RWTH Aachen

Department of Computer Science
Technical Report

Model Checking Software for Microcontrollers

Bastian Schlich, Michael Rohrbach, Michael Weber, and Stefan Kowalewski

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2006-11

RWTH Aachen · Department of Computer Science · August 2006

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/

Model Checking Software for Microcontrollers

Bastian Schlich[1*], Michael Rohrbach[1], Michael Weber[2**], and Stefan Kowalewski[1]

[1] RWTH Aachen University, Embedded Software Laboratory, Ahornstr. 55, 52074 Aachen, Germany
Schlich@informatik.rwth-aachen.de
http://www-i11.informatik.rwth-aachen.de
[2] CWI, Dept. of Software Engineering, Amsterdam, The Netherlands
Michael.Weber@cwi.nl

Abstract. A method for model checking of microcontroller code is presented. The main objective is to check embedded C code including typical hardware specific ingredients like embedded assembly statements, direct memory accesses, direct register accesses, interrupts, and timers, without any further manual preprocessing. For this purpose, the state space is generated directly from the assembly code that is generated from C code for the specific microcontroller, in our case the ATMEL ATmega family. The properties to be checked can refer to the global C variables as well as to the microcontroller registers and the SRAM. By this approach we are able to find bugs which cannot be found if one looks at the C code or the assembly code alone. The paper explains the basic functionality of our tools using two illustrative examples.

1 Introduction

In recent years industries have recognized model checking as a promising tool for the development of embedded systems. Many embedded systems are used in safety critical environments. Full testing of the systems is often not possible because it is too time consuming or too expensive. However errors in these systems may lead to fatal events. A replacement of the software while in operation is often too difficult or not possible as it is in desktop systems. Hence the industries would like to verify their systems.

In most embedded systems microcontrollers are used. The programs for microcontrollers are in the majority of cases written in the C language. Therefore it can be expected that in almost all embedded software projects in some phase there exists C code for microcontrollers. We want to use this C code for model checking. The model checking should be done without the need to manually preprocess the program. For an adoption of model checking in the industry it is not feasible that a developer uses hours of her time to prepare programs for the verification. If model checking is to be used in industry, it is mandatory that the programs that are created in the development can be used for verification without manual preparation. Furthermore, a manual preprocessing could introduce new errors or mask existing errors.

The C code used for programming software for microcontrollers often contains hardware dependent constructs as:

* Corresponding author
** This research has been partially funded by the Netherlands Organisation for Scientific Research (NWO) under FOCUS/BRICKS grant number 642.000.05N09

– direct hardware and memory accesses,
– embedded assembly instructions,
– use of interrupts and timers.

These microcontroller specific constructs are often used in software for embedded systems and important for the operation of the embedded system. They vary significantly between different microcontroller families, and can not be abstracted away in the verification procedure without the loss of crucial details.

Contribution To address these challenges, we developed a practical method to apply model checking to microcontroller programs which works directly on the level of *assembly language*. In principle, we can deal with programs written in any programming languages by first compiling them with commonly available compilers.

As platform for our experiments we have chosen the popular ATMEL ATmega family of controllers. We can handle all commonly used hardware details, and some of the more esoteric features. Despite the necessity to select a specific microcontroller family, our method is easily retargetable to others. Much of the work can be transferred straightforwardly. As expected, the main effort of supporting a new controller lies in the support for special-purpose features.

While commonly embedded systems developers have little problems dealing with microcontroller assembly, it is not very appealing to *only* work at such low levels, especially if the input program was written in another language. We exploit *debugging information* emitted by compilers during the translation phase to maintain a link between the input program and its translation into assembly.

Advantages Our approach has several advantages. The code which finally is deployed to the hardware is checked, and not an intermediate representation. All errors that could be introduced during the development process can be found, including compiler errors, misunderstanding of semantics, or misunderstanding of the behavior of the underlying hardware. *Platform-specific errors* can be found which are not visible in the original C code; an example for such an error is given in Section 6.1. In fact, not even the C source code needs to be given. Users are not forced to create an environment, make annotations, or apply manual abstractions. We support assertions about features of the C code, about features of the hardware, and about features of the assembly code.

Overview The rest of this paper is organized as follows. In the next section we review related model checkers which we evaluated before starting our own. In Section 3, we discuss in detail how current approaches fall short, and how we intend to fix the situation. In Section 4, we briefly describe the ATMEL ATmega 16 microcontroller. In Section 5, we describe two possible approaches to translate assembly programs into a representation suitable for model checking algorithms. After that, some encouraging first results of applying these approaches are presented in Section 6. We conclude with some remarks about our experiences and future work.

2 Related Work

As programs of interest to us are written in C with parts in microcontroller-specific assembly language, our work is related both to C code model checkers, and assembly code model checkers.

Today, many C code model checkers are in existence, most of them concentrating on the verification of ANSI C code. Their primary goal is to verify *hardware-independent* parts of a program. Most of them are used for verification of drivers and protocols. In contrast, we want to model check C code that is written for a specific microcontroller. In a previous study [1], we found that general purpose C code model checkers are not able to model check this specific C code. Two reasons can be identified. First, many C code model checkers restrict the set of supported ANSI C constructs. The second reason is that none of them support hardware-specific extensions, e. g., global variables which mask access to I/O ports, embedded assembly statements, interrupts and timers.

However, in embedded C programs all these features are used and therefore many of the C code model checkers are not even able to correctly parse such programs. Others abort with error messages when encountering direct memory accesses (dereference of a pointer which is a constant). None of them was able to model check a simple program written for a microcontroller [1]. Even if they would be able to build the state space for these programs, the negligence of the underlying hardware would in many cases result in a state space that is too large. Details are explained in Section 4 and 5.

For many of these C code model checkers the user has to prepare the source code with annotations, create manual abstractions, or to provide an environment to close the system [2]. In this work we try to automate this step as much as possible by exploiting detailed knowledge of the underlying hardware.

Additionally, the specification features provided by many of the C code model checkers are not adequate for creating specifications for embedded systems (e. g., only reachability, and only through source code annotations).

The *State Exploring Assembly Model Checker* StEAM [3] model checks machine code for the Internet C Virtual Machine (ICVM), which is compiled from C/C++ source code. The main focus of StEAM is model checking of parallel C++ programs. It builds the state space by monitoring a modified version of the ICVM. The ICVM simulates the program on assembly code level. StEAM depends on a modified version of the GCC compiler. If a new version of GCC has to be used, it has to be adapted again. In our approach, we can use the GCC compiler as it is. StEAM is able to model check ANSI C code and standard C++ code. The same restrictions apply which we lined out for the C code model checkers, regarding their hardware independence. However, a similar approach is used to generate the state space of a program.

3 Deriving Models from Microcontroller Programs

In this section, we review how models suitable for checking properties on have been constructed traditionally, why this is unsatisfactory for our setting, and how we can take advantage of the often under appreciated *simplicity of assembly language* to reach our goals, even with less work.

3.1 Models for Checking: Where They Come From

Explicit-state model checking is usually carried out in several steps. Traditionally, the artifact to be verified is manually described with a modeling language. From this model, a (possibly labeled) state transition system is derived (its reachable part is called *state space*), which then is used to check properties given in a temporal logic like LTL or CTL, and to provide a proof in terms of a counterexample or a *witness*. The verification algorithms for the last step are well-known. Conveniently, they can be formulated as some kind of graph search. In the remainder of this section, we will focus on how to construct a suitable input graph for these algorithms from the artifacts we want to check: microcontroller programs.

As mentioned, the construction of a model used to be a manual step. This allows the modeler to create concise models of exactly those parts of an artifact which are relevant for the properties to check. However, it is clear that this process also is a potential source of errors, as the relation between the actual artifact and the model is informal, for several reasons.

Modeling languages like finite-state machines, CCS, etc., provide a simple mathematical framework with formal semantics, and thus often do not provide sufficient expressivity to comfortably map all details of a program one-to-one into this framework. Other modeling languages, like Promela [4], are patterned to resemble conventional high-level programming languages. However, this incurs more complicated semantics. It becomes harder to convince ourselves whether our intentions are really reflected by the models. Also, the translation into transition systems becomes more involved.

Despite their extensions, we don't know of any modeling language suitable for model checking which includes direct ways to express the specifics of microcontroller hardware. We still have to resort to describing them indirectly (inconvenient), or abstracting from them completely. Furthermore, some abstraction is often needed to make the model checking task tractable, otherwise the generated state space is too big to be handled even by modern computers[3].

3.2 Software Model Checking: Eliminating the Middleman

While ultimately is seems not feasible to completely replace all manual abstractions, *software model checking* tries to eliminate one source of human error by automating the translation of the program into a model suitable for verification algorithms. The programming language in which our program is written becomes the input formalism from which a transition system can be derived.

To base ourselves on a solid foundation, we need to assign unambiguous meaning to a program through the *operational semantics* of the programming language it is written in. In an explicit-state setting, we can then obtain the transition system of a concrete program by *interpreting* it stepwise. Each step is a state in its state space. Edges of the transition system are induced by relating each two of such consecutive steps.

However, the fact that we deal with microcontroller programs complicates our setting, even if we restrict ourselves to a specific microcontroller. It is common that we have to deal with programs written in C, C++ and SystemC, often

[3] There is little practical value in drawing a distinction between infinite and simply too large state spaces.

mixed with hardware-specific assembly language. Each of the high(er)-level languages is challenging to handle in completeness, let alone formalize, as the CIL (C *Intermediate Language*) project found out [5]. In part, this is due to the fact that real-world programs often are not conforming to the respective language standard, they invoke undefined behavior, and use compiler-specific extensions. The standards themselves are large, informal, and bugged with many dark corners.

It is not surprising that conventional software model checkers have problems dealing with the intricacies of programs we are interested in [1]. In addition, they cannot deal with embedded assembly parts *at all*.

3.3 Towards a Manageable Approach

Reconsidering the setting that we find ourselves in—dealing with several input languages, embedded assembly language, and hardware specifics—it seems obvious that we cannot avoid dealing with the lowest layer: hardware-specific assembly. However, as it turns out, selecting assembly as the *only* input language that we deal with even has several advantages over existing high-level language approaches:

- Existing compilers can do the grunt work of "understanding" high-level languages. We leave this task to the respective domain experts, and rather concentrate on how to deal with their (simpler) output. Specifically, we consider only *object code*, which can be viewed as the canonical representation of a program for a specific hardware platform. As nice side effect, we can check programs for which we have no access to their original source code.
- In principle, model checking tools adopting this approach can deal with a number of languages for which exist compilers to our chosen target platform.
- Compilers implement all kinds of non-trivial optimizations on programs before emitting executable code, and interactions between such optimizations make it hard to ensure their correctness. In general, programs exhibit different run-time behavior depending on the level of optimizations in the compilation. By using the assembly output of a compiler, we implicitly make the entire compilation process part of our model. In other words, we can actually detect situations in which a *compiler* introduces a bug in an otherwise correct program, thus letting it behave as not intended.
- Assembly language is geared towards direct execution in hardware, and as such it is simple, *instructions* have well-defined and often strictly local effect on the processor state, and it bears little redundancy. In addition, reference manuals for an assembly language are sufficiently formal that it is mostly straight-forward to derive operational semantics from them. This in turn enables us to obtain a state transition system from a program by emulating the microcontroller as a whole!

However, the dependency on an external compiler brings along some additional challenges. Users expect the outcome of a model checking run to be given relative to *their* input language, that is, they would rather like to deal with properties and counterexample in terms of, e.g. C, than assembly, if possible (cf. section 6.1). Fortunately for us, most compilers can emit detailed *debugging* *information* along with the actual assembly code. Translating back and forth between input program and assembly then is a matter of threading this information through the model checking process, but does not pose too many difficulties.

Another issue is that some properties, which are easily deducible from a high-level program, are obscured by the compilation process with its possible optimizations. Here, debugging information helps to recover them as well, at least in parts. Some loss is usually unavoidable.

Programs requiring some kind of input, usually through special hardware support, also need special consideration. Such programs are considered *open*, and explicit-state model checking algorithms can only deal with *closed systems*, where an environment providing the input is part of the model.

The type of expected input is to some degree implicitly encoded in a program itself, however usually not in a conveniently accessible form. A separate line of research is devoted to methods for automatically closing open systems [2].

We stress that part of the simplicity of this approach also stems from the fact that we are dealing with a very restricted setting which plays out in our favor. The class of programs we are dealing with are usually small, largely self-contained, sequential (except for *interrupt handlers*), and finite-state. They are designed to execute within very small amounts of memory, and have limited capabilities to interact with the outside world. Recursion and dynamic memory allocation are rare as well. These traits play into our hands, as they simplify the model checking process.

4 ATMEL ATmega 16 microcontroller

We decided to use the ATMEL ATmega 16 microcontroller. It was used before at the embedded software laboratory. It has has simple and clean architecture and it is widely-used. Many real-world example can be found in the Internet. Furthermore there exists a big pool of programs at the embedded software laboratory.

The ATMEL ATmega 16 is a low-power CMOS 8 bit microcontroller based on a RISC architecture. It has inter alia the following features:

- 16 KB flash memory, 1 KB internal SRAM, 512 Bytes EEPROM
- 4 I/O Ports 8-bit, 3 Timer Counter Units, 20 vectorized interrupts

It can be programmed via assembly or C language and it is supported by the GCC compiler. It is e.g. used to control sensors and actuators on lower field-levels in industrial controll systems. Another application area are sensor networks.

Microcontrollers differ from normal microprocessor. Additional to the CPU a microcontroller consists of specific hardware features as timers, interrupts, I/O ports, etc. All these features are accessed through defined I/O registers. These I/O registers are accessed directly by their addresses. They are important for the operation of the microcontroller. Most work is done by means of these I/O registers. It is not possible to abstract from these registers. Since most of the additional hardware features are used via more than one register, these registers influence each other. E.g. a write to register DDRA influences the behavior of register PORTA. The interaction between these registers is not static, it may change during the operation of the microcontroller. A static abstraction could not reflect the correct behavior.

As the microcontroller interacts via the I/O registers with its environment and we wanted to keep the system open [2], we had to introduce nondeterminism. The nondeterminism is needed for e.g. external interrupts, read accesses to ports, and Analog to Digital Converter (ADC). In both approaches we try to limit the over-approximation induced by this nondeterminism by taking advantage of the hardware architecture. The concrete implementation differs slightly in both approaches. There are other microcontroller specific features that require extra treatment but we can not list them here due to space constraints.

5 Approaches

In this section two approaches and the implementations of theses two approaches to model check assembly code for microcontrollers are described. The first approach uses an existing simulator to build the state space. It was developed at the embedded software laboratory. The second approach translates the assembly code together with a description of the microcontroller into bytecode for an exiting virtual machine. It was developed during a diploma thesis that was done in a cooperation between the embedded software laboratory and the CWI. Both approaches use explicit state model checking. The first does CTL and the second does LTL model checking.

5.1 Using an existing Simulator

The first approach is implemented in the model checker [mc]square as proposed in [1]. [mc]square stands for Model Checking MicroControllers. In this approach parts of an existing simulator are used. This specific implementation uses parts of the cycle accurate simulation framework Avrora[4] [6]. Avrora is mainly used to simulate a network of sensor nodes. A core component of Avrora is the so called interpreter. This interpreter takes a state and evaluates the instruction of this state. In the interpreter all instructions that the ATMEL ATmega family supports are implemented. Avrora also models the complete hardware of the microcontroller. We had to change some components of Avrora to be able to use it for building the state space.

This approach is divided into the following four steps:

- Preprocess the formula.
- Build the entire state space.
- Model check the generated state space.
- Create the counterexample.

In the first step the user-provided formula is preprocessed. The formula is transformed into a formula that only contains the CTL operators EX, EU, and EG. All other temporal operator can be expressed with these operators [7].

In the next step [mc]square builds the state space. In the beginning the initial state is put on the stack. [mc]square takes the first state from the stack. Then it passes the state to Avrora and calls a step method. This step method

[4] http://compilers.cs.ucla.edu/avrora

executes the interpreter on this state. The interpreter evaluates the instruction. After that [mc]square takes the newly created state and searches in the hash table that stores the state space if this state already exists. If it exists, this state is discarded and the next state is taken from the stack. But if the state does not already exist, it is added to the state space. Before it is added to the state space, the data that is attached to this state is compressed via run-length encoding (lossless compression algorithm). The data contains the complete data address space of the microcontroller. The new state is given to Avrora and the step method is called again. This is repeated until the stack is empty and no new state is found. The build process of the state space is completed when this situation is reached.

The above described procedure applies in case only deterministic successor states occur. As described before some features of the microcontroller require nondeterminism when model checking the code. But the interpreter can not handle this nondeterminism because normally every state in Avrora only has one deterministic successor. To handle this [mc]square prepares the states that have nondeterministic successors (e.g. a read of an I/O port). If it encounters such a state, it calls a component called splitter.

The splitter is a part of [mc]square. It analyzes the state and computes the number of states that have to be created from this state. After that it creates all these states by setting the value of the nondeterministic register that is accessed in the current instruction. That means if an I/O register is read and four bits of this I/O register are nondeterministic, it creates 16 states from this state and in every of these 16 states it sets the I/O register to a different value. Thereby it creates from one state with nondeterministic successors 16 states that each have one deterministic successor. Hence all possible values are covered. The intermediate states are not added to the state space.

Through these changes the interpreter of Avrora is now able to handle these intermediate states. [mc]square iteratively passes the intermediate states to Avrora. Avrora executes the interpreter on each of these states. After that [mc]square adds the resulting states to the state space and to the stack. The state that accessed the nondeterministic register is set as the predecessor of these states. The intermediate states do not occur in the state space at all. The I/O register that was accessed is still nondeterministic. I.e. in the next read to this I/O register this procedure is repeated. The number of possible successor states that the splitter component creates depends on the hardware feature that is accessed in the corresponding instruction (cf. 4) and on the current state of this hardware feature.

After the state space is build it is checked via a standard CTL model checking algorithm. Only the three operators EX, EU, and EG are implemented. The algorithm for EG and EX were taken from [7]. The algorithm for EU was taken from [8]. We used non-recursive versions of the algorithms for performance reasons. If a formula is invalid, [mc]square generates a counterexample. This counterexample is then presented as a graph or can be traversed in the C code and assembly code file.

If this approach is to be extended to handle another microcontroller, one has to exchange the simulator and to adapt the interface to the new simulator. The model checking algorithm and most parts of the splitter can be reused. The advantage of this approach is that an existing simulator can be used to build the state space. If no simulator exists for the new microcontroller or the existing simulator can not be adapted, one has to write a basic simulator on its own. If an existing simulator is to be used, the effort that is needed to adapt it can vary. It depends on the features the simulator supports and how it implements these features. [mc]square only needs some basic functions to communicate with the simulator. A function to pass a state to the simulator, a function to get a state from the simulator, and a step function to execute an instruction.

5.2 Translation to Bytecode

The second approach is implemented in Model Checking of Embedded Systems Software (MCESS) [9]. This approach translates given assembly code together with a description of a microcontroller into the input language of an existing virtual machine for state space generation (NIPS VM) [10].

NIPS VM operates on an assembly-like bytecode, and is intended as a generic intermediate layer for the translation of modeling languages. Therefore, the main task in the second approach is to translate the microcontroller assembly code into bytecode. In addition to that, the microcontroller hardware has to be represented in the NIPS VM.

This translation is initially developed for the ATMEL ATmega 16, but it is possible to modify any microcontroller specific information, like memory layout or instructions semantics without modifying the implementation of the translation. This is done by means of a Domain-Specific description Language based on the Register Transfer Language (RTL-DSL). Some kind of register transfer language is commonly used in microcontroller documentations to describe the effects of assembler instructions on the microcontroller hardware. Thus, this information can be directly copied from the documentation to the RTL-DSL specification, to describe the instruction semantics of the microcontroller. A description of the memory layout with the number of general purpose registers, special purpose registers, SRAM size, etc. can also be taken directly from the ATMEL documentation. Because of the external RTL-DSL specification it is possible to adapt the translation and hence the model checker to other microcontrollers.

The information in the specification is used to create a representation of a microcontroller state and to perform the translation of the assembly code into the bytecode.

Up to this point the model checker has every information to compute the deterministic successor of a state resulting from the execution of one assembler instruction. However, special care has to be taken for accesses to microcontroller features which depend on interaction with the environment. For these situations additional code fragments for the access to special purpose registers have to be added to the RTL-DSL specification. These fragments are executed each time the memory address of the appropriate register is read or written instead of loading or storing the deterministic value. For example, if the program accesses the special purpose register containing the value of an extern port and this port is configured as input, the code fragment has to simulate every possible value that can possibly occur at this port. These code fragments which serve as abstractions for real accesses to memory locations are implemented directly in NIPS VM bytecode and are added during the translation of the assembly code.

The model checking process using MCESS works as follows. The C code checked is compiled to its assembly code representation and given to the translation unit, together with the specification of the microcontroller. Since MCESS does LTL model checking the user has to provide an LTL formula which specifies the desired behavior of the program. Then the assembly code and the RTL-DSL specification are used to create the model of the program running on the specified microcontroller. The formula is also encoded in the model in terms of a Buechi automaton. This automaton runs simultaneously with the system and accepts every run which falsifies the formula. Thus the formula is satisfied on all paths of the system if the resulting state space does not contain any accepting cycles [11]. This is checked with a nested depth-first search algorithm [12].

The advantage of translating assembly code into NIPS VM bytecode is that once this is done, generic algorithms can be used for the following steps in the model checking process, like static analyses and the actual LTL model checking. Also, it enables easy connection to existing distributed verification tools [13].

6 First Results

In this section two programs are presented that are used to show the basic features of both implementations. The first program was written as part of a diploma thesis. This thesis is done in corporation between the embedded software laboratory and the CWI. For clarity only a small part of this program is presented here. This part still contains an error that can not be found by C code model checking. This error is typical for software used in embedded systems. The second program was created in a laboratory course at the embedded software laboratory. It shows an error that only occurs if two interrupts are raised at the same time.

6.1 Faulty Write to an Integer

The first program is shown in Listing 1.1. Due to space constrains, we present this small part of a bigger program. This part still contains an error that is often found in programs for embedded systems.

In this program one I/O port and one external interrupt are used. In the main loop a variable `i` is incremented from 0 to 300. There is an external interrupt that may be triggered from outside. In the interrupt handler of this external interrupt port A is set to a value depending on the value of `i`. If `i < 150` port A is set to 0x00 and if `150 <=i <= 300` port A is set to 0x55. At the end of the interrupt handler port A is set to 0xff. The formula we want to verify is (1).

$$AG \quad (in\_interrupt\_handler = 1 \Rightarrow i \leq 300) \tag{1}$$

This formula states that if the program is in the interrupt handler, the variable `i` is always less or equal to 300. In the original program a bigger value than 300 resulted in an error. Looking at the C source code the program should satisfy specification (1), but both model checkers report a counterexample which shows that there exists a state where the interrupt handler is executed and the value of `i` is 511. If you take a look at the assembly code representation of the `i++;` instruction in line 16 you see that the assignment of a new value to the memory

Listing 1.1. Faulty Write to an Integer

```
1  #include <avr/io.h>
   #include <avr/interrupt.h>
   #include <avr/signal.h>

   volatile int i=0;
6  char in_interrupt_handler = 0;

   int main (void){
       DDRA = 0xff;                   // PORTA as output
       PORTA = 0xff;
11     MCUCSR = MCUCSR | (1<<ISC2); // IR 2 with falling edge
       GICR = GICR | (1<<INT2);       // ext IR 2 activated
       sei();                         // global IR enable
       while (1){
           while(i<300){
16             i++;
           }
           i = 0;
       }
   }
21
   SIGNAL (SIG_INTERRUPT2){
       in_interrupt_handler = 1;
       if (i < 150)
           PORTA = 0x00;
26     else if (i <= 300)
           PORTA = 0x55;
       PORTA = 0xff;
       in_interrupt_handler = 0;
   }
```

location of `i` is done by two subsequent assembler instruction. The first instruction stores the higher and the second instruction stores the lower byte of the integer value. The counterexample shows that an interrupt was triggered exactly in between of those two instruction when the value of `i` was 255 = `0x00FF`. This means that the first instruction already stored the high byte of the new value 256 = `0x0100` which resulted in the value 511 = `0x01FF` in the memory location of `i`. As soon as the variable `in_interrupt_handler` is set to `1` after the interrupt was triggered, the model reaches a state which falsifies specification (1).

[mc]square takes about 4 seconds to check the formula and generates 104,338 states that need 45 MB of memory. MCESS needs about 10 seconds to generate the bytecode and about 1 second to check the formula. Since MCESS uses an on-the-fly model checking algorithm it only needs to explore 20,163 states and uses approximately 23 MB of memory for storing the state space. Building the complete state space with a simple reachability formula produces 107,640 states in about 4.5 seconds and needs 119,08 MB of memory.

To fix the error in this example, we only have to deactivate interrupts while `i` is incremented. To do so, we insert the statement `cli();` just before line 16 and `sei();` just after line 16. Now the formula is found to be valid by both model checkers.

Important to notice is that the size of the state space drops to 68,340 states in mc[square] and 70,416 states in MCESS, respectively. The lower state count results in a shorter exploration time and reduced memory consumption for storing the states, mc[square] needs about 3 seconds and 30 MB of memory, MCESS requires about 10 seconds for translation, 3 seconds to check the model, and 78 MB of memory. The reason for the decreased state count is that now the interrupt can not be triggered during the execution of `i++`. On assembly level the instruction `i++` is split up into five instructions.

6.2 Traffic Light Example

In this section a program is presented that was developed as an exercise in a laboratory course at the embedded software laboratory. It is an implementation of a traffic light that is only controlled by time. The implementation is shown in Listing 1.2. The only influence from the outside is an emergency switch that puts the traffic light into an emergency off state. The normal operation is controlled by an interrupt that is triggered by a timer overflow of timer 0. The emergency button is implemented by a button that is connected to a pin at port D. If this button is pressed, it raises an interrupt. The interrupt handler of this interrupt disables the timer and sets the traffic light into the emergency state.

We want to verify that the program is in well defined states all the time and that the emergency state (`PORTA=0xf8`) is never left if it is entered once. This is encoded in formula (2).

$$\begin{aligned} AG \quad & (porta = 0 \lor porta = 0\text{xfb} \lor porta = 0\text{xfd} \lor porta = 0\text{xfe} \lor \\ & AG \quad porta = 0\text{xf8}) \end{aligned} \tag{2}$$

This formula is rejected. [mc]square needs about 1 second to build the state space and to model check it. The state space has 9,435 states and 5 MB are needed to store all states. MCESS checks the model (94 states) in less than 1 second and needs less than 1 MB. It needs 11 seconds for translation. The complete state space (9,758 states) is searched in about 1 second and 22 MB.

By inspecting the counterexample, we find out that the error happens in the interrupt handler of external interrupt 0 (`SIGNAL (SIG_OVERFLOW0) {...}`). In case both interrupts (external interrupt and timer overflow interrupt) occur at the same time, first the interrupt handler of the external interrupt is executed. In this interrupt handler the timer is deactivated. But the flag that the timer 0 overflowed is not reset. Hence after the interrupt handler of the external interrupt is finished, the interrupt handler of the timer overflow is called. This handler sets port A to 0xfd. By this the traffic light leaves the emergency state. To fix this error we insert instruction `TIFR=(1<<TOV0);` in line 33, which resets the flag storing that the timer overflow interrupt has to be called. Now the formula is verified.

[mc]square needs about 1 second to verify this formula. 9,445 states are created that need 5 MB of memory. Checking the formula with MCESS also takes about 1 second and results in 9,750 states which need 22 MB. The translation takes 10 seconds.

6.3 Remarks

Taken this first results, both implementations show similar performance. Due to the on-the-fly algorithm that MCESS uses, the size of the generated state spaces can differ drastically in case that an error is found. However, the usage of such an algorithm can lead to a blow up of the state space since it is directly

Listing 1.2. Implementation of a Traffic Light

```
#include <avr/io.h>              // access to I/O-registers
#include <avr/interrupt.h>       // interrupt handling
#include <avr/signal.h>          // signal handling

volatile char status=0;

void init(void){
    DDRA = 0xff;                 // PORTA as output
    PORTA = 0x00;
    DDRD = 0x00;                 // PORTD as Input
    PORTD = 0xff;                // activate pull-up
    TCNT0 = 0x00;                // counter register = 0
    OCR0 = 0x00;                 // compare register = 0
    TCCR0 = 0x00;                // stop timer 0
    MCUCR |= (1<<ISC01);         // enable ext IR0
    GICR |= (1<<INT0);           // enable ext IR0
    TIMSK |= (1<<TOIE0);         // enable timer 0 IR
    TCCR0 = 0x05;                // start timer 0
}

int main (void){
    init();
    sei();                       // global IR enable
    while(1){
        //do something here
    }
    return(1);
}

SIGNAL (SIG_INTERRUPT0) {
    status = 0;
    TCCR0 = 0x00;                // stop timer 0
    PORTA = 0xf8;
}

SIGNAL (SIG_OVERFLOW0){
    TCCR0 = 0x00;                // stop timer 0
    status++;                    // increment status
    if(status<80){               // choose action depending on status
        PORTA = 0xfb;            // red 1111 1011
    } else {
        if (status<115) {
            PORTA = 0xfd;        // yellow 1111 1101
        } else {
            if (status<220){
                PORTA = 0xfe;    // green 1111 1110
            } else {
                PORTA = 0xfd;    // yellow 1111 1101
            }
        }
    }
    TCNT0 = 0x00;
    TCCR0 = 0x05;                // restart timer 0
}
```

influenced by the size of the formula. The differences in memory consumption come from the fact that [mc]square uses run-length encoding. This compression algorithm causes an increase of the time that [mc]square needs for building the state space and model checking it (5–10 times). It is important to notice that the memory consumption is only measured for the complete state space. The temporary memory consumption of both implementations is higher. [mc]square needs more temporary memory since it uses a GUI and runs in the Java Virtual Machine.

Although both implementation are based on different approaches, the sizes of the entire state spaces are nearly the same. They only differ by a negligible small portion. In the programs used a considerable amount of time is consumed by the translation process in MCESS. When checking programs with larger state spaces this amount of time can be neglected since it increases only linear in program length and not in state space size.

The maximum size of state spaces that can be generated with [mc]square is about 2,000,000 states per gigabyte RAM. The maximum size of the state space for MCESS is about 800,000 per gigabyte RAM since it does not use a compression algorithm. [mc]square may produce less states per gigabyte RAM if the distribution of the values is unfortunate.

7 Conclusion & Future Work

This paper describes the basic idea to model check hardware dependent assembly code for a specific microcontroller. The advantages to model check the assembly code are manifold. The semantics of assembler code is easier and better documented than the semantics of C code. All errors that could be introduced during the development phase can be found in the assembly code file. This includes compiler errors. The assembly code is the code that finally is transferred to the microcontroller. The hardware dependency helps to decrease the size of the generated state space and makes it possible to state properties including all features that are present on a microcontroller.

Two approaches that model check the assembly code are presented. The first approach uses an existing simulation framework to build the state space. The second approach translates the assembly code together with an description of the hardware into bytecode for the NIPS virtual machine for state space generation. For both approaches a implementation is described.

Two programs were uses to demonstrate the basic features of both implementations. Only small programs were used for clarity. It is important to mention that both programs could not be checked by general purpose C code model checkers or StEAM without applying manual abstractions, creating an environment, or annotating the program. Those changes were not needed by [mc]square and MCESS. As mentioned in Sect. 6.3, both model checkers can deal with much bigger programs. Although both approaches are implemented differently, the sizes of the generated state spaces are similar. This increases the trust that the state spaces were generated correctly. Furthermore the model checking results of both approaches were equal while [mc]square does CTL and MCESS does LTL model checking.

In the future we want to compare both approaches in depth because we want to find out in which situation which approach is better suited. Additionally we want to find out whether it is desirable to use both implementations at once. Such a proceeding could increase the trust in the validity of the results.

This is a first step towards an integration of model checking into the development process for embedded systems. In both approaches model checking of the program is done without the need to prepare the program. Although the model checking is done on assembly level, information can be presented in terms of the original C code. At present both model checkers can only be used to model check code for the ATMEL ATmega 16 microcontroller. Both approaches can be extended to model check code for other microcontrollers. The effort doing this may vary.

References

1. Schlich, B., Kowalewski, S.: Model checking c source code for embedded systems. In: Proceedings of the IEEE/NASA Workshop on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2005). (2005)
2. Kupferman, O., Vardi, M.: Module checking. In: Computer Aided Verification, Proc. 8th Int. Conference. Volume 1102 of Lecture Notes in Computer Science., Springer-Verlag (1996) 75–86
3. Leven, P., Mehler, T., Edelkamp, S.: Directed error detection in C++ with the assembly-level model checker StEAM. In: Model Checking Software (SPIN). (2004) 39–56
4. Holzmann, G.J.: The SPIN MODEL CHECKER: Primer and reference manual. Addison-Wesley (2003)
5. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: Computational Complexity. (2002) 213–228
6. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable sensor network simulation with precise timing. In: Proceedings of Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Los Angeles (2005)
7. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
8. Berard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, P.: Systems and Software Verification: Model Checking Techniques and Tools. Springer (2001)
9. Rohrbach, M.: Model checking of embedded systems software. Master's thesis, RWTH Aachen University, Embedded Software Laboratory (2006)
10. Weber, M., Schürmans, S.: NIPS virtual machine and compiler implementation. `http://www.cwi.nl/~weber/nips/` (2005)
11. Vardi, M., Wolper, P.: Automata theoretic techniques for modal logics of programs. Journal of Computer and System Sciences **32(2)** (1986) 182–221
12. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In Halbwachs, N., Zuck, L.D., eds.: TACAS. Volume 3440 of Lecture Notes in Computer Science., Springer (2005) 174–190
13. Barnat, J., Brim, L., Černá, I., Šimeček, P.: DiVinE the distributed verification environment. In Leucker, M., van de Pol, J., eds.: PDMC'05, Lisbon, Portugal (2005)

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

1987-01 *	Fachgruppe Informatik: Jahresbericht 1986
1987-02 *	David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 *	Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 *	Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 *	Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 *	Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 *	Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 *	Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 *	Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 *	Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 *	Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12	J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 *	Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 *	Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 *	Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 *	Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 *	Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 *	Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 *	Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 *	Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 *	W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 *	Kai Jakobs: Towards User-Friendly Networking
1988-11 *	Kai Jakobs: The Directory - Evolution of a Standard
1988-12 *	Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 *	Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing
1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation
1988-16 * Fachgruppe Informatik: Jahresbericht 1987
1988-17 * Wolfgang Thomas: Automata on Infinite Objects
1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs
1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers
1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen
1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment
1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers
1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies
1988-24 * Uschi Heuter: Generalized Definite Tree Languages
1989-01 * Fachgruppe Informatik: Jahresbericht 1988
1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik
1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions
1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language
1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)
1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?
1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems
1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control
1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial
1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation
1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science
1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?
1989-13 * Martine Schümmer: CNC/DNC Communication with MAP
1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements
1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments
1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes
1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems
1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line
1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme
1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management
1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines
1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking
1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD
1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars
1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)
1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik
1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual
1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems
1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages
1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)
1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine
1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus
1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions
1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code
1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine
1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)
1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus
1992-19-09 D. Howe, G. Burn: Experiments with strict STG code
1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes
1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine
1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)

1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer

1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine

1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell

1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation

1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages

1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)

1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment

1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)

1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers

1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)

1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)

1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting

1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)

1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models

1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures

1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)

1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language

1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language

1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing

1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free

1992-24 K. Pohl: The Three Dimensions of Requirements Engineering

1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications

1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification

1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety

1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design

1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems
1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming
1996-01 * Jahresbericht 1995
1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees
1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability
1996-05 Klaus Pohl: Requirements Engineering: An Overview
1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs
1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming
1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents
1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines
1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation
1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell
1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems
1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming
1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management
1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement
1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools
1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996
1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases
1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation
1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations
1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems
1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto
1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality
1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design
1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996
1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization
1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting
1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets
1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases
1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations
1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
1998-01 * Fachgruppe Informatik: Jahresbericht 1997
1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes
1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments
1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future

1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases

1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML

1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

1999-01 * Jahresbericht 1998

1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager

1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference

1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie

1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL

1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools

2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

2002-07 Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities

2002-08 Markus Mohnen: An Open Framework for Data-Flow Analysis in Java

2002-09 Markus Mohnen: Interfaces with Default Implementations in Java

2002-10 Martin Leucker: Logics for Mazurkiewicz traces

2002-11 Jürgen Giesl, Hans Zantema: Liveness in Rewriting

2003-01 * Jahresbericht 2002

2003-02 Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting

2003-03 Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations

2003-04 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs

2003-05 Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard

2003-06 Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates

2003-07 Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung

2003-08 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs

2004-01 * Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: “Aachen Summer School Applied IT Security”

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-12 Neeraj Mittal, Felix Freiling, S. Venkatesan, Lucia Draque Penso: Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18 Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"

2005-19 Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers

2005-20 Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.

2005-21 Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

2005-22 Felix Freiling, Maurice Herlihy, Lucia Draque Penso: Optimal Randomized Fair Exchange with Secret Shared Coins

2005-23 Heiner Ackermann, Alantha Newman, Heiko Röglin, Berthold Vöcking: Decision Making Based on Approximate and Smoothed Pareto Curves

2005-24 Alexander Becher, Zinaida Benenson, Maximillian Dornseif: Tampering with Motes: Real-World Physical Attacks on Wireless Sensor Networks

2006-03 Uwe Naumann: Intraprocedural Adjoint Code Generated by the Differentiation-Enabled NAGWare Fortran Compiler

2006-04 Ebadollah Varnik, Uwe Naumann, Andrew Lyons: Toward Low Static Memory Jacobian Accumulution

2006-06 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Divide-and-Color

2006-07 Thomas Colcombet, Christof Löding:: Transforming structures by set interpretations

2006-08 Uwe Naumann, Yuxiao Hu: Optimal Vertex Elimination in Single-Expression-Use Graphs

2006-10 Mesut Günes, Alexander Zimmermann, Martin Wenig, Jan Ritzerfeld, Ulrich Meis: From Simulations to Testbeds - Architecture of the Hybrid MCG-Mesh Testbed

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.