RWTH Aachen

Department of Computer Science
Technical Report

Code Stabilization

Felix C. Freiling and Sukumar Ghosh

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2005-14

RWTH Aachen · Department of Computer Science · June 2005

Code Stabilization

Felix C. Freiling[1] and Sukumar Ghosh[2]*

[1] Laboratory for Dependable Distributed Systems,
RWTH Aachen University, Germany
[2] The University of Iowa, Iowa City, USA

Abstract. Dijkstra's concept of self-stabilization assumes that faults can only affect the variables of a program. We study the notion of self-stabilization if faults can also affect (i.e., augment) the program code of a system. A *code stabilizing* system automatically recovers from (almost) arbitrary perturbations of its program code. We prove some lower bounds for code stabilizing systems and argue that code stabilization has many resemblances to the area of integrity management in the domain of security.

1 Introduction

The concept of self-stabilization by Dijkstra [6] describes the fact that a system will eventually return to good behavior when starting from an arbitrary state. The arbitrary state was used as a tool to model the effects of transient faults that changed the values of variables stored in volatile memory. The program code however was always assumed to remain unchanged.

Interestingly, the assumption that the program code is not affected by faults has remained unchallenged for a long time. Usually it is argued that the program code resides in non-volatile read-only memory and can therefore be assumed to remain constant. This is however only true for small and specialized systems (like embedded systems) today. Most software which runs on PCs is stored on hard disks which—while being non-volatile—still can be subject to changes through faults. Moreover, the threats from unauthorized code alterations through malicious software (like worms or viruses) are steadily increasing. Hence we feel that it is time to investigate the notion of self-stabilization where faults can also affect the code of the program.

In this paper we ask the question: How and when can self-stabilizing systems recover not only from perturbations of the data but also from perturbations of the program code? To answer this question we first give a formal definition of what we call *code stabilization*. In analogy to self-stabilization (which we in contrast call *data stabilization*) we define code stabilization to mean eventual recovery of the program code to a "legal state". Our definition is a clean extension of Dijkstra's definition: If the legal state of the code is a self-stabilizing algorithm, then code stabilization implies also data stabilization.

We further investigate the amount of perturbation tolerable in code stabilization and prove that code stabilization is impossible if the entire code space can be perturbed. Hence, a minimal nucleus of unaltered code space must always remain. This is in clear contrast to self-stabilization where faults could affect all the variables. We show that this minimal nucleus must have a size in the order

---

* This research was supported in part by a Fellowship from the Alexander von Humboldt Foundation.

of the entire program. This result implies that code stabilization is a very costly concept. However, in a distributed system it is possible to reduce the space requirement of this nucleus to about the size of the code which is stored in only *one* process.

Finally, we relate our findings to observations made in the area of security. We discuss the area of software integrity management and argue that the concept of code stabilization underlies many practical methods used in this area.

In summary, we provide the following contributions in this paper:

- We extend the definition of self-stabilization to code perturbations.
- We prove some lower bounds for this type of stabilization.
- We relate the new type of stabilization to practical methods from the area of security.

To the best of our knowledge, the investigation of code perturbations in the context of self-stabilization is novel. In can be seen as standing in a line of research which considers stabilization as a useful abstraction in the area of security (see for example work by Gouda [10]).

The paper is structured as follows: In Section 2 we present the system model and the definition of code stabilization. In Section 3 we consider code stabilization in the context of local (non-distributed) computations and subsequently extend our findings to distributed computations in Section 4. In Section 5 we relate code stabilization to concepts from the area of security. We conclude in Section 6.

2 Code Stabilization: Definition

In this section we present a definition of code stabilization and relate it to the concept of self-stabilization.

2.1 Systems, Programs, Code, and Data

A *system* is a general purpose computing machine that consists of an execution unit and memory. Intuitively the execution unit is a microprocessor and the memory is some form of data storage like RAM, ROM or external memory (e.g. hard disk). The memory of a system is separated into two parts: a *code part* and a *data part*. The code part stores the *program* which the system should execute. We are not concerned here with the way in which the program is encoded in memory except that we assume that it be executable. To execute the program, the system chooses the next instruction from the code part, loads it into the execution unit and executes the instruction, thereby possibly changing the data or code part of memory. Choice of the next program instruction can be done deterministically (e.g. by using an explicit program counter stored in the data part) or non-deterministically (like in the language of guarded commands [7]). Note that we allow a program to update also the code part of memory, i.e., we allow programs to be *self-modifying*.

The data part of memory can hold many different values. A particular assignment of values to the variables in the data part is called a *state* of the program. Let $\mathcal{D}$ denote the set of all possible states, i.e., all possible combinations of values which may be stored in the data part.

A representation of the program in memory is called the *code of the program* (or simply *code*). The code part of memory may hold many different codes (i.e., many different programs). Let $\mathcal{C}$ denote the set of all different codes that may be stored in the code part of memory.

2.2 Distributed Systems and Executions

The definitions above can be easily extended to cover aspects of (geographical) distribution. In a *distributed system*, the concept which we called a system above is called a *process*. Each process has its individual execution unit and memory. The code part of the memory of the distributed system is the union of all the code parts of the processes. Similarly, the data part of the memory of the distributed system is the union of the data part of the memories of all processes.

In a distributed system, processes need a method to communicate. Here we assume that processes communicate through shared memory, i.e., we assume that portions of the processes' memory can be accessed by other processes. The *topology* of the distributed system defines which process has access to the memory of which other process. The type of access can be distinguished by its type (read and/or write access) and the portion of the memory which it affects (code and/or data part of the memory). We will differentiate special types of access later in Section 4 where we consider distributed systems.

In general, for any system (be it distributed or not), the state of the entire memory can be expressed as an element $(c, d) \in \mathcal{C} \times \mathcal{D}$ where c identifies the code and d identifies the data state. An *execution* of a system is a sequence $\sigma = ((c_1, d_1), (c_2, d_2), \ldots)$ of such code/data state pairs for which holds that for all i, (c_{i+1}, d_{i+1}) results from executing the fetch-execute cycle described above on state (c_i, d_i).

2.3 Memory Perturbations

We adopt here the standard fault-assumption of self-stabilization, i.e., the type of faults we assume here are transient faults that can alter the state stored in memory. This is a very general fault assumption encompassing things like transient memory faults (e.g., bit flips), faults during data transmission, brown-outs due to transiently weak power supply, and effects of cosmic rays on memories. We rule out faults that permanently affect the execution unit. We model the effect of a fault by assuming that memory can spontaneously change into a certain "bad" state. Recovery of faults is achieved if the system by itself manages to return into a "good" state, as we explain shortly. Given some type of fault, the *fault span* [4] of that fault is the largest set of memory values which can be reached by faulty behavior.

2.4 Data Stabilization

We now recall the definition of self-stabilization [3, 6]. To distinguish it from other forms of stabilization, we use the term *data stabilization* instead of self-stabilization.

Intuitively, data stabilization means that, given some set A of states, starting from a state in A, a system always eventually reaches a set of *legal* states. If it enters a legal state, then, as long as no faults occur, the next state of the system is also legal. In the following, let $D \subseteq \mathcal{D}$ denote the set of legal states.

Definition 1 (data stabilization). *Let $A \subseteq \mathcal{D}$ be a set of (data) states that includes D (i.e., $D \subseteq A$). A system* data stabilizes from A to D *if the following conditions hold for every execution $\sigma = ((c_1, d_1), (c_2, d_2), \ldots)$ of the system:*

- *(closure) for any (c_i, d_i), if $d_i \in D$ then $d_{i+1} \in D$.*
- *(convergence) for any (c_i, d_i) such that $d_i \in A$ there exists a $j \geq i$ such that $d_j \in D$.*

If $A = \mathcal{D} =$ true we omit mentioning the set A in the definition and simply say that a system data stabilizes. Data stabilization from $\mathcal{D} = A$ is equivalent to the notion of self-stabilization as introduced by Dijkstra [6].

2.5 Code Stabilization

We assume that the set of all codes $\mathcal{C}$ contains some programs that are *illegal* (they do not solve the problem for which the system was built by, e.g., going into an infinite loop). Conversely, we assume that there exists a set $C \subset \mathcal{C}$ of *legal* codes.[1]

We now define *code stabilization* in analogy to data stabilization.

Definition 2 (code stabilization). *Let $B \subseteq \mathcal{C}$ be a set of codes that includes C (i.e., $C \subseteq B$). A system* code stabilizes from B to C *if the following conditions hold for every execution $\sigma = ((c_1, d_1), (c_2, d_2), \ldots)$ of the system:*

- *(closure) for any (c_i, d_i), if $c_i \in C$ then $c_{i+1} \in C$.*
- *(convergence) for any (c_i, d_i) such that $c_i \in B$ there exists a $j \geq i$ such that $c_j \in C$.*

We define *probabilistic code stabilization* (with probability p) as code stabilization where the convergence property holds only probabilistically (i.e., with probability p). Clearly, any system that is code stabilizing is also probabilistically code stabilizing, therefore probabilistic code stabilization is a weaker concept that code stabilization.

2.6 Relations between Code and Data Stabilization

Code and data stabilization are defined independently, but they are not orthogonal since data stabilization relies on execution of correct code.

If faults are only allowed to perturb the data, then the code can be initialized to some chosen value. If the code happens to be data stabilizing algorithm, then we get the usual setting of self-stabilization. However, in the following assume that faults may happen in data *and* code. In this case, data stabilization depends on code stabilization.

Lemma 1. *For any system, if the set of legal codes C contains only data stabilizing algorithms, then the system data stabilizes only if it code stabilizes.*

---

[1] Note that our definition allows the case where more than one code is legal, e.g., if there are different syntactic representations which are semantically equivalent.

Proof. For a contradiction, assume that the code does not stabilize to a legal code in C. This means that the code remains in a state which is not data stabilizing. Hence, the system does not data stabilize. □

Note that Lemma 1 cannot be strengthened to an equivalence. To see this consider the case where a system does not code stabilize. In this case it may be stuck in an arbitrary program, e.g. one that executes an infinite loop. Clearly, such a system will not data stabilize. So data stabilization of some system is by no means sufficient for code stabilization of that system.

We define a system to be *completely stabilizing* if and only if it is code stabilizing and data stabilizing. A completely stabilizing system can tolerate a larger fault-span than a data stabilizing system because an additional level of perturbation is possible: corruptions of code space (see Figure 1). Code stabilization can therefore be explained as driving the fault-span past the border of the variable state space.

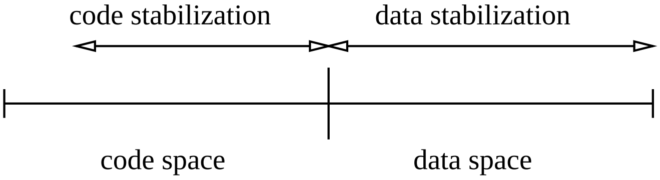

Fig. 1. Code stabilization: Moving the fault-span past to the left of the border between code and data.

3 Code Stabilization for Local Computations

In this section we consider code stabilization in a non-distributed setting, i.e., where the system consists of only one execution unit (one process).

3.1 A Technique to Establish Code Stabilization

How can code stabilizing systems be constructed? One simple way to do this is to apply a layered approach and regard the code of one layer as the data of the next layer (see Fig. 2). This approach builds upon the ideas of *fair composition* of stabilizing protocols by Dolev, Israeli, and Moran [8]. If the system at one level i is not code stabilizing, we enlarge the system by adding another code part at level $i-1$ which can modify the code at level i (the code of level i is the data of level $i-1$). Now define the correct codes of level i as the set of legal states for code at level $i-1$, then if the code of level $i-1$ is data stabilizing, the code at level i is code stabilizing.

If the code at the lowest layer (layer 1) is not affected by faults, then we can show that the entire system is code stabilizing.

Theorem 1. *Given the system as constructed in Fig. 2 in which the code of every layer is a data stabilizing algorithm. If the code of level 1 is not perturbed by faults, then the system is code stabilizing.*

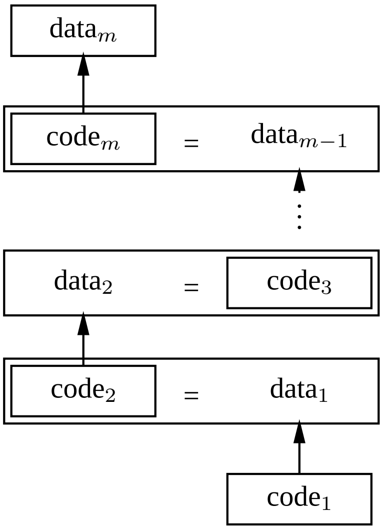

Fig. 2. Hierarchical construction of code stabilization. The code at level i is regarded as the data at level $i-1$.

Proof. The proof is similar to the proof of self-stabilizing algorithms using the idea of a *convergence stair* as introduced by Gouda and Multari [11]. The proof is by induction over the levels.

Since we assume that the code of level 1 is not perturbed by faults, this code is trivially code stabilizing, which proves the base case.

Assume that all codes up to level i are code stabilizing. Since the code at level i is data stabilizing, eventually the data of level $i+1$ will reach a legal configuration. The legal configurations however are precisely the set of codes of level $i+1$. Therefore, the code at level $i+1$ is code stabilizing, which proves the induction step. □

The construction of Theorem 1 is conceptual. It does not necessarily mean that additional execution units or memory (additional "hardware") need to be added to the system. It is just a way to structure the code and memory space of a system. Note here that this construction results in programs which are inherently self-modifying.

3.2 Minimal Requirements for Code Stabilization

One central prerequisite for Theorem 1 to hold is that the code of level m is not perturbed by faults. This raises the question whether this assumption is really necessary, i.e., is there a way to construct code stabilizing systems such that the entire code part of the memory may be perturbed by faults? Unfortunately, this is not the case, as we now explain.

The code of a program holds some form of information about this program. We define the *size* of a code as the amount of information (in bits) which it encodes. Basically, the amount of information in a code is the size of this code when compressed with an optimal compression program (e.g., one that uses Huffman codes). We now show that some minimal part of the code space must be safe from perturbations in order to achieve code stabilization.

Theorem 2. *In general, a code stabilizing system of size k requires an area of non-perturbation of size at least $O(k)$.*

Proof. The most unfavorable case is one where faults perturb the entire code and data space. Assuming that a code stabilizing system could recover from this case would mean that the information contained in the original program must be reconstructed from some source. However, if faults have perturbed the entire state space, it is impossible to recover the data from anywhere. In general, the amount of unperturbed storage corresponds directly to the amount of information which is expressed by the code. In the worst case, the code can be (almost) random data and so no more compression is possible. Hence, for a code of size k at least $O(k)$ storage needs to be maintained and this storage must be always unperturbed. □

Note that Theorem 2 is rather general. It holds for any type of system (even ones with self-modifying code) and also for probabilistic code stabilization. In a sense, it prescribes for any program of size k a "safe nucleus" of size $O(k)$ from which it can be reconstructed. This makes code stabilization fundamentally different from data stabilization because in data stabilization *all* data can be perturbed without losing the ability to stabilize.

4 Code Stabilization for Distributed Computations

We now investigate how code stabilization can be achieved in distributed systems and what the minimal requirements are to achieve code stabilization.

4.1 Uniformity Issues and Types of Remote Access

Let p and q be two processes. In the context of distributed systems with shared memory we need to distinguish different types of access from p to q. Process p has *remote read access* to q if p can read the entire code part of the memory of q. Process p has *remote write access* to q if p can write to the entire code part of q. If p has neither remote read nor remote write access to any other process, we say that p has *local access*. Note that local access does not prohibit processes to communicate since communication can still be done through some shared data part of memory.

Many distributed algorithms assume the fact that individual processes can be named using unique identifiers. Usually, these identifiers are assumed to be hard coded into the algorithm. In the terminology of this paper unique identifiers are part of the code. If faults can perturb the entire memory of a process, then also these identifiers can change. This is not a problem if the algorithm is *uniform*, i.e., it does not rely on the existence of unique identifiers and all processes in the system execute an identical copy of the same code. However, due to issues of symmetry breaking, uniform algorithms are faced with many problems. Nevertheless, in the following we focus on uniform algorithms. We discuss the impact of unique identifiers on our results later.

4.2 Techniques to Achieve Code Stabilization

Theorem 2 states that any program of size k needs an unperturbed memory portion of size $O(k)$ to code stabilize. In distributed systems with uniform algorithms, the code is stored redundantly at all processes. Therefore, it is possible to exploit this redundancy to achieve lower bounds for code stabilization than were possible in the non-distributed setting.

In the following, let k be the size of the code of an individual process. A simple and sufficient bound for code stabilization follows directly from Theorem 2. Since every process can be regarded as a non-distributed system, if all processes have only local access, then it is sufficient that all processes contain unperturbed code space of size $O(k)$. If processes have remote read and write access, this bound can be improved.

Theorem 3. *If* some *processes p has remote write access to all other processes and all other processes do not have remote write access to p, then it is sufficient that p contains unperturbed code space of size $O(k)$.*

Proof. We prove the theorem by sketching a solution that achieves code stabilization using unperturbed code space at a single process. The idea is as follows: The code of every process is augmented with a program part that regularly tries to write a copy of its own code to the code space of all other processes at once. Even if all processes have been perturbed, eventually process p will overwrite the perturbed code with an unperturbed copy of the code. Since p itself will not be perturbed, eventually all processes contain a version of the unperturbed code, yielding code stabilization. ☐

Note that Theorem 3 needs special read/write restrictions on the topology of the system. These are necessary in order to prevent a perturbed process from writing a perturbed version of the code into p. This cannot be prevented even if we assume that processes contain unique identifiers which cannot be perturbed by faults. The atomic update of the entire code state of the system is also necessary since otherwise two perturbed processes could "re-perturb" each other infinitely often if one of them is overwritten by p.

The assumption about the atomic update can be relaxed if we place restrictions on the scheduling of processes. Alternatively, we can weaken all of the above assumptions by assuming a local checking mechanism and reverting to probabilistic code stabilization, at the expense of requiring at least a constant size of unperturbed code space at *every* process.

Theorem 4. *If all processes have only remote read access to each other (and no remote write access), then it is sufficient that some process contains unperturbed code space of size $O(k)$ and all other processes contain unperturbed code space of size $O(1)$ to achieve probabilistic code stabilization.*

Proof. The central idea to construct a solution with the above characteristics is to use cryptographic hash functions [14]. A cryptographic hash function maps any finite string of bits to a fixed-size bit string, the *fingerprint*. Hash functions have the property that it is very hard to find *collisions*, i.e., two input strings that have the same fingerprint. In other words, it is very improbable that an arbitrary (random or intentional) perturbation of some bit string results in a bit string with the same fingerprint.

We augment every process with the following integrity checker program: Periodically, the process applies a cryptographic hash function to its own code and compares the resulting fingerprint with the value stored in its unperturbed code space. In case there is a mismatch, the process reads the code space of the totally unperturbed process and overwrites its own code with that copy. By doing this, any local code perturbations are erased. The only case that this does not happen is when code is perturbed to a state which has the same fingerprint as the legal code. The properties of cryptographic hash functions make this sufficiently improbable. The integrity checker together with the fingerprint can be implemented in constant space. Hence, probabilistic code stabilization with the claimed space requirements is achieved. □

In the proof of Theorem 4 it is necessary that all processes know from where to copy the unperturbed code. This information must be encoded in the constant size unperturbed part of their own code. Note also that the fingerprint must not be stored locally, it can be stored remotely at the same location where the unperturbed code resides or even can be computed on-the-fly. The method to implement the integrity check (a cryptographic hash function) can also be replaced by some form of error detecting code (like a CRC checksum) as long as faults can be assumed to be random.

5 Related Work and Concepts

The techniques described in Section 4 to achieve code stabilization in distributed systems have some similarities to other work in the area self-stabilization, namely the principle of local checking and correction [5] and work by Katz and Perry [13]. The idea is to regularly aquire a (local or global) snapshot of the state of the system and in case of discovered inconsistencies to locally correct or globally reset the system into a legal state. The problem in this area is to construct snapshot and reset procedures that are themselves self-stabilizing. In practice these methods can be found in the form of automatically generated or handcrafted runtime assertions within program code and exception handler mechanisms that perform corrective measures. However note, that all of these methods rely on the fact that the program code itself is unchanged.

Interestingly, there are close resemblances between our methods and the approaches from the area of security, more specifically from the area of (operating system) integrity management. There, *integrity* is defined as protection against unauthorized modification of the data and/or the code of a program. Integrity violations usually occur due to malicious actions by attackers. A common threat is a Trojan horse, a software which pretends to do something useful (like a screensaver or a computer game) but in fact alters your operating system in unforseen and unpleasant ways. Popular alterations are the installation of sniffers and keyloggers that capture sensitive data processed by the system, and post it on the Internet. Another typical alteration is the installation of a back door for a hacker, which enables unauthorized access to the system to outsiders. Modern operating systems have become so complex that these alterations usually are not noticed by the user or system administrator. Integrity management assumes that code is stored on writable media (like a hard disk) and aims at detecting even subtle modifications and wherever possible also to correct them.

Concepts to prevent the effect of these types of modifications are read-only files or filesystems that are supported by many of today's Unix-like operating systems (for example BSD 4.4 Unix offers read-only and append-only files, for a more involved discussion see Garfinkel, Spafford and Schwartz [9]). However, the most general approach in integrity management requires "clean" original copies of all the data and code which is part of the operating system. On a regular basis, the files of the running operating system are compared with the originals. If unauthorized alterations are found, the compromised version is replaced by the original version. The problems in integrity management correspond to the minimal requirements of code stabilization: Care must be taken that the original versions are unaltered and that the comparison and replacement software is also not compromised.

Maintaining a full clean copy of the original files and comparing it with the current ones on a computer is cumbersome in practice. This gave rise to a tool called *Tripwire* that exists in a commercial [2] and a freely available open source variant [1]. Tripwire maintains a database of cryptographic checksums of important files. This database has to be initialized by creating checksums of a known and unaltered baseline. At regular intervals, Tripwire takes snapshots of the system by comparing the checksums of the current version with the clean stored checksums. By reporting on mismatches, integrity violations can be detected or accepted changes merged into the database. Again it is vital that Tripwire itself is unaltered when it is run. Ideally, the filesystem is checked after booting a clean and original version of the operating system from CD including the Tripwire program itself. If Tripwire is executed off a compromised operating system, it may not operate in a trustworthy way [12]. The paradigm of Tripwire closely resembles the observations made in Theorem 4. Note that Tripwire needs to use cryptographic hash functions and not CRC checksums for example.

6 Conclusion

As noted by Ken Thompson in his 1984 Turing Award lecture [15], it is (almost) impossible to trust a system which you have not checked down to the transistor level. Today, integrity management software allows you to place trust on the integrity of your operating system. Integrity means prevention of unauthorized code or data modifications. Integrity is an increasingly important concern in today's computer systems, but requires a minimal amount of trustworthy code to be manageable.

In this paper we have revisited the notion of self-stabilization in a new context. Instead of allowing only data to be corrupted, we asked the question: To what extent can code corruptions be tolerated? We extended the notion of self-stabilization to also cover code corruptions. Our results on minimal unperturbed storage space and on techniques to achieve code stabilization directly reflect structures in the area of integrity management, and therefore can be used as a theoretical foundation for this important area of security.

References

1. Open source tripwire. Internet: `http://www.sourceforge.net/projects/tripwire/`.
2. Tripwire change auditing solutions. Internet: `www.tripwire.com`.
3. A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.
4. A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, Jan. 1998.

5. B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
6. E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.
7. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, Aug. 1975.
8. S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
9. S. Garfinkel, G. Spafford, and A. Schwartz. *Practical UNIX & Internet Security.* O'Reilly & Associates, 2003.
10. M. G. Gouda. Elements of security: Closure, convergence, and protection. *Information Processing Letters*, 77(2–4):109–114, 2001.
11. M. G. Gouda and N. J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, Apr. 1991.
12. halflife. Bypassing integrity checkers. *Phrack Magazine*, 7(51), Sept. 1997.
13. S. Katz and K. J. Perry. Self-stabilizing extensions for message-passing systems. *Distributed Computing*, 7:17–26, 1993.
14. A. J. Menezes, P. C. V. Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, FL, 1997.
15. K. L. Thompson. Reflections on trusting trust. *Communications of the Association for Computing Machinery*, 27(8):761–763, Aug. 1984.

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
1998-08 *	H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen
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
2000-04	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

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