Preface

On November 29, 2005 the Research Training Group (DFG-Graduiertenkolleg GK 334) "Specification of discrete processes and systems of processes by operational models and logics" at the TU Dresden organized the workshop

Methods of Category Theory in Software Engineering

in Dresden. Its scope can be characterized by the following keywords:

- Category Theory applied in Automata Theory
- Graph Transformations in a Category setting
- Dynamic, Modal, and Higher-order Logics.

This technical report contains the abstracts of the scientific talks which were presented by

H. Ehrig	B. Jacobs	H.-J. Kreowski	B. Krieg-Brückner
Chr. Lüth	T. Mossakowski	J.J.M.M. Rutten	L. Schröder.

The workshop was organized in the honour of **Prof. Dr. Horst Reichel**, who will retire end of March 2006 from his position as professor for algebraic and logical foundations of computer science at TU Dresden.

Horst Reichel has been an active member of our Research Training Group from its inception in 1997 to the present day. He has greatly contributed to the success of the group, not only by supervising several Ph.D. students, but also by strongly influencing the research direction of the group and by providing invaluable advice both to the students and to his colleagues. We will strongly miss his fruitful contributions to our scientific discussions, and his help in many different areas.

All members of the Research Training Group wish Horst Reichel a long and happy retirement, filled with all the things he has always been wanting to do (but couldn't since he had to go to our Tuesday seminars).

Dresden, November 29, 2005

Franz Baader	Manfred Droste	Bernhard Ganter
Steffen Hölldobler	Dietrich Kuske	Reinhard Pöschel
Michael Thielscher	Heiko Vogler	

Part I

Programme

Timetable

09.00	09.15	H. Vogler	*Opening*
09.15	09.50	J. Rutten	"On Stream Functions and Circuits"
09.50	10.25	H.-J. Kreowski	"A Categorial Approach to Automata"
10.25	11.00	—	*Coffee break*
11.00	11.35	C. Lüth	"Modular Modelling with Monads"
11.35	12.10	T. Mossakowski	"Monad-independent Logic for Computational Effects"
12.10	14.00	—	*Lunch break*
14.00	14.35	H. Ehrig	"Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation"
14.35	15.10	B. Jacobs	"Automata and Regular Languages"
15.10	15.35	—	*Coffee break*
15.40	16.15	L. Schröder	"Coalgebraic Modal Logic"
16.15	17.00	—	*Break*
17.00	18.00	—	*Laudatio*

Part II

Abstracts

Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation

Hartmut Ehrig

Technical University of Berlin
Berlin
Germany

ehrig@cs.tu-berlin.de

The theory of graph transformations is well-established since more than 30 years with several applications in theory and applications of Computer Science and related areas. Several variants of transformation systems based on different notions of graphs, Petri nets and algebraic specifications have been studied since about 15 years leading to the concept of high-level replacement (HLR) systems. In this lecture we introduce adhesive HLR systems as a new categorical framework for graph transformation in the double pushout (DPO) approach, which combines the well-known concept of HLR systems with the new concept of adhesive categories introduced by Lack and Sobocinski.

We show that most of the HLR properties, which had been introduced to generalize some basic results from the category of graphs to high-level structures, are valid already in adhesive HLR categories. This leads to a smooth categorical theory of HLR systems which can be applied to a large variety of graphs and other visual models. As a main new result in a categorical framework we show the Critical Pair Lemma for the local confluence of transformations. Moreover we present a new version of embeddings and extensions for transformations in our framework of adhesive HLR systems.

A detailed presentation of the classical algebraic theory of graph transformation systems and the new theory of adhesive HLR systems and typed attributed graph transformation systems is given in the new book [1].

References

[1] H. Ehrig, K. Ehrig, U. Prange, and G. Täntzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in TCS. Springer, 2005.

Automata and Regular Languages

Bart Jacobs

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.

www.cs.ru.nl/B.Jacobs

The talk reviews the classical theory of deterministic automata and regular languages from a categorical perspective. The basis is formed by RUTTEN's description of the BRZOZOWSKI automaton structure in a coalgebraic framework. We enlarge the framework to a so-called bialgebraic one, by including algebras together with suitable distributive laws connecting the algebraic and coalgebraic structure of regular expressions and languages. This culminates in a reformulated proof of KOZEN's completeness result, which can be seen as a complete axiomatisation of observational equivalence (bisimilarity) on regular expressions. We suggest that this situation is paradigmatic for (theoretical) computer science as the study of "generated behaviour".

A Categorial Approach to Automata

Hans-Jörg Kreowski

Fachbereich Mathematik und Informatik
Universität Bremen
Postfach 33 04 40, D-28334 Bremen

kreo@informatik.uni-bremen.de

Introduction

Various variants of automata have played prominent roles in several areas of computer science in the last ten to twenty years. In form of state charts, they are important kinds of UML diagrams (see, e.g., [5]). They provide as labelled transition systems the semantic foundation of concurrency and communication. Hybrid automata are used to combine discrete and continuous computation (see, e.g., [6]). As generalized sequential machines, they translate input languages into output languages yielding, for example, a characterization of recursively enumerable sets as translations of the twin shuffle language (see, e.g., [7]). And in model checking, automata model systems to be checked (see, e.g., [3]).

Therefore, it may be worthwhile to remember that various attempts were made in the 1970s toward a unified theory of automata in a categorical framework (see, e.g., [1, 2, 4]). In this paper, the basic ideas are recalled. The hope is that this may enhance the recent interest in automata.

Automata in a category

As a particular variant of automata in a category, Mealy-automata are introduced with output words instead of single outputs. The states, inputs and outputs may form just sets or sets with algebraic, probabilistic or topological structure. The state transitions and output assignments may be total functions, partial functions, relations, or structure-preserving mappings of some kind.

To cover this variety, let $\mathcal{K}$ be a monoidal category with tensor product $\otimes$ and a unit object $\mathbf{1}$. In many cases, the tensor product is based on the cartesian product of the underlying sets of the objects at hand. Moreover, it is assumed that $\mathcal{K}$ has countable coproducts that are preserved by the tensor product, i.e. $X \otimes \coprod_{i\in I} X_i = \coprod_{i\in I}(X \otimes X_i)$ and $(\coprod_{i\in I} X_i) \otimes X = \coprod_{i\in I}(X_i \otimes X)$ for all objects X and X_i, $i \in I$, where I is some countable index set. In many examples, coproducts are given by the disjoint unions of the underlying sets.

Based on these assumptions, one can construct the word monoid over some object X as the coproduct of all finite tensor products of X with itself.

$$X^* = \coprod_{i\in\mathbb{N}} X^n \quad \text{with } X^0 = \mathbf{1} \text{ and } X^{n+1} = X \otimes X^n \text{ for all } n \in \mathbb{N}.$$

The injections of X^n into X^* are denoted by in_n. The concatenation $\mu\colon X^* \otimes X^* \to X^*$ is the unique morphism induced by

$$\mu_0 = (X^0 \otimes X^* = \mathbf{1} \otimes X^* = X^* \overset{id}{\to} X^*) \quad \text{and}$$
$$\mu_{n+1} = (X^{n+1} \otimes X^* = X \otimes X^n \otimes X^* \overset{X\otimes\mu_n}{\to} X \otimes X^* \overset{incl}{\to} X^*)$$

where the inclusion $incl$ is induced by $incl_n = X \otimes X^n = X^{n+1} \overset{in_{n+1}}{\to} X^*)$. The unit $\eta\colon \mathbf{1} \to X^*$ is the injection $\mathbf{1} = X^0 \overset{in_0}{\to} X^*$.

One can now introduce generalized Mealy-automata. A *generalized Mealy-automaton* is a system $A = (I, O, S, d, l)$ where I, O and S are objects and $d\colon S \otimes I \to S$ and $l\colon S \otimes I \to O^*$ are morphisms. I is the input object, O the output object, S the state object, d the state transition and l the output assignment. One can define structure-preserving morphisms between automata in the usual way such that one gets a category of automata.

The state transition and the output assignment can be extended to input words yielding morphism $d^*\colon S \otimes I^* \to S$ and $l^*\colon S \otimes I^* \to O^*$. For the latter, one must assume that there is a natural morphism $t\colon X \to \mathbf{1}$ for each object X. In all example categories mentioned above, this is the total function from X into the singleton set $\mathbf{1}$. Moreover, one needs a diagonalization $\triangle\colon X \to X \otimes X$ for each object X, which doubles the argument in most cases. Then d^* is induced by $d_0 = (S \otimes I^0 = S \otimes \mathbf{1} = S \overset{id}{\to} S)$ and $d_{n+1} = (S \otimes I^{n+1} = S \otimes I \otimes I^n \overset{d\otimes I^n}{\to} S \otimes I^n \overset{d_n}{\to} S)$. And l^* is analogeously induced by $l_0 = (S \otimes I^0 \overset{t}{\to} \mathbf{1} {=} O^0 \overset{in_0}{\to} O^*)$ and

$$\begin{aligned} l_{n+1} = \quad & (S \otimes I^{n+1} = S \otimes I^n \otimes I \overset{\triangle\otimes I}{\to} S \otimes I^n \otimes S \otimes I^n \otimes I \\ & \overset{l_n\otimes d_n\otimes I}{\to} O^* \otimes S \otimes I \overset{O^*\otimes l}{\to} O^* \otimes O^* \overset{\mu}{\to} O^*). \end{aligned}$$

Input-output behaviour over closed categories

The extended state transitions and output assignments describe the input-output behaviour of the automata in an implicit way. A more explicit view on the semantics can be obtained if the underlying category $\mathcal{K}$ is closed, i.e. the functor $- \otimes X$ has a right adjoint functor $\langle X, -\rangle$ for every object X. Therefore, there is, for every object Y, a morphism $ev\colon \langle X, Y\rangle \otimes X \to Y$ such that, for every morphism $f\colon Z \otimes X \to Y$, a unique morphism $\overline{f}\colon Z \to \langle X, Y\rangle$ exists with $f = ev \circ \langle \overline{f} \otimes X\rangle$. Examples of closed categories are the categories of sets with total functions and with partial functions where $\langle X, Y\rangle$ is the set of all total resp. partial functions from X to Y and the universal morphism $ev\colon \langle X, Y\rangle \otimes X \to Y$ is the usual application of functions to arguments.

Closed categories provide a coalgebraic counterpart to the definiton of automata as heterogeneous algebras as given above and a semantic morphism corresponding to the extended output assignment. By using the universal properties of the assumed adjunction, each automaton $A = (I, O, S, d, l)$ induces the coalgebraic structure

$$\overline{A} = (I, O, S, \overline{d}\colon S \to \langle I, S\rangle, \overline{l}\colon S \to \langle I, O^*\rangle)$$

with $d = ev \circ (\overline{d} \otimes I)$ and $l = ev \circ (\overline{l} \otimes I)$ and the semantic morphism $SEM\colon S \to \langle I^*, O^*\rangle$ with $l^* = ev \circ (SEM \otimes I^*)$.

To get a full separation of the automata and their input-output behaviours, one may consider appropriate image factorizations of the semantic morphisms so that the images form a behaviour category and the semantics is expressed by a semantic functor from the category of automata into the behaviour category.

Input-output behaviour over pseudoclosed categories

Monoidal categories are not closed in general so that the consideration in the previous section does not always apply. This holds particularly for nondeterministic automata. But many categories with nondeterministic or stochastic functions or relations as morphisms are at least pseudoclosed. This means that the considered category $\mathcal{K}$ has got a closed subcategory $\mathcal{K}'$ (with the same class of objects and the same tensor product for objects) where the inclusion of $\mathcal{K}'$ into $\mathcal{K}$ has a right adjoint functor P. In examples, this functor is often the powerset functor or some variant of it. The universal property of this adjunction associates a unique K'-morphism $f' : X \to P(Y)$ to each $\mathcal{K}$-morphism $f : X \to Y$. In the example of power sets, this is just the one-to-one correspondence between binary relations on $X \times Y$ and the mappings that assign all related second components to each first component.

Combining the two adjunctions, automata in pseudoclosed categories allow one a coalgebraic view and an explicit behaviour construction, too.

Let $A = (I, O, S, d, l)$ be an automaton. Then its coalgebraic variant is defined by $\overline{A} = (I, O, S, \overline{d}\colon S \to \langle I, P(S)\rangle, \overline{l}\colon S \to \langle I, P(O^*)\rangle)$ with $d' = ev \circ (\overline{d} \otimes I)$ and $l' = ev \circ (\overline{l} \otimes I)$ where d' and l' are the $\mathcal{K}'$-morphisms corresponding to d and l resp. Analogously, one gets the semantic morphism $SEM\colon S \to \langle I^*, P(O^*)\rangle$ with $l^{*\prime} = ev \circ (SEM \otimes I^*)$ where $l^{*\prime}\colon S \otimes I^* \to P(O^*)$ corresponds to the extended output assignment $l^*\colon S \otimes I^* \to O^*$.

As the semantic morphisms are morphisms in the closed subcategory, one may use the same image factorization as in the previous section to turn semantic morphisms into a semantic functor.

Conclusion

Mealy automata (with output symbols instead of output words) are studied in some detail in the books by Budach and Hoehnke as well as by Ehrig, Kiermeier, Kreowski and Kühnel [2, 4]. In particular, reduction and minimization of automata are considered in various respects. The relations of reduction and minimization on one hand and bisimulation on the other hand may be investigated in future. In particular, this may shed some more light on the behaviour of automata in pseudoclosed categories including several types of nondeterministic automata.

References

[1] Michael A. Arbib, Ernest G. Manes. A categorist's view of automata and systems. In Ernest G. Manes, editor. Category Theory Applied to Computation and Control. Lecture Notes in Computer Science, vol. 25, pages 51-64, Springer, 1975.

[2] Lothar Budach, Hans-Jürgen Hoehnke. Automaten und Funktoren, Akademie-Verlag, 1975.

[3] E. Clarke, O. Grumberg, D. Peled. Model Checking. M.I.T. Press, 2001.

[4] Hartmut Ehrig, Klaus-Dieter Kiermeier, Hans-Jörg Kreowski, Wolfgang Kühnel. Universal Theory of Automata, Teubner, 1974.

[5] David Harel. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8,3. Pages 231–274, Springer, 1987.

[6] Thomas A. Henzinger. The theory of hybrid automata. In M.K. Inan, R.P. Kurshan, editors. Verification of Digital and Hybrid Systems. NATO ASI Series F: Computer and Systems Sciences, vol. 170, pages 265-292, Springer, 2000.

[7] Georghe Păun, Grzegorz Rozenberg, Arto Salomaa. DNA Computing: New Computing Paradigms. Springer, 1998.

Modular Modelling with Monads

Christoph Lüth

FB 3 — Mathematik und Informatik
Universität Bremen
Germany

http://www.informatik.uni-bremen.de/∼cxl

One useful application of category theory in computer science are *computational monads*, which are used to model diverse computational effects (such as stateful computations, exceptional behaviour or non-determinism). This goes back to MOGGI [6], and is for example used to great effect in the pure functional language Haskell [8, 11]. In this talk, we use the same technique to model computational effects in higher-order logic, and in particular the theorem prover Isabelle [7], thus shallowly embedding a small imperative language embedded shallow into Isabelle. The talk consists of three parts: we first recall the basic notions of monads, then show how the imperative language is built from three basic ingredients, and finally discuss the implementation.

Monads

A *monad* is the categorical modelling of an algebraic theory (that is, a set of operations and equations on them). One attraction of monads is that they come with a rich categorical model theory, giving a uniform treatment of both algebras and terms. The latter are given by the *Kleisi category* of a monad; this is where the computations modelled by the monad live. An adjunction between the base category and the Kleisli category gives us a notion of lifting every function from the base to a pure computation.

To combine monads, we can (under some mild preconditions on the base category) use *colimits* [3]. However, as the construction of colimits in general is quite intricate, two special cases of combinations of monads which have been considered: the coproduct, in particular of layered monads [5], and the tensor [1]. The former specifies that the sequential order of computations from the monads in question must be maintained, the latter specifies that they do not interact at all and hence their sequential order can be exchanged freely.

Modelling an Imperative Language

Our language is made up from three ingredients. Firstly, we have a core language with *iteration* and *case distinction.* The case distinction is given by coproducts from the base, and the iteration requires the existence of fixpoints.

Stateful computations can be added in two ways, either by an axiomatic description [9] or constructively by state threads [4]. The former allows us to model typed references and is more general, but may lead to arguments about consistency; the latter is more restrictively typed, but constructive and admits more equations.

Finally, exceptional behaviour is modelled by the exception monad, which is just the coproduct with a (fixed) set of exceptions.

Modelling in Isabelle

The modelling of monads in Isabelle uses a recent extension of Isabelle with parameterised theories and theory morphisms [2]. With these, we can model the monads as described above as parameterised theories and their relation by theory morphisms between them.

Syntax plays an important rôle when it comes to interactive theorem proving, as a good concise notation can make complicated proofs tractable. Therefore, we introduce a number of syntactic conventions (some as in Haskell) for monadic expressions. Moreover, we need a logic in which to reason about these; instead of e.g. Hoare logic [10] we choose a direct approach which lets us use Isabelle's automatic proof procedures.

Conclusions

We have shown how to concisely model a core imperative language which is short but, we argue, covers the essential features of most imperative languages. This can be implemented in a theorem prover to yield a shallow embedding of the core imperative language into higher-order logic; this in turn can be used to e.g. derive verification conditions, or prove program transformations. We plan to use our model in real-live program verification projects in the nearer future.

References

[1] M. Hyland, G. Plotkin, and J. Power. Combining computational effects: Commutativity and sum. In *TCS 2002, 2nd IFIP International Conference on Computer Science*, Montreal, 2002.

[2] E. B. Johnsen and C. Lüth. Theorem reuse by proof term transformation. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *International Conference on Theorem Proving in Higher-Order Logics TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer Verlag, Sept. 2004.

[3] G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. *Bulletins of the Australian Mathematical Society*, 22:1– 83, 1980.

[4] J. Launchbury and S. P. Jones. Lazy functional state threads. In *Proc. ACM Conference on Programming Languages Design and Implementation*, 1994.

[5] C. Lüth and N. Ghani. Composing monads using coproducts. In *International Conference on Functional Programming ICFP'02*, pages 133– 144. ACM Press, Sept. 2002.

[6] E. Moggi. Computational lambda-calculus and monads. In *Fourth Annual Symposium on Logic in Computer Science.* IEEE, Computer Society Press, June 1989.

[7] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer Verlag, 2002.

[8] S. Peyton Jones, editor. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[9] G. Plotkin and J. Power. Notions of computation determine monads. In M. Nielsen and U. Engeberg, editors, *Proc. FOSSACS 2002*, Lecture Notes in Computer Science 2303, pages 342– 356, 2002.

[10] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HasCASL. In M. Pezze, editor, *Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*, pages 261–277. Springer Verlag, 2003.

[11] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240– 263, 1997.

Monad-independent Logic for Computational Effects

Till Mossakowski

BISS, Department of Computer Science
University of Bremen
Germany

till@informatik.uni-bremen.de

The presence of computational effects, such as state, store, exceptions, input, output, non-determinism, backtracking etc., complicates the reasoning about programs. In particular, usually for each effect (or each combination of these), an own logic needs to be designed.

Monads, a well-known tool from category theory, have been used by Moggi [4] to model computational effects (in particular, all of those mentioned above) in an elegant way. In particular, state monads are used to emulate an imperative programming style in the functional programming language Haskell [8]. Also, a monad for effects in Java has been designed [1].

We have introduced a Hoare calculus [6] and a dynamic logic [7] that allow to reason about such effects. Semantics and proof calculus of both logics are given in a completely monad independent (and hence, effect independent) way. The particular (combination of) effects needed for an application can then be axiomatically specified in these logics.

In this work, we simplify the monad independent dynamic logic by detaching it from the HasCasl framework as used in [7]. Instead, we introduce a term language and semantics based on cartesian categories. The resulting calculus is shown to be sound and complete. (A completeness result was missing so far.)

Monads for computations

On the basis of the seminal paper [4], monads are being used for encapsulating side effects in modern functional programming languages; in particular, this idea is one of the central concepts of Haskell [2]. Intuitively, a monad associates to each type A a type TA of computations of type A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts away from particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can in fact be carried out independently of the choice of such a notion.

A monad on a given category $\mathbf{C}$ can be defined as a *Kleisli triple* $\mathbb{T} = (T, \eta, \_\_^*)$, where $T : \operatorname{Ob}\mathbf{C} \to \operatorname{Ob}\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $\_\_^*$ assigns to each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ such that

$$\eta_A^* = id_{TA}, \qquad f^*\eta_A = f, \qquad \text{and} \qquad g^*f^* = (g^*f)^*.$$

This description is equivalent to the more familiar one [3].

In order to support a language with finitary operations and multi-variable contexts (see below), one needs a further technical requirement: a monad is called *strong* if it is equipped with a natural transformation

$$t_{A,B} : A \times TB \to T(A \times B)$$

called *strength*, subject to certain coherence conditions (see e.g. [4]).

In [7], the notion of *simple* monad is defined. Roughly, an algebraic monad [3] is simple if, in each of its equations, the two sides contain the same variables. All of the following monads are simple.

Example 1 (see [4]) *Computationally relevant monads on* **Set** *(all monads on* **Set** *are strong) include*

- *stateful computations with possible non-termination:* $TA = (S \to ?\ (A \times S))$, *where* S *is a fixed set of states and* $\_\_ \to ?\ \_\_$ *denotes the partial function type;*
- *(finite) non-determinism:* $TA = \mathcal{P}_{\mathit{fin}}(A)$, *where* $\mathcal{P}_{\mathit{fin}}$ *denotes the finite power set functor;*
- *exceptions:* $TA = A + E$, *where* E *is a fixed set of exceptions;*
- *interactive input:* TA *is the smallest fixed point of* $\gamma \mapsto A + (U \to \gamma)$, *where* U *is a set of input values.*
- *non-deterministic stateful computations:* $TA = \mathcal{P}_{\mathit{fin}}(S \to (A \times S))$, *where, again,* S *is a fixed set of states;*

Dynamic logic

The right framework for reasoning about both partial correctness as well as termination or total correctness is dynamic logic as introduced in [5]. Here, we examine the infrastructure that is needed in order to develop dynamic logic in a monad-independent way, and show that this does indeed make sense when instantiated to the usual monads mentioned above.

Given a set S of *basic types*, the type system of monadic dynamic logic is generated by

$$A ::= 1 \,|\, \Omega \,|\, TA \,|\, A \times A \,|\, S$$

A *signature* $\Sigma = (S, F)$ consists of a set S of basic types and a set F of operation symbols $F : A \to B$, where A and B are types over S. The term language over a signature Σ and a context Γ of typed variables is given in Fig. 1. Repeated nestings such as do $x_1 \leftarrow p_1, \dots, x_n \leftarrow p_n$; q are somewhat inaccurately denoted in the form do $\bar{x} \leftarrow \bar{p}$; q. Term fragments of the form $\bar{x} \leftarrow \bar{p}$ are called *program sequences.* The quasi-type PDL stands for formulas of propositional dynamic logic. We let $[\bar{x} \leftarrow \bar{p}]\,\varphi$ abbreviate the formula $\Box$ do $\bar{x} \leftarrow \bar{p}$; φ, and omit the $\bar{x}$ if not occurring in φ. PDL formulas can be used in specifications; for examples, references and non-determinism have been specified in [7]. [9] contains numerous examples, including the Java monad and a parsing monad.

The language can be interpreted over a strong simple monad $\mathbb{T}$ over a cartesian category **C**. **C** is required to posses a distinguished object Ω such that $Hom(A, \Omega)$ is a Boolean algebra for all objects A. The basic sorts need to be interpreted as objects in **C**. This is

$$\text{(var)}\ \frac{}{\Gamma, x : A \rhd x : A} \qquad \text{(app)}\ \frac{f : A \to B \in \Sigma \quad \Gamma \rhd t : A}{\Gamma \rhd f(t) : B} \qquad (1)\ \frac{}{\Gamma \rhd * : 1}$$

$$\text{(pair)}\ \frac{\Gamma \rhd t : A \quad \Gamma \rhd u : B}{\Gamma \rhd \langle t, u \rangle : A \times B} \qquad \text{(fst)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd fst(t) : A} \qquad \text{(snd)}\ \frac{\Gamma \rhd t : A \times B}{\Gamma \rhd snd(t) : A}$$

$$\text{(do)}\ \frac{\Gamma \rhd p : TA \quad \Gamma, x : A \rhd q : TB}{\Gamma \rhd \text{do}\ x \leftarrow p;\ q : TB} \qquad \text{(ret)}\ \frac{\Gamma \rhd t : A}{\Gamma \rhd \text{ret}\, t : TA}$$

$$(\top)\ \frac{}{\Gamma \rhd \top : \Omega} \quad (\bot)\ \frac{}{\Gamma \rhd \bot : \Omega} \quad (\neg)\ \frac{\Gamma \rhd \varphi : \Omega}{\Gamma \rhd \neg\varphi : \Omega} \quad \text{similarly for } \wedge, \vee, \Rightarrow, \iff$$

$$(\Box)\ \frac{\Gamma \rhd \varphi : T\Omega}{\Gamma \rhd \Box\varphi : PDL} \qquad \text{(PDL)}\ \frac{\Gamma \rhd \varphi : PDL}{\Gamma \rhd \varphi : T\Omega}$$

Figure 1: Term language for propositional dynamic logic

easily extended to all types, giving an interpretation $[\![A]\!]$ for each type A. Likewise, basic operations need to be interpreted as morphisms in $\mathbf{C}$. Then a term $x_1 : A_1, \ldots x_n, A_n \rhd t : A$ can be interpreted as a morphism $[\![t]\!] : [\![A_1]\!] \times \cdots \times [\![A_n]\!] \to [\![A]\!]$, using the cartesian structure to interpret paring and projections, the monad to interpret *do* and ret, and the Boolean algebra structure to interpret the connectives. We also can interpret equations between terms as equations (or equalizers) of the arrows that are denoted by the terms. This will be needed in a moment.

PDL formulas will be interpreted as certain computations of type $T\Omega$. They are expected to have no side-effect, although they may e.g. read the state (if a notion of state is present in the monad). This is abstractly captured as follows.

A program p is called *discardable* if

$$(\text{do}\ y \leftarrow p;\ \text{ret}\, *) = \text{ret}\, *,$$

A program p is called *copyable* if

$$(\text{do}\ x \leftarrow p; y \leftarrow p;\ \text{ret}(x, y)) = \text{do}\ x \leftarrow p;\ \text{ret}(x, x).$$

p is called *deterministically side-effect free* if it is both discardable and copyable. Let PDL be the subobject of $T\Omega$ consisting of all deterministically side-effect free computations. Then $\mathbb{T}$ is said to *admit dynamic logic*, if there is an operation *Box* sending morphisms $A \to T\Omega$ to morphisms $A \to PDL$ (used to interpret $\Box$), such that

$$(\text{do}\ \bar{x} \leftarrow \bar{p}; a \leftarrow \Box\varphi;\ x_i \Rightarrow a) = \text{ret}\,\top \text{ iff } (\text{do}\ \bar{x} \leftarrow \bar{p}; a \leftarrow \varphi;\ x_i \Rightarrow a) = \text{ret}\,\top$$

Fig. 2 contains a calculus for dynamic that has been shown to be sound and complete (completeness requires restrictions on the signature, in particular, argument types of operations must not contain T). Compared with the calculus in [7], the rules for the diamond are omitted, because in classical logic, $\Diamond$ can be defined as $\neg\Box\neg$. Rules (cong), ($\Box$) and (CC) have been added for ensuring completeness. Axiom schema (CC) throws in all equalities that holds for cartesian categories, like $fst(\langle x, y \rangle) = x$. Axiom schema (Taut) includes all

Rules:

$$\text{(nec)}\ \frac{\varphi}{[\bar{x} \leftarrow \bar{p}]\,\varphi} \quad \begin{array}{c}\bar{x}\text{ not free}\\ \text{in assumptions}\end{array} \qquad \text{(mp)}\ \frac{\varphi \Rightarrow \psi; \quad \varphi}{\psi}$$

Axioms:

(K1)	$[\bar{x} \leftarrow \bar{p}]\,(\varphi \Rightarrow \psi) \Rightarrow [\bar{x} \leftarrow \bar{p}]\,\varphi \Rightarrow [\bar{x} \leftarrow \bar{p}]\,\psi$	
(K3$\Box$)	$\mathrm{ret}\,\phi \Rightarrow [p]\ \mathrm{ret}\,\phi$	
(seq$\Box$)	$[\bar{x} \leftarrow \bar{p}; y \leftarrow q]\,\varphi \iff [\bar{x} \leftarrow \bar{p}]\,[y \leftarrow q]\,\varphi$	
(ctr$\Box$)	$[x \leftarrow p; y \leftarrow q]\,\varphi \iff [y \leftarrow (\mathrm{do}\ \ x \leftarrow p;\ q)]\,\varphi$	$(x \notin FV(\varphi))$
(ret$\Box$)	$[x \leftarrow \mathrm{ret}\,a; \bar{z} \leftarrow \bar{q}]\,\varphi \iff [\bar{z} \leftarrow \bar{q}[a/x]]\,\varphi[a/x]$	
(cong)	$(\varphi \iff \psi) \implies (\chi[\varphi/x] \iff \chi[\psi/x])$	
($\Box$)	$[x \leftarrow \varphi]\,\psi \iff [x \leftarrow \Box\varphi]\,\psi$	$\varphi : T\Omega$
(CC)	$\varphi[t/x] \iff \varphi[u/x]$	for $CC \vdash t = u$
(Taut)	φ	φ a lifted tautology

$$CC = \{fst(\langle x, y\rangle) = x;\ snd(\langle x, y\rangle) = y;\ \langle fst(x), snd(x)\rangle = x;\ x : 1 = *\}$$

Figure 2: The generic proof calculus for propositional dynamic logic

tautologies lifted to PDL: The usual logical connectives are lifeted to PDL by defining e.g.

$$\varphi \Rightarrow \psi := \Box(\text{do } a \leftarrow \varphi; b \leftarrow \psi;\ \text{ret}(a \Rightarrow b)) : PDL$$

The PDL logic has so far been applied to the reasoning about Haskell and Java programs. This is supported by a coding in the theorem prover Isabelle [9]. A Hoare calculus can be built on top of PDL.

Acknowledgements

Thanks to Lutz Schröder for discussions and comments, and Erwin R. Catesbeiana for pointing out various pitfalls.

References

[1] B. Jacobs and E. Poll. Coalgebras and Monads in the Semantics of Java. *Theoret. Comput. Sci.*, 291:329–349, 2003.

[2] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Haskell 98: A non-strict, purely functional language. 1999. `http://www.haskell.org/onlinereport`.

[3] S. Mac Lane. *Categories for the Working Mathematician.* Springer, 1997.

[4] E. Moggi. Notions of computation and monads. *Inform. and Comput.*, 93:55–92, 1991.

[5] V. Pratt. Semantical considerations on Floyd-Hoare logic. In *Foundations of Conputer Science*, pages 109–121. IEEE, 1976.

[6] L. Schröder and T. Mossakowski. Monad-independent Hoare logic in HasCasl. In *Fundamental Aspects of Software Engineering*, volume 2621 of *LNCS*, pages 261–277, 2003.

[7] L. Schröder and T. Mossakowski. Monad-independent dynamic logic in HasCasl. *J. Logic Comput.*, 14:571–619, 2004.

[8] Philip Wadler. How to declare an imperative. *ACM Computing Surveys*, 29:240–263, 1997.

[9] Dennis Walter. Monadic dynamic logic: Application and implementation. Master's thesis, University of Bremen, 2005.

On Stream Functions and Circuits

Jan Rutten

CWI and VU
Amsterdam
The Netherlands

`janr@cwi.nl`

Abstract

It is with great pleasure that I dedicate this little note on stream functions and circuits to *Horst Reichel*, as a sign of my appreciation and gratitude for his work in general and his contributions to the workshop on coalgebraic methods in computer science in particular.

In this note, we study the behaviour of circuits of the following type:

$$\begin{tikzcd}
 & \circ \arrow[d, mapsto] & \circ \arrow[l, mapsto, "R"'] & \circ \arrow[l, mapsto, "a"'] & \\
\sigma \arrow[r, mapsto] & + \arrow[r] & \circ \arrow[r, mapsto] & C \arrow[u] \arrow[r] & \tau
\end{tikzcd}$$

Such a circuit inputs elements of a given set A one by one, at every step producing an output element in A at the same time. The behaviour of circuits will be described by *streams* (infinite sequences) of inputs and outputs, since the nth output will generally depend not only on the nth input, but also on all earlier inputs. Therefore we shall use stream functions $f\colon A^\omega \to A^\omega$ to express the stream of outputs as a function of the stream of inputs: $\tau = f(\sigma)$.

Circuits are built from the following four basic ingredients: a sum or $+$-gate inputs two streams and outputs their sum; a copier or C-gate inputs one stream and outputs two identical copies; a register or R-gate is a one-place memory cell with initial value 0, which inputs a stream and outputs first the initial value 0 and then, with a one step delay, the input stream; finally, an a-multiplier or a-gate multiplies its input stream elementwise by a ($\in A$). We shall in fact distinguish between three different interpretations or types of such circuit diagrams, called:

(1) signal flow graphs

(2) linear sequential machines

(3) sequential arithmetic circuits

The graphical syntax of all these circuits is the same. The difference between them consists of the type of streams they process: signal flow graphs work on streams of real (or complex) numbers whereas both linear sequential machines and sequential arithmetic circuits work on bitstreams (infinite sequences of 0's and 1's). Moreover, the interpretation of the $+$-gate is different for each type of circuit. It computes,

(1) in signal flow graphs, the elementwise sum $(\sigma + \tau)(n) = \sigma(n) + \tau(n)$ of real (or complex) numbers;

(2) in linear sequential machines, the elementwise sum modulo-2 $(\sigma+\tau)(n) = \sigma(n)\oplus\sigma(n)$ of Boolean values;

(3) in sequential arithmetic circuits, the sequential addition of bitstreams (see below).

Circuit types (1) and (2) are well-known in the literature (cf. [2] and [1]). Circuits of type (3) have been described in [3], but might very well be older than that[2]. All of these circuits are interesting for a number of reasons. They have various kinds of applications such as signal processing, switching theory, and binary arithmetic, respectively. Furthermore, they are interesting as computational devices because of the presence of memory (in the form of registers) and infinite behaviour (in the form of loops). Finally, for all three types of circuits there exists a simple and elegant mathematical description. It is the latter point with which we shall occupy ourselves here. Our main contribution will be the presentation of one uniform model for all three types of circuits at the same time.

Traditionally, circuits of types (1) and (2) are modelled with the help of *formal power series*

$$\sum_{k\geq 0} a_k D^k = a_0 + (a_1 \times D) + (a_2 \times D^2) + \cdots$$

where D is a "formal" variable (symbol) and the coefficients a_i are typically taken from some semiring or field A. Formal power series are used as representations of the streams of their coefficients $(a_0, a_1, a_2, \ldots)$. The reason that formal power series are a suitable means to describe circuits of types (1) and (2) is, in essence, two-fold. (i) First, the behavior of a register can be simply modelled by (symbolic) multiplication by D (which is also called the delay operator):

$$\alpha = \sum_{k\geq 0} a_k D^k = (a_0, a_1, a_2, \ldots) \;\longmapsto\!\!R\!\!\rightarrow\; (0, a_0, a_1, a_2, \ldots) = \sum_{k\geq 0} a_k D^{k+1} = D \times \alpha$$

(ii) Second, the +-gate can be modelled by addition of formal power series, which is defined as the elementwise addition of the respective coefficients, in the underlying semiring or field A.

The formal power series approach fails, however, for sequential arithmetical circuits. Here addition gives rise to a delay in the output because of a so-called 'carry' bit, as in

$$(1, 0, 0, 0, \ldots) + (1, 0, 0, 0, \ldots) = (0, 1, 0, 0, 0, \ldots)$$

which represents the computation $1 + 1 = 2$. As a consequence, addition of input streams is not elementwise, violating condition (ii) above.

Therefore we propose to use streams, and only streams, instead, to deal uniformly with circuits of all three types at the same time. Notably, for all of them we can define the operation of sum as a binary operation on streams. The elementwise definitions of sum and sum modulo-2 for the operation of addition in signal flow graphs and linear sequential machines was already given above. Sequential binary addition on bitstreams is not elementwise, but can be conveniently defined in terms of so-called *behavioural (stream) differential equations.* We define the stream *derivative* of $\sigma = (\sigma_0, \sigma_1, \sigma_2, \ldots)$ by $\sigma' = (\sigma_1, \sigma_2, \sigma_3 \ldots)$ and the *initial value* of σ by $\sigma(0) = \sigma_0$. Then we can define streams and stream functions

[2] I am very much interested in relevant references.

by means of differential equations and initial values, very much as in classical analysis. For the details of the differential equation for sequential binary addition, we refer to [3].

Having a binary operation of stream addition, for all three circuit types, one can define in all three cases corresponding notions of multiplication (which in essence is repeated addition), minus, and (multiplicative) inverse. If one embeds the set A into A^ω by defining $[a] = (a, 0, 0, 0, \ldots)$, for all $a \in A$, one also has constants $[0]$ and $[1]$. As it turns out, we obtain (three different versions of) an integral domain

$$(A^\omega,\ +,\ -,\ \times,\ (-)^{-1},\ [0],\ [1])$$

in which every stream $\sigma \in A^\omega$ with initial value $\sigma(0) \neq 0$ has an inverse. Sofar we already have all that is needed to model the behaviour of $+$-gates, C-gates and a-gates:

$$\begin{tikzcd}
\sigma \arrow[dr, bend left, no head] & & \\
& + \arrow[r] & \sigma + \tau \\
\tau \arrow[ur, bend right, no head] & &
\end{tikzcd}
\qquad
\begin{tikzcd}
& & \sigma \\
\sigma \arrow[r, no head] & C \arrow[ur, bend left] \arrow[dr, bend right] & \\
& & \sigma
\end{tikzcd}
\qquad
\begin{tikzcd}
\sigma \arrow[r, mapsto, "a"] & {[a] \times \sigma}
\end{tikzcd}$$

In order to model registers, we define $X = (0, 1, 0, 0, 0, \ldots)$. It will play the role of the "formal" variable D of formal power series above, but note that X is simply a constant stream. Multiplication with X gives a one step delay (in all three integral domains), which is precisely what we need for our R-gates:

$$\sigma = \ (\sigma_0, \sigma_1, \sigma_2, \ldots) \ \xmapsto{R} \ (0, \sigma_0, \sigma_1, \sigma_2, \ldots) = \ X \times \sigma$$

We can now give the mathematical semantics of our circuits, without the need to distinguish between the three different types. All that we shall use is the fact that A^ω is an integral domain together with the input-output characterisations of the four basic gates. For instance, consider again the circuit we mentioned at the beginning. In order to compute the output stream τ as a function of the input stream σ, we insert intermediate streams ρ_i as follows:

$$\begin{tikzcd}
& \rho_1 \arrow[d, no head] & \rho_2 \arrow[l, mapsto, "R"'] & \rho_3 \arrow[l, mapsto, "a"'] & \\
\sigma \arrow[r, no head] & + \arrow[r] & \rho_4 \arrow[r, no head] & C \arrow[u] \arrow[r] & \tau
\end{tikzcd}$$

This enables us to write down the following equations:

$$\rho_1 = X \times \rho_2,\ \rho_2 = [a] \times \rho_3,\ \rho_3 = \rho_4,\ \rho_4 = \sigma + \rho_1,\ \tau = \rho_4$$

leading to the following result, in which the output τ is expressed as a function of the input σ:

$$\tau = \frac{1}{1 - ([a] \times X)} \times \sigma$$

In order to give a general characterisation of arbitrary finite circuits (of again any of our three circuit types), we use the constant X for the definition of *polynomial* streams, which are of the form

$$a_0 + a_1 X + a_2 X^2 + \cdots + a_n X^n = \ (a_0, a_1, a_2, \ldots a_n, 0, 0, 0, \ldots)$$

for all $a_0, \ldots a_n \in A$ (omitting square brackets around the coefficients and simply writing a_i for the stream $[a_i]$). Correspondingly, we call a stream *rational* if it is the quotient of two polynomial streams (with the initial value of the denominator stream different from 0). Now we can formulate the following theorem.

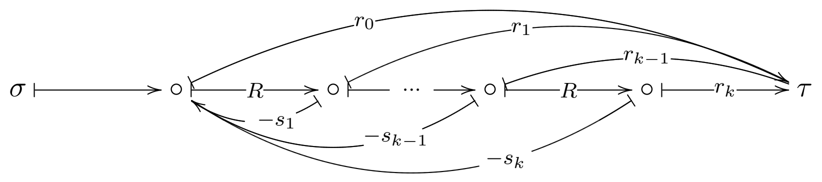
Theorem 2

(a) For every finite circuit, in which feedback loops always pass through at least one register gate, the output stream is obtained from the input stream by an application of a stream function $f\colon A^\omega \to A^\omega$ of the form, for all $\sigma \in A^\omega$: $f(\sigma) = \rho \times \sigma$, for a fixed rational stream $\rho \in A^\omega$.

(b) Conversely, any function of the form $f(\sigma) = \rho \times \sigma$, with

$$\rho = \frac{r_0 + (r_1 \times X) + \cdots + (r_{k-1} \times X^{k-1}) + (r_k \times X^k)}{1 + (s_1 \times X) + \cdots + (s_{k-1} \times X^{k-1}) + (s_k \times X^k)}$$

(where $r_i, s_j \in A$) can be implemented by a circuit of the following normal form:



(In this diagram, we use the graphical convention that arrow heads arriving in the same node are actually connected by +-gates, which are not drawn. Similarly, we omit C-gates.)

In [3], a proof of this theorem is given for the case of sequential arithmetic circuits (called 2-adic linear circuits there). The contribution of the present note is the observation that this proof, which is based on stream calculus, generalises the approach using formal power series, and works for circuits of all three types at the same time.

References

[1] Z. Kohavi. *Switching and Finite Automata Theory.* McGraw-Hill, 1978.

[2] B.P. Lahti. *Signal Processing & Linear Systems.* Oxford University Press, 1998.

[3] J.J.M.M. Rutten. Algebra, bitstreams, and circuits. In *Proceedings of the Dresden Conference 2004 (AAA68)*, volume 16 of *Contributions to General Algebra*, pages 231–250. Verlag Johannes Heyn, 2005.

Coalgebraic Modal Logic

Lutz Schröder

Department of Computer Science
University of Bremen
Germany

lschrode@informatik.uni-bremen.de

Since the seminal work of Hennessy and Milner [4], it has been apparent that reactive systems and modal logic are tightly connected. In particular, modal logic is suited as a specification language for reactive systems in that it both respects and characterizes behavioral equivalence. In recent years, coalgebra has emerged as an appropriate framework for the treatment of reactive systems in a vastly more general sense than the traditional concept of labelled transition systems [16], covering also as diverse system types as probabilistic automata, multigraphs, branching systems, and linear automata, to name just a few; this variation is achieved by parametrizing the theory by a so-called *signature functor*, which determines a datatype in which collections of successor states are organized. Despite its generality, coalgebra has proved suitable for both the definition of the central concepts of concurrency and for the proof of non-trivial results about them. E.g, coalgebra has provided a unifying perspective on notions such as coinduction, corecursion, and bisimulation.

The search for a coalgebraic analogue of Hennessy-Milner logic [5, 14, 6, 11] has led to the definition of what we shall refer to as *coalgebraic modal logic* by Pattinson [13]; this definition is based on the crucial notion of *predicate liftings*, which transform predicates on X into predicates on TX, where T is the signature functor. In comparison with the previously defined *coalgebraic logic* [9], coalgebraic modal logic is relatively easily understood and therefore usable in actual software specification; variants of coalgebraic modal logic indeed appear as features in the algebraic-specification language CoCasl [10] and in the object-oriented specification language CCSL [15]. Coalgebraic modal logic subsumes, besides standard Hennessy-Milner logic, well-known non-normal modal logics such as graded modal logic or probabilistic modal logic [8, 3]; some still relatively natural examples are even non-monotone. Coalgebraic modal logic is invariant under behavioral equivalence. Moreover, it has been shown in [13] that under suitable conditions, foremost accessibility of the signature functor (essentially a restriction on the branching degree) and the existence of ‘enough’ predicate liftings, coalgebraic modal logic is *expressive*, i.e. logical equivalence of states implies their behavioral equivalence.

The latter result has been improved in [17], where it was shown that expressivity really requires only the two explicitly mentioned essential conditions, rather than also further technical side conditions. Moreover, examples were given showing that there exist signature functors which do not admit an expressive *unary* modal logic, i.e. have too few (unary) predicate liftings. This deficiency can be remedied by moving on to *polyadic* modal logic: every accessible functor has enough polyadic predicate liftings and therefore admits an expressive polyadic modal logic. For example, an expressive logic for branching processes requires a binary modal operator which describes the next-step behaviors of both branches.

Moreover, polyadic modal logic has the added advantage of being closed under the modular construction of logics for coalgebras [2].

For various reasons, many coalgebraic modal logics fail to be compact [18], so that generic completeness results will necessessarily be limited to weak completeness, unless further restrictions are imposed on the signature functor [7]. Weak completeness for a deduction system consisting of propositional reasoning, a congruence rule, and a given set of axioms of rank 1 has been established in [12], subject to the condition that the set of axioms is complete for single-step semantic consequence; the latter property is referred to as *reflexivity*. It is shown in [18] that reflexively axiomatized coalgebraic modal logic has finite models for formally consistent formulae, a result which has weak completeness as a corollary. Moreover, it is shown that the set of all valid formulae of rank 1 is always reflexive, which then implies that coalgebraic modal logic has the *finite model property*: every satisfiable formula is satisfiable in a finite model of exponentially bounded size.

The finite model result relies on the construction of small canonical models, based on maximally consistent subsets of closed sets of formulae. The associated truth lemma can be formulated in terms of Hintikka sets; this allows the design of a generic decision algorithm for coalgebraic modal logic, which reduces the satisfiability problem to the rather simpler *one-step satisfiability* problem. One thus not only arrives at proving the decidability of a large number of modal logics, but also obtains *NEXPTIME* as a generic complexity bound, provided the one-step satisfiability problem is in *NP*. This applies in particular to Hennessy-Milner logic, graded modal logic, and probabilistic modal logic. While the former two examples are known to be *PSPACE*-complete [1, 19], no complexity bound has so far been given for probabilistic modal logic. We conjecture that the generic result can be improved to *PSPACE*; however, an adaption of the standard witness algorithm for K to the general case seems infeasible, so that new methods, possibly along the lines of [19], are required.

Ongoing work in collaboration with D. PATTINSON is concerned with a finite model construction, and thus a completeness proof, for an extended language featuring an iteration modality generalizing the *always* operator of temporal logic. Such an extension, which, in a less general framework, already features in the present design of COCASL, strongly enhances the usability of coalgebraic modal logic in system specification.

References

[1] P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic.* Cambridge, 2001.

[2] C. Cîrstea and D. Pattinson. Modular construction of modal logics. In *Concurrency Theory*, volume 3170 of *LNCS*, pages 258–275. Springer, 2004.

[3] A. Heifetz and P. Mongin. Probabilistic logic for type spaces. *Games and Economic Behavior*, 35:31–53, 2001.

[4] M. Hennessy and R. Milner. Algebraic laws for non-determinism and concurrency. *J. ACM*, 32:137–161, 1985.

[5] B. Jacobs. Towards a duality result in the modal logic of coalgebras. In *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, 2000.

[6] A. Kurz. Specifying coalgebras with modal logic. *Theoret. Comput. Sci.*, 260:119–138, 2001.

[7] A. Kurz, 2005. Personal communication.

[8] K. Larsen and A. Skou. Bisimulation through probabilistic testing. *Inform. Comput.*, 94:1–28, 1991.

[9] L. Moss. Coalgebraic logic. *Ann. Pure Appl. Logic*, 96:277–317, 1999.

[10] T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-co-algebraic specification in CoCasl. *J. Logic Algebraic Programming.* To appear.

[11] D. Pattinson. Semantical principles in the modal logic of coalgebras. In *Symposium on Theoretical Aspects of Computer Science*, volume 2010 of *LNCS*, pages 514–526. Springer, 2001.

[12] D. Pattinson. Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. *Theoret. Comput. Sci.*, 309:177–193, 2003.

[13] D. Pattinson. Expressive logics for coalgebras via terminal sequence induction. *Notre Dame J. Formal Logic*, 45:19–33, 2004.

[14] M. Rößiger. Coalgebras and modal logic. In *Coalgebraic Methods in Computer Science*, volume 33 of *ENTCS*. Elsevier, 2000.

[15] J. Rothe, H. Tews, and B. Jacobs. The Coalgebraic Class Specification Language CCSL. *J. Universal Comput. Sci.*, 7:175–193, 2001.

[16] J. Rutten. Universal coalgebra: A theory of systems. *Theoret. Comput. Sci.*, 249:3–80, 2000.

[17] L. Schröder. Expressivity of coalgebraic modal logic: the limits and beyond. In *Foundations of Software Science And Computation Structures*, volume 3441 of *LNCS*, pages 440–454. Springer, 2005.

[18] L. Schröder. A finite model construction for coalgebraic modal logic. Technical report, University of Bremen, 2005.

[19] S. Tobies. *PSPACE* reasoning for graded modal logics. *J. Logic Computation*, 11:85–106, 2001.