INSTITUT FÜR INFORMATIK

UND PRAKTISCHE MATHEMATIK

Programmiersprachen und Rechenkonzepte

22. Workshops der GI-Fachgruppe 2.1.4
Bad Honnef, 2.-4. Mai 2004

Michael Hanus, Frank Huch (Hrsg.)

Bericht Nr. 0513
Oktober 2005

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

KIEL

Institut für Informatik und Praktische Mathematik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Programmiersprachen und Rechenkonzepte

22. Workshops der GI-Fachgruppe 2.1.4
Bad Honnef, 2.-4. Mai 2004

Michael Hanus, Frank Huch (Hrsg.)

Bericht Nr. 0513
Oktober 2005

e-mail: mh@informatik.uni-kiel.de, fhu@informatik.uni-kiel.de

Dieser Bericht enthält eine Zusammenstellung der Beitrage des
22. Workshops Programmiersprachen und Rechenkonzepte,
Physikzentrum Bad Honnef, 2.-4. Mai 2005.

Vorwort

Seit 1984 veranstaltet die GI–Fachgruppe 2.1.4 “Programmiersprachen und Rechenkonzepte”, die aus den ehemaligen Fachgruppen 2.1.3 “Implementierung von Programmiersprachen und 2.1.4 “Alternative Konzepte für Sprachen und Rechner” hervorgegangen ist, regelmäßig im Frühjahr einen Workshop im Physikzentrum Bad Honnef. Das Treffen dient in erster Linie dem gegenseitigen Kennenlernen, dem Erfahrungsaustausch, der Diskussion und der Vertiefung gegenseitiger Kontakte.
In diesem Forum werden Vorträge und Demonstrationen sowohl bereits abgeschlossener als auch noch laufender Arbeiten vorgestellt, unter anderem (aber nicht ausschließlich) zu Themen wie

- Sprachen, Sprachparadigmen
- Korrektheit von Entwurf und Implementierung
- Werkzeuge
- Software-/Hardware-Architekturen
- Spezifikation, Entwurf
- Validierung, Verifikation
- Implementierung, Integration
- Sicherheit (Safety und Security)
- eingebettete Systeme
- hardware-nahe Programmierung

In diesem Technischen Bericht sind einige der präsentierten Arbeiten zusammen gestellt. Allen Teilnehmern des Workshops möchten wir danken, dass sie durch ihre Vorträge, Papiere und Diskusion den jährlichen Workshop erst zu einem spannenden Ereignis machen. Ein besonderer Dank gilt den Autoren die mit ihren vielfältigen Beiträgen zu diesem Band beigetragen haben. Ein abschließender Dank gebührt noch den Mitarbeitern des Physikzentrums Bad Honnef, die durch ihre umfassende Betreuung für eine angenehme und anregende Atmosphäre gesorgt haben.

Kiel, im Oktober 2005 Michael Hanus, Frank Huch

Inhaltsverzeichnis

Complexity-Based Denial-of-Service Attacks on Mobile Code Systems

Andreas Gal[1], Christian W. Probst[2], and Michael Franz[1]

[1] Donald Bren School of Information and Computer Science
University of California, Irvine
Irvine, CA, 92697,U.S.A.
{gal,franz}@uci.edu
[2] Informatics and Mathematical Modelling
Technical University of Denmark
2800 Kongens Lyngby, Denmark
probst@imm.dtu.dk

Abstract. Although no such event has so far been reported, we predict that we will soon witness denial-of-service attacks on mobile-code systems that will be based on algorithmic complexity. For example, the worst-case performance of the standard Java Bytecode Verification rises quadratically with program length. By sending a legal, but difficult-to-verify program to a server virtual machine, we can keep that server occupied for an inordinate amount of time, effectively making it unavailable for useful work.
The problem is not restricted to verification alone: for example, an attacker could exploit knowledge about a just-in-time compiler's register allocator by sending it a particularly difficult to solve graph-coloring puzzle. This even puts into question the premise of open-source software, since it is knowledge of the underlying algorithm that is exploited in the attack, rather than a particular implementation defect.

1 Introduction

Safe mobile code is a major accomplishment. The two leading standards, the Java Virtual Machine and the .NET Common Language Runtime, provide target-machine independence in a code distribution format that can be verified by the code recipient prior to execution. Safety in such systems is based on code verification ahead of execution and runtime monitoring and resource control during execution. The research emphasis in this area so far has been on the *correctness* of the safety mechanism and its implementation, and numerous vulnerabilities have been discovered and removed.

In this paper, we contend that mere correctness of the safety enforcement mechanism is not sufficient to defend the host computer against mobile-code based attacks. Instead, there needs to be a dual focus on the (worst-case-)*performance* of the verifier and just-in-time compiler. Otherwise, as we will show, formally correct safe-mobile-code systems are vulnerable to a new class of denial-of-service (DoS) attack that exploits the *complexity characteristics* of the underlying verification and code-generation algorithms. Since the attack is located ahead of the point at which run-time resource

control sets in, and since it attacks the very mechanism upon which safety is founded, conventional defenses cannot fully quell this threat.

For example, the standard JVM bytecode verification algorithm exhibits quadratic worst-case complexity. We have been able to construct relatively small mobile programs in the Java JAR archive format that require hours of verification on high-end workstations. The programs in question are perfectly legal JVM code, perform no malicious action on the host, and will eventually be verified as being safe. However, the process of verification itself is so costly as to effectively constitute a denial-of-service attack.

The problem with the kind of attack that we describe in this paper is that the programs in question are not "illegal" in the sense that traditional safety mechanisms would defend against. In fact, there might be completely reasonable *valid and useful* programs with verification complexities similar to our attack programs. Hence, one cannot simply deploy a traditional monitor that would abort verification when a certain time limit is exceeded—unless one wants to also accept the random rejection of potentially important non-malicious programs. The risk of rejecting certain useful programs might be particularly unacceptable for an unattended server virtual machine.

Unfortunately, attacks based on algorithmic complexity affect not just the verifier, but the complete code path on the client. For example, an adversary that knows the target virtual machine's register allocation algorithm might be able to maliciously craft a valid mobile code program containing a particularly difficult to solve graph coloring puzzle.

An interesting point to note is that "open source" systems might be more vulnerable to this kind of attack than systems that provide "security by obscurity". Advocates of "open source" development have long argued that their systems are safer because the code is audited by hundreds of people and implementation defects are hence more easily spotted and removed. Our attack, however, does not depend on any implementation defect, but only on the underlying *algorithm*, which is publicly exposed in open-source development. Hence, the open-source process simultaneously increases the vulnerability to attacks such as ours while making it impossible to quickly react to an exposed vulnerability by changing the underlying algorithm.

The remainder of this paper is structured as follows: The next section gives a short introduction to mobile code verification, followed by a description of the current state of Java security (Section 3). This chapter also discusses successful earlier attacks and the related countermeasures that have been taken. Section 4 presents a whole class of new attacks based on complexity. A summary concludes our paper (Section 5).

2 Mobile Code Verification

Static mobile code verification was introduced as an alternative to the *dynamic* checking of type safety properties at runtime through dynamic execution monitors. The basic ingredient of every JVM bytecode verifier is an abstract interpreter that works on *types*, rather than *values*.

Leroy [19] lists the minimal conditions for bytecode to be accepted by the verifier:

– *Type correctness*. Bytecode instructions are typed and must receive arguments of appropriate types.

```
1: todo ← true
2: while todo = true do
3:    todo ← false
4:    for all i in all instructions of a method do
5:       if i was changed then
6:          todo ← true
7:          check whether stack and local variable types
            match definition of i
8:          calculate new state after i
9:          for all s in all successor instructions of i do
10:            if current state for s ≠ new state derived from i then
11:               assume state after i as new entry state for s
12:               mark s as changed
13:            end if
14:         end for
15:      end if
16:   end for
17: end while
```

Fig. 1. *The standard verification algorithm found in Sun Microsystem's JVM implementations.*

- *No stack overflow or underflow*. A method must never pop a value from the empty stack or push a value onto the largest stack specified for that method.
- *Code containment*. The program counter must always stay within the code limits of the current method and must always point to the beginning of an instruction.
- *Local variable initialization*. Variables must be initialized before being used.
- *Object initialization*. Whenever an object of a class C is created, one of the class' constructors must be called.

Except for code containment, all of these conditions require tracking the types of values as they are pushed onto and popped from the Java stack and written to and read from local variables. The Java specification provides an outline of a data-flow algorithm that can be used for this purpose. A simplified description of this algorithm is shown in Figure 1.

All Java Virtual Machine implementations that we are aware of, including Sun's own CVM [27] and HotSpot virtual machines [26], use (slight variations of) this algorithm to perform bytecode verification. The verification algorithm is performed separately for every method in a Java program. For each method, it iterates over all instructions of that method until no more operand type changes are observed. For each instruction i, the verifier checks whether the abstract data associated with i has changed. If so, it checks whether the current abstract local variable and stack content allow the execution of i and computes the new local variable and stack content. Finally, this new abstract state is propagated to all successors of i.

3 The Current State of Java Security

The Java Virtual Machine and its security architecture have been under intense scrutiny since their release. Over the years, several errors on different levels have been unveiled. Figure 2 gives an overview of the structure of the Java bytecode-execution framework and references to some of the reported flaws and shortcomings in either the implementation or the architecture. The whole security concept of Java collapses at the moment at

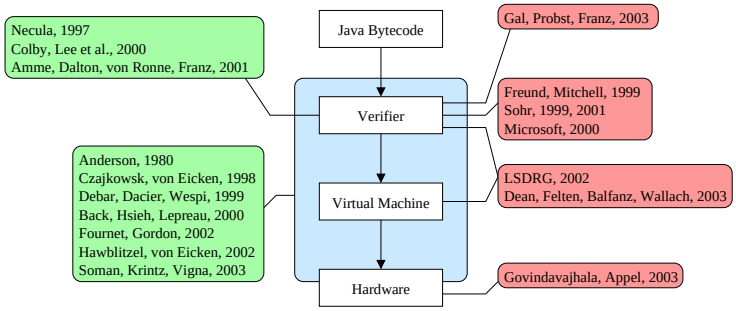


Fig. 2. *High-level structure of a bytecode-execution framework. The call-outs on the right-hand side refer to successful attacks and implementation flaws found in the JVM, the call-outs on the left-hand side to attempts to enhance the security and reliability of the virtual machine.*

which just *one* of the components is compromised. Below, we will give a brief overview over some of these attacks. It is noteworthy that all of these attacks are based on *implementation* errors rather than *conceptual* flaws. This is in contrast to the attack presented in this paper, as its enabling property is the bytecode verification algorithm itself and not a specific implementation thereof.

Beside the work described in this section, there has also been intensive research on how to actually secure the transport and the execution of mobile-code programs. Approaches include enhanced transport formats [1, 21, 6], host based intrusion-detection systems [2, 10], auditing systems [25], stack inspection [11], and extensions to the actual execution unit [3, 7, 17]. All of these approaches either try to rescue what already has been lost—the system is no longer dependable due to probable implementation faults—or try to replace the current verification scheme with mechanisms that are hopefully easier to implement and prove correct.

3.1 Implementation-Based Attacks

In attacking a mobile-code execution framework, the verifier is one of the obvious targets. The verifier has the obligation to prohibit any execution of unsafe code, where *unsafe* is defined with respect to the criteria defined in the respective framework. Due to its importance, verification can not be interrupted by the user without shutting down the whole virtual machine. Obviously, for attacking a virtual machine it is of importance to bring a hostile applet through the verification process. Since the verifier has the obligation to prohibit exactly this, this task requires profound inside knowledge and usually profits from implementation errors.

Sohr [23] presents an example for the exploit of faulty implementations. In this attack, the verifier *overlooks* certain code sequences that are passed on for execution *without having been verified*. This, in turn, allows to construct methods that use the unverified code to return objects of one type that by the type system are believed to be of another type. In later work, Sohr [24] reports on a similar problem in a version of Microsoft's bytecode verifier [20], based on incorrect handling of values of local variables in the verifier.

Freund et al. [12] exploit another flaw in a certain implementation of the verifier, which is based on the handling of subroutines and uninitialized objects in the verifier. By creating several uninitialized objects of a class and only initializing one of them, the virtual machine can be made to invoke methods on uninitialized objects. This fault is enabled by incorrect handling of initialization of newly created objects in the verifier.

Hopwood [18] presents another exploit demonstrating that certain versions of the verifier could be made to load classes with absolute class names instead of relative ones. Thus, a class could first be uploaded to a client without verification and afterwards be dynamically loaded in the virtual machine. However, since it would be loaded from the local host, there would not be any verification. This trust in local files is applied to avoid repeated re-verification of locally installed classes such as the pre-installed system classes of the APIs of mobile-code execution frameworks.

The actual hardware running the virtual machine can also be attacked, e.g. by raising the probability of a bit error [16]. In contrast to the exploits described so far, this is an area of attack that can not be handled by means of the virtual machine itself, but by taking hardware measures to minimize the probability of an uncorrectable memory error.

3.2 Class Loader-Based Attacks

The first attack that included the Java Virtual Machine itself was the *Princeton Class Loader Attack* [9]. It exploited a combination of flaws—an erroneous implementation of the verifier and the class loading mechanism that implements dynamic loading of classes in the JVM. The faulty implementation allowed the attacker to overwrite system classes with malicious code. The loaded classes were actually verified as valid while they should have been rejected, and the methods where later on called in place of the overwritten methods. The whole concept of dynamic class loading is one of the fundamental weak spots in the Java security architecture [8].

4 Complexity-based Attacks

The techniques presented in the previous section illustrate some of the documented approaches of attacking a Java Virtual Machine. We consider the complexity-based verifier attack to be more severe than these attacks, because it is enabled by the fundamental design properties of the Java bytecode execution framework, and not by errors in its implementation. The code sequences used in this exploit are *legal* and *correct* mobile code programs and as such not rejectable by the verifier. While Microsoft's .NET platform addresses some of the shortcomings that allow such complexity-based attacks to occur, it still uses fundamentally very similar verification algorithms and is very likely susceptible to similar kind of attacks.

4.1 Complexity of Verification

Regarding the complexity of verification, the analysis of straight-line code is inexpensive, since the abstract interpreter only propagates type information through the instructions and computes the abstract stack state after each instruction.

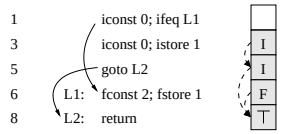


Fig. 3. *Verification of Java bytecode through iterative data-flow analysis. The verifier traverses the method from the first instruction to the last. The example code shown here contains no backward branches, and hence the analysis can be completed in a single iteration.*

The runtime of such a data-flow analysis is significantly increased if the code contains jumps, exception handlers, or subroutines, which introduce forks and joins in the control-flow graph. When separate control flows are merged together, an instruction's predecessors may have different abstract stack or variable types. After merging the state information of the two incoming control flows, the data-flow analysis has to be repeated for all instructions which are reachable from this point in the control flow of the method. For simplicity, the standard Java verifier repeats the entire data-flow analysis for every instruction of a method until there are no more changes.

For average Java programs, the verification algorithm quickly reaches a fixed point after only a few iterations. For straight-line code or code that contains only forward branches, the verification algorithm terminates already after a single iteration (Figure 3). It is obvious that—*in theory*—the Java verifier could need up to n iterations over the method, with n being the number of instructions in the method. Since for each iteration the verifier might have to visit all instructions, the overall complexity is at least $O(n^2)$. Such quadratic runtime behavior does not only exist in theory. In fact, simple Java programs can expose the worst-case scenario in practice. Figure 4 shows a very simple Java program that does nothing but store an integer into a local variable and jump backwards through the code until it eventually returns.

Due to the order in which the verification algorithm visits instructions, information is forwarded immediately to instructions that come syntactically after the current instruction. To instructions that come syntactically *before* the current instruction, the new abstractions will only be forwarded in the next iteration. Once an instruction has been visited for a particular iteration, it will not be visited again, even if new information about the operand types of that instruction is computed. For the example in Figure 4, the verifier is forced to perform an iteration for every backwards jump.

The simplistic approach of the standard Java bytecode verification algorithm to iterate over the bytecode until a fixed point is reached simplifies the generation of attacks like the one shown in Figure 4, but any other iteration order would also exhibit a particular (probably different) worst-case behavior for which a malicious program could be constructed.

4.2 Exploiting the Worst-Case Behavior

We have measured the verification time for two malicious programs designed to exhibit the worst-case performance of the Java verifier using the Sun Microsystems Java 2 HotSpot Client VM [13, 15].

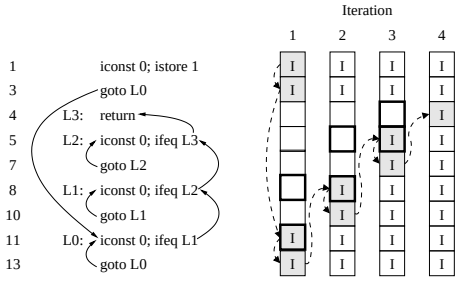


Fig. 4. *Java bytecode program that takes n iterations to be verified using Sun's standard DFA verifier approach. The entry state for each basic block depends on the successor basic block. The type of the first local variable is displayed for each iteration of the DFA. It is initially assumed to be of unknown type and is discovered to be an integer (I) during successive iterations. Shaded boxes indicate a change in the current iteration, framed boxes will be visited in the next iteration.*

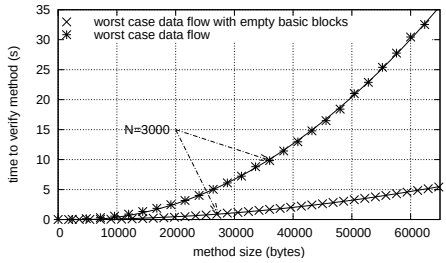


Fig. 5. *Verification time for verifying a single method containing a worst-case data-flow scenario. The x-axis indicates the length of the method bytecode in bytes, which is proportional to the number of basic blocks N used to construct the code. The arrows indicate for comparison purposes the code size for path length $N = 3000$.*

Figure 5 shows the verification time for a single method containing bytecode with an increasing maximum data-flow path of length N. This time includes only the time it takes the verifier to prove safety. The code is never actually executed or compiled to executable code. The first curve shows the verification time for a worst-case path length problem with empty basic blocks. The second curve in the graph shows the maximum flow path problem with some additional code added to each basic block, which further slows down the verifier. Both curves clearly show quadratic growth.

All measurements were taken on a 2.53 GHz Pentium 4 and the Sun HotSpot VM 1.41. The maximum verification time we observed on this machine *for a single method* was approximately 40 seconds. Since the size of method code in Java is limited, this time can not be increased. However, to achieve even longer verification times, an attacker could hide more than just one of these methods in the code. Just including 20 methods instead of one would already increase the verification time to approximately 15 minutes on the test machine we used.

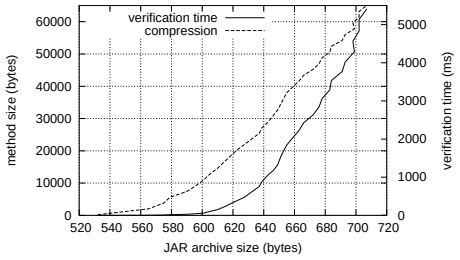


Fig. 6. *Compression of constructed code examples using the standard JAR archive format. The code is extremely well compressible as it repeats identical code patterns. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes.*

The standard JAR archive format can be used to drastically reduce the apparent size of the malicious code. The code patterns used in the presented scenarios lend themselves for compression due to their very regular structure. Figure 6 indicates the compressed size for different problem lengths N. While the verification times increases by over factor 5000, the JAR file merely grows by less than 200 bytes. The JAR archive format thus represents another example of a well-meant algorithm with appropriate average-case performance, which however exhibits very unexpected worst-case behavior.

We have used the two algorithmic shortcomings described here to construct a malicious applet [14] that disables the Java VM of web browsers for several minutes. The applet is 10kb in size and indistinguishable from regular applet code, because in the end it is a still legal and correct Java program.

Short of disabling Java applets, the user cannot prevent or interrupt the loading of this applet. In fact, existing browsers do not even allow the user to interrupt the verification because the browser implementor never considered the verification time to be costly enough. Some browsers, including some versions of the Microsoft Internet Explorer, allow the verifier to continue the verification silently and continue to hog the CPU in the background even if the user leaves a website containing an applet that takes an excessive amount of time to verify.

4.3 Attacking the Compilation Pipeline

Denial-of-service attacks are not limited to bytecode verification, which is executed early in a bytecode execution framework. Any algorithm applied to mobile code during its path from a portable bytecode format to natively executable machine code is vulnerable at its point of worst-case complexity. This applies in particular to compiler optimization algorithms, which are traditionally chosen for speed in the average case but not for worst-case performance, and some of which use heuristics to solve problems like graph coloring and instruction scheduling that are known to be NP-complete.

An example for such an attackable optimization algorithm is register allocation. Register allocation is an important component of any JIT compiler that strives to achieve good code quality. The classic register-allocating algorithm is structured after Chaitin's graph coloring allocator [5, 4]. Many improvements and variants have been proposed,

but most of this research was focused on improving the average-case performance. Poletto et al. showed that register allocation using graph-coloring has a quadratic worst-case complexity for certain pathological cases [22] and proposed a linear-scan algorithm for register allocation. This algorithm is not guaranteed to find the optimal register allocation for any given problem, but has a linear worst-case performance. To truly harden the virtual machine against complexity-based denial-of-service attacks, this principle of trading off some code quality in return for linear time complexity has to be extended to the entire code processing pipeline.

5 Conclusion

Future software-application architectures are moving to Grid- and service-based architectures, in which computations are sent to hosts for execution. Soon, these service-based execution frameworks will be omni-present, making the actual network-based execution mechanism invisible to the user. In these architectures, efficient algorithms for each step in the chain from *receiving mobile code* to *compiling it to native code* and *executing it* will be needed to protect against complexity-based attacks. The threat of these subtle denial-of-service attacks has been neglected, apparently because it does not occur in daily average-case use of mobile code. In the case of an unsupervised server at the heart of a service-based framework, however, having the framework verifying, analyzing, compiling, and executing many mobile-code programs in parallel will make each and every phase in the framework vulnerable to complexity-based attacks.

With the currently widespread mobile-code execution frameworks in place, there is no quick fix to this problem. Instead, we will need to rethink the architecture of those systems—while current systems place verification as a hurdle for incoming code and after that use fine tuned algorithms that have been selected for their average case behavior, we will need to construct systems where each step has a provable *worst-case* behavior. Until these systems are in place, open-source code development actually worsens the situation. Instead, security by obscurity actually works.

References

1. W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A Type Safe and Referentially Secure Mobile-Code Representation Based on Static Single Assignment Form. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 137–147, June 20–22, 2001. *SIGPLAN Notices*, 36(5), May 2001.
2. J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P Anderson Co., Fort Washington, PA, Apr. 1980.
3. G. Back, W. H. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in java. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI-00)*, pages 333–346, Berkeley, CA, Oct. 23–25 2000. The USENIX Association.
4. G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN 1982 Symposium on Compiler Construction (CC)*, pages 98–105, Boston, MA, June 1982.

5. G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, Martin, E. Hopkins, and P. W. Markstein. Register allocation via graph coloring. *Computer Languages*, 6(1):47–57, 1981.
6. C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. *ACM SIGPLAN Notices*, 35(5):95–107, May 2000.
7. G. Czajkowski and T. von Eicken. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices*, 33(10):21–35, Oct. 1998.
8. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, 1999.
9. D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java security: Web browsers and beyond. In D. E. Denning and P. J. Denning, editors, *Internet Besieged: Countering Cyberspace Scofflaws*, pages 241–269. ACM Press / Addison-Wesley, New York, 1998.
10. H. Debar, M. Dacier, and A. Wespi. Towards a taxonomy of intrusion detection systems. *Computer Networks*, 31(8):805–822, Apr. 1999. Special issue on Computer Network Security.
11. C. Fournet and A. D. Gordon. Stack inspection: theory and variants. *ACM SIGPLAN Notices*, 37(1):307–318, Jan. 2002.
12. S. N. Freund and J. C. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.
13. A. Gal, C. W. Probst, and M. Franz. A Denial of Service Attack on the Java Bytecode Verifier. Technical Report 03-23, University of California, Irvine, School of Information and Computer Science, 2003.
14. A. Gal, C. W. Probst, and M. Franz. An Applet performing a complexty-based Denial-of-Service attack on the verifier. Available at `http://nil.ics.uci.edu/exploit`, 2005.
15. A. Gal, C. W. Probst, and M. Franz. Integrated Java Bytecode Verification. In *Proceedings of the First Workshop on Abstract Interpretation of Object Oriented Languages*, January 2005.
16. S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 Symposium on Security and Privacy*, pages 154–165, Los Alamitos, CA, May 11–14 2003. IEEE Computer Society.
17. C. Hawblitzel and T. von Eicken. Luna: A flexible java protection system. In *Proceedings of the 5th ACM Symposium on Operating System Design and Implementation (OSDI-02)*, Operating Systems Review, pages 391–403, New York, Dec. 9–11 2002. ACM Press.
18. D. Hopwood. Java Security Bug (Applets Can Load Native Methods). *RISKS Forum*, 17(83), 1996.
19. X. Leroy. Java Bytecode Verification: Algorithms and Formalizations. *Journal of Automated Reasoning*, 30(3/4):235–269, 2003.
20. Microsoft Corporation. Microsoft Security Program: Microsoft Security Bulletin (MS99-045): Patch Available "Virtual Machine Verifier" Vulnerability, 1999.
21. G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Paris, France, January 1997.
22. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.
23. K. Sohr. Nicht verifizierter Code: eine Sicherheitslücke in Java. *JIT 1999*, 1999.
24. K. Sohr. *Die Sicherheitsaspekte von mobilem Code*. PhD thesis, Universität Marburg, 2001.
25. S. Soman, C. Krintz, and G. Vigna. Detecting malicious java code using virtual machine auditing. In *Proceedings of the 11th USENIX Security Symposium*, pages 153–168. USENIX, Aug. 2003.
26. Sun Microsystems. The Java Hotspot Virtual Machine, 2002.
27. Sun Microsystems. CDC: An Application Framework for Personal Mobile Devices, http://java.sun.com/products/cdc/, 2003.

On Simulating Nested Procedures by Nested Classes – Extended Abstract

Wolfgang Goerigk

Christian-Albrechts-Universität zu Kiel, Germany
wg@informatik.uni-kiel.de

Abstract. Nested procedures are common in imperative languages like Algol, Pascal or Modula 2. From a programming viewpoint, procedure nesting allows for information hiding, which is one key principle in software engineering. However, in modern object oriented languages, like C++ or Java, procedure nesting has been dropped like in C in favor of open subroutines and flat program structure. This is very often seen as a major drawback of object oriented languages, and the more recent Java versions reintroduce nesting on class level as a principle feature; we still do not find procedure nesting, though.
We know that procedure nesting has an important impact on the expressive power of Algol-like procedure concepts, at least from a theoretical point of view. It enables irregular calling trees if we assume the usual static scoping discipline. Even uninterpreted programs can simulate Turing machines, or more precisely, two tape pushdown automata, whereas, just by using procedures and scoping, flat programs cannot.
Computational completeness of uninterpreted programs with procedure nesting renders many formal decision problems undecidable, like e.g. formal reachability. Hence, it is an interesting question if or not the Java class nesting feature is as powerful as procedure nesting in Algol-like programs. We will give a positive answer and define a simulation of nested Algol procedures by nested Java classes. Interestingly enough, this might have a major impact on the decidability of formal Java-program properties.

1 Introduction and Motivation

The work presented in this paper is mainly motivated by an ongoing discussion on the relations between classical imperative and modern object oriented programming languages. In particular, procedure nesting and formal procedures as we find them in Algol, Pascal or Modula-2, have been left out in languages like C or Java in favor of unnested open subroutines. Moreover, the concept of *late binding* (dynamic scoping) for method invocation does not really facilitate the understanding of conceptual relations between Java-like object oriented languages and Algol-like procedural languages.

The procedure concept of imperative languages with procedure nesting and formal procedures with static scoping is well understood since quite a long time.

A whole bunch of work has been done since more than 30 years to understand and to formalize the semantics of Algol-like languages, and to develop correct and even verified implementation techniques.

From a software engineering viewpoint, nesting is a key technique for information hiding: It reduces the number of global dependencies and helps to excapsulate and to more abstractly handle implementation decisions. More recent versions of the Java language reintroduce nesting at least on class level

It is well-known that object oriented programs can be simulated by imperative languages. Late binding can either be implemented by generic procedures, which perform a runtime dispatch on the type of message receiver objects [2], or by procedure variables (i.e. formal procedures) as part of the receiver's class object [1]. C programmers often use the techniques of pointers, structures and runtime dispatch in large programs to simulate objects and message passing. Thus, Algol-like programs are sufficiently expressive to support the simulation of object oriented programs.

On the other hand, we do not directly find formal and nested procedures in languages like Java, in particular not with a similar static scoping discipline. Nesting and static scoping in Algol have a major impact both with respect to correct implementation (we need a static procedure chain in order to address global variables) and with respect to formal program properties. So, for instance, the problem of formal reachability is undecidable in full Algol, whereas it is decidable for unnested open subroutines [3, 4]. The main reason is, that calling trees can be irregular in full Algol, but with open (unnested) procedures (and thus without global formal variables) programs can only construct regular calling trees.

As a consequence, full Algol procedures are formally Turing-complete, whereas programs without procedure nesting are not. For many reasons, it is interesting to ask if Java-like object oriented programs, the other way around, can simulate full Algol programs as well. We will give a positive answer in this paper.

If people view at object oriented languages as an alternative for classical imperative languages, in particular because they find the concept of nested and formal procedures with static scoping quite hard to understand, this paper is kind of a bad news. Unfortunately, Java programs in general are not at all easier to understand.

2 Nested and Formal Procedures in Algol

Let us look at the following Algol-like program, which we want to use as a protagonist, as a running example of a (terminating) program which is programmable in Java with class nesting. Our program has nested procedures and global formal parameters, i.e. `b`. The program is formally terminating, but nevertheless recursive. It does not have the formal most-recent property. In general, such programs may have irregular calling trees, and we are not able to construct formally equivalent programs without procedure nesting.

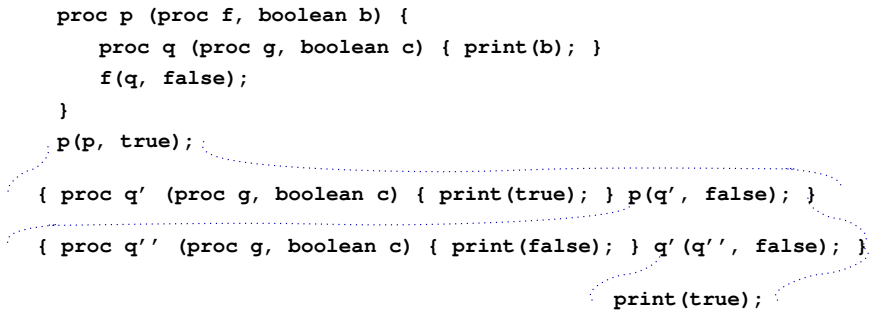
```
proc p (proc f, boolean b) {
  proc q (proc g, boolean c) { print(b); }
  f(q, false);
}
p(p, true);
```

Since the Algol procedure concept does not support higher order procedures or partial evaluation, the standard technique of transforming global parameters to regular procedure parameters does not help in order to construct a formally equivalent program without procedure nesting. Within the body of `p`, the procedure `q` depends on the global formal parameter `b`. Therefore, `q(b)` would denote the procedure with the global binding for `b`, but the corresponding partial evaluation is not allowed in Algol-programs:

```
proc p (proc f, boolean b)
  { f(q(b), false); }

proc q (boolean b, proc g, boolean c)
  { print(b); }
```

From [3, 4] we learn that it is in general impossible to formally equivalently transform full Algol-programs to programs without procedure nesting and global formal parameters. In full Algol the so-called macro property (or formal termination) is undecidable, whereas it is decidable for programs without procedure nesting. The proof shows, that (uninterpreted) full Algol programs can effectively simulate 2-tape pushdown automata, which in turn can simulate Turing-machines. Moreover, the simulating automaton has an explicit `stop`-instruction and the construction thus also shows, that formal reachability is undecidable in full Algol, whereas it is decidable in an Algol without procedure nesting or global formal parameters.



Intuitively, the main reason is that full Algol-programs can construct irregular calling trees (the procedures called again and again differ with respect to the bindings of their global formal parameters), whereas for programs without nested procedures the calling trees can at most be regular, and a program is formally non-terminating if and only if it is formally recursive. Our protagonist is a formally recursive, but nevertheless terminating program, which makes it interesting as an example for our transformation of nested Algol-procedures to nested Java-classes.

3 Classes as Procedures

Typically, methods play the rôle of procedures or functions in an object oriented language like Java. However, since methods are called using dynamic scoping (late binding), methods are not really useful to simulate Algol-procedures in our setting. Although recent Java versions allow for class nesting (local and inner classes), methods still are kind of unnested; there is no support of global formal method parameters. Classes can refer to instance variables of embracing classes, but in general not to parameters of embracing method declarations (unless they are declared *final*, so that the implementation can copy their values into the statically preceeding object).

But we can look at classes as procedures, and at instantiations as procedure calls. That is to say, nested Java classes can play the rôle of nested Algol procedures (actually of type declarations for corresponding procedure incarnations or activation records). Each Algol-procedure `p` is transformed to a corresponding Java class `Cp`, local procedures become local classes, and procedure calls of `p(`e_1`, ..., `e_n`)` are transformed to instantiations `new Cp(`e_1`, ..., `e_n`)`. Consequently, procedure bodies transform to constructor bodies in Java. Here is a simple example:

```
proc fac (int x, int y) {
   if (x == 0) print(y); else fac(x-1, y*x);
}
... fac(3, 1); ...
```

```
class Cfac {
    Cfac (int x, int y) {
        if (x == 0) print(y); else new Cfac (x-1, y*x);
    }
}
... new Cfac (3, 1); ...
```

3.1 Parameter Passing

In general, parameter passing in Java is *call-by-value* and there is no direct counterpart to the *call-by-reference* parameter passing of Algol-like procedures. But classes are reference types, and we can simulate call-by-reference parameter passing by call-by-value passing of objects. For that, for each type `T` we would introduce a class

```
class varT {
   T value;
   varT (T value) { this.value = value; }
}
```

and replace each variable declaration `T x = v;` by `varT x = new varT (v);` and each reference to `x` by a reference to `x.value`. Using this transformation, our factorial program from above would become

```
proc fac (var int x, var int y) {
   if (x != 0) { y=y*x; x=x-1; fac(x,y); }
}
var int x = 3, y = 1; fac(x, y); print(y);
```

```
class Cfac {
    Cfac (varint x, varint y) {
       if (x.value != 0) {
          y.value = y.value * x.value;
          x.value = x.value - 1;
          new Cfac (x, y);
       }
    }
}
varint x = new varint (3); varint y = new varint (1);
new Cfac (x,y); print(y);
```

Although it is not so important for the present paper, we want to make clear that there is no principle problem to transfer formal program properties expressed by a copy-rule-based (or term rewriting) semantics of Algol-programs to the corresponding Java-program. In order to define such a copy rule for Algol-programs we need procedure calls to be replaced by modified procedure bodies, and it is much more convenient to base such a definition on call-by-reference parameter passing, i.e. the textual replacement of formal procedure parameters by actual parameter variables (with an apporpriate renaming to avoid name clashes).

3.2 Class Nesting and Procedure Nesting

Procedure nesting does not as easily transform to class nesting as we might expect. The reason is, as mentioned before, that in Java local classes in general cannot refer to method parameters of embracing methods:

```
proc p (int x) {
   proc q () { print(x); }
   q();
}
p(3);
```

```
class Cp {
   Cp (int x) {
      class Cq {
         Cq () { print(x); } }
      new Cq ();
   }
}
new Cp (3);
```

From an implementation point of view we have a static chain of objects i.e. an object of a local class contains a pointer to the (necessarily instantiated) object of the embracing class. But this static chain of objects does not include method incarnations nor constructor incarnations of course. There are no global formal parameters. But we need them in order to program Algol-procedures. Therefore, we need parameters and local variables of methods to become instance variables of the corresponding embracing class.

```
proc p (int x) {
   proc q () { print(x); }
   q();
}
p(3);
```

```
class Cp {
   Cp (int x) {
      int x;
      class Cq {
         Cq () { print(x); } }
      this.x = x; new Cq ();
   }
}
new Cp (3);
```

The solution is to model procedure parameters and local variables as additional instance variables of the procedure classes. Since the constructor models procedure calls, it additionally sets these (local) variables, so that procedure class instances play the rôle of Algol procedure incarnations:

```
proc p (T1 x1, ... , Tk xk) {
   Tk+1 y1 = v1; ... ; Tk+n yn = vn;
   s
}
```

```
class Cp {
    T1 x1; ... ; Tk xk;    // parameters
    Tk+1 y1 = v1; ... ; Tk+n yn = vn;    // local variables
    Cp (T1 x1, ... , Tk xk) {    // constructor
         this.x1 = x1; ... ; this.xk = xk;
         s'
    }
}
```

3.3 Procedures as Parameters

For each procedure we additionally generate a procedure type `FCp` in order to model procedure objects.

```
proc p (T1 x1, ... , Tk xk) {
   T'1 y1 = v1; ... ; T'n yn = vn;
   s
}
```

```
class FCp implements proc_T1_..._Tk {
    public void call (T1 x1, ... , Tk xk) {
       new Cp (x1, ... , xk);
    }
}
```

A corresponding interface declaration allows for passing procedures as parameters. For each procedure type we generate an appropriate interface declaration, and for each procedure (`p`) we generate a pair of class declarations; one (`Cp`) for procedure incarnations (activation records) and one (`FCp`) for the corresponding (formal) procedure objects. The `call`-method of the latter is used to express formal procedure calls.

```
proc p (T1 x1, ... , Tk xk) {
   T'1 y1 = v1; ... ; T'n yn = vn;
   s
}
```

```
interface proc_T1_..._Tk {
    public void call (T1 x1, ... , Tk xk);
}
```

4 The Transformation

Each Algol-procedure is transformed to a pair of Java class declarations,

- one for procedure incarnations, and
- one for procedure objects, with an appropriate `call`-method.

Procedure calls become constructor calls (instantiations), and, for technical reasons, we define an interface type for each procedural type. Thus, our protagonist

```
proc p (proc f, boolean b) {
  proc q (proc g, boolean c) { print(b); }
  f(q, false);
}

p(p, true);
```

transforms to the following Java program:

```
interface proc_proc_boolean {
    public void call (proc_proc_boolean f, boolean b);
}

class FCp implements proc_proc_boolean {
    public void call (proc_proc_boolean f, boolean b) {
        new Cp(f, b);
    }
}
class Cp {

    proc_proc_boolean f; boolean b;

    public Cp (proc_proc_boolean f, boolean b) {
        this.f = f; this.b = b; f.call(new FCq(), false);
    }

    class FCq implements proc_proc_boolean { ... }
    class Cq { ... }
}
```

and the main program tranforms to

```
class main {
```

```
    public static void main (String[] argv) {
        new Cp(new FCp(), true);
    }
}
```

5 Conclusions and Future Work

Java programs with class nesting can simulate the full Algol-procedure concept. As a consequence, also for Java programs formal termination and formal reachability are undecidable. This might have harmful consequences for byte code verification. Moreover, since nested procedures cannot simply be flattened, it remains an interesting question if nested classes can be defined by corresponding unnested (flattended) classes.

References

1. Ch. Blaue. *Anforderungen bei industriellem Softwareengineering: Prozedurale Implementierungstechniken im objektorientierten Entwicklungsprozess.* PhD thesis, Technische Fakultät der Christian-Albrechts-Universität, Kiel, 2002.
2. W. Goerigk. *Korrektheit der Übersetzung objektorientierter Wissensrepräsentationssprachen mit statischer Vererbung.* PhD thesis, Math.-Nat. Fakultät der Christian-Albrechts-Universität, Kiel, 1993.
3. H. Langmaack. On Procedures as Open Subroutines I. *Acta Informatica*, 2:311–333, 1973.
4. H. Langmaack. On Procedures as Open Subroutines II. *Acta Informatica*, 3:227–241, 1974.

With-Loop Fusion in SAC

Clemens Grelck[1], Karsten Hinckfuß[1], and Sven-Bodo Scholz[2]

[1] University of Lübeck, Germany
Institute of Software Technology and Programming Languages
{grelck,hinckfus}@isp.uni-luebeck.de
[2] University of Hertfordshire, United Kingdom
Department of Computer Science
s.scholz@herts.ac.uk

Abstract. WITH-loops are versatile array comprehensions used in the functional array language SAC to implement aggregate array operations that are applicable to arrays of any rank and shape. We describe the fusion of WITH-loops as a novel optimization technique to improve the data locality of compiled code. Several experiments show the significance of WITH-loop fusion for achieving runtime performance figures that are competitive with those of low-level machine-oriented approaches.

1 Introduction

SAC (Single Assignment C) [1] is a purely functional array processing language designed with numerical applications in mind. Image processing and computational sciences are two examples of potential application domains. The language design of SAC aims at combining generic, high-level specifications of array-based algorithms with a runtime performance that is competitive with low-level, machine-oriented languages both in terms of execution time and memory consumption.

The programming methodology of SAC essentially builds upon the principles of abstraction and composition [2–4]. Unlike other array languages, SAC provides only a very small number of built-in operations on arrays, mostly for querying an array's shape, its rank, or individual elements. Aggregate array operations, e.g. subarray selection, element-wise extensions of scalar operations, rotation and shifting, or reductions, are defined in SAC itself using WITH-loops, verstile SAC-specific multi-dimensional array comprehensions. SAC allows us to encapsulate these operations in abstractions that are universally applicable, i.e. applicable to arrays of any rank and shape. More complex array operations are not to be defined by WITH-loops, but by composition of simpler array operations. Again, they are encapsulated in functions that may abstract from concrete ranks and shapes of argument arrays as far as possible and useful.

Following this technique, entire application programs typically consist of various logical layers of abstraction and composition. This style of programming leads to highly generic implementations of algorithms and provides good opportunities for code reuse on each abstraction layer. As a very simple example,

consider a function `MinMaxVal` that yields both the least and the greatest element of an argument array. Rather than implementing this functionality directly using WITH-loops, our programming methodology suggests to define the function `MinMaxVal` by composition of two simpler functions `MinVal` and `MaxVal` that yield the least and the greatest element, respectively:

```
int, int MinMaxVal( int[*] A)
{
  return( MinVal( A), MaxVal( A));
}
```

Direct compilation of programs designed on the principles of abstraction and composition generally leads to poor runtime performance. Massive creation of temporary arrays as well as repeated traversals of the same arrays are the main reasons. Computing the minimum and the maximum value of `A` individually requires the processor to load each value of `A` into a register twice. Whereas this is fairly efficient for small arrays that entirely fit into the processor-local cache memory, larger arrays lead to expensive 2nd-level cache or even more expensive main memory transactions. With increasing discrepancy between processor and memory speeds, the above implementation of `MinMaxVal` results in a performance penalty that approaches a factor of two with respect to a direct implementation that computes both values in a single sweep.

Our programming methodology represents a classical trade-off between modular, reusable code design on the one side and runtime performance on the other side. Whereas in many application domains a performance degradation of a factor of 2 or more in exchange for improved development speed, maintainability, and code reuse opportunities may be acceptable, in numerical computing it is not. Hence, in practice abstraction and composition are only useful to the extent to which accompanying compiler optimization technology succeeds in systematically transforming programs from a representation amenable to humans into a representation that is suitable for efficient execution on computing machinery.

In the past, we have developed two complementary optimization techniques that avoid the creation of temporary arrays at runtime: WITH-loop folding [5, 6] and WITH-loop scalarization [7]. In our current work we address the problem of repeated array traversals, as illustrated by the example above. We propose WITH-*loop fusion* as a novel technique to avoid costly repeated array traversals at runtime. To make fusion of WITH-loops feasible, we extend the internal representation of WITH-loops in order to accomodate computation of multiple values by a single WITH-loop, named *multi-operator* WITH-loop. We introduce WITH-loop fusion as a high-level code transformation on intermediate SaC code. The essence of fusion is formally defined in a very restricted setting. Additional preprocessing techniques that systematically generate optimization cases are outlined.

The remainder of this paper is organized as follows. Section 2 provides a brief introduction into WITH-loops. In Section 3 we extend the internal representation of WITH-loops to multi-operator WITH-loops. The base case for WITH-loop fusion is described in Section 4. More complex cases are reduced to the base case using techniques described in Section 5. Section 6 reports on a series of experiments

evaluating WITH-loop fusion while Section 7 concludes and outlines directions of future research.

2 With-loops in SAC

General introductions to SAC and its programming methodology may be found in [2, 1, 3]. Here, we restrict ourselves to the explanation of WITH-loops, more precisely WITH-loop expresssions. Their syntax is defined in Fig. 1.

WithLoopExpr	⇒	**with** ⌈ *Generator* **:** *Expr* ⌋$^{+}$ *Operation*
Generator	⇒	**(** *Expr* `<=` *Identifier* `<` *Expr* ⌈ *Filter* ⌋ **)**
Filter	⇒	**step** *Expr* ⌈ **width** *Expr* ⌋
Operation	⇒	**genarray** **(** *Expr* ⌈ **,** *Expr* ⌋ **)**
		| **fold** **(** *FoldOp* **,** *Expr* **)**

Fig. 1. Syntax of with-loop expressions.

A WITH-loop consists of three parts: a *generator*, an *associated expression* and an *operation*. The operation determines the overall meaning of the WITH-loop. There are two variants: `genarray` and `fold`. With `genarray(` ***shp***`,` ***default***`)` the WITH-loop creates a new array of shape ***shp***. With `fold(` ***foldop***`,` ***neutral***`)` the WITH-loop specifies a reduction operation. In this case, ***foldop*** must be the name of an appropriate associative and commutative binary operation with neutral element given by the expression ***neutral***.

The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to integer vectors of equal length, define lower and upper bounds of a rectangular index vector range. For each element of this set of index vectors the associated expression is evaluated. Depending on the variant of WITH-loop, the resulting value is either used to initialize the corresponding element position of the array to be created (`genarray`) or it is given as an argument to the fold operation (`fold`). In the case of a `genarray`-WITH-loop, elements of the result array that are not covered by the generator are initialized by the (optional) default expression in the operation part. For example, the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
genarray( [3,5], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0\\ 0&2&3&4&0\\ 0&3&4&5&0 \end{pmatrix}$ while the WITH-loop

```
with ([1,1] <= iv < [3,4]) : iv[0] + iv[1]
fold( +, 0)
```

evaluates to 21. An optional filter may be used to further restrict generators to periodic gridlike patterns, e.g.

```
with ([1,1] <= iv < [3,8] step [1,3] width [1,2]) : 1
genarray( [3,10], 0)
```

yields the matrix $\begin{pmatrix} 0&0&0&0&0&0&0&0&0&0 \\ 0&1&1&0&1&1&0&1&0&0 \\ 0&1&1&0&1&1&0&1&0&0 \end{pmatrix}$.

Furthermore, a single WITH-loop may consist of multiple generators defining pairwise disjoint index sets, each being associated with a different expression.

3 Multi-operator with-loops

The aim of WITH-loop fusion is to compute multiple values in a single sweep in order to avoid the repeated traversal of identical argument arrays. Hence, a major prerequisite for the fusion of WITH-loops is the ability to represent the computation of multiple values by a single WITH-loop. Regular WITH-loops, as described in the previous section, always compute either a single new array or a single reduction. To overcome this limitation we extend the internal representation of WITH-loops to *multi-operator* WITH-loops:

$$\mathit{MultiOpWL} \quad \Rightarrow \textbf{ with } \lceil \mathit{Generator} : \mathit{Expr} \lceil \ , \ \mathit{Expr} \rfloor^{*} \rfloor^{+} \ \lceil \mathit{Operation} \rfloor^{+}$$

Internal multi-operator WITH-loops differ from language-level WITH-loops essentially in two aspects. First, each generator is associated with a comma-separated list of expressions rather than a single expression. Second, a single WITH-loop may have a sequence of operations rather than exactly one. All generators must be associated with the same number of expressions, and this number must match the number of operations. More precisely, the first operation corresponds to the first expression associated with each generator, the second operation corresponds to each second expression, etc. For example, computing both the minimum and the maximum element of an argument array of any rank and shape can be specified by the following multi-operator WITH-loop:

```
int, int MinMaxVal( int[*] A)
{
  Min, Max = with (0*shape(A) <= iv < shape(A)) : A[iv], A[iv]
             fold( min, MaxInt())
             fold( max, MinInt());
  return( Min, Max);
}
```

In this case, the multi-operator WITH-loop yields two result values, which must be bound to two variables using simultaneous assignment. While this simple example only uses `fold` operations, `fold` and `genarray` operations are generally mixed. We do not feature multi-operator WITH-loops on the language level because they run counter the idea of modular generic specifications. We consider the above representation of `MaxMinVal` the desired outcome of an optimization process, not a suitable implementation.

4 With-loop fusion — the base case

In the following we describe WITH-loop fusion as a high-level code transformation. The optimization base case is characterized by two WITH-loops that have the same sequence of generators and no data dependence, i.e., none of the variables bound to individual result values of the first WITH-loop is referred to within the second WITH-loop. A formalization of WITH-loop fusion for this base case is shown in Fig. 2.

We define a transformation scheme $\mathcal{WLFS}$ that describes the context-free substitution of a SAC intermediate code pattern by some other intermediate SAC code provided that certain guard conditions are met. In the example of Fig. 2 guard conditions for the application of the transformation scheme $\mathcal{WLFS}$ are the absence of data dependences as mentioned before and the additional property that all `genarray` operations create arrays of the same shape. The fact that all generators must be identical as well, is expressed by using the same identifiers in the pattern part of the transformation scheme. We assume that sequences of generator/expressions pairs are sorted in some systematic way to avoid explicit handling of permutations. Furthermore, we have simplified the definition of $\mathcal{WLFS}$ by leaving out step and width specifications in WITH-loop generators.

WITH-loop fusion systematically examines intermediate SAC code to identify pairs of suitable WITH-loops. They do not need to be adjacent in code as is indicated by the dots in between the two WITH-loops in the upper part of Fig. 2. If the required conditions are met, $\mathcal{WLFS}$ takes two assignments with WITH-loops on their right hand sides and concatenates

1. the sequences of assigned identifiers,
2. the sequences of expressions associated with each generator, and
3. the sequences of operations.

Since WITH-loop fusion can be applied repeatedly, we define the transformation scheme $\mathcal{WLFS}$ on multi-operator WITH-loops and simply interpret the initial single-operator WITH-loops as a special case.

In general, index variables introduced by WITH-loops are likely to have different names. In fact, preceding steps in the SAC compilation process transform code into a variant of static single assignment form [8]. Hence, in the intermediate code representation relevant for WITH-loop fusion index variables of different WITH-loops are guaranteed to have different names. In the transformation scheme $\mathcal{WLFS}$ we address this issue by taking the index variable of the first WITH-loop and a systematic α-conversion of all associated expressions that originally stem from the second WITH-loop. In Fig. 2, this is denoted by $[expr]^{iv_a}_{iv_b}$ meaning that all free occurrences of iv_a in $expr$ are replaced by iv_b.

5 Enabling with-loop fusion

The transformation scheme $\mathcal{WLFS}$, as outlined in the previous section, is only applicable in a very restricted setting. In particular, the need for identical gen-

$$
\mathcal{WLFS}\left[\!\!\left[
\begin{array}{l}
Idents^{(a)}\ \texttt{= with} \\
\qquad \texttt{(}\ lb_1\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ ub_1\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(a)}_{1,1}, \ldots, expr^{(a)}_{1,m} \\
\qquad\quad \ldots \\
\qquad \texttt{(}\ lb_k\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ ub_k\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(a)}_{k,1}, \ldots, expr^{(a)}_{k,m} \\
\qquad operation^{(a)}_1 \\
\qquad\quad \ldots \\
\qquad operation^{(a)}_m\ \texttt{;} \\
\ldots \\
Idents^{(b)}\ \texttt{= with} \\
\qquad \texttt{(}\ lb_1\ \texttt{<=}\ iv^{(b)}\ \texttt{<}\ ub_1\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(b)}_{1,1}, \ldots, expr^{(b)}_{1,n} \\
\qquad\quad \ldots \\
\qquad \texttt{(}\ lb_k\ \texttt{<=}\ iv^{(b)}\ \texttt{<}\ ub_k\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(b)}_{k,1}, \ldots, expr^{(b)}_{k,n} \\
\qquad operation^{(b)}_1 \\
\qquad\quad \ldots \\
\qquad operation^{(b)}_n\ \texttt{;}
\end{array}
\right]\!\!\right]
$$

$$
=
\left\{
\begin{array}{l}
Idents^{(a)},\ Idents^{(b)}\ \texttt{= with} \\
\qquad \texttt{(}\ lb_1\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ ub_1\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(a)}_{1,1}, \ldots, expr^{(a)}_{1,m}, \\
\qquad\quad [expr^{(b)}_{1,1}]^{iv^{(a)}}_{iv^{(b)}}, \ldots, [expr^{(b)}_{1,n}]^{iv^{(a)}}_{iv^{(b)}} \\
\qquad\quad \ldots \\
\qquad \texttt{(}\ lb_k\ \texttt{<=}\ iv^{(a)}\ \texttt{<}\ ub_k\ \texttt{)}\ \texttt{:} \\
\qquad\quad expr^{(a)}_{k,1}, \ldots, expr^{(a)}_{k,m}, \\
\qquad\quad [expr^{(b)}_{k,1}]^{iv^{(a)}}_{iv^{(b)}}, \ldots, [expr^{(b)}_{k,n}]^{iv^{(a)}}_{iv^{(b)}} \\
\qquad operation^{(a)}_1 \\
\qquad\quad \ldots \\
\qquad operation^{(a)}_m \\
\qquad operation^{(b)}_1 \\
\qquad\quad \ldots \\
\qquad operation^{(b)}_n\ \texttt{;}
\end{array}
\right.
$$

$$
\left|
\begin{array}{l}
\forall i, j \in \{1 \ldots n\}\ : \\
\qquad operation_i \equiv \texttt{genarray(}shape_i\texttt{)} \\
\qquad \wedge\ operation_j \equiv \texttt{genarray(}shape_j\texttt{)} \\
\qquad\quad \Longrightarrow \quad shape_i = shape_j \\
\\
Idents^{(a)}\ \cap\ \bigcup_{i=1}^{k} \bigcup_{j=1}^{n} \mathcal{FV}\texttt{(}\ expr^{(b)}_{i,j}\ \texttt{)}\ =\ \emptyset
\end{array}
\right.
$$

Fig. 2. Basic WITH-loop fusion scheme.

erator sets is difficult to meet in practice. Hence, many useful optimization cases are not handled appropriately so far. Instead of extending our existing transformation scheme to cover a wider range of settings, we accompany $\mathcal{WLFS}$ by a set of preprocessing code transformations that create application scenarios for $\mathcal{WLFS}$. Due to the limitation of space we here only sketch out some of the preprocessing steps.

- We make the default case in `genarray`-WITH-loops explicit. Rather than having a default rule that implicitly covers all index positions of the new array not covered explicitly by one of the generators, we add additional generator/expression pairs so that eventually each legal index of the array to be created is covered by exactly one generator.
- If the generator set of a `fold`-WITH-loop is a subset of that of a `genarray`-WITH-loop, we introduce the missing generators to the `fold`-WITH-loop with the neutral element of the folding operation as associated expression. Special care is taken in the code generation phase to avoid costly execution of the folding operation at runtime in these cases.
- We unify generator sets of two WITH-loops by systematically computing intersections of each pair of generators from the first and the second WITH-loop. A threshold value sets limits to the number of generators actually created in this way and, hence, avoids explosion of the number of generators in individual WITH-loops.
- We eliminate data dependencies between two WITH-loops which we find useful to fuse by replacing all references to the first WITH-loop found in the second WITH-loop by the corresponding computation.

6 Experimental evaluation

We have conducted several experiments in order to quantify the impact of WITH-loop fusion on the runtime performance of compiled SAC code. Our test system is a 1100MHz Pentium III based PC running SuSE LINUX, and we used gcc 3.3.1 as backend compiler to generate native code from SAC specifications.

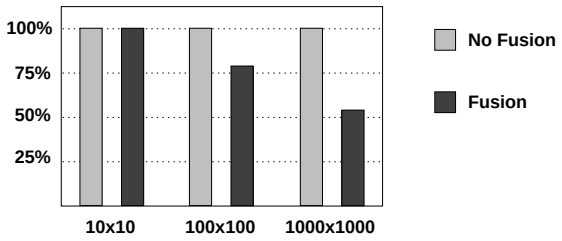


Fig. 3. Impact of WITH-loop fusion on program execution times for computing minimum and maximum element values (`MinMaxVal`) of matrices of varying size.

The first experiment involves our initial motivating example: computing minimum and maximum values of an array. Fig. 3 shows runtimes for three different problem sizes with and without application of WITH-loop fusion. As expected, there is almost no improvement for very small arrays The benefits of fusion in this example are twofold. We do save some loop overhead, but our experiments show this to be marginal. Therefore, the main advantage of fusion in this example is that we can avoid one out of two memory accesses. However, as long as an argument array easily fits into the L1 cache of the processor, the penalty turns out to be negligible. As Fig. 3 shows, this situation changes in steps as the array size exceeds L1 and later L2 cache capacities. In the latter case, WITH-loop fusion reduces program execution time by almost 50% since with frequent slow main memory accesses the application becomes totally memory-bound.

```
double[+] convolution( double[+] A, double eps)
{
  do {
    B = A;
    A = relax( B);
  }
  while (continue( A, B, eps));

  return( A);
}
```

Fig. 4. Convolution with convergence criterion.

Our second benchmark program is a convolution algorithm with convergence test, as sketched out in Fig. 4. Without WITH-loop fusion this implementation of convolution essentially leads to two array operations for each iteration of the outer sequential loop: the convolution step `relax` yielding a new value `A` and the evaluation of the convergence criterion `continue` whose result either terminates or continues the iteration. For computing the convergence criterion

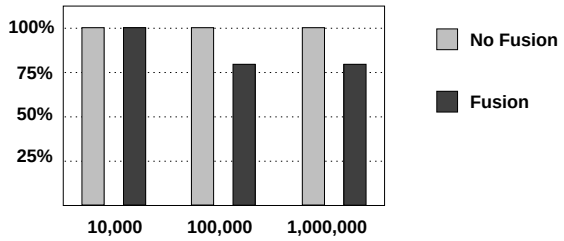


Fig. 5. Impact of WITH-loop fusion on program execution times for computing convolution with convergence test (Fig. 4) on vectors of varying length.

we must reload the corresponding elements of `A` and `B` into registers, although both have been stored in registers already during computation of the convolution step. Applying WITH-loop fusion combines both the convolution step and the convergence criterion in a single step. The experimental data displayed in Fig. 5 backs these considerations. For a small problem size fusion again has no visible impact on performance, but with growing problem size a 25% reduction can be observed for this example.

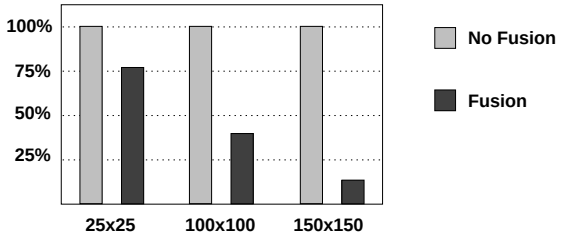


Fig. 6. Impact of WITH-loop fusion on the SPEC benchmark `tomcatv` for varying problem sizes.

The last experiments are based on a SAC implementation of the SPEC floating point benchmark `tomcatv`. As shown in Fig. 6 substantial performance gains can be observed for this benchmark with growing problem size. The background for improvements of up to 80% by WITH-loop fusion for this benchmark is the fact that in contrast to the preceding examples more than two WITH-loops are fused showing the full potential of this optimization.

7 Conclusion and future work

Engineering application programs based on the principles of abstraction and composition, as propagated by SAC, leads to well-structured and easily maintainable software. However, the downside of this approach is that it requires non-trivial compilation techniques which systematically restructure entire application programs into a form that allows for efficient execution on real computing machinery.

In the current work, we have described WITH-loop fusion as one mosaic stone of this code restructuring compiler technology. WITH-loop fusion takes two WITH-loops without a data dependence and transforms them into a single generalized variant named multi-operator WITH-loop, which we have intoduced as a compiler internal intermediate code representation for exactly this purpose. The positive effect of WITH-loop fusion is to avoid repeated traversals of the same array and replace memory load and store operations by equivalent but much faster register accesses.

In several experiments we have demonstrated the potential of WITH-loop fusion to achieve substantial reductions of execution times. WITH-loop fusion

has proved to be one important prerequisite to make the modular programming style of SaC feasible in practice.

In the future, we plan to expand the applicability of WITH-loop fusion to additional cases, e.g. to fusion of `genarray`-WITH-loops that define arrays of non-identical but similar shape. In this situation we could create a joint WITH-loop that operates on the convex hull of the individual WITH-loop's iteration spaces. A special value `none` or `noop` would be used as "initialization" value for index positions that do not exist in one or the other target array. Another subject of future research is the guided selection of WITH-loops that may be fused. Since fusion of two WITH-loops may prevent further fusion with a third WITH-loop, we plan to develop heuristics that guide the sequence of fusion steps based on a cost model.

References

1. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming **13** (2003) 1005–1059
2. Grelck, C.: Implementing the NAS Benchmark MG in SAC. In Prasanna, V.K., Westrom, G., eds.: Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02), Fort Lauderdale, Florida, USA, IEEE Computer Society Press (2002)
3. Grelck, C., Scholz, S.B.: Towards an Efficient Functional Implementation of the NAS Benchmark FT. In Malyshkin, V., ed.: Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia. Volume 2763 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2003) 230–235
4. Grelck, C., Scholz, S.B.: Generic Array Programming in SAC. In Goerigk, W., ed.: 21. Workshop der GI-Fachgruppe 2.1.4 Programmiersprachen und Rechenkonzepte, Bad Honnef, Germany. Volume 0410 of Technischer Bericht., University of Kiel, Institute of Computer Science and Applied Mathematics (2005) 43–53
5. Scholz, S.B.: With-loop-folding in SAC — Condensing Consecutive Array Operations. In Clack, C., Davie, T., Hammond, K., eds.: Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, Selected Papers. Volume 1467 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (1998) 72–92
6. Scholz, S.B.: A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in SAC. In Hammond, K., Davie, T., Clack, C., eds.: Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, Selected Papers. Volume 1595 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (1999) 216–228
7. Grelck, C., Scholz, S.B., Trojahner, K.: With-Loop Scalarization: Merging Nested Array Operations. In Trinder, P., Michaelson, G., eds.: Proceedings of the 15th International Workshop on Implementation of Functional Languages (IFL'03), Edinburgh, Scotland, UK, Revised Selected Papers. Volume 3145 of Lecture Notes in Computer Science., Springer-Verlag, Berlin, Germany (2004)
8. Appel, A.: SSA is Functional Programming. ACM SIGPLAN Notices **33** (1998) 17–20

Open Types and Bidirectional Relationships as an Alternative to Classes and Inheritance

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. The basic idea of open types is to separate the definition of types from the definition of their constituents, i. e., their base types (or superclasses to use object-oriented terminology) and their data members (or fields). This is in complete contrast to traditional record types and object-oriented classes, which are *closed* in the sense that the set of their constituents is fixed once the type has been defined. It will be shown, however, that this alternative approach opens the door to greatly enhanced expressiveness and increased flexibility. Even though the concept of open types is presented in this paper as a language extension for C++, the basic principles are actually language-independent and could be incorporated into any imperative programming language.

1 Open Types

1.1 Type and Attribute Definitions

An *open type* is defined by declaring its name with the keyword `typename`, e. g.:

```
typename Person;
typename Car;
```

Afterwards, a *single-valued attribute* such as `name` − corresponding to a data field in record notion − can be defined by declaring it as a kind of mapping from `Person`s to `string`s:

```
Person -> string name;
```

Here, the right hand side of the definition (`string name;`) looks identical to a C++ (member) variable definition.

Similarly, a *multi-valued attribute* such as `gnames` (given names) − corresponding to a data field whose type is an array or container type − is defined by using a double instead of a single arrow to indicate the multi-valuedness:

```
Person ->> string gnames;
```

1.2 Constructors and Mutators

To create, initialize, and modify objects of an open type `T`, the following *constructors* and *mutators* are provided.

The *parameterless constructor* `T()`, which might either be called explicitly or is called implicitly for variables of type `T` which are not initialized explicitly [8], returns the *null object* of type `T`, i. e., actually *no object*. In contrast, the *attribute-initialization constructor* `T(@attr, val)` creates a distinct *new object* of type `T`, i. e., an object that is different from null and any other object, and initializes its attribute `attr` with value `val`.

Similarly, the *attribute mutator* `obj(@attr, val)` sets the value of attribute `attr` of object `obj` to `val` (if `attr` is a single-valued attribute) or adds `val` to `obj`'s values of attribute `attr` (if `attr` is a multi-valued attribute) and returns the object `obj`. This allows straightforward combinations of a constructor call with one or more mutator calls to create an object with multiple initial attribute values, e. g.:

```
Person p = Person(@name, "Hoare")(@gnames, "Charles")
 (@gnames, "Anthony")(@gnames, "Richard");
```

Here, the constructor call `Person(@name, "Hoare")` creates a new `Person` object, initializes its attribute `name` with the string `"Hoare"`, and returns the object. This object is directly used in the mutator call `...(@gnames, "Charles")` which initializes its attribute `gnames` with `"Charles"` and returns the same object. This is again used in and returned by the subsequent mutator calls `...(@gnames, "Anthony")` and `...(@gnames, "Richard")` which in turn add the strings `"Anthony"` and `"Richard"` to the values of attribute `gnames`. Finally, the object returned by the last mutator call is assigned to the `Person` variable `p`.

To create an *empty object*, i. e., a distinct object which is different from null and any other object, but does not possess any attribute values yet, the *Boolean constructor* `T(flag)` can be used. If the Boolean value `flag` is `true`, a unique empty object is created, while otherwise a null object is returned, i. e., `T(false)` is equivalent to just `T()`. On the other hand, a call `T(@attr, val)` to the attribute-initialization constructor is actually just a shorthand for `T(true)(@attr, val)`, i. e., a call to the Boolean constructor followed by an appropriate mutator call.

In addition to these predefined constructors of open types, it is possible to define arbitrary *user-defined constructors*, e. g.:

```
// Create person with given name g and name n.
Person (string g, string n) {
  return Person(@name, n)(@gnames, g);
}
```

In contrast to normal C++ constructors (and constructors in other object-oriented programming languages) which must be defined (or at least declared) inside their class and must not explicitly return anything, but rather initialize the implicitly available object `this`, user-defined constructors of open types are much like ordinary (global) functions whose result type and name coincide (and therefore only one of them is specified in their definition). In particular, there is no implicitly available object `this`, and the constructor must explicitly (create and) return an object, typically by calling one of the predefined constructors. Furthermore, just like attributes, constructors can be defined *successively* on demand.

1.3 The Attribute Inspection Operator `@`

To inspect the attribute values of a given object, the *attribute inspection operator* `@` can be used, quite similar to the way the dot operator is used to access class members in C++ and other languages, e. g.:

```
string n = p@name;
```

For a single-valued attribute such as `name`, its current value is returned, i. e., the value that has been set for this attribute by the most recent mutator (or attribute-initialization constructor) call for this object. If none of these operations has been executed for the object yet, i. e., the attribute does not possess any value, a well-defined *default value* is returned that is obtained by calling the parameterless *default constructor* of the attribute's type (i. e., `string` in the current example). Ideally, this constructor should return *null* to indicate the absence of any real value [2], but in principle any value (e. g., an empty string or zero for numeric types) is acceptable.

If a multi-valued attribute such as `gnames` is inspected with the `@` operator, the values added to this attribute by all mutator (and attribute-initialization constructor) calls performed for this object so far are returned as an *ordered sequence*. Even though it is possible to grasp such a sequence as a whole, it is typically processed element by element using a tailored iteration statement, e. g.:

```
for (string g : p@gnames) cout << g << " ";
```

This prints `p`'s given names in the order in which they have been added, i. e., `Charles Anthony Richard`. Alternatively, it is possible to directly inspect individual values of such a sequence by applying the well-known index operator, e. g.:

```
string g2 = p@gnames[2];
```

to obtain the second given name of `p`, i. e., `"Anthony"`. Similarly to inspecting the value of a non-existent single-valued attribute, inspecting a non-existent value of a multi-valued attribute by using an out-of-range index yields a well-defined default value that is obtained in the same way as described above. Therefore, expressions such as `p@gnames[0]` or `p@gnames[4]` will return a null string in the current example.

It should be noted that the attribute inspection operator always returns an *R-value* [8], i. e., a value which must not occur on the left hand side of an assignment operator. Therefore, attribute update operations must only be performed by mutator calls, not directly by assignments such as:

```
p@name = "Hoare"; // Syntax error!
```

1.4 Inspecting and Modifying Null Objects

Trying to inspect or modify a member of "object null," i. e., the "object" referenced by a null pointer, is illegal in C++ and many other languages and usually leads to a run

time error such as a `SIGSEGV` (segmentation violation) signal or a `NullPointerException`, since a null pointer actually does not refer to any object.

In contrast to that, inspecting and modifying attributes of open types is well-defined even for null objects: While inspecting such an attribute is equivalent to inspecting a non-existent attribute, i. e., returns the attribute's default value, modifying such an attribute simply has no effect. The main reason for these unusual definitions is convenience, since they allow to omit many otherwise necessary checks. To test, for example, whether `p`'s name is `"Hoare"`, one can simply write `if (p@name == "Hoare")` − instead of `if (p && p@name == "Hoare")` − even if `p` might be null; if it actually is, `p@name` is null, too, and therefore, as expected, different from `"Hoare"`. Simultaneously, programs tend to become more robust since inadvertently omitted checks will not lead to run time errors, but usually merely to unsatisfied conditions.

Similarly, the definition that mutator calls on null objects are silently ignored frequently reduces the need to explicitly distinguish between real and null objects, and again, inadvertently omitting such distinctions does not lead to run time errors (cf. [2]).

1.5 Object Deletion

In contrast to normal C++ objects, which must be explicitly deleted by the programmer to reclaim their storage, objects of open types are automatically *garbage-collected* when they have become unreachable, quite similar to objects of classes in Java, Eiffel, Smalltalk, and many other programming languages.

In addition to and independently from this automatic storage reclamation, it is also possible to explicitly *delete* objects − even while they are still referenced. Although this might appear strange at first sight, there are reasonable practical examples where this is useful. If, for instance, a car has been scrapped, it does not exist anymore, even though it might still appear in the list of all cars of its (previous) owner.

In contrast to C++, however, where the deletion of an object might lead to dangerous dangling pointers, deletion of an open type object causes all remaining references to the object to become null immediately and automatically. By that means, it is always possible to reliably detect that an object has been deleted. Furthermore, since null objects can be safely inspected and modified, too, neither run time errors nor undefined behaviour will occur if deleted objects are used without care. Since object deletions might be performed unexpectedly, this is another strong argument for the definitions given in Sec. 1.4.

2 Bidirectional Relationships

Basically, a *bidirectional relationship* between two types is also a kind of mapping from one type to the other, with the additional possibility to directly access the *inverse mapping*. Since both of these mappings might be either single- or multiple-valued, there are four different kinds of relationships altogether, one to one, one to many, many to one, and many to many, expressed by corresponding bidirectional arrow symbols `<->`, `<->>`, `<<->`, and `<<->>`, respectively. Furthermore, there are two special

kinds, i. e., *symmetric* one-to-one and many-to-many relationships, where the inverse mapping is equivalent to the original mapping.

For example, a one-to-many relationship between `Person` and `Car` called `cars` resp. `owner` can be defined as follows:

```
Person owner <–>> Car cars;
```

Reading this from left to right (and omitting the name on the LHS) yields a multi-valued attribute of type `Person`:

```
Person –>> Car cars;
```

while reading from right to left (and omitting the name on the RHS) yields a single-valued attribute of type `Car`:

```
Car –> Person owner;
```

representing the inverse mapping. However, only by combining both attribute definitions into a single relationship definition as shown above, they are actually treated as mutually inverse mappings, which means that a call to one of the mutators automatically implies a corresponding call to the other mutator with reversed roles.

For example, a mutator call such as `p(@cars, c)` adding `c` to `p`'s sequence of cars, implies the call `c(@owner, p)` assigning `p` as `c`'s owner, and vice versa. Furthermore, if `c` already possesses another owner `q` when either such call is made, `c` is first removed from the sequence of `q`'s cars.

3 Anonymous and Automatic Attributes and Relationships

3.1 Basic Principles

If the name of a single-valued attribute is omitted, it implicitly possesses the name of its target type, e. g.:

```
typename Address;
Person –> Address;

Person p = Person(@Adress, Address(...));
Address a = p@Adress;
```

Similarly, it is possible to omit one or both names of a bidirectional relationship.

If the arrow in an attribute declaration is followed by an exclamation mark, the attribute might be applied *automatically* on demand to perform an *implicit type conversion* from its source type (left of the arrow) to its target type (right of the arrow), e. g.:

```
Person –>! int pid;
```

This declares an `int` attribute `pid` of type `Person` which can be used just like any other attribute, with the additional property that an expression of type `Person` is implicitly convertible to an `int` value by automatically applying this attribute.

Similarly, it is possible to declare automatic relationships by adding an exclamation mark before or after the bidirectional arrow, depending on which direction of the relationship should be automatically applicable.

3.2 Modeling Type Hierarchies

Automatic one-to-one relationships can be exploited to model object-oriented type hierarchies with subtype polymorphism without requiring any additional mechanisms. For example, a new type `Student` (with a regular attribute `number` denoting the matriculation number) might be defined as a "subtype" of `Person` by declaring a one-to-one relationship between these types that is automatically applicable from the derived type to the base type:

```
// Declare Student as a "subtype" of Person.
typename Student;
Student -> string number;
Student <->! Person;
```

Typically, but not necessarily, such "subtype" relationships are anonymous.

A typical constructor for `Student` might be defined as follows:

```
// Create student with given name g, name n,
// and matriculation number m.
Student (string g, string n, string m) {
  // Create person subobject.
  Person p = Person(@name, n)(@gnames, g);

  // Create and return student object connected with p.
  return Student(@Person, p)(@number, m);
}
```

Now, a student named Peter Clark with matriculation number 777 can be created and used as follows:

```
// Create student.
Student s = Student("Peter", "Clark", 777);

// Print name and matriculation number.
cout << "Name: " << s@name << endl;
cout << "Number: " << s@number << endl;
```

Because the relationship between `Student` and `Person` is applied automatically on demand, the subexpression `s@name` is actually replaced by `s@Person@name`. Furthermore, all functions accepting `Person` arguments can be called with `Student` objects, too, and finally, a `Student` object can be used polymorphically as a `Person` object.

The fact that the relationship between `Student` and `Person` is bidirectional can be exploited to check whether a given `Person` object "is" actually a student (i. e., to perform a *dynamic type test*) and to access its student attributes if appropriate (i. e., to perform a *downcast*):

```
// Polymorphically use a student as a person.
Person p = Student("Peter", "Clark", 777);

// Check whether p is actually a student s ...
if (Student s = p@Student) {
  // ... and access its matriculation number.
  cout << "Number: " << s@number << endl;
}
```

This corresponds roughly to a `dynamic_cast` in C++ which returns a valid pointer to an object of a derived class if the cast has been successful and a null pointer otherwise.

By employing automatic relationships to model object-oriented type hierarchies, the traditionally distinct or even conflicting concepts of *aggregation* and *inheritance* have been merged into a single coherent concept. Furthermore, the fact that relationships can be defined incrementally, allows "supertypes" of a type to be declared later on, e. g.:

```
// Declare Vehicle as a "supertype" of Car.
typename Vehicle;
Car <–>! Vehicle;
```

Despite its practical usefulness, such a possibility is missing in most object-oriented programming languages.

3.3 Multiple Inheritance

Of course, it is possible to use automatic relationships to model type hierarchies with multiple inheritance, too. For example, one might define a type `EmployedStudent` that is derived from both `Student` and another type `Employee`:

```
// Declare Employee as a subtype of Person.
typename Employee;
Employee <–>! Person;

// Attributes and constructors of Employee.
Employee –> string company;
Employee (......) { ...... }

// Declare EmployedStudent as a
// subtype of Student and Employee.
typename EmployedStudent;
EmployedStudent <–>! Student;
EmployedStudent <–>! Employee;
```

Since both of these types in turn "inherit" from `Person`, the typical question arises whether an `EmployedStudent` object should possess one or two `Person` subobjects, i. e., whether `Person` is, in C++ terminology, a *virtual* base type or not. In C++, the corresponding decision must be taken when the types `Student` and `Employee` are

defined, even though it does not make any difference for *these* types. Therefore, it would be much more logical to answer the question when `EmployedStudent` is defined, because only for this type (and possible subtypes of it) the distinction is relevant. However, the concept of automatic relationships does not provide a way to specify the difference at the level of *declarations*: The four `<->!` relationships between the types `Person`, `Student`, `Employee`, and `EmployedStudent` merely specify that there are two ways to convert an `EmployedStudent` to a `Person`, either via `Student` or via `Employee`, but they do not specify whether these ways lead to the same destination, i. e., to the same `Person` object, or not. Even though this appears to be disadvantageous at first sight, it will turn out to be the most flexible approach possible.

To actually distinguish between virtual and non-virtual inheritance, one simply creates either one or two `Person` "subobjects" when creating an `EmployedStudent` object in a constructor, e. g.:

```
// Create employed student with given name g, name n,
// matriculation number m, and company c.
EmployedStudent (string g, string n, string m, string c) {
  Person p = Person(@name, n)(@gnames, g);
  Student s = Student(@Person, p)(@number, m);
  Employee e = Employee(@Person, p)(@company, c);
  return EmployedStudent(@Student, s)(@Employee, e);
}
```

Here, a *single* `Person` object `p` is created that is passed to both the `Student` and `Employee` constructors to create `Student` and `Employee` objects `s` and `e`, respectively, which *share* the subobject `p`. Afterwards, an `EmployedStudent` object with subobjects `s` and `e` is created and returned. Therefore, converting an `EmployedStudent` object created by this constructor to type `Person` always yields the same `Person` subobject, no matter whether the conversion is done via `Student` or via `Employee`.

3.4 Dynamic Object Evolution

The fact that an object of a derived type such as `Student` or `EmployedStudent` is actually a network of interconnected subobjects − even though this remains invisible except when constructing the objects −, can be exploited in a straightforward manner to implement *dynamic object evolution*. For example, it is almost trivial to transform an object that has been initially created as a bare person into a student, an employee, or even an employed student later, by simply creating additional associated subobjects, e. g.:

```
// Create a person object p.
Person p = Person("Peter", "Clark");

// "Transform" p to a student.
p(@Student, Student(@number, 777));
```

Conversely, it is also possible to delete subobjects to transform a specialized object to

a more general one, e. g.:

```
// "Transform" p back to a person.
delete p@Student;
```

Here, it does not matter whether `p` has been originally created as a `Person` or a `Student` (or something else).

By explicitly deleting the student subobject associated with `p`, all "student references" to this person automatically become null. Otherwise, if only the association between `p` and its student object would have been cut, these references would remain valid, but refer to a degenerate student object that does not possess an associated person object anymore. (According to the rules given in Sec. 1.4, accesses to this person object and its attributes would still be well-defined, however.)

It is even possible to create "hybrid" objects, such as a person that is both a student and an employee, even if no common "subtype" of these types (such as `EmployedStudent`) would exist.

4 Related Work

Aspect-oriented programming languages such as AspectJ [4] or AspectC++ [7] provide so-called *inter-type member declarations* or *introductions* to retroactively extend existing data structure definitions without needing to change the code of the original definitions. Nevertheless, some kind of recompilation or "weaving" is required by all these approaches: While the AspectC++ compiler needs the source code of the original definition together with all extension code to produce a new definition that is actually compiled by a C++ compiler, the AspectJ compiler is able to perform the combination on the byte code level. Frameworks such as JMangler [5] are even able to delay the final composition until load time, but in either case a class remains fixed once it has been loaded. Open types, on the other hand, allow attributes to be loaded dynamically, even for types which have already been instantiated.

Actually, aspect-oriented approaches still adhere to the traditional concept of records as fixed data structures and only make their definition more flexible, while open types support truly flexible objects whose storage size might vary over time.

The *Common Lisp Object System* (CLOS) [3] (and other languages based on similar ideas) deviate from the typical object-oriented approach that everything belonging to a class must be defined (or at least declared) in the class, by allowing methods (of so-called *generic functions*) to be defined separately and incrementally. However, the set of data fields making up a class must still be defined at once and cannot be extended later, except by redefining the whole class. In contrast, open types apply the "generic function principle," i. e., the possibility to define methods separately and independently, to data fields, too.

Description logic systems such as Classic [1] and Loom [6] provide data models which are very similar in nature to open types and have in fact influenced some of their ideas. They provide *concepts* (corresponding to open types) and *roles* (corre-

sponding to attributes and relationships), which are defined separately and independently, and roles might possess *inverse roles*. Furthermore, a running system can be extended by new definitions at any time.

However, since description logic systems are actually AI tools, providing powerful reasoning capabilities such as subsumption checking, automatic instance classification, and truth maintenance, using them as bare data models of a programming language would mean to break a fly upon the wheel. Therefore, open types might be viewed as the result of reducing a description logic system to a simple data representation system by stripping off all AI functionality.

References

[1] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, L. A. Resnick: "Living with CLASSIC: When and How to Use a KL-ONE-Like Language." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 401−456.

[2] C. Heinlein: "Null Values in Programming Languages." In: H. R. Arabnia (ed.): *Proc. Int. Conf. on Programming Languages and Compilers (PLC'05)* (Las Vegas, NV, June 2005), 123−129.

[3] S. E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*. Addison-Wesley, Reading, MA, 1989.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 − Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327−353.

[5] G. Kniesel, P. Costanza, M. Austermann: "JMangler − A Powerful Back-End for Aspect-Oriented Programming." In: R. E. Filman, T. Elrad, S. Clarke, M. Aksit (eds.): *Aspect-Oriented Software Development*. Pearson International, 2004, 311−342.

[6] R. MacGregor: "The Evolving Technology of Classification-Based Knowledge Representation Systems." In: J. F. Sowa (ed.): *Principles of Semantic Networks. Explorations in the Representation of Knowledge*. Morgan Kaufmann Publishers, San Mateo, CA, 1991, 385−400.

[7] O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53−60.

[8] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

A Pattern Logic for Lazy Assertions in Haskell

Olaf Chitil
University of Kent, UK
oc@kent.ac.uk

Frank Huch
CAU Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract. Assertions test expected properties of run-time values without disrupting the normal computation of a program. Here we present a library for enriching programs in the lazy language Haskell with assertions. Expected properties are written in an expressive *pattern logic* that combines pattern matching with logical operations and predicates. The presented assertions are lazy: they do not force evaluation but only examine what is evaluated by other parts of the program. They are also prompt: assertion failure is reported as early as possible, before a faulty value is used by the main computation. The implementation is based on lazy observations and continuation-based coroutines.

1 Introduction

Large programs are composed of algorithms and numerous (more or less) abstract data types which interact in complex ways. A bug in the implementation of a basic data structure can result in the whole program going wrong. Such a bug can be hard to locate, because the faulty data structure may not be part of the wrong result, it may just be an intermediate data structure. Even worse, the program may produce wrong results for a long time before the user even notices.

Testing abstract data types exhaustively is difficult. However, interesting test cases often occur when data structures are used within other algorithms. Hence it is a good idea to check for bugs in basic data structures and functions during the execution of larger programs. Using *assertions* is a common approach to do so (e.g., see [3]). The programmer specifies *intended properties* of data structures and functions by writing assertions. During program execution, these assertions are tested and failure of an assertion is reported to the programmer. Examples of assertions are restricting the square root function to positive arguments or the property of being sorted for a search tree.

Recently, in functional programming languages assertions became famous in form of assertion-based contracts for Scheme [2]. Also the Glasgow Haskell Compiler (ghc) provides the possibility to define assertions:

```
assert :: Bool -> a -> a
```

The first argument is the asserted property. If this property evaluates to True, then `assert` behaves like the identity function. Otherwise, an error is reported with detailed information about the source code position of the failed assertion. For example, consider an assertion that checks whether a list is sorted:

```
checkSorted :: Ord a => [a] -> [a]
checkSorted xs = assert (sorted xs) xs

sorted :: Ord a => [a] -> Bool
sorted (x:(y:ys)) = x<=y && sorted (y:ys)
sorted _          = True
```

Unfortunately `assert` is strict in its boolean argument which clashes with Haskell's laziness. The asserted property is evaluated and the tested data structure is evaluated as

far as necessary to decide the property. Hence, programming with assertions will result in strict programs with loss of the expressive power of laziness, e.g., the use of infinite data structures.

We conclude that assertions in lazy languages should respect laziness. They should only be evaluated as far as possible, i.e., the assertion should only be checked for that part of the data structure which is evaluated during the computation. A first approach for lazy assertions is [1]. There, an assertion is introduced by

```
assert :: String -> (a -> Bool) -> a -> a
```

The first parameter is a label which names the assertion. When an assertion fails the computation aborts with an appropriate message that includes the assertion's label. As further parameters, `assert` takes the property and the value on which it behaves as a partial identity.

To prevent an assertion from evaluating too much, the property has to be defined as predicate on the tested data structure. The context in which the application of `assert` appears determines how far the tested data structure is evaluated, and only the evaluated part is passed as argument to the predicate.

We can redefine `checkSorted` as follows: `checkSorted xs = assert "sorted" sorted xs`

Applying `checkSorted` to the list `[1,3,2,4]` yields:

```
Assertion (sorted) failed: 1:3:2:_
```

The failure is reported as early as possible, before the whole list is evaluated. However, the approach has a major drawback. If we evaluate only the tail of the observed list, no failure occurs:

```
> tail (checkSorted [1,3,2,4])
[3,2,4]
```

The reason for this behaviour is that `(&&)` is sequentially defined. The assertion is suspended on checking the sorted property for the first two elements of the list. The conjunction is never evaluated to `False`, although there are two elements in the evaluated part, which are not in order.

In practice, many lazy assertions are suspended exactly for this reason. Many asserted properties may not hold for evaluated parts of data structures, but the assertions do not fail and hence, the programmer wrongly believes their program to be correct. Lazy evaluation does involve a *sequential* evaluation order.

In this paper we introduce a new approach for lazy assertions. The basic idea is to define assertions by means of a pattern logic instead of arbitrary Haskell functions. In this logic, we express properties with parallel versions of `(&&)` and `(||)`. If any of the arguments of such a parallel operator makes the whole assertion fail, then this is reported independently of the other parts of the assertion.

Although this approach may in some cases be more complicated than defining assertions within the programming language Haskell itself, there is also an opportunity. Our pattern logic is more a specification language than a programming language. Hence, properties are asserted in a completely different style to ordinary programs. So it is unlikely that programmers will make the same mistakes in the assertions as in the program, which may happen easily using the same language for programming as for specifying properties.

Beside reporting failed assertions, reporting how many and which assertions have succeeded may also be useful. We collect succeeded assertions in a file, so that the programmer can later analyse which assertions succeeded. However, not every assertion is supposed to

succeed in the presence of laziness. The user must be aware, that in many cases checking assertions suspends and cannot be decided on the evaluated parts of the data structures. This behaviour is even required when a property shall be tested on a never fully evaluated infinite data structure. However, if an assertion fails because of any part of an evaluated data structure, then this is reported immediately.

Our assertions have the following properties:

- they do not modify the lazy behaviour of a program
- whenever some part of a data structure is evaluated and this part violates an asserted property, this is promptly reported to the programmer
- assertions are implemented as a library and do not need any compiler or run-time modifications
- the only extension to Haskell 98 used for the implementation are `unsafePerformIO` and `IORef`s.

2 Programming with Assertions

In the following subsections we introduce our pattern logic step by step and justify our design decisions through several examples.

2.1 Patterns

Pattern matching is a powerful feature of modern functional languages. The pattern is a kind of prototype of a function's argument. For example, it allows a simple definition of a function that tests whether a list has exactly two elements:

```
hasTwoElements :: [Int] -> Bool
hasTwoElements (_:_:[]) = True
hasTwoElements _        = False
```

We can define a function that is basically the identity function on lists but additionally asserts that the argument has exactly two elements as follows:

```
twoElements :: [Int] -> [Int]
twoElements = assert "two elements" (p_ <:> p_ <:> pNil)
```

So what are the new functions used in this definition? We cannot use built-in pattern matching for prompt lazy assertions and we do not want to extend the language Haskell. Therefore, we implement our pattern logic using an abstract type constructor `Pat`. We provide functions for constructing `Pat`s:

> `p_:: Pat a` is the wildcard pattern that matches anything;
> `pNil :: Pat [a]` and `(<:>) :: Pat a -> Pat [a] -> Pat [a]` construct patterns that match the two data constructors of the list type. Using these pattern constructors we can write `p_ <:> p_ <:> pNil` to express a `Pat` similar to the pattern `_:_:[]` used in the definition of `hasTwoElements`.

Whereas `hasTwoElements` forces the evaluation of the list constructors of its argument to perform pattern matching, `twoElements` is lazy: the argument is only evaluated as far as its result is demanded by the caller of `twoElements`.

In many cases it will be useful to combine patterns by means of the logical conjunction and disjunction operators:

```
(|||) :: Pat a -> Pat a -> Pat a          (&&&) :: Pat a -> Pat a -> Pat a
```

For instance, we can now define an assertion which expresses that a list contains less than two elements:

```
shortList :: [a] -> [a]
shortList = assert "length less than two" (pNil ||| p_ <:> pNil)
```

2.2 Context Patterns

When specifying properties of large data structures, it is not sufficient to match a finite initial part of the data structure. We would like to be able to match patterns in arbitrarily deep contexts, for example, to select an arbitrary element of a list. Hence we provide *context patterns* within our pattern logic. The pattern constructor `pListC :: Pat [a] -> Pat [a]` matches an arbitrary sublist of a list. For example

```
oneTrue :: [Bool] -> [Bool]
oneTrue = assert "True in list" (pListC (pTrue <:> p_))
```

asserts that there exists an element `True` in the argument list.

2.3 Universal and Existential Quantification

Why does the preceding example assert that there *exists* an element `True`? Could it not mean that *all* elements should be `True`? Indeed we will sometimes want to assert a property for all sublists and sometimes want to assert that there exists a sublist with a given property. Hence we introduce the quantifier patterns: `forAll :: Pat a -> Pat a` and `exists :: Pat a -> Pat a` which change the meaning of context patterns within their scope. So `exists (pListC (pTrue <:> p_))` asserts that there exists an element `True` whereas `forAll (pListC ((exists pTrue) <:> p_))` asserts that all list elements are `True`.

Why is there a nested `exists` in the last example? Because quantifiers do not only change the semantics of context patterns, but also of normal patterns. Within the scope of `forAll` a constructor pattern such as `pTrue` matches any other constructor. Because of the quantifier `forAll` the context pattern `pListC` has to match *all* sublists with its argument pattern. In any finite list one sublist will be `[]`. We could list this alternative in our definition:

```
forAll (pListC (pTrue <:> p_ ||| pNil))
```

This is acceptable for lists, but not for more complex types with more constructors, such as abstract syntax trees. We would have to add a disjunction for every constructor and the size of assertions would blow-up unacceptably. Therefore we decided that within the scope of `forAll` a pattern built from `(<:>)` also matches the empty list. In contrast, in an existential context the pattern describes which structure is supposed to exist. Hence, non-matching sub-data-structures should not match the pattern inside `exists`. So within the scope of `forAll` the pattern `(exists pTrue) <:> p_` matches both the empty list and a non-empty list that does start with `True`.

The function `assert` implicitly surrounds its pattern by `exists`. Hence in the preceding subsection the pattern context is existentially quantified.

2.4 Unary Predicates

Pattern matching cannot express properties of primitive types, such as a number being positive or a number being greater than another. For expressing such properties, Haskell enriches standard pattern matching with guards, in which the programmer specifies restrictions for the bound variables.

Because we cannot define a new variable binding construct within Haskell, we cannot bind normal variables in our patterns. Instead, we introduce a new pattern `val` that represents binding a value to a variable. To check a property of such a "variable" we provide a function `check`.

For example, we define an assertion checking whether a number is positive:

```
posInt :: Int -> Int
posInt = assert "positive" (check val (>0))
```

We can define a more complex assertion that checks whether all elements of a list are positive:

```
allPos :: [Int] -> [Int]
allPos = assert "all positive" (forAll (pListC ((check val (>0)) <:> p_)))
```

2.5 Predicates with several Arguments

Unary predicates are not very expressive. For instance, it is not possible to compare two elements of a data structure, as is necessary to express the property of being sorted. Hence we extend the function `check` so that values from different `val`s can be compared in a predicate:

```
sortedList :: [Int] -> [Int]
sortedList = assert "sorted" (forAll (check (pListC (val <:> (pListC (val <:> p_))))
                                            (<=)))
```

We select two elements within a list (respecting their positions in the list) by means of two list contexts, and check whether these two elements are in order. The assertion is checked for every possible combination of elements in the list. Evaluating `sortedList [2,4,6,3,5]`, the following failure is reported:

```
Assertion (sorted) failed: 2: 4 :6: 3 :_
```

The result of the application is the list itself. For printing this list, the list has to be evaluated from left to right. When the list element 3 is evaluated the assertion fails. The list elements which cause the assertion to fail are highlighted. Because the remaining list is not evaluated at all, an underscore is presented to the user for the unevaluated tail of the list. With a different evaluation order of the values within the list other failure positions may be reported. However, our assertions are prompt. When an assertion fails during the evaluation of a data structure, this is reported to the user. The data structure is not evaluated any further.

Checking `sortedList` is expensive in time ($\mathcal{O}(n^2)$, where n is the length of the list). Using the transitivity of `(<=)`, we can define a linear version instead:

```
sortedLin :: [Int] -> [Int]
sortedLin = assert "sortedLin" (forAll (check (pListC (val <:> val <:> p_)) (<=)))
```

However, assertions should be seen as high-level specifications for which it is more important to be understandable and correct than to be efficient. Furthermore, this more efficient implementation has another drawback. If only every second element of the list is evaluated, then `sortedLin` cannot be checked, i.e., for a list which is only evaluated to `1:_:2:_:1:_` `sortedLin` does not fail, whereas the less efficient assertion `sortedList` would fail. On the other hand, in practice evaluation orders like this one are uncommon and failure of `sortedLin` will in most cases be detected as early as failure of `sorted`.

2.6 The Pattern Type

When introducing predicates with more than one argument, we have to extend he definition of patterns (`Pat`) as well. Applying `check` to a pattern and a predicate function, we have to guarantee that the predicate takes as many arguments as `val`s occur in the pattern. Furthermore, the type of each value matched by `val` and the corresponding argument of the predicate must agree. In other words, `check` should have a type like

```
check :: Pat a (b1,...,bn) -> (b1->...->bn->Bool) -> Pat a ()
```

where $b_1,\ldots,b_n$ are the types of the values matched by `val`s. How can such a type be expressed within Haskell 98? We want `check` to work with predicates of any arity. Even a set of `check` functions indexed by arity would not do as a first take at the type of a simple constructor pattern demonstrates:

```
(<:>) :: Pat a (b1,...,bn) -> Pat [a] (bn+1,...,bm) -> Pat [a] (b1,...,bm)
```

How shall we handle all these varying numbers of arguments collected by `val` for the predicate tested by `check`? The solution is to extend the type `Pat` not by one but by two type arguments. The first is the type of a predicate passed as input to the pattern and the second it the type of a predicate resulting from the pattern. We revise the types as follows:

```
check :: Pat a (b1->...->bn->Bool) Bool -> (b1->...->bn->Bool) -> Pat a Bool Bool
```

```
(<:>) :: Pat a (b1->...->bm->Bool) (bn+1->...->bm->Bool) ->
         Pat [a] (bn+1->...->bm->Bool) Bool
         -> Pat [a] (b1->...->bm->Bool) Bool
```

These are still not Haskell 98 types, but they are instances of types that we can use:

```
check :: Pat a b Bool -> b -> Pat a c c
(<:>) :: Pat a b c -> Pat [a] c d -> Pat [a] b d
```

So the second type argument of `Pat` is the type of a value passed into the pattern and the third type argument is the type of a value passed back out of the pattern, if the pattern matches. We always use patterns for which these passed values are predicates or simply boolean values.

The type of `check` expresses that the predicate of type `b` has to be applied to all its arguments in the pattern to return a boolean value. The variable bindings within `check` are encapsulated. Also, while `check` tests the predicate for its argument pattern, it also accepts a predicate as input which it passes unchanged back, if the pattern matches.

We have the following type for the variable pattern:

```
val :: Pat a (a -> b) b
```

This type expresses that the input function is applied to the matched value and the result passed back.

To make our assertions lazy, `val` can only be performed if the selected data structure is fully evaluated. Otherwise, the predicate would be tested on partially evaluated values, which could involve further evaluation destroying the laziness of our assertions. The pattern `val` is usually used for values of primitive types, which cannot be evaluated partially at all.

2.7 Example: Equal Sets

Let's define the property that two sets (implemented as unordered lists without repeated items) contain the same elements. A simple way to describe this property would be the following:

> For each element of the first list, there exists an equal element in the second list and for each element of the second list, there exists an equal element in the first list.

Using our quantifiers, the first of these two assertions can easily be defined as follows:

```
equalSets :: ([Int],[Int]) -> ([Int],[Int])
equalSets = assert "already subset" (check (pTuple (forAll (pListC (val <:> p_)))
                                                   (exists (pListC (val <:> p_)))) (==))
```

The quantifiers are nested with respect to the order in which they appear within the linearly written formula. Hence, for every element of the first list an equal element within the second list has to exist. Expressing the other direction is more difficult, because the nesting of quantifiers ($\forall\exists$) has to be applied in the reverse order of the tuple elements. We need to first select any element of the second list and then check whether there exists the same element within the first list. This can be expressed by matching the same data structure twice, by means of a modified conjunction operator

```
(+++) :: Pat a b c -> Pat a c d -> Pat a b d
```

which applies both argument patterns to the same data structure and collects all `val`s within the two argument patterns (all combinations — like a product) to apply a predicate to these by means of `check`. Using this operator, we can define the complete assertion as:

```
equalSets :: ([Int],[Int]) -> ([Int],[Int])
equalSets = assert "equal sets" (check (pTuple (forAll (pListC (val <:> p_)))
                                               (exists (pListC (val <:> p_))))
                                   &&& (pTuple p_ (forAll (pListC (val <:> p_)))
                                    +++ pTuple (exists
                                    (pListC (val <:> p_))) p_))
                                (==))
```

Evaluation of `equalSets ([1,2,3],[3,2,2,1])` just yields the tuple of sets, whereas the call `equalSets ([1,2,3],[3,2,4,2,1])` aborts with the message:

```
Assertion (equal sets) failed: (1:(2:(3: [] )),3:(2:( 4 :_)))
```

For the element 4 of the second list, no element was found in the first list. In the presence of existential properties it is not so easy to show the programmer where an assertion failed. The first list does not contain the element 4. Marking the first list completely would present the reason for the failure of the existentially quantified part. However, this would often mean that the whole data structure is marked. Hence, we decided to mark only that part of the data structure, at which the failure of the existential pattern is observed. To distinguish these sub-terms from those causing failure of a universally quantified `val` we use a lighter colour for marking. In this application, the lists were evaluated from left to right. As a consequence, the empty list made the decision that the assertion fails possible and we mark it. If the elements of the list were evaluated in another order, another element might be highlighted.

2.8 Functions

So far, our approach allows programmers to annotate arbitrary data structures with assertions. However, where should a programmer add such assertions? To express pre- and post-conditions, it would be nice to add assertions directly to functions. Furthermore, in a higher order language, it should be possible to add assertions to functional arguments, functional return values, and functions within data structures as well.

In our pattern logic, we handle functions just like any other data structure. The idea is that a function can be seen as a set of argument/result pairs, which are matched by the function pattern

```
(-->) :: Pat a c d -> Pat b d e -> Pat (a -> b) c e
```

The first argument of `(-->)` is matched against the argument the function is applied to. The second argument is matched against the function result. An assertion for functions will usually contain predicates relating arguments and results. Hence, its type is similar to any pattern constructor of arity two.

As `b` can again be a functional type, patterns for functions with higher arity can be defined by nested `(-->)` applications. As an example we consider the greatest common divisor (*gcd*) of two numbers. A reasonable assertion for gcd is that the result is a factor of both arguments:

```
gcd :: Int -> Int -> Int
gcd = assert "result is factor of arguments"
  (forAll (check (val-->val-->val) (\x y res -> mod x res==0 && mod y res==0))) gcd'

gcd' :: Int -> Int -> Int
gcd' n m = let r = n `mod` m in if r == 0 then m else gcd n r
```

The algorithm is implemented by the function `gcd'`. For the assertion, we add a wrapper `gcd` which checks every application of `gcd'`. The function works correctly for many arguments, but we finally get a report like:

```
Assertion (result is factor of arguments) failed:  6  ->  9  ->  6
```

The function `gcd` applied to the arguments 6 and 9 yields 6, which is wrong, because 6 is not a factor of 9. The reason is the wrong argument of `gcd` in the recursive call to `gcd`: we wrote `n` instead of `m`. After fixing the bug, the assertion is always satisfied.

In contrast to data structures, which are only evaluated once during the computation, functions can be applied many times. The assertion is checked for each application and any failure is reported to the programmer.

The definition of `gcd` demonstrates how programmers should add assertions to their functions. The defined function is renamed (here to `gcd'`) and a wrapper with the original name (`gcd`) is defined.

Because `(-->)` is just a standard pattern constructor, its usage is not restricted to top-level function definitions. We can also use it for asserting properties of functional arguments and results as well as for functions occurring within data structures.

2.9 Negation and Implication

Finally we add negation to the logic: `neg :: Pat a b Bool -> Pat a b Bool`

We restrict negation to boolean formulas, because using values selected by both negated and non-negated patterns in the same predicate does not make sense. We can, for example, define implication in the common way:

```
(==>) :: Pat a b Bool -> Pat a b Bool -> Pat a b Bool
(==>) pat1 pat2 = neg pat1 ||| pat2
```

2.10 Defining new Patterns

Using our library does not come for free. The user has to define pattern constructors for their own data types. For each algebraic data type they usually have to define: a context pattern, pattern constructors for all its constructors, and an instance for the class `Observable`, defined in our library.

To make these definitions as simple as possible, we provide a set of combinators.[1] As an example, we introduces a data type `Tree` for polymorphic trees and show the definitions the programmer has to write for the pattern logic:

```
data Tree a = Node (Tree a) a (Tree a) | Empty

instance Observe a => Observe (Tree a) where
  observe (Node lt n rt) = o3 Node "Node" lt n rt
  observe Empty = o0 Empty "Empty"

pNode :: Pat (Tree a) b c -> Pat a c d -> Pat (Tree a) d e -> Pat (Tree a) b e
pNode = pat3 (\t -> case t of Node tl n tr -> Just (tl,n,tr)
                              _            -> Nothing)

pEmpty :: Pat (Tree a) b b
pEmpty = pat0 (\t -> case t of Empty -> Just ()
                               _     -> Nothing)

pTreeC :: Pat (Tree a) b c -> Pat (Tree a) b c
pTreeC = patContext (\t -> case t of Node tl n tr -> [(0,tl),(2,tr)]
                                    Empty -> [])
```

First, the programmer has to define an instance of the class `Observe`: for each constructor, they have to define an observation function. We provide generic observers for constructors of any reasonable arity. These observers have to be applied to the constructor function itself, a string representation of the constructor and the arguments obtained from pattern matching. The programmer also has to define the pattern constructors. Again, we provide generic versions for pattern constructors (`pat`n) for each arity. The only argument of these generic patterns is a function which makes pattern matching a total function by means of a `Maybe` type and a tuple of the same arity as the constructor. Finally, the programmer has to define the context pattern for their new type. They should use the generic function `patContext`, which takes a function that determines all arguments in which the type is recursive. We encode these arguments as a list of the argument number and the corresponding actual argument. Note, that descending within a data type only makes sense for arguments of the same data type. Whenever descending another type, it is necessary to add a context of this type. For instance, consider a tree of lists of `Int`s. An arbitrary `Int` within this tree can be selected by the pattern

```
pTreeC (pNode p_ (pListC (val <:> p_)) p_)
```

2.11 Positions in Data Structures

For tree-like data structures it can be useful to compare positions of selected values in the structure. We provide positional information by means of

[1] Alternatively, all definitions could easily be derived by a tool like DrIFT or various generic programming extensions of Haskell. Here we show that even without such tools or language extensions the required effort is reasonable.

```
valPos :: Pat a ((Pos,a) -> b) b
```

where Pos is an abstract data type which can be compared by functions such as

```
moreLeft :: Pos -> Pos -> Bool          above :: Pos -> Pos -> Bool
```

p1 `` `moreLeft` `` *p2* is true iff in an in-order traversal of the data structure *p1* is reached before *p2* is reached. *p1* `` `above` `` *p2* is true iff position *p2* is within the substructure at position *p1*.

For example, using positions, the property of being sorted can be defined as follows:

```
sortedPos = assert "sortedPos"
   (forAll (check (pListC (valPos <:> p_) +++ pListC (valPos <:> p_))
                  (\ (p1,x1) (p2,x2) -> p2 `moreLeft` p1 || x1<=x2))
```

We non-deterministically select two elements of the list and compare them taking their positions into account.

2.12 Deactivating Assertions

Any system supporting assertions in some language enables the programmer to easily deactivate assertions. Hence we provide a module `AssertWithoutCheck` with a function `assert` that is just implemented as the identity function on its third argument and does not check any assertion. To deactivate assertions the programmer replaces in their program `import Assert` by `import AssertWithoutCheck`.

While it may be advisable to leave simple assertions ("argument greater zero") in production code, our pattern logic encourages the formulation of properties of large data structures. Testing these properties is inherently time consuming. For example, it is infeasible in practice to check in a compiler after every update of the symbol table that the whole table is sorted with respect to a key.

3 Conclusions

We have presented a new approach for assertions in lazy functional programming languages such as Haskell. Our assertions do not modify the run-time behaviour of the lazy execution (unless a predicate fails to terminate). Assertions are implemented by means of a pattern logic, a high level, abstract specification language. Assertions provide a parallel implementation of conjunction and disjunction, which makes it possible to report failure of assertions promptly, before faulty values can effect the rest of the computation. Our approach is implemented as a library, without any modification of the compiler or the run-time system, and only needs common extensions of Haskell 98.

References

1. O. Chitil, D. McNeill, and C. Runciman. Lazy assertions. In P. Trinder, G. Michaelson, and R. Pena, editors, *Implementation of Functional Languages: 15th International Workshop, IFL 2003*, LNCS 3145, pages 1–19. Springer, November 2004.
2. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 48–59. ACM Press, 2002.
3. D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.

Über die formale Beschreibung räumlicher Netze

Hermann von Issendorff
Institut für Netzwerkprogrammierung
Hauptstr. 40, D-21745 Hemmoor

1 Einleitung

Thema dieses Berichts ist eine grundlegende formale Sprache, mit der sich die räumliche Struktur und die Funktionalität aller natürlichen diskreten Systeme beschreiben lassen. Diskrete Systeme sind solche, die sich aus Komponenten zusammensetzen. Die Komponenten können dynamisch oder statisch sein. Sind sie dynamisch, d.h. produzieren sie materielle Objekte oder werten sie Funktionen aus, dann sind sie zeitlich gerichtet. Sind sie statisch, dann lässt sich der Menge der Komponenten eine Aufbauordnung zuweisen, die ebenfalls eine zeitliche Richtung induziert.

Abstrahiert man ein diskretes System, beispielsweise einen Computer, von seiner Metrik, dann bleibt ein dreidimensionales Netz der Funktionen übrig, die in den Komponenten ausgeführt werden können. Abstrahiert man ein diskretes System, wieder beispielsweise einen Computer, von seiner Funktionalität, dann bleibt ein dreidimensionales Netz von Bausteinen übrig, die die Abmessungen der Komponenten haben. Abstrahiert man zugleich von Metrik und Funktionalität, dann bleibt ein dreidimensionales Netz von Knoten übrig. Dieses Knotennetz ist das Skelett, auf dessen Beschreibung die formale Sprache aufbaut.

Die formale Sprache wird Aktonalgebra genannt, wobei die Bezeichnung "Algebra" bedeutet, dass auf ihr funktionserhaltende Transformationen möglich sind. In ihrem Kern beschreibt sie das Skelett, auf das sich alle diskreten Systeme durch metrische und funktionale Abstraktion reduzieren lassen. Auf diesem Skelett können durch Hinzufügung von speziellen semantischen Eigenschaften formale Sprachen zu jedem funktionalen und/oder metrischen diskreten System generiert werden.

Die Klasse der diskreten Systeme ist ausserordentlich gross. Um zu verdeutlichen, was alles dieser Klasse zuzuordnen ist, nennen wir einige Beispiele: Programmiersprachen jeglicher Provenienz; neuartige Sprachen für die Beschreibung des Layout und der Funktionalität analoger oder digitaler Schaltungen, neuartige Sprachen zur vollständigen Beschreibung der physikalischen Struktur und der Funktionalität von Rechnern und Rechnernetzen; Sprachen zur algorithmischen Beschreibung mathematischer Systeme wie der Booleschen Algebra oder der klassischen Arithmetik; formale Sprachen für Verwaltungs-, Planungs- und Produktionssysteme; formale Sprachen für Molekülstrukturen, darunter insbesondere eine Sprache zur gemeinsamen Beschreibung von Aminosäureketten und der daraus gebildeten Proteine.

Die Aktonalgebra wurde ursprünglich als Programmiersprache für Rechnernetze entwickelt. Seit längerer Zeit ist erkannt, dass Aktonalgebra mit verschiedenen Semantiken ausgestattet werden kann und so z.B. die Eigenschaften einer klassischen Programmiersprache, einer Programmiersprache für digitale oder analoge Schaltungen oder einer Layout-Sprache annehmen kann. Unklar war bisher, wie die gemeinsame Basis aller dieser Anwendungssprachen formal erfasst werden kann. Mit diesen

Grundlagen, in denen der dreidimensionale Raum eine fundamentale Bedeutung hat, befasst sich dieser Bericht.

Obgleich das abstrakte räumliche Knotennetz eine einheitliche Basis aller diskreten Systeme darstellt, ist es - nach Kenntnis des Autors - bisher nicht wissenschaftlich untersucht worden. Dies ist besonders erstaunlich, da das abstrakte Knotennetz in der Informatik, dem wichtigsten technischen Wissensbereich der heutigen Zeit, eine besondere Rolle spielen sollte. Bildet es doch eine Brücke zwischen der Datenverarbeitung auf der eine Seite und der physikalischen Struktur der Maschinen, auf denen die Datenverarbeitung stattfindet, auf der anderen. Bisher fand dieser Zusammenhang offenbar keine weitere Beachtung. Z.B. enthält das Knotennetz genau die Strukturinformation, die ein optimales Layout von elektronischen Schaltungen ermöglicht. Bisher verzichtet man aber auf diese Strukturinformation und versucht stattdessen, diese mit ausserordentlich aufwändigen spieltheoretischen Verfahren angenähert zu rekonstruieren.

In der Vergangenheit hat es einige Ansätze gegeben, die Struktur elektronischer Schaltungen formal zu beschreiben, wobei zwischen der Funktionsstruktur und der Layout-Struktur zu unterscheiden ist. Alle diese Ansätze beschränken sich auf die Beschreibung planarer Strukturen. Beschrieben werden also nicht die realen räumlichen Strukturen, sondern nur deren Projektion auf eine Ebene, was erhebliche Informationsverluste mit sich bringt.

Ein früher Ansatz [2] beschränkt sich auf die formale Beschreibung rechtwinkliger Layouts, die mit Hilfe eines Nord/Süd- und eines Ost/West-Operators aus quadratischen Verdrahtungsbausteinen zusammengefügt werden. Eine Beziehung zum Knotennetz der Aktonalgebra besteht damit nur implizit und partiell.

Ruby [1] ist ein Ansatz, der neben der Funktionsbeschreibung auch die Anordnung der Funktionskomponenten in der Fläche einbezieht. Ruby ist eine relationale Programmiersprache, die Zustandsautomaten beschreibt und auf die Beschreibung synchroner Hardware-Schaltungen beschränkt ist. Die zunächst nur funktionale Beschreibung wird im Nachhinein semantisch in ein zweidimensionales Bezugssystem eingebettet, was als Layout-Eigenschaft interpretiert wird. Ruby wird dadurch vergleichsweise kompliziert und schwerfällig.

Ein dritter Ansatz, zu dem es eine Vielzahl von Publikationen gibt, verzichtet ganz auf die formale Beschreibung des Layouts und beschränkt sich auf die funktionale Beschreibung dynamischer Netze. Das Thema wird im Buch "Network Algebra" von G. Stefanescu systematisch und umfassend behandelt [3]. Network Algebra basiert auf einer regulären Sprache, in der schwarze Kästen, die über einen Input und einen Output verfügen, verknüpft werden können. Zu den schwarzen Kästen gehören insbesondere die parametrisierten Strukturelemente Kreuzung, Verzweigung, Vereinigung und Rückkopplung, die zur Übertragung von Strings dienen. Network Algebra kommt darin der Aktonalgebra nahe, auch wenn sie wegen der fehlenden Raumsemantik dreidimensionale Strukturen nicht analytisch beschreiben kann und dadurch deutlich komplizierter ist.

2. Topologisches Bezugssystem und homöomorphe Abbildung

Eine formale Sprache dient zur Erstellung von Texten. Texte sind Zeichenfolgen, die einen Anfang und ein Ende haben und Zeichen für Zeichen, d.h. zeitlich geordnet, gelesen werden. Abstrakt ausgedrückt sind Texte geordnete eindimensionale Strukturen. Seiner zeitlichen Ordnung wegen kann ein Text als Anweisungsfolge sowohl zum schrittweisen und damit konstruktiven Aufbau eines Systems als auch zur schrittweisen Änderung von lokalen Zuständen, z.B. der Auswertung von Funktionen, verstanden werden.

Die formale Sprache der Aktonalgebra bildet die dreidimensionale gerichtete Struktur eines dynamischen diskreten Systems auf die geordnete eindimensionale Struktur eines Textes ab. Der Kern der Aktonalgebra beschreibt das abstrakte Knotennetz, d.h. die topologische Struktur, die sich ergibt, wenn man ein dynamisches diskretes System von allen Funktionen und Abmessungen befreit. Die Abstraktion auf ein Knotennetz ist in Abbildung 1 am Beispiel eines *RS*-Flipflops demonstriert.

Das *RS*-Flipflop dient in dieser Arbeit als "running example". Es wurde einerseits deshalb gewählt, weil es mit seiner zyklischen Struktur bisher mit keiner Sprache annalytisch beschreibbar war und andererseits, weil es alle Basisstrukturen enthält, auf denen die Aktonalgebra aufbaut.

Links in der Abbildung 1 ist das Schaltbild gezeigt, das von der Metrik des realen Systems abstrahiert, rechts ein Layout, das von der Funktion abstrahiert. Beide münden bei weiterer Abstraktion in das in der Mitte gezeigte Knotennetz. Die Knoten sind die dimensionslosen Abstraktionen der Systemkomponenten, wobei zu beachten ist, dass auch jede physikalische Verbindung zwischen Systemkomponenten selbst eine Komponente ist. Die in den bildlichen Knotennetzen gezeigten Richtungspfeile stellen dagegen lediglich Abhängigkeitsbeziehungen dar.

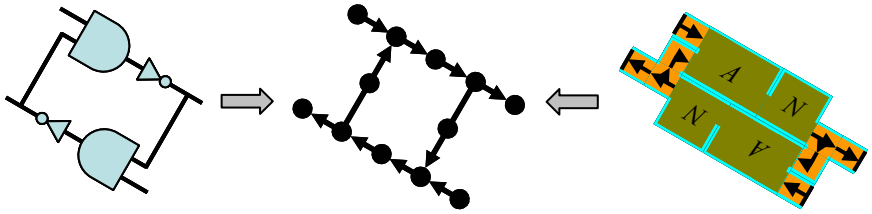


Abbildung 1: Knotennetz als Abstraktion von der Schaltung (links) und der Metrik (rechts).

Das Knotennetz, was im allgemeinen Fall räumlich ist, kann sodann homöomorph und damit bijektiv auf die eine Dimension eines Textes abgebildet werden. Die homöomorphe Abbildung lässt nicht nur die Dehnung und Streckung des Knotennetzes zu, sondern auch das Hinzufügen von funktionslosen Knoten und deren Aufspaltung in Knotenpaare.

Da das abstrakte Knotennetz keine Metrik hat, kommt für die Beschreibung seiner räumlichen Eigenschaften nur ein topologisches Bezugssystem in Betracht. Ein solches lässt sich mittels eines Beobachters definieren, der in natürlicher Weise Raum in den Relationen rechts bzw. links, oben bzw. unten und vorne bzw. hinten wahrnimmt. Der in der westlichen Welt üblichen Leserichtung entsprechend wird *Zeit* in

diesem Bezugssystem als Bewegung von links nach rechts interpretiert, womit *links* zusätzlich die Bedeutung *früher* erhält und *rechts* die Bedeutung *später*.

Die homöomorphe Abbildung der dreidimensionalen Raumstruktur des Knotennetzes auf die eindimensionale Struktur des Textes geschieht in zwei Schritten: Im ersten Schritt erfolgt die Abbildung auf eine Beobachtungsebene zwischen dem Beobachter und dem Knotennetz, im zweiten die Abbildung der ebenen Struktur auf die lineare Struktur des geordneten Textes.

Die Abbildung des Knotennetzes auf die Beobachtungsebene beinhaltet die Projektion der Raumstruktur unter Orientierung aller Knoten von links nach rechts (Abbildung 2). Dabei können, wie im betrachteten Beispiel, Kreuzungen entstehen. Diese sind Überbleibsel der ursprünglichen Raumstruktur und müssen so umgeformt werden, dass die Darstellung des Knotennetzes insgesamt planar ist. Dies lässt sich dadurch erreichen, dass die untere der beiden kreuzenden Verbindungen aufgeschnitten und die Schnittenden durch ein Knotenpaar ersetzt werden, das die Verbindung symbolisiert. In der Abbildung 2 ist das Knotenpaar mit einer gepunkteten Linie verbunden.

Ein ähnliches Knotenpaar ist erforderlich, wenn Zyklen oder andere planare Strukturen aufgeschnitten und orientiert dargestellt werden sollen. Dieses Knotenpaar kann aber auch zur willkürlichen symbolischen Teilung einer Struktur in Teilstrukturen verwendet werden. In der Abbildung 2 ist ein solches Knotenpaar durch eine gestrichelte Verbindungslinie gekennzeichnet. Sind alle Kreuzungen und Zyklen auf diese Weise umgeformt, hat man die angestrebte orientierte ebene Struktur.

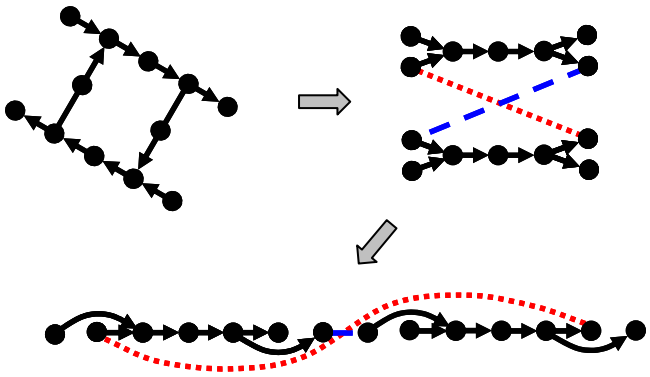


Abbildung 2: Zweistufige homöomorphe Abbildung: 1. Orientierung des Knotennetzes in der Beobachtungsebene von links nach rechts, 2. Geordnete Linearisierung der Knoten

Die Abbildung der ebenen Struktur auf die lineare Struktur des Textes geschieht durch Aufreihung aller Knoten auf der Links/Rechts-Achse. Dies erfolgt unter der Vereinbarung, dass obere Knoten bzw. Knotenstrukturen nach links und untere Knoten bzw. Knotenstrukturen nach rechts auseinander gezogen werden. Es gibt jedoch spezielle Basisstrukturen, so genannte traversierte Maschen, bei denen diese Linearisierung nicht direkt möglich ist. Sie werden aber linearisierbar, wenn sie wie Zyklen aufgeschnitten werden. Das Resultat ist eine Kette, die aus Gruppen unabhängiger und abhängiger Knoten besteht.

Die formalen Mittel zur Planarisierung bzw. Linearisierung der drei Basisstrukturen Kreuzung, Zyklus und traversierte Masche werden im folgenden Abschnitt näher behandelt.

3. Operatoren der Aktonalgebra

In dem Bezugssystem definieren wir zunächst zwei orthogonale binäre Operatoren, die zur algebraischen Beschreibung von unmittelbaren Nachbarschaften in der Beobachtungsebene dienen. Der erste Operator, *Next* genannt und durch das Trennzeichen "-" dargestellt, beschreibt die *Links/Rechts*- bzw. *Früher/Später*-Beziehung. Der Ausrichtung des Knotennetzes im Bezugssystem entsprechend beschreibt *Next* eine Beziehung zwischen abhängigen Elementen. Der Ausdruck e_1-e_2 bedeutet, dass e_2 unmittelbar auf e_1 folgt. Der zweite Operator, *Juxta* genannt und durch den Schrägstrich "/" dargestellt, beschreibt eine Oben/Unten-Beziehung und damit eine Beziehung zwischen unabhängigen Elementen. Der Ausdruck e_1/e_2 bedeutet, dass e_1 unmittelbar oberhalb von e_2 liegt.

Als weitere Grundlage zur Formalisierung der Knotennetze definieren wir die Netzknoten. Netzknoten bilden die Konstanten (*0*-Operatoren) der Aktonalgebra:

- Jeder Knoten besteht aus einem Eingang, einem Ausgang und einem knoteninternen Netzwerk, das Ein- und Ausgang verbindet.
- Die Ein- und Ausgänge sind Schnittstellen, d.h. linear geordnete Listen (strings) endlicher Länge. Innerhalb des Bezugssystems sind die Schnittstellenelemente von oben nach unten geordnet. Dazu dient der *Juxta*-Operator.
- Die Schnittstellen werden aus drei Sorten von Elementen gebildet:

  p (*pin*) für Kontakte in der Beobachtungsebene,

  v (via) für Kontakte unterhalb der Beobachtungsebene und

  λ als Platzhalter für nicht vorhandene Kontakte.
- Eine leere Schnittstelle, d.h. eine Schnittstelle ohne Kontakte, enthält mindestens einen Platzhalter. Formalsprachlich bilden *p*, *v* und λ das Alphabet, über dem die Schnittstellen generiert sind.
- Die knoteninternen Netzwerke werden aus drei Sorten gerichteter Strukturelemente gebildet

  link (1,1-Verbindung), *fork (1,2-Verbindung)* und *join (2,1-Verbindung)*
- Knoteninterne Netzwerke sind irreflexiv, asymmetrisch und intransitiv.
- Primitive Knoten sind solche, die genau ein *link*, *fork* oder *join* enthalten.
- Knoten und ihre funktionalen oder metrischen Konkretisierungen werden Aktonen genannt.

Betrachtet man das Knotennetz des *RS*-Flipflops, dann findet man darin alle Sorten primitiver Knoten, mit denen sich jedes beliebige Knotennetz beschreiben lässt. Insgesamt gibt es 8 Sorten primitiver Knoten , die mit *Entry*, *Exit*, *Up*, *Down*, *Link*,

Fork, *Join* und *AS* (vollständiges Akton-System) bezeichnet werden. Sie bilden die Basismenge für die mehrsortige Aktonalgebra.

Die Basismenge der Aktonen wird mit A_0 bezeichnet, die Basismenge der Schnittstellen mit I_0 und die Basismenge der knoteninternen Netzwerke mit R_0.

Die 8 Knotensorten unterscheiden sich zum einen durch ihre interne Netzstruktur und zum anderen durch ihre Eingangs- und Ausgangsschnittstellen. Die primitiven Knoten sind in Tabelle 1 definiert:

Tabelle 1: Die Basisaktonen der 8 Knotensorten

Entry	*Exit*	*Up*	*Down*	*Link*	*Fork*	*Join*	*AS*
λ link p	*p link λ*	*v link p*	*p link v*	*p link p*	*p fork (p/p)*	*(p/p) join p*	*λ link λ*

Die Bezeichner der internen Netzstrukturen werden im Unterschied zu den teilweise gleichnamigen Aktonen klein geschrieben.

Down- und *Up*-Aktonen treten immer paarweise auf. Sie dienen dem Zweck, Kreuzungen als orientierte planare Strukturen darzustellen. Das geschieht dadurch, dass jede unterhalb der Beobachtungsebene liegende Verbindung aufgetrennt und durch ein *Down/Up*-Paar ersetzt wird. Mit *Down* taucht die Verbindung über ein Via im Ausgang aus der Beobachtungsebene ab und mit dem *Up* taucht die Verbindung über ein Via im Eingang an anderer Stelle wieder auf. Der Eingang von *Down* ist gleich dem Ausgang von *Up*. Ein *Down/Up*-Paar hat damit die äusserlichen Eigenschaften eines *Link*.

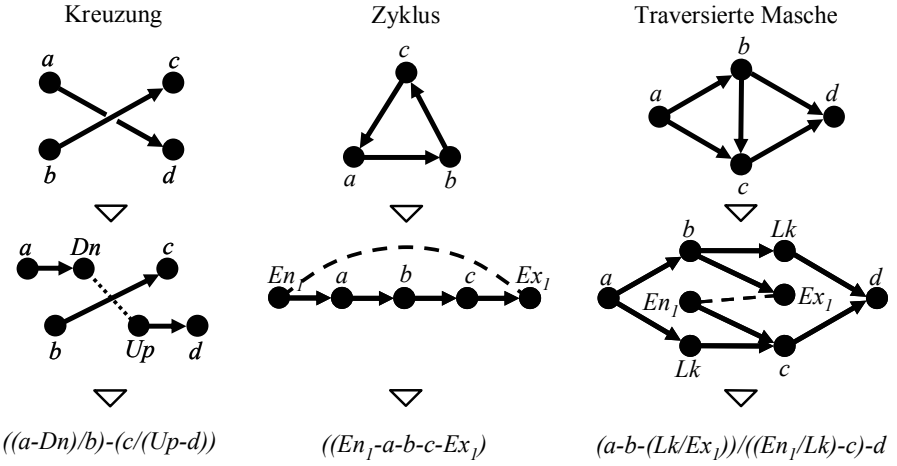


Abbildung 3: Linearisierung von Basisstrukturen durch Schnitte

In gleicher Weise lassen sich auch Verbindungen, die in der Beobachtungsebene liegen, durch gleichindizierte $Exit_j/Entry_j$-Paare symbolisch beschreiben. Die Gleichindizierung bedeutet dabei Gleichheit des Eingangs von $Exit_j$ und des Ausgangs von $Entrj_i$. Ein gleichindiziertes *Exit/Entry*-Paar hat damit wie ein *Down/Up*-Paar ebenfalls die äusserlichen Eigenschaften eines *Link.*

Gleichindizierte *Exit/Entry*-Paare dienen zum einen dazu, Zyklen symbolisch auftrennen und so in eine orientierte lineare Darstellung überführen zu können Zum anderen gibt es spezielle planare Strukturen, sogenannte traversierte Maschen, die durch ein $Exit_j/Entry_j$-Paar aufgetrennt werden müssen, um aktonalgebraisch beschreibbar zu werden.

Abbildung 3 zeigt die drei gerichteten Basisstrukturen, die sich nur mit Hilfe von *Down/Up*- bzw. gleichindizierten *Exit/Entry*-Paaren linearisieren lassen. Oben dargestellt sind jeweils die Ausgangsstrukturen, in der folgenden Zeile die orientierte planare Struktur und unten die orientierte lineare Struktur des algebraischen Ausdrucks. Die räumliche Kreuzung links wird durch ein *Down/Up*-Paar planarisiert. Die planaren aber nichtorientierten Basisstrukturen Zyklus und traversierte Masche lassen sich durch ein $Exit_j/Entry_j$-Paar in linearisierbare orientierte Strukturen überführen.

4. Definition der Aktonalgebra

Seien

$\{I,/\}$ eine Schnittstellensprache,
$\{R,/,-\}$ eine Vernetzungssprache und
$\{A,/,-\}$ eine Relationensprache (Aktonsprache),

I, R, A sind durch Komposition erweiterbare Mengen von Konstanten, die auf den bereits im letzten Abschnitt eingeführten Basismengen der Schnittstellen I_0, der internen Vernetzungen R_0 und der Aktonen A_0 aufbauen, d.h.

$I \supseteq I_0,\ I_0=\{(p/p),p,v,\lambda\}$,
$R \supseteq R_0,\ R_0=\{link\ fork,join\}$,
$A \supseteq A_0\ ,\ A_0=\{Link,Fork,Join,Entry,Exit,Up,Down,AS\}$

Zu den oben definierten Sprachen lassen sich jeweils die algebraischen Universen I^+, R^+ und A^+ generieren, d.h. die Mengen aller Strings über I, R und A.

Alle drei Sprachen enthalten den *Juxta*-Operator, die beiden letzten zusätzlich den *Next*-Operator. *Juxta* und *Next* sind irreflexiv, asymmetrisch und intransitiv. Da sie zueinander orthogonal sind, sind I^+, R^+ und A^+ freie Halbgruppen.

Ein Akton a ist über A, R und I definiert, d.h. durch
$a = (in_a\ r_a\ out_a)$, mit $a \in A$, $r_a \in R$ *und* $in_a, out_a \in I$

Ein Aktonterm x ist über A^+, R^+ und I^+ definiert, d.h. durch
$x = (in_x\ r_x\ out_x)$, mit $x \in A^+$, $r_x \in R^+$ *und* $in_x, out_x \in I^+$

Aktonen und Aktonterme sind gültige Relationen zwischen einem Input und einem Output.

Die Verknüpfung von primitiven Aktonen zu Aktontermen führt zu weiteren Sorten. Die Zahl der Aktontermsorten ist daher grösser als die der primitiven Aktonen. Die Gesamtzahl der Aktontermsorten ergibt sich aus der folgenden Betrachtung: Jede Schnittstelle $i \in I^+$ enthält eine endliche Zahl von Elementen aus der Menge $\{p,v,\lambda\}$. Enthält sie weder p- noch v-Elemente, dann zumindest einen Platzhalter λ. Neben

Platzhaltern kann sie nur *p*-Elemente, nur *v*-Elemente, beide gemeinsam oder keine von beiden enthalten. Bezüglich *p* und *v* gibt es damit *4* Sorten von Schnittstellen.

Für einen Aktonterm ergeben sich mit einem Input und einem Output *4*4=16* Sorten. Im Input und Output eines Aktonterms zugleich auftretende *v*-Elemente stammen von *Down/Up*-Paaren, deren *v*-Elemente sich gegenseitig kompensieren und daher nach aussen nicht in Erscheinung treten. Diese *v*-Elemente müssen daher aus den Schnittstellen entfernt und durch Platzhalter ersetzt werden. Das reduziert die Zahl der Sorten eines Aktonterms um *4*, d.h. die Zahl der Aktontermsorten auf *12*.

Um alle *v*-Elemente in den Schnittstellen eines Aktonterms durch Platzhalter zu ersetzen, führen wir eine Funktion *v-del* ein:

$$\begin{array}{ll} v\text{-}del: I^+ \rightarrow I^+ & del: i \rightarrow i \\ v\text{-}del(i) = \forall v \in i.\ del(v) & del(v)=\lambda,\ del(p)=p,\ del(\lambda)=\lambda \end{array}$$

Wir haben damit alle Mittel verfügbar, um die Aktonalgebra

$$\langle A^+, \{A, /, -\} \rangle$$

zu definieren:

A basiert auf der Menge der Knotensorten A_0

$$A \supseteq A_0,\ A_0 = \{Link, Fork, Join, Entry, Exit, Up, Down, AS\}$$

Juxta verknüpft zwei Aktonterme, die keine gemeinsame Schnittstelle haben.

$$Juxta : A^+ \times A^+ \rightarrow A^+$$
$$(x/y) = ((in_x/in_y)\ (r_x/r_y)\ (out_x/out_y)) \text{ if } \neg(\exists v \in in_{(x/y)} \wedge \exists v \in out_{(x/y)})$$
$$\text{else } (v\text{-}del(in_x/in_y)\ (r_x/r_y)\ v\text{-}del(out_x/out_y))$$

Next verknüpft zwei Aktonterme, die eine gemeinsame Schnittstelle haben, die mindestens ein *p* enthält.

$$Next : A^+ \times A^+ \rightarrow A^+$$
$$(x\text{-}y) \text{ iff } (out_x \equiv in_y \wedge \exists p \in out_x)$$
$$(x\text{-}y) = (in_x\ (r_x\text{-}r_y)\ out_y) \text{ if } \neg(\exists v \in in_{(x/y)} \wedge \exists v \in out_{(x/y)})$$
$$\text{else } (v\text{-}del(in_x)\ (r_x\text{-}r_y)\ v\text{-}del(out_y))$$

Um Klammern einzusparen, wird festgelegt, dass *Juxta* stärker bindet als *Next*.

Schliesslich führen wir noch eine Funktion *compose* ein, mit der sich die Basismenge *A* der Akronalgebra erweitern lässt. Mit Hilfe von *compose* kann im Prinzip jeder aus zwei Aktonen bestehender Aktonterm zu einem Akton verkapselt und die Darstellung von Aktonstrukturen damit beliebig vergröbert werden. Diese Eigenschaft ist unentbehrlich für die hierarchische Beschreibung komplexer Systeme. Die Funktion *compose* verschmilzt jeweils zwei *Juxta*- oder *Next*-verknüpfte Aktonen zu einem Akton. Sie ist definiert durch:

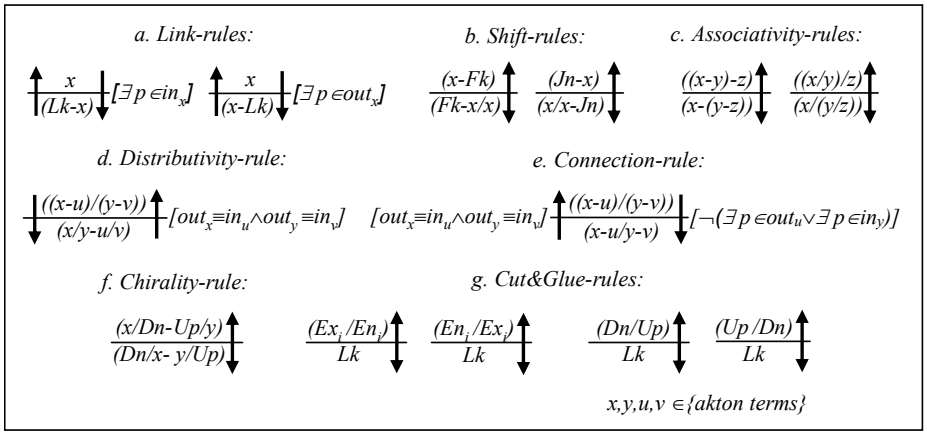
$$compose \circ Juxta: A \times A \rightarrow A,$$
$$compose(a/b)=c \text{ mit } (in_a/in_b)=in_c,\ (r_a/r_b)=r_c,\ (out_a/out_b)=out_c,$$
$$compose \circ Next: A \times A \rightarrow A,$$
$$compose(a\text{-}b)=c \text{ mit } in_a=in_c,\ (r_a\text{-}r_b)=r_c,\ out_b=out_c,$$

5. Funktionsinvariante Strukturtransformationen

Diskrete Systeme lassen sich im Allgemeinen strukturell modifizieren, ohne dass dabei die Funktionen des Systems verändert werden. In der Aktonalgebra geschieht das durch Termersetzungsregeln, die in der Tabelle 2 aufgeführt sind. Die Termersetzungsregeln ersetzen die Axiome, mit denen Algebren mathematisch definiert werden. Axiome sind aber lediglich zulässig, wenn es nur um Funktionen geht. In der Aktonalgebra dagegen werden die physikalischen Strukturen beschrieben, die für die Ausführung bestimmter Funktionen erforderlich sind, d.h. genau die Strukturen, von denen in der Mathematik abstrahiert wird.

Jede Termersetzungsregel besteht aus zwei durch eine horizontale Linie getrennte Aktonterme. Die Pfeile geben an, in welcher Richtung die Aktonterme ausgetauscht werden dürfen. Ist Austausch nur unter einer Bedingung zulässig, dann ist diese neben dem Pfeil ausgewiesen. Würde man von der physikalischen Struktur abstrahieren, dann wären die bedingten Termersetzungsregeln Implikationen und alle beidseitig unbedingten Termersetzungsregeln Äquivalenzen.

Tabelle 2: Auflistung der funktionsinvarianten Strukturtransformationen



Die unter *a.*, *b*. und *c.* aufgeführten Regeln sind unmittelbar verständlich: Die Regeln *a.* besagen, dass ein Term *x* durch ein *Link* erweitert werden kann, wenn die betreffende Schnittstelle mindestens ein *p* enthält. Umgekehrt können vorstehende oder nachfolgende immer beseitigt werden. Regeln *b.* besagen, dass ein Term verdoppelt wird, wenn er mit einem nachfolgenden *Fork* (vorstehenden *Join*) vertauscht wird und umgekehrt. Die Regeln *c.* definieren Assoziativität für *Next* und *Juxta*.

Die Distributivregel *d.* findet man in vielen Veröffentlichungen, jedoch immer ohne die einseitige Bedingung. In Abwärtsrichtung beschreibt sie die Aufspaltung von zwei *Juxta*-verknüpften *Next*-Termen in zwei *Next*-verknüpfte *Juxta*-Terme. Diese Strukturtransformation lässt sich zur Sequentialisierung der parallelen Terme *x/y* nutzen: Dazu erweitert man den Parallelterm *x*/*y* zunächst unter Anwendung der

beiden Regeln a. zu dem Term *(x-Lk)/(Lk-y)*, aus dem durch Anwendung von Regel d. *(x/Lk-Lk/y)* wird. Die Terme *x* und *y* sind darin *Next*-geordnet, d.h. sequentialisiert. Die umgekehrte Strukturtransformation, die im behandelten Beispiel einer Parallelisierung entspricht, ist dagegen nur zulässig, wenn die Schnittstellen zwischen den Termen *x* und *u* identisch sind und ebenso die zwischen den Termen *y* und *v*, was nicht der Fall sein muss.

Die Bindungsregel *e.* erlaubt die Vereinigung von zwei *p*-unabhängigen Termen zu einem Term. Die Umkehrung hat die gleiche Bedingung wie Aufwärtsbedingung in d.. Die Chiralitätsregel *f.* erlaubt die Änderung des Drehsinns einer Kreuzung. Der Drehsinn ist eine physikalische Kreuzungseigenschaft, die nur in Erscheinung tritt, wenn Kreuzungen als Projektionen im Raum betrachtet werden.

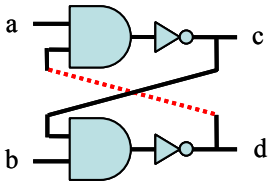
Die *Cut&Glue* genannten Regeln *g.* sind ein wesentlicher Bestandteil der homöomorphen Abbildung, die der Aktonalgebra zugrunde liegt. Sie können in der gleichen Weise verwendet werden, um ein gegebenes diskretes System beliebig zu partitionieren.

6. Beschreibung realer diskreter Systeme

Aktonalgebra ist nicht nur zur Beschreibung abstrakter Knotennetze geeignet, sondern auch zur Beschreibung realer diskreter Systeme. Dies lässt sich lediglich dadurch erreichen, dass die 8 Konstanten der Aktonalgebra als Sortenbezeichner interpretiert werden und damit den Charakter von Variablen annehmen. Jede der 8 Sorten ist dann eine Menge spezieller Elemente, die die Struktureigenschaften der Sorte haben. Betrachten wir zur Verdeutlichung wieder das Beispiel *RS*-Flipflop.

Spezifikation:

$Entry \in \{En_i\}$	$Up \in \{U\}$	$Fork \in \{F\}$
$Exit \in \{Ex_j\}$	$Down \in \{D\}$	$Join \in \{And\}$
$i,j \in \{a,b,c,d\}$	$Link \in \{Wire,Not\}$	



En_a/U-And-Not-F-Ex_c/Wire/En_b-And-Not-F-D/Ex_d

Abbildung 4: Spezifikation einer speziellen Aktonalgebra zur Schaltbildbeschreibung. Darunter die struktural und funktional vollständige Beschreibung des *RS*-Flipflop-Schaltbildes

Will man es als Schaltung, d.h. funktional spezifizieren, dann ist *Link* der Sortenbezeichner für alle elektronischen 1,1-Elemente, die in der Schaltung entweder Leitungen oder Inverter sein können. Formal ist das durch $Link \in \{Wire,Not\}$ darstellbar. In der gleichen Weise lassen sich alle anderen Sorten spezifizieren. Für das *RS*-Flipflop genügt z.B. die *Join*-Spezifikation $Join \in \{And\}$. *Entry* und *Exit* sind im Beispiel freie Leitungsenden, können aber ebenso auch Steckkontakte oder vielleicht sogar Funkverbindungen sein. Abbildung 4 zeigt die Spezifikation aller 8 Aktonsor-

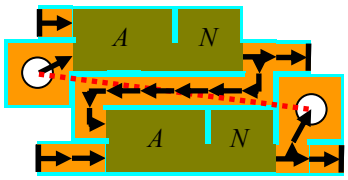
ten für eine primitive Schaltbildsprache, und darunter das Schaltbild des *RS*-Flipflop zusammen mit der akton-algebraischen Beschreibung seiner Funktion und Topologie. Im Prinzip reicht diese primitive Schaltbildsprache bereits aus, einen Rechner vollständig zu spezifizieren.

In der gleichen Weise lässt sich auch das Layout des *RS*-Flipflops beschreiben. Der Einfachheit halber nehmen wir an, dass das Layout in ein quadratisches Raster eingebettet ist. Dann lässt sich definieren $Link \in \{W_s, W_l, W_r\}$ und $Fork \in \{Fk_{sl}, Fk_{sr}, Fk_{lr}\}$, wobei die Indizes *s, l* und *r straight, left-turn* und *right-turn* bedeuten sollen. Jedes dieser Elemente belege genau ein quadratisches Rasterelement. Um ein gewünschtes Layout zu erreichen, müssen im Allgemeinen sehr viele *W*-Elemente *Next*-verknüpft werden. Zur Schreibvereinfachung wird deshalb eine Abzählkonvention eingeführt. *And* und *Not* werden beim Layout zu Flächenelementen, die ein rechteckiges Vielfaches des Rasters belegen. Für *Up* und *Down* wird in Anlehnung an die Realität eine 2×2-Rasterfläche angenommen. Damit gibt es für *Up* und *Down* zwei verschiedene Schnittstellen (p/λ) und (λ/p), je nachdem, ob die *p*-Verbindung oben oder unten liegt. Im Layout von Abbildung 5 haben *Up* und *Down* beide die Ausgangs- bzw. Eingangsschnittstelle (p/λ). Wo erforderlich, können die Schnittstellen der Aktonen auch explizit in die Beschreibung aufgenommen werden. Hier verzichten wir darauf.

Abbildung 5 zeigt die Spezifikation der Layoutsprache und darunter die Beschreibung eines Layouts zum *RS*- Flipflop-Beispiel in dieser Sprache.

Spezifikation:

$Entry \in \{E_i\}$	$Up \in \{U\}$	$Fork \in \{Fk_{sl}, Fk_{sr}, Fk_{lr}\}$
$Exit \in \{X_j\}$	$Down \in \{D\}$	$Join \in \{A\}$
$i,j \in \{a,b,c,d\}$	$Link \in \{W_s, W_l, W_r, N\}$	$nx=(n-1)x-x,\ n>1$
		$1x=x,\ n \in \mathbf{N},\ x \in A^+$



$$E_a/U\text{-}A\text{-}N\text{-}F_{sr}\text{-}X_c/W_r\text{-}4W_s\text{-}2W_l/(E_b\text{-}W_s)\text{-}A\text{-}N\text{-}F_{sl}\text{-}D/X_d$$

Abbildung 5: Spezifikation einer speziellen Aktonalgebra zur Layout-Beschreibung. Darunter ein Layout zum *RS*-Flipflop-Beispiel

Literatur

[1] Jones, G., Sheeran, M.: Circuit Design in Ruby. In: Formal Methods for VLSI Design. ed . J. Staunstrup, Horth Holland (1990)

[2] Kolla, R., Molitor, P., Osthoff, H.G..: Einführung in den VLSI-Entwurf. Leitfäden und Monographien der Informatik. B.G.Teubner (1989)

[3] Stefanescu, Gh..: Network Algebra. Springer, Berlin 2000

Formal Result Checking for Unverified Theorem Provers

Nicole Rauch

rauch@informatik.uni-kl.de
Technische Universität Kaiserslautern, Germany

1 Introduction

The importance of formal software verification is growing steadily. Software companies that develop safety- or security-critical applications start to introduce formal verification into their software production processes [ACL03,BRL03]. Usually, special-purpose theorem provers are used for this task. The vast majority of these provers has not been formally verified for various reasons. This is a huge drawback because these tools are not formally trustworthy. The work presented in this paper shows how this drawback can be circumvented. The approach presented here increases the credibility of the verification of programs performed by unverified proof tools, without having the need to formally verify these tools. The basic idea is to transform a proof that has been performed in an unverified prover to a description of the proof that can be checked by a trusted proof checker. To further increase the credibility of this approach, different trusted checkers can be used. The approach is applied to the Jive tool as unverified prover and to Isabelle/HOL as trusted proof checker to demonstrate its usability.

2 Formal Result Checking

It is highly desirable that a verification process yields a trustworthy result. In order to achieve this, a first naïve approach might be to verify the verification tool itself. This immediately requires to verify other tools and libraries involved in the verification, to verify the operating system, and to verify the hardware. This poses the first problem because the verification tool alone usually is already a large application, perhaps with a graphical interface. To verify such a system is very expensive and may take a long time. To verify a whole standard operating system like Windows or Linux currently seems infeasible. An annoying side-effect of the verification process is that further development of any component of the system is blocked because if the code is being changed, the verification usually has to be repeated at least partially. – But even if we assume that we are able to handle this task, the second problem arises: What would be a good and usable specification? A desired specificaton might be "All proofs are correct.". But how is this expressed in terms of pre- and postconditions of single methods? How can we make sure that the specification of a certain method adds to this goal if the method, say, displays a dialog that allows the user to enter a formula?

– But even if we assume that we are able to handle this task as well, a third problem appears: How should we prove this specification? We can (a) prove the specification of the proof tool in the proof tool itself, but this proof does not give us the desired result because errors in the tool may produce faulty proof trees that are incorrectly accepted as valid; (b) prove it in another proof tool, but then we would have to formally verify the other tool, which brings us back to the beginning; (c) prove it in a formal theorem prover, but then we would have to prove that the formalization of our proof tool in the formal theorem prover is related to our implementation of our tool; or (d) prove it manually, which imposes the same problem as (c) and is less credible than (c) for many people. Thus, none of these approaches is feasible in practice. – But even if we assume that we are able to handle this task, too, we may want to find a cheaper way of assuring the trustworthiness of the proof result.

The work presented in this paper shows how we can circumvent the problems above and at the same time drastically reduce the cost. The credibility of the proof results performed by unverified proof tools is increased without requiring to formally verify these tools. The basic idea is to use an unverified proof tool to perform a proof task. We then add an independent proof checker. It is a general observation that in many cases, the effort to solve a problem is much higher than it is to check whether a certain solution for a problem is correct. For example, it is much harder to determine the solution of a linear equation system than to check whether a given solution is correct. In theorem proving, this discrepancy is even more drastic because not every step of the proof process is decidable, thus there cannot exist a fully automatic proof generator that proves all given problems. To check an existing proof, on the other hand, is fairly trivial compared to that because one only has to ensure that all proof rule applications have been performed correctly. The mechanization of such a checker is straightforward.

In order to be used in our approach, the proof checker needs to have a formalization of the underlying logic of the unverified prover. Each proof goal of the unverified prover is transformed into a proof goal for the checker. This transformation should be as marginal as possible. This way, it is possible for a human verification engineer to ascertain that the semantics of the proof goal of the original prover is carried over to the proof checker without alterations. Each of the proofs that have been performed in the unverified prover is then translated by a proof transformer tool to some proof description for the checker, e.g. a proof script. Finally, this proof description is checked with the trusted checker. This way, the checker's only task is to check whether the proof produced by the transformer tool is valid for the transformed proof goal. It is neither required that the source proof that was performed in the unverified prover is correct nor that the transformer tool works correctly. As a result, we know that the transformed proof goal holds if the checker succeeds. The only transformation step that must be correct is the transformation of the proof goal. That is why we require the transformed proof goal to be as similar to the original goal as possible. This allows to compare both proof goals with simple means, e.g. by inspecting them visually or by applying simple tools such as textual comparison tools. If we can

ensure that the semantics of the transformed proof goal is identical to the semantics of the original proof goal, we know that the original proof goal holds as well, even though we do not know whether the original proof is correct. Fig. 1 shows how the unverified prover, proof transformer and trusted checker interact with the two proof representations.

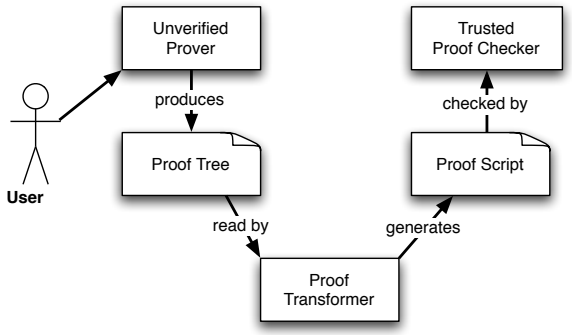


Fig. 1. Interaction of the various tools and proof representations

This approach seems to impose a hen and egg problem at first sight because now we need a trusted checker instead of a trusted verification tool. The advantage is that the proof checker can be much smaller than the verification tool because there is no need for a graphical user interface, for automatic proof generation support, for proof manipulation etc. Therefore, it might be feasible to perform a verification of this proof checker with reasonable cost to achieve a high level of trust. Another approach is to embed such a checker into a formally rigorous theorem prover which is considered trustworthy in a certain community. To further increase the credibility of this approach, different trusted checkers, implemented by different people in different programming languages or theorem provers, running on different operating systems and different hardware, can be used.

A highly positive effect of this approach is that it does not impose a problem on the process if the unverified theorem prover is being modified at some point in time. In contrast to this, if we had performed a verification of the tool, we would be required to repeat at least parts of those proofs. Therefore, our approach can even be used for verification tools that are still under development. Changes to the prover, e.g. optimizations, can immediately be validated. Thus, our approach is also a valuable aid in testing the prover during development if the process of generating a checker script is already established during the development process.

In certain settings, the unverified prover might even act as a user-friendly interface to the proof checker, e.g. if the unverified prover has a sophisticated graphical user interface and the proof checker is embedded into a formally rig-

orous theorem prover with just a text-mode user interface. Using both in combination, the user can produce proofs in the checker although only interacting with the more convenient unverified prover.

In the following, we use an unverified Hoare-logic program verification tool and a checker that has been embedded into a formal theorem prover to demonstrate the application of our approach. Nonetheless, the approach can be adapted to any kind of proof tool and trusted checker.

Related Work. Runtime result checking has first been introduced by Blum [WB97]. It was introduced as a means of assuring software reliability by adding runtime error-identification and possibly error-correction. Their focus is on checking the critical parts of a program and asserting their correctness. This was extended by Goerigk, Gaul and Zimmermann [GGZ98]. They combined verification and runtime checking techniques to prove the correctness of a program. They argue that program parts that are checked at runtime need not be verified if the checker is correct. Thus, their approach partly substitutes program verification by runtime checking. The approach presented here lifts the runtime checking approach to the next level: not the result of a program is checked but the result of the verification of a program. Verification is enhanced by checking to improve the credibility of verification results that are produced by an unverified tool.

3 The Jive Tool

Jive [MPH00] is a verification system that is being developed at the University of Kaiserslautern and at the ETH Zürich. It is an interactive special-purpose theorem prover for the verification of object-oriented programs on the basis of a partial-correctness Hoare-style programming logic. Jive operates on Java-KE [PHGR05], a desugared subset of sequential Java which contains all important features of object-oriented languages (subtyping, exceptions, static and dynamic method invocation, etc.). It is written in Java and currently has a size of about 40,000 lines of code.

Jive is able to operate on completely unannotated programs, allowing the user to dynamically add specifications. It is also possible to preliminarily annotate programs with invariants, pre- and postconditions using the specification language JML [LBR99]. In practice, a mixture of both techniques is employed, in which the user extends and refines the pre-annotated specifications during the verification process. The program to be verified, together with the specifications, is translated to Hoare sequents. Program and pre-annotated specifications are translated during startup, while the dynamically added specifications are translated whenever they are entered by the user. Hoare sequents have the shape $\mathcal{A} \rhd \{ \mathbf{P} \}$ `pp` $\{ \mathbf{Q} \}$ and express that for all states S that fulfill $\mathbf{P}$, if the execution of the program part `pp` terminates, the state that is reached when `pp` has been evaluated in S must fulfill $\mathbf{Q}$. The so-called assumptions $\mathcal{A}$ are used to prove recursive methods.

Jive's logic contains so-called Hoare rules and axioms. The rules consist of one or more Hoare sequents that represent the assumptions of the rule, and

a Hoare sequent which is the conclusion of the rule. Axioms consist of only one Hoare sequent; they do not have assumptions. Therefore, axioms represent the known facts of the Hoare logic. To prove a program specification, the user directly works on the program source code. Proofs can be performed in backward direction and in forward direction. In backward direction, an initial open proof goal is reduced to new, smaller open subgoals by applying a rule. This process is repeated for the smaller subgoals until eventually each open subgoal can be closed by the application of an axiom. If all open subgoals are proven by axioms, the initial goal is proven as well. In forward direction, the axioms can be used to establish known facts about the statements of a given program. The rules are then used to produce new facts from these already known facts. This way, facts can be constructed for parts of the program.

A large number of the rules and axioms of the Hoare logic is related to the structure of the program part that is currently being examined. Besides these, the logic also contains rules that manipulate the pre- or postcondition of the examined subgoal without affecting the current program part selection. A prominent member of this kind of rules is the rule of consequence[1]:

$$\frac{\mathbf{PP} \Rightarrow \mathbf{P} \qquad \mathcal{A} \mid\!\rhd \{\ \mathbf{P}\ \}\ \mathtt{pp}\ \{\ \mathbf{Q}\ \} \qquad \mathbf{Q} \Rightarrow \mathbf{QQ}}{\mathcal{A} \mid\!\rhd \{\ \mathbf{PP}\ \}\ \mathtt{pp}\ \{\ \mathbf{QQ}\ \}}$$

It plays a special role in the Hoare logic because it additionally requires implications between stronger and weaker conditions to be proven. If a JIVE proof contains an application of the rule of consequence, the implication is attached to the proof tree node that represents this rule application; these attachments are called lemmas. JIVE sends these lemmas to an associated general purpose theorem prover. The user is then required to prove them. Currently, JIVE supports ISABELLE/HOL as associated prover. It is required that all lemmas that are attached to any node of a proof tree are proven before the initial goal of the proof tree is accepted as being proven.

One of JIVE's main strengths is its dedicated user interface. At all times, the user has full visual control over the whole proof process. Applicable proof operations can be selected in menus. Powerful strategies allow for partial proof automation. JIVE features two representation modes for the proofs that are being performed: Tree view and text view. In the tree view, each node of the tree contains a Hoare sequent that is to be proven. The goal of a proof represents the root of a proof tree, and the subgoals that are constructed during the proof are represented as the child nodes of the proof tree. The leaves are either open subgoals which still need to be proven, or subgoals that have been closed (i.e. proven) by the application of an axiom. If a rule or axiom has been applied to a node, the following information is stored in the node: the name of the rule or axiom, any parameters that have been entered by the user, and any lemmas

[1] In JIVE, the rule of consequence is part of a larger rule which serves several purposes at once. Since we want to focus on the rule of consequence, we left out the parts that are irrelevant in this context.

that are required to be proven (currently these are the implications that stem from strengthening or weakening, but other lemmas might be attached by future rules). In the text view, the program has a textual representation, just as in an editor. The pre- and postconditions of the initial goal as well as of the open subgoals are inserted into this text. This way, the proof engineer can concentrate on those program parts that still need to be proven, while the full program context remains in view.

Internally, JIVE is equipped with an encapsulated proof container [Sch04], and only the Hoare rules are allowed to modify the proof state stored in this container. But still there are many opportunities to invalidate the resulting proof, e.g. in case there are bugs in the implementation of the Hoare rules in JIVE. The approach presented in this paper relieves us from the need to formally verify the application in order to achieve trustworthy results.

4 Isabelle/HOL

ISABELLE [Pau94] is an interactive generic theorem prover that can be instantiated with numerous logics. Proof rules are represented as propositions of a so-called meta-logic or logical framework. The object-logics are formalized in this meta-logic. ISABELLE/HOL is the instantiation of ISABELLE with Church's Higher-Order Logic, a classical logic with equality that is based on typed λ calculus. ISABELLE has an LCF-style architecture, which means that the system has a very small kernel, and only this kernel produces valid theorems. The system is being developed since 1986, and since it is in the open source, we believe that it is quite reliable because a number of persons have audited the code so far.

To use ISABELLE/HOL as trusted checker, we enhanced it with a formalization of JIVE's logical foundations. JIVE is based on an object-oriented data and store model [PH97]. The ISABELLE/HOL formalization of this model is presented in [SR05]. In the store model, all types of the program are mapped to one sort in ISABELLE/HOL, called *Value* [PH97, p. 41]. The abstract syntax of the programming language JAVA-KE is formalized as a number of datatypes. This formalization is extended by a term-position algebra over these datatypes that allows to distinguish different program positions, even if the program terms of these positions are identical [PHR04]. The operational semantics and Hoare logic of JAVA-KE are formalized according to Nipkow [Nip02] as inductive sets. For the operational semantics, this set contains all valid combinations of program parts, pre- and poststates $\langle \mathtt{pp}, s_0 \rangle \longrightarrow s_1$. For the Hoare logic, the set contains all valid combinations of assumptions, program parts, pre- and postconditions $\mathcal{A} \models \{ \mathbf{P} \}\ \mathtt{pp}\ \{ \mathbf{Q} \}$.

5 Formal Checking of Jive's Proofs

In this section we describe how JIVE proofs are translated to ISABELLE/HOL proof scripts.

In JIVE, proofs are represented as trees where each proof tree node contains the Hoare sequent which is being proven by the subtree originating from that node, the name of the rule or axiom that was applied to that node, any parameters that have been passed by the user such that the rule could be applied, and any lemmas that are required to be proven. In the translation to an ISABELLE/HOL proof script, it is sufficient to give the initial proof goal as theorem to be proven, and then to apply the Hoare rules to this proof goal in backward direction. During these backward proofs, it is not necessary to specify any intermediate proof goal. Therefore, it suffices to translate the sequent that is contained in JIVE's proof tree root to ISABELLE/HOL. The other sequents are not needed in the ISABELLE/HOL proof script.

In JIVE, Hoare logic rules and axioms can be applied in forward and backward direction. The parameters the user enters if a rule is applied may depend on whether the forward or the backward direction has been chosen. In an ISABELLE/HOL proof script, the Hoare rules are always applied in backward direction. Therefore, the parameters stored in the JIVE proof tree nodes which describe a forward application of a Hoare rule may need to be modified when the rule application is translated to ISABELLE/HOL.

In JIVE, in the Hoare sequents the state is implicitly present. The semantics of a Hoare sequent $\mathcal{A} \models \{ \mathbf{P} \}\ \mathtt{pp}\ \{ \mathbf{Q} \}$ is given by

$$UC(\forall S, SQ : \mathbf{P}[S] \wedge rsem(N, S, \mathtt{pp}, SQ) \Longrightarrow \mathbf{Q}[SQ])$$

where N is a logical variable[2] not occurring free in $\mathbf{P}$ or $\mathbf{Q}$, UC denotes the universal closure over all logical variables except N, $F[S]$ denotes a formula or term in which each occurrence of a program variable v is substituted by the application of the state S to v, and *rsem* is a big-step operational semantics relation where the first parameter N captures the maximal depth of nested method calls that is allowed during execution of the statement (for further details see [PHGR05]). To represent the pre- and postconditions of the Hoare sequents in ISABELLE/HOL, we had to make the treatment of the state and the logical variables explicit. We represent the pre- and postconditions of the Hoare sequents as functional abstractions over the state. Program variables occurring in the conditions are replaced by the application of the state to them. To handle the logical variables, we use an extended state space. The concept of an extended state space was introduced by Apt and Meertens [AM80] and was adapted to Hoare logic by Kleymann [Kle99]. We add a functional abstraction over this extended state space to our condition representation. Again, occurrences of logical variables in the conditions are replaced by applications of the extended state space to them. To give an example, the formula $\mathtt{excV'} = \mathtt{null}\ \ \&\&\ \ b1 = \mathtt{true}$ is represented in Jive as

```
excV' = nullV  &  aB b1 = True
```

while it is transformed to ISABELLE/HOL as

[2] Logical variables are also referred to as auxiliary variables, e.g. by Kleymann [Kle99].

```
\<lambda> S. \<lambda> V. apply_v S excV' = nullV
                       &  aB (apply_l V b1) =  True
```

In these translations, `\<lambda> S` and `\<lambda> V` denote the functional abstractions over the state and the extended state space, respectively. The function `apply_v` applies the state to a program variable, and `apply_l` applies the extended state space to a logical variable. Both functions return an element of the store type *Value*. If this element represents a Java boolean value, it can be mapped to Isabelle/HOL's type *bool* by the abstraction function `aB`. `excV'` denotes a special program variable that captures the current exception object; otherwise it is `null`. Apart from the above-mentioned transformations, the formulae are identical, thus the transformation satisfies the requirement of only marginally changing the proof goal, as required above.

In Jive, the program is internally represented as abstract syntax tree. In the graphical user interface, it is unparsed with Java concrete syntax. In Isabelle/HOL, it is again represented as abstract syntax tree. Jive is designed to eventually handle a richer language, therefore the internal Jive AST is more complex than the Isabelle/HOL AST. This does not change the program since it must currently adhere to the smaller language Java-KE in order to be proven in Jive. A technical detail is that statement sequences are represented differently in the two abstract syntaxes (as a flat list vs. a combed list), but the transformation is straightforward. Therefore, compared to Jive's internal representation, the requirement of only marginally changing the program in the transformation is fulfilled, too.

If a Jive proof tree contains nodes that have lemmas attached to them, Jive requires the user to prove these lemmas in Isabelle/HOL; otherwise, the proof tree is not accepted as proven. These lemma proofs are currently not recorded in Jive. Since it is not possible in general to automatically produce proofs for these lemmas, the lemmas are currently assumed as being valid in the checking process. In the future, the lemma proofs will be recorded for each lemma and stored with it in its proof tree node. Then, the checker can reuse the stored proof. This does not check the correctness of the lemma proof, but this proof has already been performed in Isabelle/HOL, and the goal of the checking procedure described here is to check the results of Jive, not of Isabelle/HOL. Nevertheless, it adds to the trustworthiness of the proof performed in Jive if we repeat the lemma proofs in Isabelle/HOL and if we use these actually proven lemmas in the checking of the Jive proof tree because then we can be sure that Jive handled the lemmas and their proofs correctly and did not introduce errors, e.g. by erroneously recording that a lemma has successfully been proven while in fact it has not.

The proof tree transformer [Sch05] generates a number of Isabelle/HOL theories. The first theory is called Preamble_⟨file⟩.thy. It contains the declarations of all program variables and logical variables that appear in the program. Additionally, all subterms of the program (including the whole program itself) and all program positions of the term-position algebra that has been formalized in Isabelle/HOL [PHR04] are explicitly assigned a name. This way, program

parts and proofs related to them can be built up incrementally, using the program part definitions and proofs of the subterms or parent positions. The theory also contains helper lemmas for the terms and positions, e.g. proofs of definedness. For each proof tree in the container, two theories are generated. The first theory contains the lemmas for the implications. The second theory contains the proof goal and its proof. This proof consists of the Hoare rule applications as stored in the JIVE proof tree. Most of the Hoare rules produce a new sugboal and additional assumptions. The latter are solved by applications of the helper lemmas generated in the preamble theory.

The approach is linear in the number of nodes in the program tree plus the number of nodes in the proof trees. For a given program, each proof tree can have arbitrary size because there are Hoare rules whose assumptions contain the same program part as the conclusion. These rules can be applied an arbitrary number of times (e.g. strengthening, weakening, assumption-introduction). In practice, the number of nodes in a program tree will be dominated by the number of Hoare rule applications that reduce the size of the program part. These rules can only be applied once for each statement in the tree. For each rule application, the number of helper lemma applications is limited by a constant factor. The length of the helper lemma proofs is constant because there exists a helper lemma for each program term or position, and the lemmas of the smaller structures are used in the proofs of the lemmas of the larger structures. Thus, in practice the number of nodes in one proof tree is of the order of the number of nodes in the program tree. Thus we can state that in practice the approach is linear in the number of nodes in the program tree times the number of proof trees.

6 Results

What do we gain from this technique? On the one hand, we are able to verify specified properties of a program by using a special-purpose theorem prover like e.g. JIVE which has been designed to help the user in the verification process by making the proof task easier and more convenient. On the other hand, we get a fully formal proof script for a trusted proof checker like e.g. ISABELLE/HOL. This proof script is checked against a formalization of the Hoare logic which has been formally proven sound in ISABELLE/HOL against the formalization of the underlying operational semantics. This way, we combine convenient proof creation with formally rigorous proof checking.

And what about the details? If we generate an ISABELLE/HOL theory that contains a Hoare logic proof, and if this proof is accepted by ISABELLE/HOL, then we know that the program contained in this theory matches its specification contained in this theory – provided that the formalization of the Hoare Logic rules was correct (which can be verified by a correctness proof), and provided that ISABELLE/HOL is implemented correctly.

If we are very precise, we note that even if we assume that our checker is correct, we do *not* know that our unverified prover is correct because we only check the prover's results, not the tool itself. Furthermore, we do *not* know that

the proof that was created in the unverified prover is valid, because the prover might be implemented incorrectly, and the check theory might be generated incorrectly, in such a way that the faulty generator produces a correct proof for the checker from the incorrect proof performed in the unverified prover. But we *do* know that there exists a valid proof of the proof goal contained in the generated theory, which is finally what we want to know.

7 Conclusions

In this paper, we presented a method to increase the confidence in results provided by unverified theorem provers. The method is derived from runtime result checking. It implies creating a proof with an unverified prover and checking this proof in a trusted proof checker. This approach renders the requirement of verifying the untrusted theorem prover unnecessary. We applied this approach to the JIVE tool as unverified theorem prover and used ISABELLE/HOL as trusted proof checker.

References

[ACL03] J. Andronick, B. Chetali, and O. Ly. Using coq to verify java card applet isolation properties. In *16th Conference on Theorem Proving in Higher Order Logics (TPHOLs 2003)*, volume 2758 of *Lecture Notes in Computer Science*, pages 335–351. Springer-Verlag, 2003.

[AM80] Krzysztof R. Apt and Lambert Meertens. Completeness with finite systems of intermediate assertions for recursive program schemes. *SIAM Journal on Computing*, 9:665–671, 1980.

[BRL03] Lilian Burdy, Antoine Requet, and Jean-Louis Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods Europe*, volume 2805 of *Lecture Notes in Computer Science*, pages 422–439. Springer-Verlag, 2003.

[GGZ98] Wolfgang Goerigk, Thilo Gaul, and Wolf Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In R. Berghammer and Y. Lakhnech, editors, *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, pages 108 – 122, Wien, New York, 1998. Springer Verlag.

[Kle99] Thomas Kleymann. Hoare logic and auxiliary variables. *Formal Aspects of Computing*, 11(5):541–566, 1999. Extended version of Schreiber, Auxiliary Variables and Recursive Procedures.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, chapter 12, pages 175–188. Kluwer, 1999.

[MPH00] Jörg Meyer and Arnd Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 1785 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2000.

[Nip02] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[Pau94] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, NY, USA, 1994.

[PH97] Arnd Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitiationsschrift, Technische Universität München, 1997.

[PHGR05] Arnd Poetzsch-Heffter, Jean-Marie Gaillourdet, and Nicole Rauch. A Hoare Logic for a Java Subset and its Proof of Soundness and Completeness. Internal report, University of Kaiserslautern, Germany, 2005. To appear.

[PHR04] Arnd Poetzsch-Heffter and Nicole Rauch. Application and formal specification of sorted term-position algebras. In José L. Fiadeiro, editor, *17th International Workshop on Recent Trends in Algebraic Development Techniques, WADT 2004, Barcelona, Spain*, volume 3423 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, March 2004.

[Sch04] Jan Schäfer. Encapsulation and specification of object-oriented runtime components. Master's thesis, University of Kaiserslautern, September 2004.

[Sch05] Christian Schmidt. Automatische Validierung von Hoare-Logik Beweisen durch Transformation in Isabelle/HOL Beweisskripte. Master's thesis, University of Kaiserslautern, May 2005.

[SR05] Norbert Schirmer and Nicole Rauch. Jive data and store model. Internal report, University of Kaiserslautern, 2005. To appear.

[WB97] Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

Temporale Assertions mit AspectJ

Volker Stolz und Eric Bodden

Lehrstuhl für Informatik II, RWTH Aachen, 52056 Aachen

Zusammenfassung In diesem Artikel stellen wir ein *Runtime Verification Framework* für Java-Programme vor. Eigenschaften können in *Linear Time Logic* (LTL) über AspectJ-Pointcuts spezifiziert werden. Diese Eigenschaften werden zur Laufzeit durch einen automatenbasierten Ansatz überprüft, in welchem Zustandsübergänge durch Aspekte ausgelöst werden. Unser Ansatz benötigt nicht notwendigerweise den Java-Quelltext der zu instrumentierenden Anwendung, da AspectJ auf dem Bytecode arbeitet und somit auch Anwendungen Dritter instrumentiert werden können.

1 Einführung

Viele Softwareprodukte enthalten sog. *Assertions* (Zusicherungen), welche an bestimmten Punkten in der Programmausführung Bedingungen testen und im Fehlerfall entweder die Ausführung abbrechen oder eine Fehlerbehandlung durchführen. Diese Assertions beschränken sich normalerweise auf das Testen von Boolschen Ausdrücken. Oft wäre es jedoch praktisch, nicht nur über einen einzelnen Zustand argumentieren zu können, sondern über eine Folge von Zuständen. Dies erlaubt dem Entwickler, Assertions über den Kontrollfluß und Ausführungspfade zu formulieren.

In unserer vorherigen Arbeit [6] haben wir einen symbolischen Checker für parametrisierte LTL-Formeln über endlichen Pfaden entwickelt, mit dem z.B. ein in nebenläufigen Programmen häufiges Problem, das sog. *lock-order reversal*, untersucht werden kann. Zur Laufzeit wird der Entwickler über (möglicherweise) verletzte Assertions benachrichtig. Am Lehrstuhl wurde ein funktionsfähiger Prototyp mit ähnlicher Funktionalität für Java-Anwendungen entwickelt. Die Hauptneuheit ist, das wir eine alternative, effizientere Implementierung basierend auf alternierenden Automaten benutzen [5].

Der Nachteil der vorherigen Version ist, dass die den Checker steuernden Annotationen in den Quelltext der zu instrumentierenden Anwendung eingefügt werden müssen. Für die Standardbibliotheken stellt das Werkzeug annotierte Versionen bereit. Anwendungen müssen also neu übersetzt werden, um untersucht werden zu können.

In objektorientierten Programmiersprachen haben Quelltext-basierte Instrumentierungsansätze außerdem das Problem, daß eine für eine Schnittstelle spezifizierte Eigenschaft auch für Unterklassen gelten soll. Unser Ansatz wird diesem Umstand gerecht, indem Spezifikationen für eine bestimmte Klasse automatisch auch für Unterklassen überprüft werden. Entsprechendes gilt für Interfaces.

Der Erfolg anderer Test-Werkzeuge wie etwa Valgrind [4] zeigt, das es deutlichen Bedarf an besserer Unterstützung beim Debugging gibt, im Gegensatz zu „Offline“-Analysen wie Model Checking. Zwar wurde vor kurzem der Java Pathfinder [7] freigeben, der von den Autoren der NASA erfolgreich eingesetzt wurde, jedoch muß sich erst noch zeigen, inwiefern er sonstigen Einsatz findet.

2 Instrumentierung

Aspekt-orientierte Programmierung trennt Klassen-übergreifende Belange so auf, das sie nicht an über den Quelltext verteilten Stellen behandelt werden, sondern in einem separaten Modul gewartet werden können und dann mit dem Programm an den entsprechenden Stellen „verwoben“ werden. Die zugrundeliegende Programmiersprache wird dabei durch eine Aspekt-orientiere Programmiersprache erweitert, so daß durch deklarative Konstrukte beschrieben wird, welche Programmfragmente zur Laufzeit wann ausgeführt werden sollen.

Heutzutage ist die Erweiterung von Java zu AspectJ [2] die am weitesten verbreitete Aspekt-orientierte Programmiersprache. Ursprünglich in den späten 90er Jahren in Xerox PARC entwickelt, tragen auch heute noch Firmen und Wissenschaftler zur Weiterentwicklung bei. AspectJ ist auch Bestandteil großer Produkionssysteme wie beispielsweise von IBM Websphere.

Der dynamische Kontrollfluß einer Anwendung zur Laufzeit wird durch eine Menge von sog. *Joinpoints* gebildet. Ein *Aspekt* wird aus *Pointcuts* modelliert, welche man als Prädikate mit Variablen über Joinpoints sehen kann. Folgende Konstrukte und ihre Boolschen Verknüpfungen stehen zur Verfügung (weitere Pointcuts adressieren beispielsweise Exceptions, sind für unser Werkzeug aber nicht von Interesse):

Pointcut...	adressiert...
`execution(`*MethodSignature*`)`	Ausführung einer Methode
`call(`*MethodSignature*`)`	Aufruf einer Methode
`set/get(`*FieldSignature*`)`	Attributzugriff
`cflow(`*Pointcut*`)`	Kontrollfluß innerhalb anderen Pointcuts
`if(`*BooleanExpression*`)`	bei erfüllter Bedingung; der Ausdruck hat Zugriff auf statische Java-Objekte und durch den Pointcut gebundene Variablen

Mit Pointcuts ist es möglich, Punkte im dynamischen Kontrollfluß herauszugreifen. Wir setzen sie als Propositionen unserer Logik ein: Ein Pointcut gilt an einem gegebenen Punkt genau dann wenn er den aktuellen Joinpoint matcht.

Platzhalter (*wildcards*) in Pointcuts können dazu benutzt werden, um von bestimmten Teilen der zu matchenden Signatur zu abstrahieren. Damit ist es möglich, Mengen von Joinpoints aufzusammeln, während die Anwendung läuft. Um nun das Verhalten einer Anwendung zu verändern, wird ein *Advice* mit solch einem Pointcut verknüpft. Immer wenn der Pointcut einen Joinpoint im dynamischen Kontrollfluß matcht, wird der Advice zur angegebenen Zeit ausgeführt:

```
pointcut auth(User u):
  call(* Authentication.login(User)) && args(u);

after returning(): auth(User user) {
  SecurityLog.log("User " + user.getId() + " logged in");
}
```

Abbildung 1. AspectJ Pointcut und Advice zur Protokollierung

Dies kann *vor*, *nach* oder *statt* dem Joinpoint sein. Abb. 1 zeigt ein Beispiel, welches Authentifizierungen protokolliert.

Die in dem Beispiel zusätzlich verwendeten Pointcuts `this`, `target` und `args` ermöglichen dem Advice Zugriff auf den aktuellen Zustand. Dies kann auch in unserem Ansatz verwendet werden. Desweiteren sei noch angemerkt, daß das Weben der Aspekte und des Advice normalerweise zur Compilezeit erfolgt.

3 LTL und der Zustandsraum eines Java-Programmes

Ein Lauf eines Java-Programmes ergibt einen Pfad, in dem jeder Schritt einem Zustand der virtuellen Maschine entspricht. Diese Zustand beeinhaltet z.B. den Befehlszähler, den aktuellen Stack und Heap (für eine detaillierte Beschreibung s.h. [3]). Jede Bytecode-Instruktion (Attributzugriff, Methodenaufruf etc.) bewirkt einen Zustandsübergang.

Dieses Modell ist zu fein-granular, beispielsweise ist es nicht nötig, über den tatsächlichen Wert des Befehlszählers Aussagen zu machen. Es reicht aus, wenn wir uns auf die durch AspectJ-Pointcuts selektierbaren Zustände beschränken. Damit können durch Boolsche Kombination von Pointcuts auch nicht zusammenhängende Zustandsmengen beschrieben werden. Im Allgemeinen gibt es keinen direkten Zusammenhang zwischen Joinpoints und sowohl Bytecode-Instruktionen als auch Quelltext-Positionen, was jedoch für unsere Argumentation keine Rolle spielt.

Betrachten wir folgendes Beispiel einer temporalen Eigenschaft: Jeder Aufruf einer Methode `C.f()` soll irgendwann von einem Aufruf von `C.g()` gefolgt werden. Als LTL-Formel ausgedrückt:

$$\mathbf{G}\ (\texttt{call}(\texttt{C.f}()) \longrightarrow \mathbf{F}\ \texttt{call}(\texttt{C.g}())\ )$$

Das äußere *Globally* sorgt dafür, daß die Formel auch bei jedem Aufruf von `C.f()` überprüft wird, da sonst nur der Pfad zutreffen würde, in dem der Methodenaufruf im ersten Schritt erfolgt.

4 Implementierung

Wir benutzen den *abc*-Compiler [1] zum Parsen der Formeln. Der abstrakte Syntaxbaum wird dann auf Syntax- und Typfehler geprüft. Die Konvertierung der Formel in einen Aspekt geht wie folgt von statten:

1. Eine Variable enthält eine Referenz auf den aktuellen Zustand des sich aus der LTL-Formel ergebenden Automaten [5]. Sie wird mit dem Anfangszustand initialisiert.
2. Eine Tabelle wird aufgestellt, welche $n + 1$ Advice für n in der Formel vorkommende verschiedene Propositionen (Pointcuts) enthält. Insbesondere sind dies:
   - Ein Advice für jede Pointcut-Proposition, welcher registriert, dass der Pointcut matcht, also die Proposition im aktuellen Zustand gilt; und
   - ein Advice, welcher alle obigen Propositionen matcht und die Zustandstransition in den neuen Zustand mit den akkumulierten Propositionen auslöst.

Dabei ist zu beachten, daß Advice in der Reihenfolge des Vorkommens im Aspekt ausgeführt wird, d.h. der letzte Advice, der die Transition auslöst, wird auch tatsächlich nach allen anderen ausgeführt.

Der generierte Aspekt wird dann mit der Anwendung verwoben. Abb. 2 gibt einen Überblickt über die verschiedenen Bearbeitungsschritte.

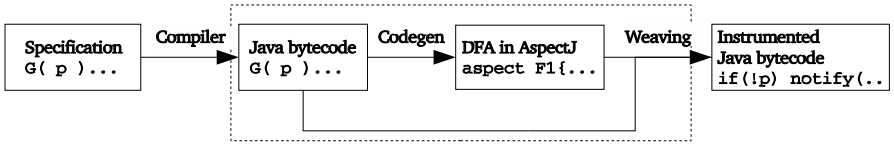


Abbildung 2. Instrumentierungsschritte

4.1 Ausführung und Overhead

Damit die instrumentierte Anwendung überprüft werden kann, muss sie lediglich ausgeführt werden. Sollte eine Formel verletzt werden, gibt das System eine entsprechende Meldung aus. Es werden natürlich nur Programmpfade überprüft, die auch tatsächlich durchlaufen werden. Deshalb ist es notwendig, diese Technik mit anderen zu kombinieren, die während des Testens eine Abdeckung möglichst vieler Pfade gewährleistet.

Jede Überprüfung zur Laufzeit (*runtime verification*) bedeutet notwendigerweise einen gewissen Overhead für die instrumentierte Anwendung, wenn eine aktive Proposition „gefeuert“ wird. Dieser zusätzliche Aufwand ist linear zur Anzahl der verwendeten Propositionen. Das Berechnen der Nachfolger eines Zustands ist ungleich aufwändiger, außerdem kann jeder Zustand *mehrere* Nachfolger haben.

5 Zusammenfassung

Wir haben ein Werkzeug vorgestellt, welches LTL-Formeln über AspecJ-Pointcuts als Propositionen zur Laufzeit mit Hilfe alternierender Automaten überprüft.

Aus einer Formel wird ein Aspekt generiert und mit der Anwendung verwoben. Sollte eine Spezifikation zur Laufzeit verletzt werden, wird eine entsprechende Meldung generiert. Der Prototyp mit dem Namen Java Logical Observer (JLO) und weitere Informationen sind unter http://www-i2.informatik.rwth-aachen.de/JLO/ zu finden.

Literatur

1. P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *AOSD'05: Proceedings of the Fourth international conference on Aspect-oriented software development*. ACM Press, 2005.
2. R. Laddad. *AspectJ in Action*. Manning Publications, 2003.
3. T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1997.
4. N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. In *Third Workshop on Runtime Verification (RV'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2003.
5. V. Stolz and E. Bodden. Temporal Assertions using AspectJ. In *Proc. of the Fifth Workshop on Runtime Verification (RV'05), to be published in ENTCS*, Edinburgh, UK, 2005.
6. V. Stolz and F. Huch. Runtime Verification of Concurrent Haskell Programms. In K. Havelund and G. Roşu, editors, *Fourth Workshop on Runtime Verification (RV'04)*, volume 113 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2004.
7. W.Visser, K.Havelund, G.Brat, and S.Park. Model Checking Programs. In *Proc. of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000.

Spezifikation und Verifikation in regelbasierten Beratungssystemen auf der Grundlage Hybrider Automaten

Elke Tetzner and Günter Riedewald

Universität Rostock, IEF
D-18051 Rostock
(tetzner|gri)@informatik.uni-rostock.de,
http://www.informatik.uni-rostock.de/~tetzner

Zusammenfassung. Technische und verwaltungstechnische Prozesse, in denen Ereignisse im Zusammenhang mit erwünschten bzw. kritischen Zeitbedingungen ihres Auftretens betrachtet werden, können problemadäquat durch hybride Systeme auf der Grundlage hybrider Automaten modelliert werden. Hierbei entspricht die analoge Größe der Zeit einer rein kontinuierlichen Komponente, welche ohne künstliche Diskretisierung mit dem durch die Ereignisse beschriebenen diskreten Ablauf verbunden werden kann. Hybride Automaten ermöglichen die Untersuchung von Prozessen aus sprachtheoretischer Sicht und den Nachweis von Eigenschaften auf der Basis von Zeitwörtern, einer Folge von Tupeln, welche aus Ereignissen und der Zeit des Auftretens dieser Ereignisse bestehen. Aus der Akzeptanz bzw. Nichtakzeptanz solcher Zeitwörter durch hybride Automaten werden Schlussfolgerungen bezüglich zeitkritischer Eigenschaften der Systeme gezogen. Gegenstand des Artikels sind Methoden einer nutzerfreundlichen Spezifikation hybrider Systeme zur Modellierung regelbasierter Anwendungen mit beratender Funktion sowie Strategien zum Aufbau, zur Erweiterung und Änderung der hybriden Systeme. Dabei werden Konzepte der Wiederverwendung und Verfeinerung als auch besondere Merkmale der Spezifikation im Bereich regelbasierter Beratungssysteme herausgearbeitet. Eine für den Bereich regelbasierter Beratungssysteme typische Klassifikation von Eigenschaften und Ansätze zur Formalisierung bilden den Abschluss der Arbeit.

1 Einleitung

Hybride Systeme [ACHH93] auf der Grundlage hybrider Automaten [Hen96] haben sich als problemadäquate Modelle für automatisierte technische Systeme, die stark in einer sich kontinuierlich verändernden Umwelt eingebettet sind, erwiesen [TKRE00]. Wie in der Arbeit gezeigt wird, stellen hybride Systeme auch für automatisierte verwaltungstechnische Abläufe, welche entsprechend eines Regelwerkes mit zusätzlichen Plänen und Festlegungen unter Beachtung zeitlicher Bedingungen ausgeführt werden, eine dem Problem sehr naheliegende Lösung zur Modellierung dar. Zeitliche Einschränkungen, welche in Regelwerken wie Ordnungen, Verordnungen bzw. Gesetzen auftreten, werden als absolute Zeitpunkte oder als in sich geschlossene, kontinuierliche Zeitabschnitte defi niert. Hierbei werden Informationen wie Pausenzeiten oder Nachtstunden, die für das menschliche Zusammenleben selbstverständlich sind, vernachlässigt und die künstliche Diskretisierung durch eine tageweise Betrachtung vermieden. Die adäquate Modellierung von Prozessen, die Abläufe bezüglich der genannten Regelwerke und zugehörigen Plänen und Festlegungen zulassen, muss die Zeit als rein kontinuierliche Komponente und die von der Zeit abstrahierten Abläufe als diskrete Komponenten behandeln können. Hybride Automaten bieten die Möglichkeit, die Zeit mit Lokationen zur Darstellung kontinuierlicher Verläufe und die von der Zeit abstrahierten Abläufe mit Transitionen zur Darstellung diskreter Ereignisse ohne Zeitverzug zu beschreiben.
Soll ein automatisiertes verwaltungstechnisches System, wie zum Beipiel ein Studiensystem [STG01], eine beratende Funktion zu Fragen bestehender Abläufe, möglicher Fortsetzungen von Abläufen, der Erstellung und Übereinstimmung von Plänen und Regelwerken ausführen, so besteht die Notwendigkeit des Nachweises zeitkritischer Eigenschaften. Durch hybride Automaten können zeitkritische Eigenschaften auf der Basis von Zeitwörtern mit Hilfe der symbolischen Simulation [Rie95,RU95] nachgewiesen werden. Zeitwörter beschreiben eine Folge von Tupeln, wobei die Tupel aus Ereignissen und der Zeit des Auftretens dieser Ereignisse bestehen. Die Menge aller verwaltungstechnischen Abläufe, die als Zeitwörter von einem hybriden Automaten akzeptiert werden, bilden eine formale Sprache, die das mögliche Verhalten eines verwaltungstechnischen Systems widerspiegelt. Aus der Akzeptanz bzw. Nichtakzeptanz von Abläufen als Zeitwörter lassen sich Aussagen zu erwünschten und kritischen Eigenschaften des modellierten Systems ableiten. Werden die Zeiten des Auftretens der Ereignisse in Abläufen durch Variablen beschrieben, die eine unendliche Menge von Werten eines Zeitintervalls repräsentieren, so entsprechen die Abläufe symbolischen Abläufen, die unendlich viele konkrete Abläufe zusammenfassen. Symbolische Abläufe werden während der symbolischen Simulation ausgeführt. Die symbolische Simulation ist eine Art

der automatischen Verifikation, welche wie das Model Checking [CGP99] die Frage beantwortet, ob ein gegebenes zustandsbasiertes System eine in einem logischen Formalismus spezifizierte Eigenschaft erfüllt. Die symbolische Simulation wird wie das Model Checking in unterschiedlichen Bereichen und mit unterschiedlichen Mitteln ausgeführt. Während [Rie95,RU95] das zustandsbasierte System auf der Grundlage hybrider Automaten beschreiben, werden in [BSW02] kontinuierliche Aktionssysteme zur Beschreibung des zustandbasierten Modelles verwendet. [LC05] dagegen haben Zeiglers DEVS [Zei76] um symbolische Variablen erweitert und somit eine Möglichkeit zur symbolischen Simulation geschaffen.
Um zu zeigen, wie die symbolische Simulation im Bereich regelbasierter Beratungssysteme eingesetzt werden kann, wird in Kapitel 2 der Begriff des regelbasierten Beratungssystems und die Modellierung mittels hybrider Systeme, welche durch unsere nutzerfreundlichen Sprachen MODEL-HS [TR99,Sik05] und VYSMO [TBR01] unterstützt wird, genauer betrachtet. Im Kapitel 3 werden Konzepte der Abstraktion und Verfeinerung, ähnlich [Hen00,AGH$^+$00,AGLS01], im Zusammenhang mit besonderen Merkmalen der Spezifikation hybrider Systeme im Bereich der regelbasierten Beratungssysteme behandelt. Die Klassifikation von Eigenschaften und deren Formalisierung ist Thema des Kapitels 4. Abschließend erfolgt eine Zusammenfassung und ein Ausblick auf weiterführende Arbeiten.

2 Hybride Systeme als Modell regelbasierter Beratungssysteme

Im Mittelpunkt unserer Untersuchungen stehen Abläufe, welche eine Folge von Ereignissen verbunden mit der Zeit ihres Auftretens bilden. Die Ereignisse und zugehörige Zeiten sind entweder durch Daten und Dokumente manuell belegt oder über einen Regelmechanismus automatisch erzeugt. Auf diese Art und Weise ist jeder Ablauf in einen Kontext eingebettet.

Begriff 21 Ein *Kontext* K zu einem Ablauf ρ ist ein 2-Tupel $\langle P,\ \omega\rangle$, wobei:

- P das Regelwerk als eine Menge von Produktionsregeln gültiger abstrakter Abläufe darstellt und
- ω eine Menge von aktuellen Daten und Dokumenten, wodurch Instanzen der abstrakten Abläufe abgeleitet werden, repräsentiert.

Das Regelwerk bildet theoretisch eine Zeitgrammatik [RU95], in der die Produktionsregeln diskrete Übergänge, welche die Ereignisse eines Ablaufes und die Zeit des Auftretens der Ereignisse widerspiegeln, mit einem vorausgehenden, zeitlich kontinuierlichen Fortschritt kombinieren. Praktisch basiert das Regelwerk auf einer Hierarchie von Regeln. Die oberste Stufe beinhaltet die Regeln der Rahmen- bzw. Metaordnungen, die mittlere Stufe die Regeln der Ordnungen sowie Verordnungen und die unterste Stufe die Regeln der Durch- bzw. Ausführungsbestimmungen. Sollen diese Regeln in einem konkreten Fall angewendet werden, so müssen weiterhin Daten und Dokumente ω wie aktuelle Pläne und Termine, individuelle Durch- und Ausführungsentscheidungen als auch IST- und SOLL-Abläufe berücksichtigt werden.

Beispiel 21 *Ein Ablauf für das Erbringen eines Leistungsnachweises 'LN' unter der Voraussetzung 'VS' im Fach 'Mathematik' kann in folgendem Kontext ausgeführt werden:*
1. Regeln der Studien- und Prüfungsordnung:
<Start> $\rightarrow$T$_k$, $T_k.t \leq 24$, <VS>,<Rest>, *Rest.t:= VS.t ;* <Rest> $\rightarrow$T$_k$, $T_k.t \leq 24$ *-Rest.t,* $\langle$LN.b, T$_{LNb}\rangle$
2. Regeln der Durchführungsordnung für die Voraussetzung als erfüllte Hausaufgaben 'HS' oder bestandenes Kurztestat (KT):
<VS> $\rightarrow$ $\langle$HS.b,T$_{HA}\rangle$, *VS.t :=* $T_{HA}.t$; <VS> $\rightarrow$ $\langle$KT.b,T$_{KT}\rangle$, *VS.t :=*$T_{KT}.t$
3. Zeitraum und Art des Ablegens der Voraussetzung, die durch den Dozent festgelegt wurden:
Kurztestat im 11. oder 12. Monat zum Ende des 1. Studienjahres
Die angegebene Grammatik ist eine Zeitgrammatik, deren Regeln auf dem Wechsel von kontinuierlichen und diskreten Anteilen im Zusammenhang mit Attributierungsregeln über einem Zeitwert 't' basieren. T$_k$ *stellt einen kontinuierlichen Zeitraum dar, welcher in der Startregel kleiner oder gleich 24 Monaten (4 Semester) sein muss. Nach einer bestandenen Voraussetzung ist in der Restfolge der Leistungsnachweis abzulegen. Aus Gründen der Vereinfachung werden nur bestandene Leistungsnachweise erzeugt. Entsprechend 'Rest.t:= VS.t' und '*T_k*.t < 24 -Rest.t' kann der Leistungsnachweis kann nur noch in der verbleibenden Zeit vom Bestehen der Voraussetzung bis zum Ende des 24. Monats abgelegt werden. Die Voraussetzung kann als erfüllte Hausaufgaben zu einem Zeitpunkt '*T_{HA}*' oder als Kurztestat zu einem Zeitpunkt '*T_{KT}*' bestanden werden. Aus der Grammatik lassen sich 2 abstrakte Abläufe ableiten: 1) Erst sind der Hausaufgaben innerhalb von 24 Monaten zu erfüllen und im verbleibenden Ablauf bis zu vollen 24 Monaten ist der Leistungsnachweis abzulegen oder 2) Erst ist das Kurztestat innerhalb von 24*

Monaten zu erfüllen und im verbleibenden Ablauf bis zu vollen 24 Monaten ist der Leistungsnachweis abzulegen. Der Dozent schränkt mit der gestellten Forderung die Ablaufmenge auf den 2. Ablauf ein und instantiiert Ablauf 2 durch die Bedingung, dass der Abschluss des Kurztestates zwischen dem 11. und 12. Monat vorgenommen werden muss.

In der Arbeit dienen die regelbasierten Systeme dem Zweck einer beratenden Tätigkeit. Auf der Grundlage der oben aufgeführten Regeln, aktuellen Daten und Dokumente sollen automatisch Empfehlungen erarbeitet werden, die der Unterstützung von Benutzern bei der Durchführung von Abläufen, der Sicherung von Transparenz und der Absicherung von Garantien dienen. Für ein Studiensystem können dabei zum Beispiel folgende Aufgaben gelöst werden:

a) Berechnen und Anzeigen der IST-Studienabläufe von Studierenden,
b) Berechnen möglicher Fortsetzungen von Abläufen bezüglich gegebener Studien- und Prüfungsordnungen,
c) Unterstützung bei der Erarbeitung von Lehrplänen und Prüfungsterminen so, dass Abläufe in festgelegten Zeiträumen durchführbar sind,
d) Beurteilung individueller Durchführungsentscheidungen von Dozenten sowie Vergleich von IST- und SOLL-Abläufen,
e) Überprüfung der Konsistenz in Studien- und Prüfungsordnungen sowie Machbarkeit von Lehrplänen und Prüfungsterminen in Bezug auf gegebene Studien- und Prüfungsordnungen, wodurch Garantien für sicherheits- und zeitkritische Eigenschaften, wie z.B. Einhaltung von Regelstudienzeiten, gegeben werden können.

Anwendungen regelbasierter Beratungssysteme bestehen in den Bereichen der Aus- und Weiterbildung, der Verwaltung, des Rechtswesens, der Polizei und zeitlich gebundener Abläufe der Industrie, welche aus einer Vertragsbindung entstehen. Im folgenden Verlauf werden alle Erkenntnisse am Beispiel eines Studiensystems dargelegt. Ein Studiensystem besitzt verschiedene Nutzergruppen: i) die Verwaltung, ii) die Dozenten und iii) die Studierenden, welche unterschiedliche Ziele verfolgen und auf verschiedene Dokumente Zugriff besitzen. Die Verwaltung nutzt Studien-, Prüfungs- und Durchführungsordnungen und erarbeitet Lehrpläne und Prüfungstermine. Die Dozenten beschäftigen sich mit Lehrplänen und Prüfungsterminen und gestalten individuell die Durchführung von Lehrveranstaltungen. Die Studierenden sind an bereits vorliegenden Leistungen und an der möglichen Fortsetzung der Studienabläufe unter gegebenen Bedingungen interessiert. Zur Unterstützung der genannten Aufgaben durch automatisch erzeugte Empfehlungen müssen die Dokumente und Informationen in einer formalen Notation vorliegen. Da Studien-, Prüfungs- und Durchführungsordnung formalisiert einer Zeitgrammatik entsprechen, können die Regeln dieser Ordnungen äquivalent mit Hilfe von Automaten beschrieben werden. Die zu einer Zeitgrammatik äquivalenten Automaten sind hybride Automaten, die durch sequentielle und parallele Komposition [AKY99] hybride Systeme bilden. Lehrpläne und Prüfungstermine liegen als Daten in einer Datenbank vor. Individuelle Durchführungsentscheidungen der Dozenten sind Regeln, welche als Verfeinerungen hybrider Automaten im Sinne der sequentiellen Komposition spezifi ziert werden. Die Gesamtheit der Leistungen eines Studierenden mit den zugehörigen Zeitpunkten des Ablegens dieser Leistungen stellen den Studienablauf des Studierenden dar. Die Leistung basiert dem Namen der Lehrveranstaltung, der Art des Leistungsnachweises (Prüfung, Beleg, Praktikum), der Anzahl an Versuchen und dem Erfolg des Ablegens.
Die hybriden Automaten eines Studiensystems müssen in der Lage sein, eine Folge von Leistungen im Zusammenhang mit dem zeitlichen Auftreten der Leistungen zu akzeptieren. Wird von der Zeit abstrahiert, so kann ein Prüfungsvorgang, der zum Ablegen einer Leistung führt, wie in Abbildung 1 mit einem endlichen Automaten beschrieben werden. Der endliche Automat zeigt den Ablauf von Prüfungsphasen als Zustände im Wechsel mit dem Ablegen von Leistungen als Übergänge für das Fach Allgemeinmedizin aus dem 2. Abschnitt eines Medizinstudiums. Hier wird am Anfang ein Zustand für einen Zeitraum reserviert, in welchem von einem Dozenten festgelegte Voraussetzungen für die Zulassung zur eigentlichen Prüfung zu erfüllen sind.
Zur Verbindung des durch den endlichen Automaten beschriebenen Prüfungsablaufes mit der Zeit muss geklärt werden, in welchen Zeiträumen, modelliert durch Lokationen eines hybriden Automaten, Leistungen laut Studien- und Prüfungsordnung abgelegt werden *müssen* und zu welchen Zeitpunkten, modelliert durch Übergänge, Leistungen laut festgelegten Prüfungszeiträumen abgelegt werden *können*. Entsprechend der Abbildung 2 kann ein erfolgreicher Abschluss des Faches Allgemeinmedizin in der Zeit von Beginn des 46. Monats bis zum Ende des 72. Monats vorgewiesen werden. Durch Zusatzbedingungen wie i) das vorlesungsbegleitende Ablegen von Prüfungen, ii) Zeitbegrenzungen von Wiederholungen auf höchstens ein Jahr und iii) 2 Wiederholungen als maximale Anzahl an Wiederholungen, wobei die 2. Wiederholung nur nach einer Genehmigung erfolgen darf, lässt sich das Zeitintervall in mehrere kontinuierliche Abschnitte unterteilen. Diese Unterteilung führt zu den einzelnen Zeiträumen zwischen dem Ablegen und Zeitpunkten des Ablegens von Leistungen, die sich mit den Lokationen und Übergängen eines hybriden Automaten wie in Abbildung 3 ausdrücken lassen.

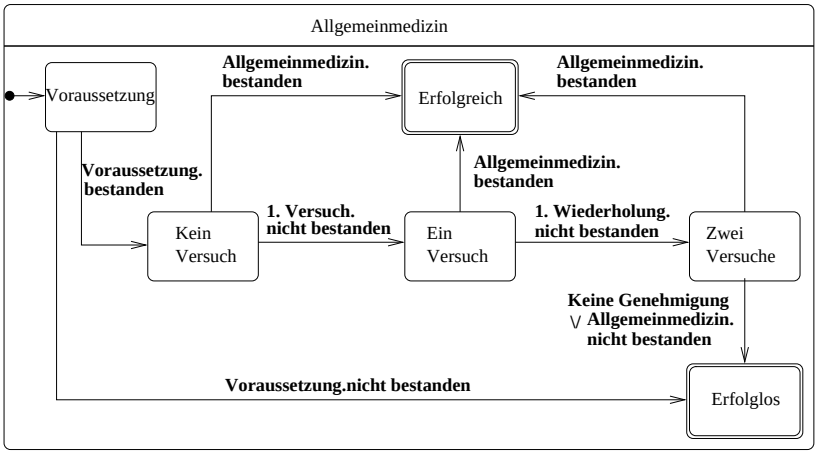


Abb. 1. Endlicher Automat für den Prüfungsvorgang des Faches Allgemeinmedizin

Der hybride Automat wurde in unserer visuellen Notation VYSMO spezifiziert. MODEL-HS existiert als äquivalente textuelle Notation zu VYSMO, so dass der Nutzer eine entsprechend gegebenen Fähigkeiten und Ressourcen angepasste Spezifikation erstellen kann. Aus Platzgründen wird hier auf die textuelle Darstellung in MODEL-HS verzichtet. Wie beim endlichen Automaten ist im Kopfteil des hybriden Automaten der Name des Faches 'Allgemeinmedizin' angegeben. Im darunterliegenden Abschnitt werden Parameter spezifiziert, welche für den durch den hybriden Automaten beschriebenen Vorgang in der Umgebung festgelegt worden sind und sich während eines Ablaufes nicht ändern. Das Fach 'Allgemeinmedizin' besitzt keinen solchen Parameter. Der 3. Abschnitt dient der Deklaration von Variablen. Die Uhrenvariable 'X' ist für die 'Allgemeinmedizin' mit dem Schlüsselwort 'Clock' gekennzeichnet. Im 4. Abschnitt wird der Vorgang als Verhalten des Automaten beschrieben. Die Anfangslokation, hier 'Voraussetzung', ist durch einen auf diese Lokation zeigenden Pfeil gekennzeichnet, der von einem schwarz gefüllten Kreis ausgeht, einer Wartelokation. Die Wartelokation ist nur dann von Bedeutung, wenn der Automat, ähnlich kommunizierenden Automaten in UML, durch Synchronisationsbedingungen später als zu Beginn der globalen Laufzeit des gesamten Systems aktiviert werden soll. Jede Lokation ist mit dem Namen, einer Invariante und einer Aktivität gekennzeichnet. Die Invariante gibt an, in welchen Wertgrenzen der mit der Lokation verbundenen kontinuierlichen Variablen das Verbleiben in der Lokation möglich ist. Nur in den angegebenen Wertgrenzen darf die Lokation betreten werden bzw. muss verlassen werden, wenn die kontinuierlichen Variablen die Grenzen unter- oder überschritten haben. Die Aktivitäten beschreiben als Differentialgleichungen den Fortschritt der kontinuierlichen Grössen je Zeiteinheit. Zum Beispiel muss die Lokation 'Voraussetzung' entsprechend der gegebenen Invariante '$0 \leq X < 46$' vor dem Beginn des 46. Monats verlassen werden. Auch in der Lokation 'Kein Versuch' ist die Invariante mit der unteren Grenze '0' angegeben. Ist die Menge der Voraussetzungen leer, so wird die Lokation 'Voraussetzung' ohne Zeitverzug mit der Leistung 'Voraussetzung.bestanden' zur Lokation 'Kein Versuch' verlassen. Da die Zeiteinheit in Monaten festgelegt ist, wird der Wert der Uhr 'X' laut der Aktivitäten '$\dot{X} = 1$' pro Monat um 1 erhöht und beinhaltet somit die Anzahl der verstrichenen Monate.

Im Gegensatz zum endlichen Automaten bestehen beim hybriden Automaten im Zusammenhang mit der Zeit 3 statt vorher 2 Übergänge von einer Lokation zu folgenden Lokationen. Der 3. Übergang resultiert aus der Unterscheidung von Ereignissen, die durch die manuelle Bereitstellung der Daten und Dokumente in einem gegebenen Zeitraum erzeugt wurden und Ereignissen, die automatisch bei Erreichen eines Zeitpunktes eintreten. Die manuell erzeugten Ereignisse werden als empfangene Signale, gekennzeichnet durch einen nach unten gerichteten Pfeil, und die automatisch auftretenden Ereignisse durch gesendete Signale, gekennzeichnet durch einen nach oben gerichteten Pfeil, symbolisiert. In der Lokation 'Voraussetzung' kann einer der 3 Übergänge zu den Lokationen i) 'Kein Versuch', wenn die Voraussetzung innerhalb des Zeitraumes bis zu Beginn des 46. Monats bestanden wurde, ii) 'Erfolglos', wenn die Voraussetzung innerhalb des Zeitraumes bis zu Beginn des 46. Monats nicht bestanden wurde, und iii) 'Erfolglos', wenn die Voraussetzung innerhalb des Zeitraumes bis zu Beginn des 46. Monats nicht absolviert wurde, ausgeführt werden. Der Unterschied zwischen dem 2. und 3. Fall besteht in der Art der Ereignisse und somit der Art der Auslösung der Übergänge. Während das Nichtbestehen der Voraussetzung im 2. Fall

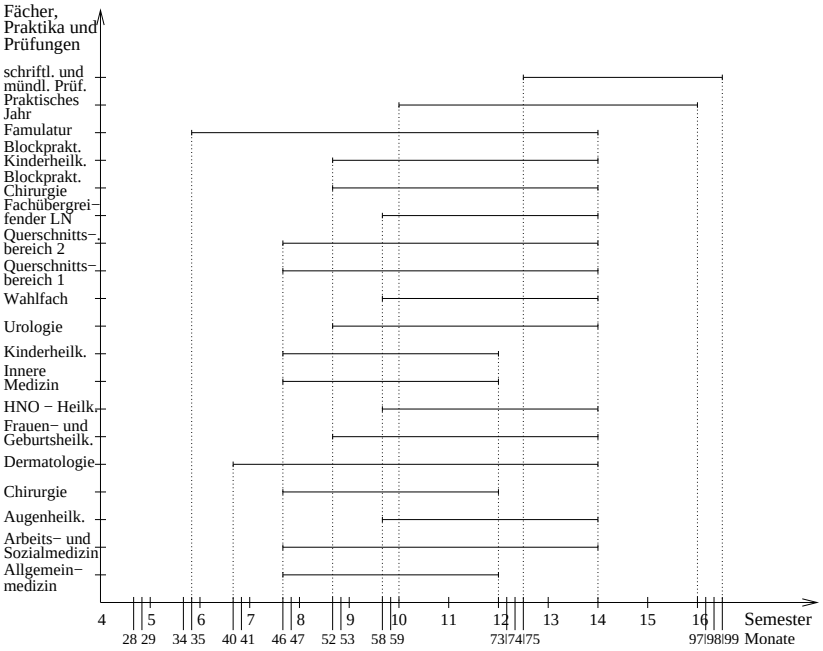


Abb. 2. Zeiträume erfolgreicher Abschlüsse der Fächer des 2. Abschnittes der Ärztlichen Prüfung

durch einen in der Umgebung erfolgten manuellen Eintrag in eine Datenbank ausgelöst wurde, tritt der 3. Fall genau dann ein, wenn der 1. und 2. Fall niemals vorlagen oder ausgeführt wurden und die Zeit den Beginn des 46. Monats überschritten hat. Im Fall 3 kann der Übergang dann automatisch ausgelöst werden. Weitere Regeln von Grundfächern lassen sich in der gleichen Weise wie die Regeln des Prüfungsablaufes des Faches 'Allgemeinmedizin' formalisieren. Ein hybrides System kann bereits solch einem hybriden Automaten entsprechen bzw. durch Synchronisation mehrerer Automaten, die selbst auf durch Synchronisation kombinierten Automaten bestehen und verfeinert sein können, geschaffen werden..

3 Abstraktion und Verfeinerung

Abstraktion im Sinn der parallelen Komposition mittels Synchronisation wird in unseren Spezifi kationen zum Zweck der Erhaltung der Übersichtlichkeit, der Möglichkeit des Bottom-Up-Entwurfes und der Wiederverwendung bestehender hybrider Automaten eingesetzt. Die Abläufe in Automaten können vollständig unabhängig voneinander ausgeführt oder mit Hilfe empfangener oder gesendeter Signale synchronisiert werden. Dadurch wird die gleichzeitige Ausführung von Ereignissen der Abläufe erreicht, welche eine angenommene Voraussetzung für den Nachweis von Eigenschaften mit der symbolischen Simulation darstellt. Zum Aufbau eines Systemes, wie das Studiensystem des 2. Abschnittes der Ärztlichen Prüfung, kann zu Beginn auf die genaue Beschreibung detaillierter Prüfungsabläufe zu Grund- und Komplexfächern verzichtet werden. Der in Abbildung 4 dargestellte Automat beschreibt einen in VYSMO spezifi zierten, hybriden synchronisierenden Automat (HSA) für den 2. Abschnitt der Ärztlichen Prüfung.
Ein HSA wird ähnlich einem hybriden hierarchischen Automaten (HHA), wie solch ein Automat in Abbildung 3 für das Fach 'Allgemeinmedizin' gezeigt wurde, in 4 Abschnitte unterteilt. Der HSA ist jedoch im Namensfeld mit doppelten Linien an den Seiten gekennzeichnet und anstelle der Verhaltensbeschreibung ist eine Synchroniationsbeschreibung hybrider Automaten aufgeführt. Die Synchronisationsbeschreibung enthält Automaten, welche auf HSA oder auf elementaren HHA basieren und durch Verbindungen zur Synchronisation miteinander in Beziehung stehen. Während hybride hierarchische Automaten die Möglichkeit zur Verfeinerung bieten, entstehen hybride synchronisierende Automaten aus der Gruppierung von Automaten und der Schaffung von Synchronisationsbeziehungen zu anderen Gruppen von Automaten. Die Synchronisationsbeziehungen können durch Bedingungen zur

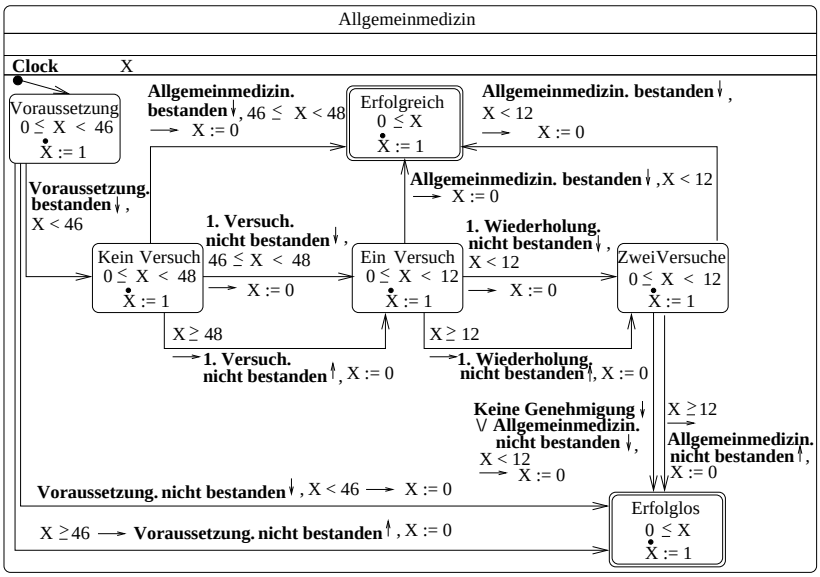


Abb. 3. Hybrider Automat für den Prüfungsvorgang des Faches Allgemeinmedizin

Aktivierung eines Automaten oder einer Automatengruppe festgelegt werden und Zuweisungen zu Variablen enthalten.

Die Aktivierung des Gesamtsystems 'Studienverlauf' wird durch einen Pfeil symbolisiert, welcher von einem gefüllten schwarzen Kreis ausgeht. An dieser Stelle lassen sich Vorbedingungen und Initialwerte von Variablen des Systems spezifizieren. Zum Beispiel wird in Abbildung 4 die Variable 'GF' mit der Menge aller bestehenden Grundfächer initialisiert. Die Synchronisationsbeschreibung weist die Abstraktion aller Fächer, Fachgruppen und Praktika auf, die zum erfolgreichen Absolvieren des 2. Abschnittes der Ärztlichen Prüfung beitragen. Dabei existieren Automaten, wie der Automat der 'GF_I' vom Typ 'Grundfächer', die selbst Instanzen eines HSA und Automaten, wie der Automat der 'Famu_I' vom Typ 'Famulatur', die Instanzen eines HHA sind. Hier besteht kein notationeller Unterschied. Erst der detaillierte Typ lässt auf die Art des Automaten schliessen. Die Nachweise zu den Fächern und Fachgruppen können je nach Synchronisationsbeziehung unabhängig parallel oder müssen nacheinander folgend erbracht werden. Die Prüfungen zur Gruppe der Grundfächer, wozu auch ein durch den Parameter 'FLN' festgelegter fachübergreifender Leistungsnachweis gehört, kann vollständig unabhängig von dem Absolvieren der Famulatur ausgeführt werden. Das Wahlfach dagegen kann entsprechend der Aktivierungsbedingung 'activ_WF' nur dann unabhängig von der Gruppe der Grundfächer abgelegt werden, wenn das Fach Sportmedizin oder Zahnmedizin/Mund-, Kiefer- und Plastische Chirurgie gewählt wurden. Ist das Wahlfach 'WF' laut der Bedingung an der Synchronisationsverbindung zwischen den Grundfächern und dem Wahlfach in der Menge der Grundfächer 'GF' enthalten, so muss das Grundfach erst bestanden sein, bevor das Wahlfach darauf aufbauend abgelegt werden kann. Um das Praktische Jahr zu absolvieren, müssen laut Aktivierungsbedingung 'aktiv_PJ' und den Bedingungen an den Synchronisationsverbindungen alle Prüfungen bis auf die abschliessenden schriftlichen und mündlichen Prüfungen bestanden sein. Aktivierungsbedingungen werden später in den detaillierten Automaten als formale Parameter an den Eingangsübergängen festgehalten, so dass hier der aktuelle Wert mit zum Beispiel 'aktiv_PJ' übergeben werden muss.

Verfeinerungen dienen der sequentiellen Komposition, wodurch in einem Top-Down-Entwurf das detaillierte Verhalten der Lokationen von hybriden hierarchischen Automaten beschrieben werden kann. Angenommen, ein für das Grundfach 'Allgemeinmedizin' verantwortlicher Dozent gestaltet die Voraussetzungen des hybriden Automaten aus Abbildung 3 wie folgt:

> Als Zulassungsvoraussetzung für die Prüfung der 'Allgemeinmedizin' ist ein Leistungsnachweis in Form von erreichten 60% der Punkte für erfolgreich gelöste Hausaufgaben oder eines Kurztestates, welches einmalig innerhalb von 6 Monaten wiederholt werden darf, zu erbringen.

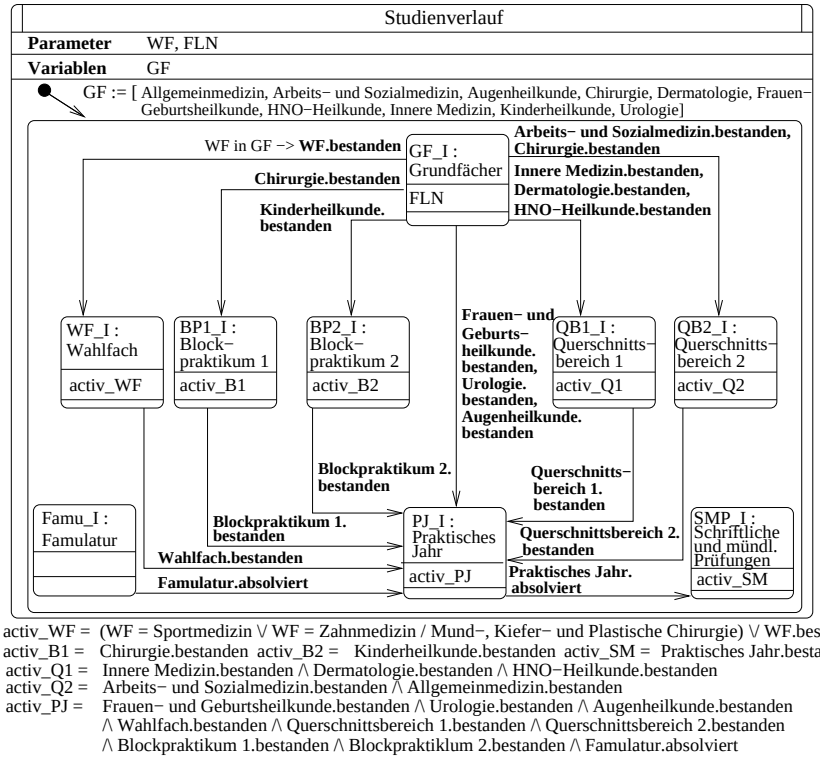


Abb. 4. Hybrider Synchronisierender Automat (HSA) zum 2. Abschnitt der Ärztlichen Prüfung

Unter dieser Annahme kann die Lokation 'Voraussetzung' durch den hybriden hierarchischen Automaten aus Abbildung 5 ersetzt werden. Technisch wird die Ersetzung als Instanziierung realisiert, wozu vorher im Automaten für 'Allgemeinmedizin' syntaktische Veränderungen vorgenommen werden müssen, die hier nicht ausgeführt sind. Der Automat zur Verfeinerung der Voraussetzungen besitzt Merkmale, die an dieser Stelle kurz angesprochen werden. Die Darlegung im Einzelnen erfolgt jedoch in weiteren Arbeiten. Die Lokation 'Voraussetzung' des HHA 'Allgemeinmedizin' ist mit der Invariante '$0 \leq X < 46$' verbunden. Das bedeutet als 1. Merkmal, das Überschreiten der Zeitgrenze von 46 Monaten ist nicht erlaubt, sonst muss die Lokation verlassen werden. Diese Invariante gilt auch für den hybriden Automaten, der zur Verfeinerung der Lokation 'Voraussetzung' dient. Alle Invarianten von Lokationen und Bedingungen der Übergänge dürfen die Invariante '$0 \leq X < 46$' nicht verletzen. Da dem Modellierer die Überprüfung manuell nicht zumutbar ist, kann nur ein automatisches Verfahren, wie die symbolische Simulation, den Nachweis unterstützen. Das 2. Merkmal bezieht sich auf die Art der Modellierung von hybriden Systemen, wodurch Konflikte entstehen können. Wie in technischen Systemen, für welche die Aktionen des Rechnersystems als diskretes Zustandsmodell und die zu überwachende und steuernde Umgebung als Menge von Gleichungen und Ungleichungen über kontinuierlichen Variablen unabhängig voneinander beschrieben und danach kombiniert werden, werden in regelbasierten Beratungssystemen Leistungen konkret festgelegter Zeiträume und Zeitpunkte als diskretes Zustandsmodell und die Regeln von Ordnungen usw. als Menge von Gleichungen und Ungleichungen über Uhrenvariablen unabhängig voneinander beschrieben und danach kombiniert. Daraus kann sich ein Konflikt zwischen Übergangs- und Lokationsbeschreibungen ergeben. In Abbildung 5 führt zum Beispiel die Bedingung '$Y \geq 46$' an dem Übergang von Lokation 'Voraussetzung zu erfüllen' zu 'Voraussetzung nicht erfüllt' dazu, dass die Lokation 'Voraussetzung nicht erfüllt' mit der Invariante '$Y < 46$' gar nicht betreten werden kann. Hier liegt eine Art des Zeno-Verhaltens [Yov97] vor, welches von der symbolischen Simulation erkannt werden muss und eine Änderung des Modells erfordert. Die Änderung stellt aufgrund der Invarianten im Moment ein offenes Problem dar. Das 3. Merkmal betrifft das Senden von Signalen an den umgebenden, verfeinerten HHA wie 'Voraussetzung.bestanden' am Übergang von 'Voraussetzung zu erfüllen' zu 'Voraussetzung erfüllt' an den umge-

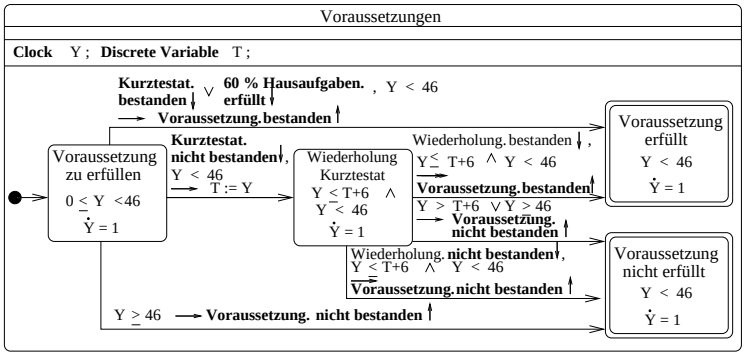


Abb. 5. Hybrider hierarchischer Automat zur Verfeinerung der Lokation Voraussetzung des HHA für 'Allgemeinmedizin'

benden, verfeinerten HHA der 'Allgemeinmedizin'. Dieses Signal dient als Impuls, um das sequentielle Verhalten des übergeordneten Automaten fortsetzen zu können.

4 Klassifikation von Eigenschaften und deren Formalisierung

Im Bereich der regelbasierten Beratungssysteme haben sich 2 Gruppen von Eigenschaften herausgebildet, zum einen die Konsistenz bestehender (Ver-)Ordnungen und zum anderen die Machbarkeit konkreter Aussagen gegenüber solchen (Ver-)Ordnungen. Die beiden Begriffe lassen sich wie folgt erklären:

Begriff 41 Eine (Ver-)Ordnung heisst *konsistent*, wenn genau diejenigen Abläufe akzeptiert werden, die gegebenen Konsistenzbedingungen entsprechen. Als Konsistenzbedingungen gelten dabei Widerspruchsfreiheit in der Logik der Regeln selbst, Regeln der Rahmen- und Durchführungsordnungen sowie Regeln aus Zusatzbestimmungen.

Begriff 42 Die *Machbarkeitsanalyse* umfasst die Überprüfung der Kohärenz [STG01] von aktuellen Plänen und Terminen gegenüber bestehenden Ordnungen, der individuellen Durch- und Ausführung gegenüber Regeln bestehender Ordnungen und die Überprüfung von SOLL-Abläufen gegenüber bestehenden Ordnungen und aktuellen Regeln aus Plänen, Terminen, individuellen Durch- und Ausführungsentscheidungen.

Theoretisch werden beide Klassen von Eigenschaften auf der Basis von konkreten und symbolischen Abläufen formalisiert.

Begriff 43 Ein *konkreter Ablauf* ρ ist ein Zeitwort: $\langle \Lambda_1,\ X_1,\ \omega_1\rangle$, $\langle \Lambda_2,\ X_2,\ \omega_2\rangle$, $\langle \Lambda_3,\ X_3,\ \omega_3\rangle$, ..., wobei:
Λ_i eine Menge von Leistungen L,
X_i eine Menge von Uhren, die die Menge von Zeitwerten T widerspiegelt, und
ω_i eine Abbildung von Uhrenvariablen auf konkrete Zeitwerte mit i = 1,2,3,... darstellen.

Begriff 44 Ein *symbolischer Ablauf* R ist ein Zeitwort: $\langle \Lambda_1,\ X_1,\ \Omega_1\rangle$, $\langle \Lambda_2,\ X_2,\ \Omega_2\rangle$, $\langle \Lambda_3,\ X_3,\ \Omega_3\rangle$, ..., wobei:
Λ_i und X_i wie für den konkreten Ablauf definiert sind und
Ω_i eine Abbildung von Uhrenvariablen auf Zeitintervalle unendlich vieler Zeitwerte mit i = 1,2,3,... darstellen.

Die Abläufe können dabei, wie im folgenden Beispiel gezeigt, als temporallogische Formeln beschrieben werden.

Beispiel 41 *In einer übergeordneten Rahmenordnung sei folgende Regel für das Studium der Medizin enthalten:*
$\Diamond T_i.Mathematik.LN.(bestanden|nicht\ bestanden) \wedge$
$\Diamond T_j.Physik.Prüfung.(bestanden|nicht\ bestanden)\ \wedge\ T_i < T_j < 4$ *wobei:*
$\Diamond T.Leistung$ *bedeutet: Irgendwann zu einem Zeitpunkt 'T' in der Zukunft ist die 'Leistung' nachzuweisen.*
Die Regel ist eine Konsistenzbedingung für untergeordnete Studien- und Prüfungsordnungen, welche besagt, dass innerhalb von 4 Semestern ein Leistungsnachweis im Fach Mathematik zu erbringen und danach eine Prüfung in

Physik abzulegen ist. Die temporallogische Aussage spiegelt alle symbolischen Abläufe folgender Form wider:
$\langle\{Leistungen\},\ T_1,\ Bedingung\ für\ T_1\rangle$,..., $\langle\{Leistungen\},\ T_{i-1},\ Bedingung\ für\ T_{i-1}\rangle$,
$\langle\{Mathematik.LN.(bestanden|nichtbestanden)\},\ T_i,\ Bedingung\ für\ T_i\rangle$,
$\langle\{Leistungen\},\ T_{i+1},\ Bedingung\ für\ T_{i+1}\rangle$,..., $\langle\{Leistungen\},\ T_{j-1},\ Bedingung\ für\ T_{j-1}\rangle$,
$\langle\{Physik.Prüfung.(bestanden|nichtbestanden)\},\ T_j,\ Bedingung\ für\ T_j\rangle$,... .
Die alte Studien- und Prüfungsordnung erforderte einen Nachweis je Fach Mathematik, Physik und Chemie:
1. Ein Leistungsnachweis ist im Fach Mathematik bis zum Ende des 4. Semesters zu erbringen.
2. Nach dem erfolgreichen Abschluss des Faches Mathematik sind unabhängig voneinander Prüfungen in den Fächern Physik und Chemie bis zum Ende des 4. Semesters abzulegen.
In der neuen Studien- und Prüfungsordnung wurde bezüglich dieser Regel folgende Änderung vorgenommen:
1. In einer Gesamtprüfung müssen 2 der Fächer Mathematik, Physik und Chemie wahlweise abgelegt werden.
2. Danach ist das 3. Fach in einer separaten Prüfung abzulegen.
Die Änderung in der neuen Studien- und Prüfungsordnung führt zur Verletzung der Konsistenzbedingung aus der Rahmenordnung.

Die Überprüfung der Machbarkeit basiert auf der Instantiierung von Regeln der Ordnungen, die eine Menge von symbolischen Abläufen bilden, durch aktuelle Informationen, wie gegenwärtig gültige Pläne, festgelegte Termine, individuelle Durchführungsentscheidungen und SOLL-Anforderungen.

Beispiel 42 *Eine mögliche Anfrage nach der Machbarkeit eines Studienablaufes für einen Studierenden des Medizinstudiums im 2. Abschnitt kann folgendermaßen formuliert werden:*

```
machbar (Studien_Ablauf, S_P_Ordnung, Lehrplan, Immat)
        :-
              Studien_Ablauf = [Beginn, <´ASM.bestanden´,T1>,
                                Zwischen, <´Wahlfach.bestanden´,T2>, Ende],
              Wahlfach = ASM,
              52 =< T1 < 54 ,
              Ende= [RestLeist,<Letzte,Tn>],
              Tn < 75.

S_P_Ordnung = Studien- und Prüfungsordnung,
Immat = Immatrikulationsdatum des anfragenden Studierenden
```

Der Studienablauf soll das erfolgreiche Bestehen des Grundfaches 'Arbeits- und Sozialmedizin' (ASM) und des vertiefenden Wahlfaches 'Arbeits- und Sozialmedizin' enthalten. Entsprechend der Bedingung '52 =< T1 < 54' soll das Fach 'ASM' innerhalb der 2 Monate vor Beendigung des 9. Semesters bestanden werden. Der gesamte Studienablauf ist laut 'Tn < 75' und Abbildung 2 innerhalb der Regelstudienzeit zu beenden. Aus der Studien- und Prüfungsordnung ist bekannt, das ein auf einem Grundfach aufbauendes Wahlfach erst belegt werden kann, wenn das Grundfach erfolgreich abgeschlossen worden ist. Der Lehrplan lässt jedoch die Belegung des Wahlfaches 'ASM' zur Zeit nicht zu. Alle Studierenden des jetzigen 10. Semesters, welche das Wahlfach 'ASM' belegen möchten, können das Studium nicht wie entsprechend der vorgegebenen Studien- und Prüfungsordnung innerhalb der Regelstudienzeit beenden. Die Kohärenz zwischen Lehrplan und Studien- und Prüfungsordnung ist verletzt.

5 Zusammenfassung und Ausblick

In dem Beitrag wurde gezeigt, wie hybride Systeme zur problemadäquaten Modellierung verwaltungstechnischer Abläufe, insbesondere in regelbasierten Anwendungen mit beratender Funktion, genutzt werden können. Verwaltungstechnische Abläufe werden dabei im Kontext zweier Bereiche ausgeführt, i) des Regelwerkes, welches aus Ordnungen, Verordnungen und Gesetzen besteht und ii) den aktuellen Daten und Dokumente wie gegenwärtig gültige Pläne und Termine, individuelle Durch- und Ausführungsentscheidungen als auch IST- und SOLL-Abläufe. Das Regelwerk wird formal als Zeitgrammatik, welche die Ausführung der Regeln auf der Grundlage hybrider Automaten zulässt, betrachtet. Die Regeln des Regelwerkes sind abstrakte Regeln, welche durch die Informationen der aktuellen Daten und Dokumente, wie in [TLR05], verfeinert und instantiiert werden. Die Kombination von kontinuierlichem und diskretem Verhalten führt zur unabhängigen Beschreibungsmöglichkeit der Regeln des Regelwerkes, zugehörigen Verfeinerungen und Instantiierungen auf der einen Seite und aktuell festgelegter Termine und Zeiträume auf der anderen Seite. Die Regeln des Regelwerkes *müssen* eingehalten werden. In den festgelegten Zeiträumen *können* Leistungen abgelegt werden. Mit Hilfe der symbolischen Simulation werden zeit- und

sicherheitskritische Eigenschaften wie Konsistenz und Machbarkeit nachgewiesen, um eine beratende Tätigkeit auszuführen. Da die symbolische Simulation auf einem sprachtheoretischen Ansatz basiert, müssen die hybriden Systeme an allen Übergängen Signale zur Beschreibung abgelegter Leistungen aufweisen und jeder Automat muss mit Endlokationen ausgestattet sein. Zur Ausführung der symbolischen Simulation in der constraint-logischen Sprache PrologIV ist die Architektur eines Werkzeuges Rossy [TLR04] entstanden, wobei Anfragen und Programme der nutzerfreundlichen Sprache MODEL-HS in PrologIV-Programme übersetzt werden. Anfragen, die als Eingabeparameter das hybride System und erwünschte Bedingungen für den Ablauf erhalten, werden von einem symbolischen Simulator gestartet, der als Ergebnis die Bedingungen berechnet, unter welchen das hybride System den gegebenen Ablauf akzeptieren kann.
Aus theoretischer Sicht wird der Zusammenhang zwischen Zeitgrammatiken und hybriden Automaten in zukünftigen Arbeiten anhand weiterer Beispiele vertieft. Aus Ergebnissen, welche bei der Ausführung hybrider Automaten erreicht werden, sollen Schlussfolgerungen auf die Eigenschaften der Grammatik gezogen werden. Praktisch ist die konzeptionelle Architektur von Rossy durch eine Implementation zu unterstützen, wodurch sich theoretische Ergebnisse verifizieren lassen. Für die Nutzerfreundlichkeit sollen dazu bedienerfreundliche Oberflächen und Anfragemasken [LTR03], die eine Auswahl aus gegebenen Listen ermöglichen, zur Verfügung gestellt werden.

Literatur

[ACHH93] R. Alur, C. Courcoubetis, T.A. Henzinger, and P.-H. Ho. Hybrid Automata: An Approach to the Specification and Verification of Hybrid Systems. In *Hybrid Systems*, number 736 in LNCS, pages 209–229. Springer, 1993.

[AGH+00] R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular Specification of Hybrid Systems in CHARON. In *3rd Int. Workshop of Hybrid Systems:Computation and Control*, number 1790 in LNCS, pages 6–19. Springer, 2000.

[AGLS01] R. Alur, R. Grosu, I. Lee, and O. Sokolsky. Compositional Refinement of Hierarchical Hybrid systems . In *4th Int. Workshop of Hybrid Systems:Computation and Control*, number 2034 in LNCS, pages 33–48. Springer, 2001.

[AKY99] R. Alur, S. Kannan, and M. Yannakakis. Communicating Hierarchical State Machine. In *Automata, Language and Programming*, number 1644 in LNCS, pages 169 – 178. Springer, 1999.

[BSW02] R. J. Back, C. Cerschi Seceleanu, and J. Westerholm. Symbolic Simulation of Hybrid Systems. In P. Strooper and P. Muenchaisri, editors, *APSEC 2002*, pages 147–155, Queensland, Australia, 2002. IEEE Computer Society Press.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1 edition, 1999.

[Hen96] T. A. Henzinger. The theory of hybrid automata. In *11th Annual IEEE Symposium on Logic in Computer Science*, IEEE, pages 278–292. IEEE Computer Society Press, 1996.

[Hen00] T. A. Henzinger. Masaccio: A formal model for embedded components. In *IFIP*, number 1872 in LNCS, pages 549–563. Springer, 2000.

[LC05] J. K. Lee and S. D. Chi. Use of the Symbolic DEVS Simulation in Generating Optimal Traffic Signal Timings. *Simulation*, 81(2):153–170, 2005. ISSN: 0037 - 5497.

[LTR03] A. Lantsmann, E. Tetzner, and G. Riedewald. Ubiquitäre Studienplanung aus Sicht der Konsistenz und Sicherheit, 2003. ISSN 0233-0784.

[Rie95] G. Riedewald. A little bit modified Railroad Crossing. Preprint CS-05-95, University Rostock, Department Computer Science, 1995.

[RU95] G. Riedewald and L. Urbina. Symbolische Simulation Hybrider Systeme in Constraint Logic Programming. Preprint CS-02-95/CS-03-95, University Rostock, Department Computer Science, 1995.

[Sik05] M. Sikos. Fallstudie zur Beschreibung hybrider Systeme anhand der Sicherheitsanlagen eines Verkehrstunnels. Studienarbeit, Juni 2005.

[STG01] R. Sebastiani, A. Tomasi, and F. Giunchiglia. Model Checking Syllabi and Student Careers. In *TACAS2001*, number 2031 in LNCS, pages 128–142. Springer, 2001.

[TBR01] E. Tetzner, R. Brauch, and G. Riedewald. Visual Modeling of Hybrid Systems for Symbolic Simulation in VYSMO, 2001. ESM2001.

[TKRE00] E. Tetzner, A. Kunert, G. Riedewald, and N. Erdmann. Symbolic Simulation for Cultivation of Biosensor Cells. In *FOODSIM2000*, SCS Publication, 2000.

[TLR04] E. Tetzner, W. Lohmann, and G. Riedewald. Rossy: A Tool for Symbolic Simulation of Hybrid Systems. In *Industrial Simulation Conference 2004*. EUROSIS, June 2004.

[TLR05] E. Tetzner, A. Lantsmann, and G. Riedewald. Modellierung und Symbolische Simulation des Grundstudiums der Informatik an der Universität Rostock, 2005. in preparation.

[TR99] E. Tetzner and R. Riedewald. MODEL-HS - An Approach to Specify Hybrid Systems for Symbolic Simulation. In A. Poetsch-Heffter and W. Goerigk, editors, *ATPS 99*, number 258 in Informatik-Berichte, pages 201–224. University Hagen, 1999.

[Yov97] Sergio Yovine. Kronos: A Verification Tool for Real-Time Systems. *International Journal of Software Tools for Technology Transfer*, 1, 1997.

[Zei76] B. P. Zeigler. *Theory of Modelling and Simulation.* John Wiley and Sons, New York, 1976.

A Virtual Machine for State Space Generation

Michael Weber and Stefan Schürmans

Lehrstuhl für Informatik II
RWTH Aachen University
Aachen, Germany
{michaelw,stes}@i2.informatik.rwth-aachen.de

Abstract. The semantics of modelling languages are not always specified in a precise and formal way, and their rather complex underlying models make it a non-trivial exercise to reuse them in newly developed tools. We present a virtual machine-based approach for state space generation. The virtual machine's (VM) byte-code language is straightforwardly implementable, facilitating reuse and making it an adequate target for translation of higher-level languages like the SPIN model checker's Promela. As added value, it provides executable and formal operational semantics for modeling languages. Benchmarks show that a VM is competitive in speed for state space generation.

1 Introduction

Common approaches in state-based model checking employ modeling languages like CSP [13], LOTOS [5], Murϕ [8], DVE [4], or Promela [14] to describe actual state spaces. These languages are usually non-trivial: in addition to concepts found in programming languages (scopes, variables, expressions etc.) they often provide features like process abstraction, non-determinism, guarded commands, synchronisation and communication primitives, timers, etc. Implementing an operational model of such languages for use in verification tools is consequently not straightforward, even more so if the language is described informally only, and their static and operational semantics are incomplete at best, outdated, or entirely unavailable.

That being said, when developing verification tools it is highly desirable to reuse an already existing popular modeling language like e.g. Promela, which has been used in a sizeable number of real-world case studies. These models can be used to benchmark new tools against old ones in a fair way. Promela has wide industry acceptance, allowing modelers to try out compatible tools without having to re-specify models in yet another formalism. Lastly, since these tools are developed *for the application* of formal methods, it is worthwhile and only fair to treat them with the same rigor: a shared underlying virtual machine would make it possible to compare different algorithms fairly and easier to test a new algorithm for conformance against existing algorithms.

This begs however the question, why existing tools are not simply extended and thus the whole static and dynamic semantics machinery is reused? The most trivial answer is that new approaches might be implemented in a different programming language, for a variety of reasons [16]. Also, many verification tools can

be considered research prototypes which are not developed with extensibility first in mind: modifications become more time-consuming. Furthermore, different approaches often admit different tool architectures: parallel and distributed model checkers often need small *mobile data structures*, so that computations can be relocated to other processors. Nevertheless, we stress that despite different designs all these tools have in common the need for a state space generator component.

Another commonly found reuse pattern is the translation of other formalisms to modeling languages like PROMELA [21, 10], using SPIN [12] as verification backend, and hence restricting the choice of analyses to offerings of a single tool again, or getting trapped in *abstraction inversion* (non-trivial encodings of constructs) [2].

It comes as no suprise that researchers often find it easier to invent their own modeling language with informally specified semantics incompatible to existing tools which, as argued above, puts additional burden on end-users to switch tools, benchmark them, or consider them at all.

We strive to remedy current shortcomings by proposing a virtual machine-based approach to state space generation, in which high-level modeling languages are translated to byte-code instructions. Subsequent execution of such byte-code programs with a virtual machine yields state spaces for further use in model checking tools.

Contributions We describe a virtual machine model suitable for state space generation. The operational semantics of our virtual machine are straightforward, and hence easy to derive an implementation from (indeed, this is exactly what we did, cf. section 3). Moreover, our machine model can be augmented to handle timers, probabilities etc., in a compositional way by adding instructions, and keeping the rest of the model unchanged.

Most byte-codes are simple operations, and benchmarks show that our machine is competitive in state space generation to SPIN. For distributed state space generation our machine offers additional benefits: it can be restarted from machine state snapshots, which have a self-contained, contiguous and platform-independent representation, and thus can be send across networks without further serialization efforts.

Organisation In section 2 we describe the virtual machine model and byte-code semantics. Section 3 presents benchmark results for our implementation. We conclude with a brief summary of related and future work in sections 4 and 5.

2 Virtual Machine Specification

For the remainder of the paper, we will consider a concrete instance of a state space generating VM as presented in [28]. Due to space constraints, we cannot present details of the machine. Instead, we will highlight its distinguishing features informally, its global and local state, invariants which translations must preserve, and crucial operations.

We are concerned with state space generation, hence our virtual machine has a couple of features not all of which are commonly found at byte-code level in conventional VM architectures like the JavaVM [22], in particular:

Non-determinism If non-deterministic choice is encountered during executing, the machine offers all possible continuations to the scheduler who then decides which path to take.

Concurrency A builtin `run` byte-code allows to spawn processes at run-time.

Communication Both, synchronous and asynchronous channel objects are provided for inter-process communication.

First-class channels Like in PROMELA and π-calculus [20], our machine allows channels to be sent over channels.

Priority scheme Our byte-code allows to specify which actions have to be given preference. Together with explicit control over externally visible actions, this allows to encode high-level constructs like PROMELA's `atomic` and `d_step`.

Speculative execution Certain code sequences are executed speculatively, and changes to the global state are rolled back if the sequence does not run to completion.

External Scheduling Scheduling decisions are delegated to host applications. This allows for implementation of different scheduling policies which is needed to cater for simulation (interactive scheduling) vs. state space exploration with some search strategy (breadth-first, depth-first, or combinations thereof).

Although we are confident that our VM is already a suitable target for quite a number of high-level languages, we do not claim exhaustiveness here. That is, depending on features of a high-level model, adjustments and additions might be needed, or an entirely different state representation could even be beneficial. However, we would like to stress the generic nature of our approach which is in no way restricted to this specific example.

2.1 Architecture

The design of our VM was mainly driven by pragmatic decisions: it was our intention to create a model that is simple, efficient and embeddable as component into host applications, with implementation effort split between the VM and compilers targeting it. For example, many instructions make use of the VM's stack because it is trivial for compilers to generate stack-based code for expression evaluation. On the other hand, a stack-based architecture alone is inconvenient for translation of counting loops, thus registers were added. The RISC-like instruction set is motivated by the need for fast decoding inside the instruction dispatcher, the VM's most often executed routine.

Although our machine is a mixture of register-based and stack-based architecture, we are nevertheless dealing with finite state models in this paper.

Because our machine model supports non-determinism, we cannot merely execute instructions like commonly known VMs, e. g. the JavaVM. Instead, the machine executes a step and yields a set of possible successor states. An external scheduler finally decides which successor states (possibly more than one) are to be executed further. In verification context, these scheduling decisions are induced e. g. by model checking algorithms.

For presentational reasons we chose to specify successor states by combining several (labeled) relations between states. Here, we mention only internal steps

which are unobservable to the scheduler, and a prioritised step which concludes a number of internal steps to finally render the resulting machine state visible as successor.

As concurrency model we chose *interleaving* semantics for our machine, although *true concurrency* semantics are conceivable as well.

2.2 Machine State

The machine's global state $\Gamma = (\Pi, e, G, \Phi)$ consists of a finite set of processes Π, the identifier of a process with exclusive execution privileges $e \in Pid_\perp$ ($\perp$ if none), global variable store G, and set of existing channels Φ.

Channels are used to model message-passing synchronisation. In our machine, communication channels are typed and bounded, and we distinguish between rendezvous channels and asynchronous channels. Rendezvous channels have zero capacity. Nevertheless, a single message can temporarily be stored in a channel during synchronous communication. Such states are internal to the virtual machine and unobservable to the outside.

Our unit of execution, a process $\pi = (p, M, \Lambda') \in \Pi$, is represented by its globally unique identifier p, execution mode M (Normal, Atomic, Deterministic, Terminated), and $\Lambda' = (L, m)$ as the local state of a process. L denotes the process-local variable store and its program counter m the instruction executing next. When a process becomes *active*, its state Λ' is augmented with (zeroed) registers R_0 and a stack $D_\epsilon = \epsilon$ to its *active local state* $\Lambda = (L, m, R_0, D_\epsilon)$.

On purpose, the design of our VM allows it to be interrupted and restarted from snapshots of its global state Γ at a later time, possibly even on a different processor. This property allows us to embed our VM in distributed model checking tools [9]. In our implementation we eliminated any overhead of taking state snapshots, by having the VM work on a self-contained and contiguous binary state representation which e.g., can be directly stored or sent over a network without further marshalling.

2.3 Invariants

Translations to our byte-code language must guarantee the following invariants: a process becoming active again always resumes execution with register set R_0 and the empty stack D_ϵ. Conversely, at those points in the program when a process may become inactive, the contents of registers and stack are discarded and need not matter for the rest of its execution. Because the number of local variables is fixed, a local state Λ' thusly occupies constant space only.

2.4 Byte-code Semantics

Global and active local machine state is modified by instructions from our byte-code language. Among them are operations one would usually expect for translation of arithmetic and boolean expressions (`NEG`, `ADD`, `MUL`, `LT`, `CMP`), for loading constants and variables (`LDC`, `LDV`), other stack and register handling (`PUSH`, `POP`), and (un)conditional jumps for control flow (`JMP`, `JMPZ`).

Table 1. Crucial byte-code operations

`NDET` a	non-deterministic jump $(L, m, R, D) \rightarrow_{int} (L, m+1, R, D)$ $(L, m, R, D) \rightarrow_{int} (L, a, R, D)$
`NEX`	step not executable $(L, m, R, D) \not\rightarrow$
`STEP` r, M'	step complete with priority $r \in \Sigma_{pri}$ and mode M' $(\{(p, M, (L, m, R, D))\} \cup \Pi, e, G, \Phi)$ $\xrightarrow{r}_{pri} (\{(p, M', (L, m+1))\} \cup \Pi, e', G, \Phi)$ $e' := p$ if $M' \in \{\underline{\mathrm{A}}, \underline{\mathrm{D}}\}, \perp$ otherwise

We will not go into details of any of the above mentioned operations nor of more complicated operations on channels, mainly because they are not relevant for our argument. They are documented more detailed in [28]. Instead, we focus on three crucial operations given in table 1, which account for most of the flexibility of our VM.

First, we deal with *internal* steps denoted by relation $\rightarrow_{int}$. Through local state operation `NDET` a we express non-determinism explicitly in our machine. Informally, program execution for the currently active process continues from this point both at the following instruction, and in an alternative future at instruction a, thus yielding two possible successors (not yet visible to a scheduler). This allows us to express a variety of non-deterministic high-level constructs in our language.

We can translate conditions on input variables common in *guarded-command* languages by first translating the input action A, then the condition E to be satisfied, and then adding instructions which abort this internal executing path if E is not satisfied: $trans(A); trans(E);$ `NEXZ` (`NEXZ` being the conditional variant of `NEX`). Informally, this means that unless E is satisfied, A retroactively has never happened.

So far, all actions were considered internal. With `STEP` r, M, we allow a global state change (and its associated sequence of actions) to be observed by a scheduler as an r-prioritized step $\xrightarrow{r}_{pri}$. As minor detail, we specify execution mode M for the *following* sequence of actions. This explicit notion of observability allows a flexible granularity specification of actions, thus allowing conditions and actions to be intertwined, as well as providing support for a rich language of actions. Duplication of conditions can now easily be avoided by combining `STEP` with common control flow instructions to translate condition cascades.

We emphasize that our intermediate language is not a toy example. It is, for instance, expressive enough to encode Promela. A translation procedure is outlined in [28, sec. 3].

3 Benchmarks

We implemented the virtual machine sketched in the previous section to confirm the practicality of our approach. Our efforts took roughly one person-week for the

Table 2. State generation: measured in states per second

Model	VM	SPIN: `pan.c`
eratosthenes(7)	181818	18461
eratosthenes(14)	106949	108667
eratosthenes(17)	90419	123698
eratosthenes(30)	45499	11310
sort(3)	245977	34833
sort(5)	132161	191120
sort(6)	101267	169009
leader(6)	66625	122360
snoopy	55607	210507
pftp	142660	117556

implementation of a working version, amounting in about 3,000 lines of C code. It turns out that this prototype performs competitive even when compared to state-of-the-art tools like SPIN.

Contrary to SPIN, the sole task of our VM is state space generation. Additional functionality like model checking, possibly together with partial order reduction [6] etc., is duty of other components not covered here.

For our comparison to SPIN we employed standard breadth-first search with full state space storage. We used the same hash function as SPIN does (due to Jenkins).

Unfortunately, we cannot compare state space size directly, because in general we generate more states[1] than SPIN due to our finer-grained program counter. We expect this to be cleaned up by subsequent optimisations passes in our compiler (along with statement merging etc.) by reducing the number of `STEP` instructions. Hence, in order to get a meaningful idea on the speed of our VM, we measured the rate with which successor states are generated (table 2).

We used SPIN 4.2.5 to translate Promela models into corresponding C files. By default, SPIN uses data-flow optimisations and statement merging. Our compiler does not yet have such optimisation passes, thus—to get a fairer comparison—we disabled them in SPIN as well (`-o1 -o3`). To match our test setup we compiled the generated `pan.c` files with `-O2` (C optimisations), `-DNOREDUCE` (disabling partial-order reduction) and `-DBFS` (enabling breadth-first search). The resulting executable was used for benchmarking.

For our tests we compiled models coming with the SPIN distribution into our byte-code language and subsequently executed them on our VM. Our experiments show that we are competitive in state space generation speed to SPIN. For very small state spaces (below a few thousand states) our VM is ahead of SPIN (e.g., `erathosthenes(7)`), but this hardly matters because the overall runtime is in fractions of seconds.

Larger parameters for `erastothenes` seem to be slightly in our favour as well. The generally decreasing rate of generated states for both contestants is due to an increasing numbers of transitions which have to be handled.

[1] which multiplies due to interleaving semantics

A notable deficiency becomes obvious through the `snoopy` model. Our VM is roughly 3.7 times slower than SPIN. We identified extensive use of communication channels as the culprit. Incidentally, communication is one of costliest operations in our VM, and clearly could benefit from optimisations. On the other hand, our VM outperforms SPIN on `pftp`, which makes use of six channels after all (state space size as reported by SPIN: 439895).

Overall, with the exception of `snoopy` (for known reasons), speed of state space generation for other example models is within a factor of at most two when compared to SPIN, which is good enough for our purposes already. For example, state space generation costs with our VM is already insignificant compared to communication costs in a distributed model checker.

The size of states, which contain all information needed to restart the virtual machine from (global and local variables, channels, processes), is typically within a few bytes of what SPIN reports.

4 Related Work

Virtual Machines Virtual machines have been used extensively in Computer Science. A well-known example is e.g., the work of Wirth for the Pascal programming language [29].

Independent to our work, two (unpublished, to the best of our knowledge) attempts of virtual machine models for restricted PROMELA-like languages have been made [11, 24]. In [11], the virtual machine is described as part of the general design of a model checker, while our paper is focused on providing a reusable component for state space generation.

ESML [7], the high-level language translated into byte-code is restricted in several ways when compared to PROMELA, and its underlying virtual machine inherits some of these restrictions e.g., it lacks support for asynchronous channels, shared variables and dynamic process creation.

In [24, sec. 8] itself, Rosien describes some shortcomings of his attempt, e.g., lack of arrays, no support for data types beyond integers, unclear semantics for `do` loops or handshake communication inside atomic blocks ("[...] causes undesired results, unexpected atomic deadlocks or otherwise erratic behavior.").

Besides that, we are in doubt that the architecture of Rosien's design can be adapted easily to e.g., distributed settings where successive states may be generated on different computers. This use case was specifically taken into account in the design of our VM.

Both papers do not provide a complete formal model of their VM or of the translation into their byte-code language, making it non-trivial to derive implementations from their work, neither are implementations readily available.

BACI The *Ben-Ari Concurrent Interpreter* (BACI) suite in its latest version compiles a version of Wirth's Pascal enriched with concurrency constructs into the byte-code language PCODE, which is then executed on a virtual machine. BACI is widely used as teaching device for concurrency, not for verification purposes,

hence the byte-code language is still relatively high-level and not stream-lined for simplicity and efficient execution. Also, the virtual machine does not allow to specify the granularity of visible actions, as is the case with ours.

To the best of our knowledge, formal semantics of the byte-code language or virtual machine are not available.

TyCO VM In [18, 19], a virtual machine for the process algebra of *Typed Concurrent Objects* (TyCO), a close relative to asynchronous π-calculus, is presented. Features include a process concept, communication channels, and a notion of atomic execution (coined *thread*). Since its virtual machine is meant for program execution rather than verification, it lacks nondeterminism, an external scheduler and invisible states, when compared to our work. Also, TyCO's more complex machine state is not designed for snapshotting and restarting. We stress the authors' report that their virtual machine executes efficiently, also due to optimisations carried out at byte-code level.

Probmela A probabilistic extension of PROMELA is presented in [3]. Through private communication with one of the authors (Ciesinski) we recently learned about their endeavour to implement a virtual machine. No published work of these efforts is available so far, but the cited PROBMELA paper reveals a number of simplifying deviations from PROMELA semantics, e.g. atomic regions always running to completion, making them equal to PROMELA's `d_step` and thus obviating the need for priorities on byte-code level.

However, we see the existence of their project as evidence that we are on the right track, and we are confident that probabilistic extensions can be fitted into our virtual machine model. This is left as future work for a possible collaboration.

Java Path Finder and Bandera Java Path Finder 2 [27] translates Java byte-code into Bandera intermediate representation (BIR), which then can be model-checked using Bogor [23], or translated to PROMELA, using SPIN as back-end model checker. The intermediate representation is a high-level guarded command language, not unlike PROMELA. While it can be translated further down to a certain extent, constructs like arrays, locks, exceptions, and high-level control constructs remain, complicating an implementation of its operational semantics. On the other hand, we are confident that BIR can be translated further down to an extended version of our byte-code language.

The Bogor framework consists of a large Java code base, which we conjecture is not easy to replicate in another language if needed. Again, from the tool point of view, our aim is not to beat the Bogor framework in terms of features, but rather to provide a small but versatile component which can easily be reused, or written from scratch based on a formal specification.

5 Conclusions and Future Work

We presented a virtual machine-based approach to state space generation. The machine's semantics turn out to be straight-forwardly implementable, thus encourag-

ing reuse of our specification. Among the byte-code instructions are all operations commonly needed for the specification of concurrent systems: non-determinism, process creation, communication primitives, and a way to express scheduler constraints (atomic regions). As such, our byte-code language doubles as a general framework for the assignment of semantics to high-level modeling languages for concurrent systems.

To confirm our design, in [25] we detail how to assign understandable and *executable* operational semantics to the full PROMELA language by translation to our byte-code language.

We implemented the PROMELA compiler and the corresponding virtual machine [1]. Benchmarks showed that it is a viable alternative to SPIN's state space generator in terms of speed, and superior for embedding into third-party model checkers. Although a *Just-in-time* compiler for our byte-code is conceiveable to further increase VM speed, we believe the extra complexity is not worth the effort for now.

Future Work An implementation of our virtual machine is currently being integrated with the DIVINE [9, 17] framework for distributed model checking.

Additionally, we are looking into an optimisation phase for our PROMELA compiler along the lines of [30].

We are also looking into extensions of our virtual machine with notions of time [26], probabilities [3], or dynamic memory allocation [15], and their effect on its complexity.

References

[1] Virtual machine and compiler implementation. available from http://www-i2.informatik.rwth-aachen.de/Research/vmssg/.

[2] J. Augusto, M. Butler, C. Ferreira, and S. Craig. Using SPIN and STeP to verify StAC specifications. In *PSI'03*, number 2890 in LNCS, pages 207–213. Springer Verlag, July 2003.

[3] C. Baier, F. Ciesinski, and M. Größer. PROBMELA: a modeling language for communicating probabilistic systems. In *Proc. MEMOCODE*, 2004.

[4] J. Barnat, L. Brim, I. Černá, and P. Šimeček. DiVinE – Distributed Verification Environment. Accepted for PDMC'05's short presentations, 2005.

[5] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. In *The Formal Description Technique LOTOS*, pages 23–73. Elsevier Science Publishers North-Holland, 1989.

[6] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled. State space reduction using partial order techniques. 2(3):279–287, Nov. 1999.

[7] P. de Villiers and W. Visser. ESML—a validation language for concurrent systems. pages 59–64. 7-th Southern African Computer Symposium, July 1992.

[8] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid, 1992.

[9] DiVinE. http://anna.fi.muni.cz/divine.

[10] D. D'Souza and M. Mukund. Checking consistency of SDL+MSC specifications. In T. Ball and S. K. Rajamani, editors, *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2003.

[11] J. Geldenhuys. Efficiency issues in the design of a model checker. Msc. thesis, University of Stellenbosch, South Africa, November 1999.

[12] J.-C. Grégoire, G. J. Holzmann, and D. A. Peled, editors. *The Spin Verification System*, volume 32 of *DIMACS series*. American Mathematical Society, 1997. ISBN 0-8218-0680-7, 203p.

[13] C. A. R. Hoare. *Communcating Sequential Processes*. Prentice Hall, 1985.

[14] G. J. Holzmann and V. Natarajan. Outline for an operational-semantics definition of PROMELA. Technical report, Bell Laboratories, July 1996.

[15] F. Lerda and R. Sisto. Distributed-memory model checking with SPIN. In *Proceedings of the 5th International SPIN Workshop*, volume 1680 of *Lecture Notes in Computer Science*, pages 22–39. Springer, 1999.

[16] M. Leucker, T. Noll, P. Stevens, and M. Weber. Functional programming languages for verification tools: A comparison of ML and Haskell. *Software Tools for Technology Transfer*, 7(2):184–194, 2005.

[17] M. Leucker, M. Weber, V. Forejt, and J. Barnat. DivSPIN – a SPIN compatible distributed model checker. Accepted for PDMC'05's short presentations, 2005.

[18] L. Lopes, F. Silva, and V. T. Vasconcelos. A virtual machine for the TyCO process calculus. In *PPDP'99*, volume 1702 of *Lecture Notes in Computer Science*, pages 244–260. Springer-Verlag, September 1999.

[19] L. Lopes and V. T. Vasconcelos. TyCO abstract machine — the definition. DCC 97–1, DCC-FC & LIACC, Universidade do Porto, May 1997.

[20] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS–LFCS–91–180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, Oct. 1991.

[21] P. P. P. Inverardi, H. Muccini. Automated check of architectural models consistency using spin. San Diego, California, 2001.

[22] Z. Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.

[23] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. *SIGSOFT Softw. Eng. Notes*, 28(5):267–276, 2003.

[24] M. Rosien. Design and implementation of a systematic state explorer. Msc. thesis, University of Twente, The Netherlands, March 2001.

[25] S. Schürmans and M. Weber. A virtual machine for state space generation. full version (draft), 2005.

[26] S. Tripakis and C. Courcoubetis. Extending promela and spin for real time. In *Proceedings of TACAS '96*, volume 1055 of *LNCS*, 1996.

[27] W. Visser, K. Havelund, G. Brat, and S. Park. Java pathfinder - second generation of a java model checker, 2000.

[28] M. Weber and S. Schürmans. A virtual machine for state space generation. unpublished, April 2005.

[29] N. Wirth. Pascal-s: A subset and its implementation. In D. W. Barron, editor, *Pascal - The Language and its Implementation*, pages 199–259. John Wiley, 1981.

[30] K. Yorav and O. Grumberg. Static analysis for state-space reductions preserving temporal logics. *Form. Methods Syst. Des.*, 25(1):67–96, 2004.

Ein striktes coalgebraisches Berechnungsmodell

Baltasar Trancón y Widemann

Institut für Softwaretechnik und theoretische Informatik
Technische Universität Berlin
bt@cs.tu-berlin.de

Zusammenfassung. Freie Datentypen und pure Funktionen können *fundiert* oder *nichtfundiert* betrachtet werden. Dabei muß zwischen den unvereinbaren Vorteilen von abschätzbarem Verhalten und einfacher Implementierung einerseits und der Unterstützung potentiell unendlicher Daten und Berechnungen andererseits abgewägt werden. Zyklische Daten werden aber in beiden Ansätzen innerhalb der *Algebra* weder besonders unterstützt noch überhaupt als solche erkannt. Hier hilft die duale Theorie *Coalgebra*, die auch zyklische und unendliche Objekte adäquat beschreibt. Der endlich darstellbare Anteil einer Coalgebra kann im Wesentlichen mit den gleichen rekursiven Techniken wie eine fundierte Algebra verarbeitet werden. Läuft die Berechnung in einen Zyklus, kann dieser anhand der Informationen auf dem Stack erkannt und behandelt werden. Das Prinzip der coalgebraischen Berechnungstechnik und die Implementierung als virtuelle Maschine wird in dieser Arbeit anhand von elementaren, aber nichttrivialen Berechnungen auf periodisch unendlichen Listen demonstriert.

1 Einführung

1.1 Einordnung des Ansatzes

Der hier vorgestellte Ansatz versucht, eine Lücke in den Reihen der bekannten Programmier-Paradigmen zu schließen. Ziel ist die Kombination von Vorteilen der folgenden Ansätze:

1. Sicherheit, Skalierbarkeit und Eleganz des deklarativen, funktionalen Stils.
2. Natürliche Implementierung und Codierung der strikten Auswertung.
3. Explizite Zyklizität der Objektorientierung (*Implementierungssicht*).
4. Implizite Zyklizität der Coalgebra (*Benutzersicht*).

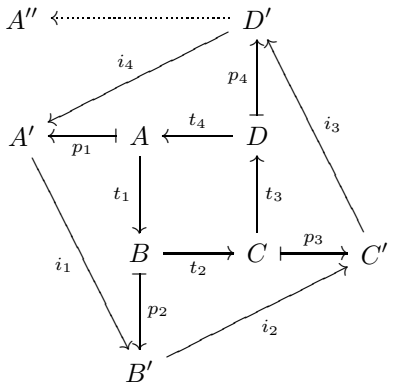
1.2 Algebra und Coalgebra

Ohne die volle Tiefe beider Theorien auszuloten, soll dieser Abschnitt den Unterschied zwischen Algebra und Coalgebra soweit veranschaulichen, wie es für das Verständnis der nachfolgenden Ergebnisse notwendig ist. Für eine formalere Behandlung sei auf [2] verwiesen.

Charakteristisch für die Algebra ist eine kausale Abhängigkeit der Identität eines Objektes von seinem Aufbau durch *Konstruktion*. Dieser Abhängigkeit folgt die *rekursive* Berechnung bzw. *induktive* Definition, indem sie komplexe Ausdrücke von innen nach außen (*bottom-up*) auswertet. Die zentrale Eigenschaft gutartiger algebraischer Operationen ist die *Terminierung*, also die Endlichkeit der ganzen Operation.

Dem gegenüber steht bei der Coalgebra die Ableitung des Aufbaus eines Objektes von seiner Identität durch *Beobachtung*. Dementsprechend geht die *corekursive* Berechnung bzw. *coinduktive* Definition von außen nach innen (*top-down*) vor. Die zentrale Eigenschaft ist dabei die *Produktivität*, also die Endlichkeit jedes beobachtbaren Operationsschritts.

1.3 Corekursives Berechnen



Man betrachte den Ablauf einer Berechnung, die den Zyklus $ABCD$ (innen im Bild) in einen Zyklus $A'B'C'D'$ (außen) derselben Form übersetzt. Auf der Argumentseite sind dazu mindestens die *Traversierungsschritte* $t_1 - t_4$ nötig, um die gesamte Struktur zu erfassen. Die Ausgabe wird erzeugt durch die *Produktionsschritte* $p_1 - p_4$, welche Zellen erzeugen, und die *Initialisierungsschritte* $i_1 - i_4$, welche die Zellen verbinden.

Eine algebraische Implementierung würde zunächst rekursiv alle durch einen Traversierungsschritt erreichbaren Teiloperationen durchführen, um zum Schluß Produktion und Initialisierung der Wurzelzelle in einen *Konstruktorschritt* zu vereinen. Zyklen können auf diese Art nicht erzeugt werden.

Das charakteristisch corekursive Vorgehen ist dagegen, pro Eingabezelle zuerst einen Produktionsschritt auszuführen und die Abbildung von Ein- auf Ausgabezelle geeignet zu speichern. Dann führt ein Traversierungsschritt zur nächsten Eingabezelle, und die Prozedur wird wiederholt. Initialisierungen finden zu nicht näher bestimmten Zeitpunkten *nach* Produktion der beteiligten Ausgabezellen statt. Eine korrekte Reihenfolge der Schritte im Bild mit frühestmöglicher Initialisierung ist also:

$$p_1 t_1 p_2 i_1 t_2 p_3 i_2 t_3 p_4 i_3 t_4 \star i_4$$

Ein Zyklus kann durch Suche auf dem Aufrufstack nach wiederholten Funktionsargumenten erkannt werden. Wurde die dazugehörige produzierte Ausgabe ebenfalls auf dem Aufrufstack abgelegt, kann der Zyklus sofort geschlossen werden. Dies geschieht im obigen Ablauf an der mit $\star$ markierten Stelle. Wird der Zyklus nicht oder nicht an dieser Stelle erkannt, entsteht statt eines Zyklus eine Spirale $A'B'C'D'A''\ldots$. In einem nichtstrikten algebraischen Modell wird diese Divergenz in Kauf genommen und durch globale Verzögerung der Auswertungsstrategie kompensiert.

Der Verlust der Zyklenstruktur ist nicht tragisch, solange ein unendliches Datum nur als Generator endlicher Anfangsstücke betrachtet wird. Im folgenden Abschnitt wird jedoch ein Problem vorgestellt, für das die Zykleninformation von essentieller Bedeutung ist, und deren Verlust zur Undefiniertheit führt.

1.4 Coinduktives Entscheiden

Ein System voneinander abhängiger logischer Aussagen kann ebenfalls als Datenstruktur aufgefaßt werden, wobei eine Implikation $p \Rightarrow q$ als Abhängigkeitskante $q \rightarrow p$ dargestellt wird. Der Beweis einer Aussage p wird damit zum Suchproblem mit der Wurzel p.

Solange die Abhängigkeiten azyklisch sind, ist eine Entscheidungsprozedur und damit eine konsistente Belegung eindeutig festgelegt. Im Fall von zyklischen Abhängigkeiten können viele konsistente Belegungen existieren, die sich als Fixpunkte eines Deduktionsoperators charakterisieren lassen. Einer Tradition folgend, die sich bis zur griechischen Antike zurückverfolgen läßt, wird meistens der kleinste Fixpunkt bevorzugt, und insbesondere zur Semantik einer *induktiven* Definition erklärt. Eine *coinduktive* Definition dagegen hat natürlicherweise den größten Fixpunkt als Semantik.

Der kleinste Fixpunkt hat die Eigenschaft, daß nur endlich ableitbare Aussagen gelten, zyklisch abhängige folglich nicht. Führt man zur Entscheidung eine Suche aus und erkennt einen Zyklus, kann man ihn also korrekt als erfolglose Suche abbrechen. Im größten Fixpunkt dagegen gelten nur endlich widerlegbare Aussagen nicht, zyklisch abhängige folglich schon. Man implementiert ihn also ebenfalls durch Zyklenerkennung, allerdings wird ein Zyklus als Erfolg gewertet.

Ein besonders relevanter Spezialfall einer coinduktiven Definition ist die *Bisimilarität* oder Verhaltensgleichheit[1] coalgebraischer Elemente. Sie ist leicht entscheidbar, kann aber nicht effektiv als kleinster Fixpunkt dargestellt werden.

2 Technik

2.1 Operationen

Zur Implementierung coalgebraischer Berechnungen sind lediglich zwei unkonventionelle Operationen erforderlich: Eine zur *Erkennung* und eine zur *Reproduktion* von Zyklen.

[1] in der Objektorientierung, besonders bei Datenbanken, bekannt als *tiefe Gleichheit*

Erkennung Die Erkennung eines Zyklus erfolgt beim Aufruf einer Funktion durch Suche auf dem Aufrufstack nach einer identischen Inkarnation. Diese Suche kann entweder exakt oder bis auf Bisimilarität erfolgen, wobei zweiteres etwas mächtiger, aber auch mit erheblich mehr Aufwand verbunden ist.

Wird ein Zyklus erkannt, kann dieser behandelt werden. Dazu wird ein alternativer Funktionsrumpf betreten. Im Fall eines Fixpunktproblems liefert dieser *falsch* für kleinste und *wahr* für größte Fixpunkte. Im Fall einer corekursiven Funktion muß der Eingabezyklus reproduziert werden.

Reproduktion Diesem Zweck dient ein spezieller Ausdruck *dito*, welcher stets zu dem Ergebnis auswertet, das für den Eintrittspunkt des Zyklus auf dem Stack abgelegt wurde. Indem dieses auch Ergebnis des Austrittspunktes wird, ist der Ausgabezyklus geschlossen.

2.2 Optimierung

Für eine detaillierte Beschrweibung der möglichen Optimierungen sei auf [1] verwiesen.

Statische Optimierung Durch statische Analyse des Aufrufgraphen kann die Suche an garantiert nichtrekursiven Aufrufen abgeschnitten werden.

Dynamische Optimierung Durch geschickte Markierung von Zellen und Referenzen kann die Anzahl der SUchvorgänge auf einen pro Zyklus reduziert werden. Außerdem wird so die Elimination von *tail*-Rekursionen ermöglicht, die bei corekursiven Funktionen überaus häufig sind.

2.3 Implementierung

Anforderungen Das korrekte Arbeiten der *dito*-Operation setzt voraus, daß für jeden Corekursionsschritt die Wurzelzelle des Ergebnisses vor dem Abstieg feststeht. Die Initialisierung dieser Zelle muß also verzögert und gegebenenfalls an Unteraufrufe delegiert werden. Für die Klassen der *primitiv* corekursiven Funktionen ist diese Verzögerung immer möglich.

Die Trennung von Erzeugung und Initialisierung einer Zelle gilt allgemein als verdammenswerte Praktik. In die objektorientierte Programmierung wurden künstlich Konstruktoren eingeführt um zu verdecken, daß diese Trennung in Implementierungen stattfindet. Für die Konstruktion von Zyklen ist sie allerdings unbedingt notwendig.

Für die Speicherung des Ergebnisses muß Platz auf dem Stack vorgesehen sein, was bei den meisten konventionellen Zielplattformen nicht der Fall ist. Auch die Traversierung des Stacks als regulärer Bestandteil des Programmablaufs wird allgemein nicht unterstützt. Daher führt kaum ein Weg an der Definition einer virtuellen Maschine als maßgeschneiderte Ausführungsplattform vorbei.

Entwurf Die Zielplattform stellt eine einfache, statisch getypte Stack-Maschine dar, die an die Java- oder .NET-Maschinen angelehnt ist, aber nur über die für deklarative Programmierung notwendigen, minimalen Kontrollstrukturen verfügt. Als Eigenheiten besitzt sie eine Aufrufkonvention, welche den oben definierten Anforderungen zur Zyklenerkennung genügt, sowie alternative Funktionsrümpfe, welche auch die *dito*-Operation ausführen können. Die Delegation von Initialisierungen erfolgt, indem Ausgaben über Referenzparameter zugewiesen werden.

Stand der Entwicklung Die aktuelle Implementierung umfaßt das semantische Modell, also *byte-code* und Typsystem, eine textuelle Repräsentation mit zugehörigem Assembler, einen visuellen Interpreter, sowie einen Compiler nach C++. Die Erkennung und Behandlung von Zyklen wird voll unterstützt, inklusive der Auswertung zyklischer Konstanten zur Ladezeit. Die Interpreter-Umgebung ist voll reflektiv, so daß Typen und Funktionen als Datenstrukturen verarbeitet und zur Laufzeit erzeugt werden können. Parametrische Polymorphie und Funktionen höherer Ordnung werden ansatzweise unterstützt; dies ist der derzeitige Schwerpunkt der Weiterentwicklung.

Derzeit nicht vorhanden, aber geplant ist ein Verifikationswerkzeug, um Typkorrektheit und referentielle Transparenz von Programmen zu beweisen. Der Compiler ist ein minimaler Prototyp, der nicht optimiert und über keinerlei Laufzeitsystem, wie etwa automatische Speicherverwaltung, verfügt.

3 Anwendung

Die vorgestellten Techniken sollen nun auf praktische Standard-Probleme der Listenprogrammierung angewendet werden, allerdings in der Verallgemeinerung auf unendliche Listen. In einem strikten Berechnungsmodell bedeutet Unendlichkeit notwendigerweise Periodizität, da nur eine Liste mit endlich vielen *verschiedenen* Zellen tatsächlich *by-value* übergeben werden kann. Für weiterführende Beispiele auf periodischen Dezimalbrüchen sei auf [3] verwiesen.

Die Algorithmen werden in einer funktionalen Pseudosprache dargestellt, deren Syntax an der bekannten nichtstrikten Sprache Haskell orientiert ist. Das erlaubt eine relativ kompakte und selbsterklärende Formulierung. Die bekannte Syntax sollte aber nicht darüber hinwegtäuschen, daß die verwendete Berechnungstechnik eine strikte Auswertungsstrategie voraussetzt. Das bedeutet insbesondere, daß zu jedem Zeitpunkt der vollständige Stack der umgebenden Funktionsinkarnationen verfügbar ist, und daß deren Argumente in Normalform[2] vorliegen. Die Sprache wurde um einige wenige Elemente erweitert, die der Zyklenbehandlung dienen:

1. Das Pseudo-Pattern $\circlearrowleft$ akzeptiert jeden Argumentvektor, der bereits für eine umgebende Inkarnation derselben Funktion auf dem Stack liegt. Eine Gleichung mit $\circlearrowleft$ leistet also Zyklenerkennung.

[2] Volle Normalform für reine Daten, Kopf-Normalform für Funktionen

2. Der Pseudo-Wert **dito** repliziert das Ergebnis der mit $\circlearrowleft$ gefundenen umgebenden Inkarnation. Es wird vorausgesetzt, daß diese vor dem corekursiven Abstieg auf dem Stack abgelegt wurden.

Nicht alle Gleichungen der folgenden Algorithmen erfüllen die Bedingung, daß das Ergebnis vor dem corekursiven Abstieg festgelegt werden kann. Es handelt sich also nicht um nachweislich primitiv corekursive Funktionen. Wird trotzdem **dito** verwendet, wird der Algorithmus partiell. Darauf wird an gegebener Stelle näher eingegangen.

3.1 Map

Der Ablauf von *map* entspricht exakt dem oben diskutierten Beispiel für Corekursion. Wenn ein Zyklus erreicht ist, wird dieser geschlossen, ansonsten wird ein Element transformiert und der Rest der Liste corekursiv weiterverarbeitet. Bei endlichen Listen tritt der gewöhnliche Basisfall ein, und das Ende der Liste ist irgendwann erreicht.

$$
\begin{array}{lll}
map\ \circlearrowleft & = \textbf{dito} & \text{— Abbruch: unendlich} \\
map\ f\ [] & = [] & \text{— Abbruch: endlich} \\
map\ f\ (x\ :\ ys) & = f\ x\ :\ map\ f\ ys &
\end{array}
$$

3.2 Filter

Das Filtern von Listen gehört zu den ausgesprochenen Schwachstellen der nichtstrikten Sprachen. Dabei ist nicht von Belang, ob die Eingabeliste endlich oder undendlich ist. Probleme macht lediglich eine unendliche Folge von ausgefilterten Elementen: in diesem Fall ist das Verfahren nicht produktiv, und die Berechnung des nächstens Ausgabeelements gelingt nicht in endlicher Zeit.
Der Ablauf von *filter* gliedert sich in zwei Phasen:

$$filter\ p\ =\ filter_2\ .\ filter_1\ p$$

Die erste Phase invalidiert die auszufilternden Elemente, ohne die zugehörigen Listenzellen zu entfernen. Dadurch ist gewährleistet, daß das Prädikat p pro Element nur einmal ausgewertet wird.

$$
\begin{array}{lll}
filter_1\ \circlearrowleft & = \textbf{dito} & \text{— Abbruch: unendlich} \\
filter_1\ p\ [] & = [] & \text{— Abbruch: endlich} \\
filter_1\ p\ (x\ :\ ys) \mid p\ x & = Just\ x\ :\ filter_1\ p\ ys & \\
\quad \mid otherwise & = Nothing\ :\ filter_1\ p\ ys &
\end{array}
$$

Die zweite Phase streicht die invalidierten Elemente. Hier zunächst die naive Implementierung mit dem oben beschriebenen Terminierungsproblem:

$$
\begin{array}{lll}
filter_2\ \circlearrowleft & = \textbf{dito} & \text{— Abbruch: unendlich}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{filter}_2\ [] & = [] & \text{— Abbruch: endlich} \\
\mathit{filter}_2\ (\mathit{Nothing}\ :\ ys) & = \mathit{filter}_2\ ys & \\
\mathit{filter}_2\ (\mathit{Just}\ x\ :\ ys) & = x\ :\ \mathit{filter}_2\ ys &
\end{array}
$$

Eine unendliche Folge invalidierter Elemente kann nur die Form eines Zyklus haben. Dieser kann erkannt und durch die korrekte, leere Ergebnisliste ersetzt werden. Man ersetze also die zweite Gleichung durch:

$$\mathit{filter}_2\ xs \mid \mathit{nomore}\ xs = [] \quad \text{— Abbruch: unproduktiv}$$

Bleibt noch die Erkennung des Abbruchfalls zu implementieren:

$$
\begin{array}{lll}
\mathit{nomore}\ \circlearrowleft & = \mathit{True} & \text{— Abbruch: unendlich oft } \mathit{Nothing} \\
\mathit{nomore}\ [] & = \mathit{True} & \text{— Abbruch: endlich} \\
\mathit{nomore}\ (\mathit{Nothing}\ :\ ys) & = \mathit{nomore}\ ys & \\
\mathit{nomore}\ (\mathit{Just}\ x\ :\ ys) & = \mathit{False} &
\end{array}
$$

3.3 Quicksort

Das folgende Programm zeigt eine mögliche Implementierung des Quicksort-Algorithmus für endliche Listen:

$$
\begin{array}{lll}
\mathit{qsort}\ [] & = xs & \text{— Abbruch: endlich} \\
\mathit{qsort}\ xs & = \textbf{let} & \\
 & \quad p = \mathit{head}\ xs & \\
 & \quad ls = \mathit{filter}\ (<\ p)\ xs & \\
 & \quad es = \mathit{filter}\ (==\ p)\ xs & \\
 & \quad gs = \mathit{filter}\ (>\ p)\ xs & \\
 & \textbf{in} & \\
 & \mathit{qsort}\ ls\ +\!\!+\ es\ +\!\!+\ \mathit{qsort}\ gs &
\end{array}
$$

Die Verallgemeinerung auf unendliche Listen ist nicht trivial, da die rekursive Teilung nicht mehr zwangläufig auf eine kleinere oder dieselbe Liste führt. Die rekursiv zu sortierenden Teillisten enthalten aber garantiert weniger *verschiedene* Elemente. Daher kann gilt als im Unendlichen hinreichende Abbruchbedingung, daß die Liste unbeschränkt oft dasselbe Element enthält. Man ersetze also die Abbruchgleichung durch die folgende:

$$\mathit{qsort}\ xs \mid \mathit{allsame}\ xs = xs \quad \text{— Abbruch: trivial sortiert}$$

Bleibt noch die Implementierung des Abbruchprädikats. Dazu müssen alle Listenelemente mit dem ersten verglichen werden:

$$
\begin{array}{ll}
\mathit{allsame}\ [] & = \mathit{True} \\
\mathit{allsame}\ (x\ :\ ys) & = \mathit{allequal}\ x\ ys
\end{array}
$$

Dieser Vergleich ist ein einfacher größter Fixpunkt:

$$
\begin{array}{lll}
allequal\ \circlearrowleft & = True & \text{— Abbruch: unendlich} \\
allequal\ x\ [] & = True & \text{— Abbruch: endlich} \\
allequal\ x\ (y\ :\ zs) & = (x\ ==\ y)\ \&\&\ allequal\ x\ zs &
\end{array}
$$

Das Verketten zweier potentiell unendlicher Listen ergibt sich als triviale Verallgemeinerung des endlichen Algorithmus. Da nur die erste Liste traversiert wird, bedeutet ein Zyklus die Unendlichkeit der ersten Liste, und die zweite ist irrelevant. Der Zyklus kann also einfach geschlossen werden, und liefert somit eine Kopie der ersten Liste:

$$
\begin{array}{lll}
\circlearrowleft\ +\!\!+ & = \textbf{dito} & \text{— Abbruch: unendlich} \\
[]\ +\!\!+\ ys & = ys & \text{— Abbruch: endlich} \\
(x\ :\ xs)\ +\!\!+\ ys & = x\ :\ (xs\ +\!\!+\ ys) &
\end{array}
$$

Das Sortieren einer unendlichen Liste mag als akademische Spielerei gelten. Die folgenden Algorithmen sind aber in jedem Fall relevant, und erlauben die Manipulation periodischer Listen, wie sie in vielen Planungsproblemen nützlich sind, auf deutlich elegantere Art als die herkömmliche imperative Lösung über explizite Ringlisten.

3.4 Insert

Die folgende Funktion realisiert das periodische Einfügen eines Elementes x in die Liste ys alle n (Original-)Elemente und beginnend nach i Elementen:

$$
\begin{array}{lll}
insert\ \circlearrowleft & = \textbf{dito} & \text{— Abbruch: unendlich} \\
insert\ n\ i\ x\ [] & = [] & \text{— Abbruch: endlich} \\
insert\ n\ 0\ x\ ys & = x\ :\ insert\ n\ n\ x\ ys & \\
insert\ n\ i\ x\ (y\ :\ ys) & = y\ :\ insert\ n\ (i\ -\ 1)\ x\ ys &
\end{array}
$$

Dabei ist die Liste nicht das einzige potentiell unendliche Argument! Für $i = \omega$ (und folglich $i - 1 = \omega$) ergibt sich als Spezialfall das Kopieren der Liste ohne einzufügen. Für $n = \omega$ ergibt sich demnach als Spezialfall das einmalige Einfügen nach i Elementen, da danach nur noch kopiert wird.

Ein komplementärer Spezialfall ist $n = 0$. Hier wird, beginnend nach i Elementen der Eingabeliste, das Element x periodisch wiederholt.

3.5 Delete

Das Gegenstück zum periodischen Einfügen ist das periodische Löschen:

$$
\begin{array}{lll}
delete\ \circlearrowleft & = \textbf{dito} & \text{— Abbruch: unendlich} \\
delete\ n\ i\ [] & = [] & \text{— Abbruch: endlich} \\
delete\ n\ 0\ (y\ :\ ys) & = delete\ n\ (n-1)\ ys & \\
delete\ n\ i\ (y\ :\ ys) & = y\ :\ delete\ n\ (i-1)\ ys &
\end{array}
$$

Hier gilt ebenfalls, daß der Algorithmus für $i = \omega$ zum Kopieren und für $n = \omega$ zum einmaligen Löschen degeneriert. Für $n = 0$ ist der Ausdruck $n - 1$ und damit der Algorithmus undefiniert. Für $n = 1$ ist der Algorithmus unproduktiv; dieser Fall kann leicht gesondert behandelt werden, und das Ergebnis ist stets leer.

3.6 Select

Auch der Zugriff auf eine Liste kann als periodisch aufgefaßt werden:

$$
\begin{array}{lll}
\mathit{select}\ \circlearrowleft & = \mathbf{dito} & \text{— Abbruch: unendlich} \\
\mathit{select}\ n\ i\ [] & = [] & \text{— Abbruch: endlich} \\
\mathit{select}\ n\ 0\ (x\ :\ xs) & = x\ :\ \mathit{select}\ n\ (n-1)\ xs & \\
\mathit{select}\ n\ i\ (x\ :\ xs) & = \mathit{select}\ n\ (i-1)\ xs & \text{— unproduktiv}
\end{array}
$$

Für $n = 0$ ist der Algorithmus aus erwähntem Grund undefiniert. Für $n = 1$ degeneriert die Selektion zur Kopie.

Für $i = \omega$ oder $n = \omega$ ist der Algorithmus unproduktiv: Ein Zyklus wird zwar erreicht und erkannt, es ist aber kein replizierbares Ergebnis vorhanden, da die Corekursion über die letzte, unproduktive Gleichung erfolgt. Eine Implementierung muß entweder diesen Fall zur Laufzeit diagnostizieren, oder die Definition als nicht primitiv corekursiv zurückweisen.

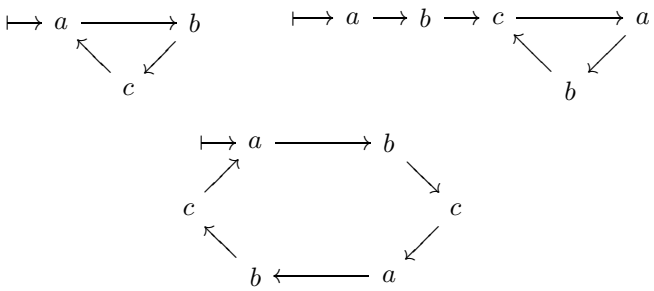
Gibt man aber die Analogie zu *insert* und *delete* auf und orientiert sich stattdessen an *filter*, dann läßt sich auch eine produktive Implementierung von *select* finden.

3.7 Ring

Mit Hilfe von *select* läßt sich auch elegant[3] bestimmen, ob eine Liste xs beginnend nach i Elementen eine Periodizität von n besitzt:

$$
\begin{array}{ll}
\mathit{ring}\ \circlearrowleft & = \mathit{True} \\
\mathit{ring}\ n\ i\ [] & = \mathit{False} \\
\mathit{ring}\ n\ 0\ xs & = \mathit{allsame}\ (\mathit{select}\ n\ 0\ xs)\ \&\&\ \mathit{ring}\ n\ 1\ xs \\
\mathit{ring}\ n\ i\ (x\ :\ xs) & = \mathit{ring}\ n\ (i-1)\ xs
\end{array}
$$

Dabei muß n nicht notwendigerweise die Periodizität n_0 der *Zellen* der Liste sein. Einerseits sind alle Vielfachen $k \cdot n_0$ ebenfalls gültige Periodizitäten, andererseits kann die Liste selbst mehrere Perioden der Elemente hintereinander oder ein Abwicklung vor dem eigentlichen Zyklus enthalten. Die folgenden Listen haben alle die minimale Periodizität 3:



[3] sprich: klar und ohne Rücksicht auf Effizienz

4 Zusammenfassung

In dieser Arbeit wurde dargestellt, wie sich wohlverstandene algebraische Berechnungstechniken natürlich auf coalgebraische Probleme fortsetzen lassen. Das Bindeglied ist hierbei der konstruktive Umgang mit Zyklen. Es wurde gezeigt, daß für eine Implementierung nur elementare Techniken erforderlich sind. Die resultierenden Berechnungsabläufe folgen Strategien, die eher dem imperativen als dem deklarativen Paradigma zugehörig scheinen.

Die strikte coalgebraische Berechnung verhält sich in Mächtigkeit und technischer Umsetzung komplementär zu Nichtstriktheit. Zwar erweitert die strikte Coalgebra nicht den Raum der implementierbaren Funktionen, aber die coalgebraische Darstellung ist für zyklische Probleme entschieden natürlicher und adäquater als eine explizite algebraische Codierung.

4.1 Zukünftige Arbeit

Die Nutzung der bekannten Optimierungspotentiale der Zyklenerkennung ist ein wesentliches ungelöstes Problem. Des weiteren ist die Qualität der Compilation verbesserungsfähig. Insbesondere wäre eine Laufzeitumgebung für compilierten Code mit automatischer Speicherverwaltung wünschenswert, um die Berechnungstechnik realistisch evaluieren und mit anderen Systemen vergleichen zu können.

Ein offenes Problem theoretischer Natur ist die Frage, inwieweit ein Stück imperativer *byte-code* ein deklaratives Programm implementiert, also referentiell transparent ist. Die statische Analyse scheint praktikabel, da die Maschine nur über stark eingeschränkten Kontrollfluß verfügt. Zu guter Letzt bleibt als Herausforderung der Entwurf einer funktionalen Programmiersprache, welche die Möglichkeiten der Maschine adäquat abbildet.

Literatur

1. B. Trancón y Widemann. Stacking cycles: Functional transformation of circular data. In *Implementation of Functional Languages*, volume 2670 of *LNCS*. Springer, 2002.
2. B. Trancón y Widemann. Advanced strict corecursion. In *Draft Proceedings of the IFL 2003*. Department of Computer Science, Heriot-Watt University, Edinburgh, 2003.
3. B. Trancón y Widemann. Advanced strict corecursion. In *Implementation of Functional Languages*, volume 3474 of *LNCS*. Springer, 2005.