a built-in OpenMP-support for single tape adjoint computation. Potential data races are taken care of internally. The more efficient alternative is, of course, handling the parallelization of adjoints explicitly using the multiple tape support of dco/c++ (see Section 3.3).

The built-in support exploits the parallelism during tape recording (forward section) while preaccumulating the local gradient of each statement (see Section 3.2). This yields better speed-up for right-hand sides with a greater work load. We show the speed-up of the code

```cpp
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    y[i] = exp(3 * sin(sin(x1[i]) + cos(x2[i]) + exp(x3[i]))) + atan(x4[i]);
}
```

with problem size n, input variables x1[n], x2[n], x3[n] and x4[n], and output variable y[n]. For the measurements we chose n=21E6 on a core-i5, 2.5 GHz with 4 cores. In Fig. 6(a) the speed-up for a passive function evaluation is shown as well as the speed-up of the overloaded function call. The latter is split into the speed-up of the forward section (tape recording – see Section 2) labeled with “dco/c++ tape recording” and the global speed-up including also the reverse section (tape interpretation). The tape interpretation step in the built-in OpenMP-support is currently not parallelized and therefore a constant offset to the speed-up. In Fig. 6(b) the ratios

\[ R_S = \frac{\text{Run time of one OpenMP accelerated adjoint computation}}{\text{Run time of one serial function evaluation}} \]

and

\[ R_P = \frac{\text{Run time of one OpenMP accelerated adjoint computation}}{\text{Run time of one OpenMP accelerated function evaluation}} \]

A Using OpenMP

dco/c++ offers a built-in OpenMP-support for single tape adjoint computation. Potential data races are taken care of internally. The more efficient alternative is, of course, handling the parallelization of adjoints explicitly using the multiple tape support of dco/c++ (see Section 3.3).

The built-in support exploits the parallelism during tape recording (forward section) while preaccumulating the local gradient of each statement (see Section 3.2). This yields better speed-up for right-hand sides with a greater work load. We show the speed-up of the code

```cpp
#pragma omp parallel for
for (int i = 0; i < n; ++i) {
    y[i] = exp(3 * sin(sin(x1[i]) + cos(x2[i]) + exp(x3[i]))) + atan(x4[i]);
}
```

with problem size n, input variables x1[n], x2[n], x3[n] and x4[n], and output variable y[n]. For the measurements we chose n=21E6 on a core-i5, 2.5 GHz with 4 cores. In Fig. 6(a) the speed-up for a passive function evaluation is shown as well as the speed-up of the overloaded function call. The latter is split into the speed-up of the forward section (tape recording – see Section 2) labeled with “dco/c++ tape recording” and the global speed-up including also the reverse section (tape interpretation). The tape interpretation step in the built-in OpenMP-support is currently not parallelized and therefore a constant offset to the speed-up. In Fig. 6(b) the ratios

\[ R_S = \frac{\text{Run time of one OpenMP accelerated adjoint computation}}{\text{Run time of one serial function evaluation}} \]

and

\[ R_P = \frac{\text{Run time of one OpenMP accelerated adjoint computation}}{\text{Run time of one OpenMP accelerated function evaluation}} \]
are shown. As the speed-up for the overloaded function is a little lower than the speed-up of the passive function, also the ratio $R_P$ gets a little worse with a growing number of cores. Nevertheless it is important to note that the ratio $R_S$ is falling below 1 (exactly 0.91), which means that the adjoint projection using the OpenMP-accelerated dco/c++ version is faster than a serial passive execution. The very good ratios are obtained due to the relatively complex right-hand side in the given example.

### B Using AdjointMPI

In the context of the case study presented in Section 4, we also implemented an MPI enabled version based on a domain decomposition of the given grid. At each update of the grid values, they are synchronized at the boundaries. This is achieved through non-blocking MPI calls. Finally, global values are synchronized by relying on MPI reduction routines.

As AD implies a complete data flow reversal this especially holds true for the MPI communication. Essentially, sending data in between processes amounts to assignments of variables across the processes. Therefore, the adjoint of MPI communication amounts to an incremental assignment for each communicated adjoint. This logic is handled by the Adjoint MPI library (AMPI) [8]. AMPI is coupled with dco/c++ internally through a well-defined interface. All MPI routines are to be replaced by the corresponding AMPI (Adjoint MPI) routines all named with a leading A (e.g. MPI_Init() → AMPI_Init()).

AMPI applied to our case study Section 4 is still work in progress in terms of scalability. Hence, we present here the run time results of a matrix multiplication based on an implementation of the Cannon algorithm without AD in Fig. 7(a) and with AD in Fig. 7(b). It is adjoined using dco/c++, AMPI and dcc[7].

---

(a) Speed-Ups of the passive and the overloaded function.

(b) Run time ratios $R_S$ and $R_P$.

**Fig. 6.** Speed-Ups and run time ratios.
Fig. 7. Run time for a Cannon matrix multiplication $C = A \cdot B$ with the size of the matrices set equal to $A(64,200000),B(200000,64)$ and $C(64,64)$. 
Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

http://aib.informatik.rwth-aachen.de/

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

2009-01 * Fachgruppe Informatik: Jahresbericht 2009
2009-02 Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Quantitative Model Checking of Continuous-Time Markov Chains Against Timed Automata Specifications
2009-03 Alexander Nyßen: Model-Based Construction of Embedded Real-Time Software - A Methodology for Small Devices
2009-05 George B. Mertzios, Ignasi Sau, Shmuel Zaks: A New Intersection Model and Improved Algorithms for Tolerance Graphs
2009-06 George B. Mertzios, Ignasi Sau, Shmuel Zaks: The Recognition of Tolerance and Bounded Tolerance Graphs is NP-complete
2009-07 Joachim Kneis, Alexander Langer, Peter Rossmanith: Derandomizing Non-uniform Color-Coding I
2009-08 Joachim Kneis, Alexander Langer: Satellites and Mirrors for Solving Independent Set on Sparse Graphs
2009-09 Michael Nett: Implementation of an Automated Proof for an Algorithm Solving the Maximum Independent Set Problem
2009-10 Felix Reidl, Fernando Sánchez Villaamil: Automatic Verification of the Correctness of the Upper Bound of a Maximum Independent Set Algorithm
2009-11 Kyriaki Ioannidou, George B. Mertzios, Stavros D. Nikolopoulos: The Longest Path Problem is Polynomial on Interval Graphs
2009-12 Martin Neuhausser, Lijun Zhang: Time-Bounded Reachability in Continuous-Time Markov Decision Processes
2009-13 Martin Zimmermann: Time-optimal Winning Strategies for Poset Games
2009-14 Ralf Huuck, Gerwin Klein, Bastian Schlich (eds.): Doctoral Symposium on Systems Software Verification (DS SSV’09)
2009-15 Joost-Pieter Katoen, Daniel Klink, Martin Neuhausser: Compositional Abstraction for Stochastic Systems
2009-16 George B. Mertzios, Derek G. Corneil: Vertex Splitting and the Recognition of Trapezoid Graphs
2009-17 Carsten Kern: Learning Communicating and Nondeterministic Automata
2009-18 Paul Hänsch, Michaela Slaats, Wolfgang Thomas: Parametrized Regular Infinite Games and Higher-Order Pushdown Strategies
2010-01 * Fachgruppe Informatik: Jahresbericht 2010
2010-02 Daniel Neider, Christof Löding: Learning Visibly One-Counter Automata in Polynomial Time
2010-03  Holger Krahn: MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering
2010-04  René Wörzberger: Management dynamischer Geschäftsprozesse auf Basis statischer Prozessmanagementsysteme
2010-05  Daniel Retkowitz: Softwareunterstützung für adaptive eHome-Systeme
2010-06  Taolue Chen, Tingting Han, Joost-Pieter Katoen, Alexandru Mereacre: Computing maximum reachability probabilities in Markovian timed automata
2010-07  George B. Mertzios: A New Intersection Model for Multitolerance Graphs, Hierarchy, and Efficient Algorithms
2010-08  Carsten Otto, Marc Brockschmidt, Christian von Essen, Jürgen Giesl: Automated Termination Analysis of Java Bytecode by Term Rewriting
2010-09  George B. Mertzios, Shmuel Zaks: The Structure of the Intersection of Tolerance and Cocomparability Graphs
2010-10  Peter Schneider-Kamp, Jürgen Giesl, Thomas Ströder, Alexander Serebrenik, René Thiemann: Automated Termination Analysis for Logic Programs with Cut
2010-11  Martin Zimmermann: Parametric LTL Games
2010-12  Thomas Ströder, Peter Schneider-Kamp, Jürgen Giesl: Dependency Triples for Improving Termination Analysis of Logic Programs with Cut
2010-13  Ashraf Armouhosh: Design Patterns for Safety-Critical Embedded Systems
2010-14  Michael Codish, Carsten Fuhs, Jürgen Giesl, Peter Schneider-Kamp: Lazy Abstraction for Size-Change Termination
2010-15  Marc Brockschmidt, Carsten Otto, Christian von Essen, Jürgen Giesl: Termination Graphs for Java Bytecode
2010-17  Hans Grönniger: Systemmodell-basierte Definition objektbasierter Modellierungssprachen mit semantischen Variationspunkten
2010-18  Ibrahim Armacı: Personalisierte eHomes: Mobilität, Privatsphäre und Sicherheit
2010-19  Felix Reidl: Experimental Evaluation of an Independent Set Algorithm
2010-20  Wladimir Fridman, Christof Löding, Martin Zimmermann: Degrees of Lookahead in Context-free Infinite Games
2011-01  * Fachgruppe Informatik: Jahresbericht 2011
2011-02  Marc Brockschmidt, Carsten Otto, Jürgen Giesl: Modular Termination Proofs of Recursive Java Bytecode Programs by Term Rewriting
2011-03  Lars Noschinski, Fabian Emmes, Jürgen Giesl: A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems
2011-04  Christina Jansen, Jonathan Heinen, Joost-Pieter Katoen, Thomas Noll: A Local Greibach Normal Form for Hyperedge Replacement Grammars
2011-07  Shahar Maoz, Jan Oliver Ringert, Bernhard Rumpe: An Operational Semantics for Activity Diagrams using SMV
2011-08  Thomas Ströder, Fabian Emmes, Peter Schneider-Kamp, Jürgen Giesl, Carsten Fuhs: A Linear Operational Semantics for Termination and Complexity Analysis of ISO Prolog
2011-09  Markus Beckers, Johannes Lotz, Viktor Mosenkis, Uwe Naumann (Editors): Fifth SIAM Workshop on Combinatorial Scientific Computing
2011-10 Markus Beckers, Viktor Mosenkis, Michael Maier, Uwe Naumann: Adjoint Subgradient Calculation for McCormick Relaxations
2011-12 Ingo Felscher, Wolfgang Thomas: On Compositional Failure Detection in Structured Transition Systems
2011-13 Michael Förster, Uwe Naumann, Jean Utke: Toward Adjoint OpenMP
2011-14 Daniel Neider, Roman Rabinovich, Martin Zimmermann: Solving Muller Games via Safety Games
2011-16 Niloofar Safiran, Uwe Naumann: Toward Adjoint OpenFOAM
2011-18 Kamal Barakat: Introducing Timers to pi-Calculus
2011-24 Callum Corbett, Uwe Naumann, Alexander Mitsos: Demonstration of a Branch-and-Bound Algorithm for Global Optimization using McCormick Relaxations
2011-25 Callum Corbett, Michael Maier, Markus Beckers, Uwe Naumann, Amin Ghobeity, Alexander Mitsos: Compiler-Generated Subgradient Code for McCormick Relaxations
2011-26 Hongfei Fu: The Complexity of Deciding a Behavioural Pseudometric on Probabilistic Automata
2012-01 * Fachgruppe Informatik: Annual Report 2012
2012-02 Thomas Heer: Controlling Development Processes
2012-03 Arne Haber, Jan Oliver Ringert, Bernhard Rumpe: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems
2012-04 Marcus Gelderie: Strategy Machines and their Complexity
2012-05 Thomas Ströder, Fabian Emmes, Jürgen Giesl, Peter Schneider-Kamp, and Carsten Fuhs: Automated Complexity Analysis for Prolog by Term Rewriting
2012-06 Marc Brockschmidt, Richard Musiol, Carsten Otto, Jürgen Giesl: Automated Termination Proofs for Java Programs with Cyclic Data

* These reports are only available as a printed version.
Please contact biblio@informatik.rwth-aachen.de to obtain copies.