Electronic Communications of the EASST
Volume 26 (2010)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday

Algebraic Model Checking

Peter Padawitz

22 pages

Guest Editors: Frank Drewes, Annegret Habel, Berthold Hoffmann, Detlef Plump
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Algebraic Model Checking

Peter Padawitz

TU Dortmund, Germany

Abstract: Three algebraic approaches to model checking are presented and compared with each other with respect to their range of applications and their degree of automation. They have been implemented and tested in our Haskell-based formal-reasoning and -presentation system Expander2. Besides realizing and integrating state-of-the-art proof and computation rules the system admits the co/algebraic specification of the models to be checked in terms of rewrite rules and functional-logic programs. It also offers flexible features for visualizing and even animating models and computations. This paper does not present purely theoretical work. Due to the increasing abstraction potential of programming languages like Haskell, traditional gaps between specification formalisms and their executable implementations as well as between systems developed in different communities are going to vanish. The topics discussed in this article and the way of their presentation reflect this fact.

Keywords: model checking, algebra, coalgebra, functional programming, induction, coinduction, fixpoint theorems

1 Introduction

Model checking means proving properties of labelled or unlabelled transition systems (TRS). Modal, temporal or dynamic logics have been developed to formalize the properties and provide methods for proving them (see, e.g., [4, 14, 29]). In contrast to classical predicate logic, modal logics hide the relations (here: the transition systems) they are talking about. Translations of the latter into the former are well-known (see, e.g., [1, 20]), but did not affect very much the direction of research in model checking. With the invention of *coalgebraic* logics (see, e.g., [15, 28, 16, 11, 2]) the direction of translation is reversed: these logics generalize the 'relation-hiding' concept of modal logics from merely unstructured states and transitions to arbitrary *destructor-based* types and thus open up alternatives to classical predicate-logic-based data type verification. Moreover, the use of coalgebraic concepts reveals the intrinsic algebraic flavor of modal logics: their formulas denote (unary) relations; the logical operators (including fixpoint operators!) are functions building relations from relations. This approach is sometimes called "global" in contrast to the "local" one that starts out from a satisfiability relation between states and formulas (see, e.g., [18]). Mathematically, both views on the semantics of modal logics are equivalent: the global one just turns the satisfiability relation of the local view into a function from formulas to sets of states. In both cases the data modal logics deal with states and thus with elements of a destructor-based type, or with *paths*, which also form a destructor-based type.

We have investigated and implemented in the prover part of Expander2 [22, 23, 24] four approaches to model checking. The first one may be called purely algebraic because the proof of a formula boils down to *term evaluation*. In the second one, formulas are proved by solving sets of

regular equations represented by *data flow graphs*. The third technique uses *simplification* rules and must accompany the first one if, for instance, the underlying type has infinitely many elements (such as the set of paths of a TRS). The fourth method applies *co/Horn logic*, extends the others by powerful inference rules (mainly *parallel co/resolution* and *incremental co/induction*) and thus imposes the fewest restrictions on the formulas to be proved. On the other hand, this technique requires more manual control of the proof process than the others.

For lack of space the present paper skips the data flow approach. The other methods are illustrated mainly with a couple of axiomatic specifications of small Kripke structures and the verification of properties given by *state* or *path formulas*. More and larger examples can be found in [25] and the *Examples* directory of Expander2. This system has also created all graphics and proof records presented in this paper. To a great extent, Expander2 specifications follow the syntax of the functional programming language Haskell (see haskell.org) with which we assume a little familiarity. We also use Haskell for some definitions that involve data structures like lists or trees. Neither a purely set-theoretical notation nor an—unfortunately still prevailing—imperative syntax can cope with the elegance and adequacy of Haskell.

Although it is long ago, the extremely inspiring work with Hans-Jörg Kreowski (and my supervisors Hartmut Ehrig and Dirk Siefkes) at the computer science department of the Technical University of Berlin, lasting from 1974 to 1983, have influenced the direction of my research over the entire subsequent 25 years. We worked in three areas: automata theory, graph grammars and algebraic software specification. In all of them, constructions and methods from universal algebra played the key rôle. My additional work on Horn logic and rewrite systems was also led by the algebraic viewpoint. Last not least, graph grammar concepts left their mark on the treatment of term graphs in Expander2.

2 Kripke structures in Expander2

Since we want to use the same techniques for several variants of transition systems and modal logics, the following definitions take into account deterministic *and* nondeterministic, labelled *and* unlabelled systems as well as state *and* path formulas:

A **Kripke structure** $K = (Q, At, Lab, trans, transL, value, valueL)$ consists of sets Q, At, Lab of **states**, At of **atoms** and Lab of **labels** (actions, input, output, etc.), respectively, **transition relations** $trans : Q \to \wp(Q)$ and $transL : Q \times Lab \to \wp(Q)$ and **atom valuations** $value : At \to \wp(Q)$ and $valueL : At \times Lab \to \wp(Q)$. Usually, either $trans$ or $transL$ and either $value$ or $valueL$ are empty. For an empty $transL$, the set of *paths* of K is given by

$$path(K) \;=\; \{p \in Q^{\mathbb{N}} \mid \forall\, i \in \mathbb{N} : p_{i+1} \in trans(p_i)\} \cup \bigcup_{n \in \mathbb{N}} \{p \in Q^n \mid \forall\, 1 \le i < n : p_{i+1} \in trans(p_i)\}$$

and analogously for an empty $trans$. Given a function $f : Q \to \mathscr{P}(Q)$,

$$\begin{array}{lcl} imgsShares(qs)(f)(qs') & = & \{q \in qs \mid f(s) \cap qs' \neq \emptyset\}, \\ imgsSubset(qs)(f)(qs') & = & \{q \in qs \mid f(s) \subseteq qs'\} \end{array}$$

denote the sets of states $q \in qs$ such that at least one resp. all f-images of q are in qs'. Expander2 admits the specification of Kripke structures in terms of rewrite rules as in the following example.

```
-- TRANS
defuncts:  drawFT                    defined functions
fovars:    x y                       first-order variables
axioms:    states == [0] &           initial states
           (x < 6 & x `mod` 2 = 0    ==> x -> branch$[x,x+1]) &
           (x < 6 & x `mod` 2 =/= 0 ==> x -> x+1) &
           6 -> branch$[1..5]++[7..10] &
           7 -> 14 &
           drawFT == wtree$fun(sat$x,frame$text$x,x,x)
```

After *TRANS* has been parsed, the simplifier of Expander2 constructs a Kripke model from a list of initial states (here: [0]) and (Horn clause) axioms for the built-in binary predicate $\rightarrow$. The set of states that are reachable w.r.t. $\rightarrow$ from the initial ones is assigned to the constant *states*. The resulting transition relation is presented in Fig. 1. *TRANS* has no atoms. Since we perform modal-logic reasoning within predicate logic, atom valuations are usually represented in terms of predicates on states (like (< 4) in Fig. 3) and not in terms of functions as in the definition of a Kripke structure. However, if an atom valuation shall be displayed or manipulated, we also need a functional representation of the predicates—as, for instance, in the specification *MUTEX* of Section 3.

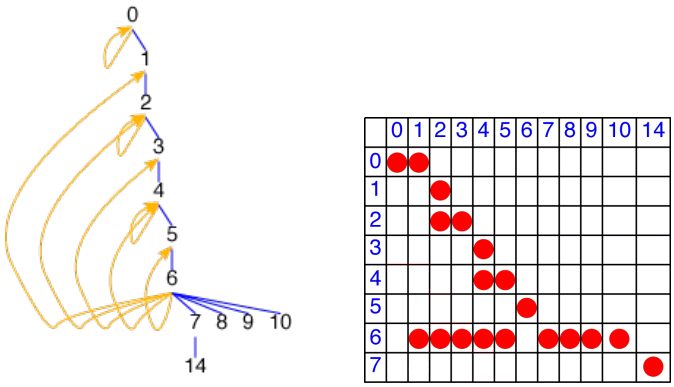

Fig. 1. The term graph representing the transition relation derived from TRANS and its interpretation by the matrix interpreter of Expander2

branch, *wtree*, *sat*, *frame* and *text* are built-in constructors. An implicational axiom of the form $\varphi \Rightarrow t \rightarrow branch[t_1,\ldots,t_n]$ means that for all ground instances q of the term (= state pattern) t that satisfy φ, the corresponding instances of $t_1,\ldots,t_n$ are direct successors of q. *sat* is attached to all nodes of the transition graph that represent states satisfying a given formula (see Figs. 3 and 6). A term of the form $wtree(f)(t)$ is simplified into a term with graphical attributes (here: *frame* and *text*) at some nodes of t by applying the function f to each node. For instance, in the axiom for *drawFT*, f is given by the term *fun(sat$x,frame$text$x,x,x)*, which represents the λ-abstraction

$$\lambda n.\ case\ n\ of \quad sat(x) \rightarrow frame(text(x)) \qquad n \rightarrow x.$$

The attributes are interpreted by the painter module: if a node (term) n has the form $sat(t)$, the subterm t is turned into its text representation and framed by a rectangle, while other nodes do

not obtain a graphical attribute and thus will be displayed by default. Fig. 3 provides an example of a term t and the result of interpreting the simplication of *drawFT(t)*.

& and | denote conjunction resp. disjunction. Equational axioms involving == are used as simplification rules (see below). The apply-operator \$ and list functions like concatenation (++), *map* and *filter* are defined as usually.

The solver module of Expander2 always produces resp. transforms term graphs like the one in Fig. 1. Basically, term graphs are trees, but they may involve additional edges (those with tips). The solver module may display further term representations of a binary or ternary relation: a list of pairs resp. triples and a conjunction of regular equations (equations with a variable on one side).

3 Modal logic and algebra

We present well-known modal and temporal operators (see, e.g., [14, 29]) in a rigorously algebraic fashion that allows us to model-check finite Kripke structures by pure term evaluation. The corresponding implementation in Expander2 is illustrated at the specification *TRANS* of the previous section and a further one (*MUTEX*) based upon [14], Example 3.1.1.

Let *Var* be a set of variables denoting sets of states or paths (sequences of states). The words generated from *sf* resp. *pf* by the following context-free rules are called **state formulas** resp. **path formulas**: Let $at \in At$, $lab \in Lab$ and $x \in Var$.

$$
\begin{array}{ll}
 & sf \rightarrow at \mid true \mid false \mid \neg sf \mid sf \vee sf \mid sf \wedge sf \mid sf \Rightarrow sf \\
(1) & sf \rightarrow EX\ sf \mid AX\ sf \mid \langle lab \rangle sf \mid [lab] sf \\
(2) & sf \rightarrow x \mid \mu x.sf \mid \nu x.sf \\
 & sf \rightarrow EF\ sf \mid AF\ sf \mid EG\ sf \mid AG\ sf \mid sf\ EU\ sf \mid sf\ AU\ sf \\
 & pf \rightarrow at \mid true \mid false \mid \neg pf \mid pf \vee pf \mid pf \wedge pf \mid pf \Rightarrow pf \\
(3) & pf \rightarrow next\ pf \mid \langle lab \rangle pf \mid [lab] pf \\
(4) & pf \rightarrow x \mid \mu x\ pf \mid \nu x\ pf \\
 & pf \rightarrow F\ pf \mid G\ pf \mid pf\ U\ pf
\end{array}
$$

Some of the above operators are subsumed by others. This is intended because the user shall be allowed to formalize conjectures as adequately as possible. The reduction to a minimal set of operators should be left to the model checker. Ours will turn all formulas into equivalent ones that consist of propositional, next-step ((1) resp. (3)) and fixpoint operators ((2) resp. (4)).

Like every context-free grammar the one above defines an algebraic **signature** $\Sigma = (PS, S, OP)$ with a set PS of *primitive sorts* (here: *at*, *lab* and *x*), a set S of further sorts, one for each nonterminal of the grammar, and a set OP of operators, one for each rule of the grammar: a rule $A \rightarrow w$ becomes an operator of type $v \rightarrow A$ where v is the word consisting of the nonterminals of w ($\varepsilon \rightarrow A$ is the type of a constant). In the above case, Σ-terms represent formulas, and proving the latter means evaluating the former with respect to a suitable interpretation of Σ, i.e. a **Σ-algebra**, say A.

Each sort $s \in PS \cup S$ is interpreted by a 'carrier' set s^A and each operator f by a function f^A whose domain and range comply with the interpretation of the sorts involved in the type of f. The nature of primitive sorts is to have the same interpretation in every Σ-algebra A. Hence *at*,

lab and x are always interpreted as the given sets At, Lab and Var of atoms, labels and variables, respectively. The interpretation of sf and pf in A leads to *functional* domains:

$$\begin{aligned} sf^A &= (Var \to \mathscr{P}(Q)) \to \mathscr{P}(Q), \\ pf^A &= (Var \to \mathscr{P}(path(K))) \to \mathscr{P}(path(K)). \end{aligned}$$

In Σ, each atom *at* becomes a constant of sort sf and also a constant of sort pf. Both fixpoint operators (μ and ν) have the types $Var \times sf \to sf$ and $Var \times pf \to pf$. Analoguous *binding* operators occur in other term languages as well, e.g., the *abstraction* and least-fixpoint operators λ resp. μ for building higher-order functions or the quantification operators $\forall$ and $\exists$ that come with an algebraic view on predicate logic.

Fixpoint operators are the main model builders. Be it single objects (including functions of arbitrary order), types (sets of objects) or relations (predicates) of arbitrary arity, whatever cannot be constructed by simply combining given objects (resp. sets) conjunctive- or disjunctively, is defined as a solution of a system of regular equations between variables on the left- and terms/formulas on the right-hand side, i.e. as a fixpoint of the function induced by the equations. From the classical theory of recursive functions via the semantics of logic programming languages up to domain theory and universal co/algebra, fixpoints provide the link between description, computation and proof in all these approaches.

The existence of a fixpoint requires the monotonicity of the functions used in the equations to be solved. Its stepwise constructability requires the stronger property of (upward or downward) continuity. In the case of a modal formula φ, monotonicity is ensured if each free occurrence of $x \in Var$ in φ has *positive polarity*, i.e. the number of negations on the path from the binder of x (μ or ν) to the occurrence is even. Continuity is guaranteed if, in addition to the monotonicity requirement, the transition relation is *image finite*, i.e. for all $q \in Q$ and $lab \in Lab$, $trans(q)$ resp. $transL(lab)(q)$ is finite. Hence we assume that Q is finite and all free variable occurrences in φ have positive polarity so that φ can be evaluated in the following extension of A to a Σ-algebra, called the **modal algebra over** K. We omit the interpretation of temporal, i.e. path formula operators because—due to the infinity of $path(K)$—it cannot be implemented as directly as the interpretation of state formula operators.

Let $s \in Q$, $lab \in Lab$, $\varphi, \psi \in pf^A$ and $b : Var \to \mathscr{P}(Q)$.

$$\begin{array}{lll} at^A(b) & =_{def} & value(at) \\ true^A(b) & =_{def} & Q \\ false^A(b) & =_{def} & \emptyset \\ \neg^A(\varphi)(b) & =_{def} & Q \setminus \varphi(b) \\ (\varphi \vee^A \psi)(b) & =_{def} & \varphi(b) \cup \psi(b) \\ (\varphi \wedge^A \psi)(b) & =_{def} & \varphi(b) \cap \psi(b) \\ \varphi \Rightarrow^A \psi & =_{def} & \neg^A(\varphi) \vee^A \psi \\ EX^A(\varphi) & =_{def} & imgsShares(Q)(sucs) \circ \varphi \qquad \text{“}exists\ next\text{”} \\ AX^A(\varphi) & =_{def} & imgsSubset(Q)(sucs) \circ \varphi \qquad \text{“}always\ next\text{”} \\ \langle lab \rangle^A(\varphi) & =_{def} & imgsShares(Q)(sucsL(lab)) \circ \varphi \\ [lab]^A(\varphi) & =_{def} & imgsSubset(Q)(sucsL(lab)) \circ \varphi \\ x^A(b) & =_{def} & b(x) \end{array}$$

$$(\mu x)^A(\varphi)(b) \quad =_{def} \quad lfp(\varphi(\lambda y.b[y/x]))(\emptyset)$$
$$(\nu x)^A(\varphi)(b) \quad =_{def} \quad gfp(\varphi(\lambda y.b[y/x]))(Q)$$

$f[a/x]$ denotes an update of (the valuation or substitution) f: $f[a/x](x) = a$ and for all $y \neq a$, $f[a/x](y) = f(y)$. The functions *lfp* (least fixpoint) and *gfp* (greatest fixpoint) are defined (in Haskell) as follows:

```
lfp, gfp :: Eq a => [a] -> ([a] -> [a]) -> [a]
lfp f s = if fs `subset` s then s else lfp f fs where fs = f s
gfp f s = if s `subset` fs then s else gfp f fs where fs = f s
```

They transform a finite set by repeatedly applying f until it does not change any more. If $lfp(f)$ is applied to $\emptyset$ or $gfp(f)$ to Q, the iteration terminates and—by Kleene's fixpoint theorem—returns the least resp. greatest solution of the equation $x = \varphi$ in $\mathscr{P}(Q)$. All further operators of Σ can be reduced to fixpoints, as one knows from the μ-calculus of modal logic (see, e.g., [29, 20]):

$EF(\varphi)$	$=$	$\mu x(\varphi \vee EX(x))$	"*exists finally*"
$AF(\varphi)$	$=$	$\mu x(\varphi \vee (EX(true) \wedge AX(x)))$	"*always finally*"
$EG(\varphi)$	$=$	$\nu x(\varphi \wedge (AX(false) \vee EX(x)))$	"*exists generally*"
$AG(\varphi)$	$=$	$\nu x(\varphi \wedge AX(x))$	"*always generally*"
$\varphi\ EU\ \psi$	$=$	$\mu x(\psi \vee (\varphi \wedge EX(x)))$	"*exists until*"
$\varphi\ AU\ \psi$	$=$	$\mu x(\psi \vee (\varphi \wedge AX(x)))$	"*always until*"

As demonstrated in Section 2, the simplifier of Expander2 derives K from a specification like *TRANS*. Any modal formula φ can then be evaluated in the modal algebra over K by applying rule (1) below to the expression $sols(\varphi)$ or rule (2) to the expression $solsG(\varphi)$:

$$(1)\ \frac{sols(\varphi)}{\varphi^A} \qquad (2)\ \frac{solsG(qs)}{\textit{transition graph of } K \textit{ with each state } q \in qs \textit{ replaced by } sat(q)}$$

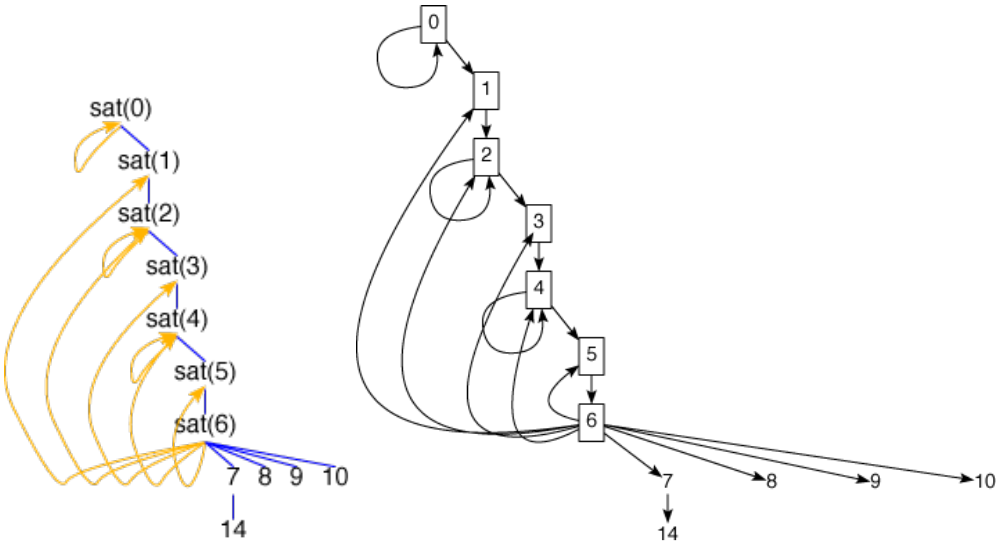

Fig. 3. Results of simplifying solsG$EF(<4) (left) and graphically interpreting the simplification of *drawFT$solsG$EF(<4)* (right) w.r.t. the specification TRANS of Section 2: $0,\ldots,6$ *are all states from which a state less than 4 is reachable.*

The Kripke structure derived from the following specification *MUTEX* models a system of n processes accessing a critical region. The model is specified along the lines of [14], Example 3.3.1, where the system is described for two processes. In *MUTEX*, *procs* denotes the actual list of all processes. States are pairs (xs, ys) consisting of the list xs of waiting processes and the list ys of processes in the critical region. Given two lists s and s', $s - s'$ returns all elements of s that are not in s'. *MUTEX* has atoms $idle(x)$, $wait(x)$ and $crit(x)$ for each process x. Like transition relations, atom valuations are specified in terms of the built-in binary predicate $\rightarrow$: $t \rightarrow branch[t_1, \ldots, t_n]$ means that for all ground instances *at* of the term (= atom pattern) t, the corresponding instances of the terms (= state patterns) $t_1, \ldots, t_n$ satisfy *at*. The higher-order predicate *Atom* turns Kripke structure atoms into atomic formulas. Logical operators are introduced as higher-order predicates and thus applied and composed like and in combination with higher-order functions. For instance, $and\$map(not.Crit)[x, y, z]$ denotes a ternary predicate that is satisfied by all triples of processes outside the critical region.

```
-- MUTEX
constructs: idle wait crit
preds:      Idle Wait Crit Crit' Atom live nonBlock noSeq
            /\ \/ `then` not and or EX EF AF AG `EU`
defuncts:   procs drawK
fovars:     xs ys at ats

axioms:
states == [([],[])] &                                          initial states
atoms == map($) $ prodL[[idle,wait,crit],procs] &

(xs,ys) -> branch $ map(fun(x,(x:xs,ys))) $ procs-xs-ys &      x waits
(xs =/= [] ==> (xs,[]) -> (init(xs),[last(xs)])) &             last(xs) enters
(xs,[x]) -> (xs,[]) &                                          x leaves

(Idle(x)(xs,ys) <==> x `in` procs-xs-ys) &
(Wait(x)(xs,ys) <==> x `in` xs) &
(Crit(x)(xs,ys) <==> x `in` ys) &

(Atom$idle$x <==> Idle$x) &
(Atom$wait$x <==> Wait$x) &
(Atom$crit$x <==> Crit$x) &

(at `in` atoms
   ==> at -> branch $ filter(Atom$at) $ states) &              atom valuation

(live$x     <==> AG $ Wait(x) `then` AF$Crit$x) &              no infinite waiting
(nonBlock$x <==> AG $ Idle(x) `then` EX$Wait$x) &              no blocking
(noSeq$x    <==> AG $ EF $ Crit(x) /\
                           (Crit(x) `EU` (not(Crit$x) /\
                                          (Crit'(x) `EU` crit$x)))) &
                  no sequencing: a process may leave the critical region and enter it again
                  before another process does so.
(Crit'$x <==> and(map(not.Crit) $ procs-[x])) &
```

```
drawK == wtree $ fun((xs,ys),frame$matrix[wait$xs,crit$ys],
                      sat((xs,ys),ats),
                         frame$matrix[wait$xs,crit$ys,satisfies$ats]) &
```

The graphical attribute *matrix* causes the elements of its argument list to be displayed as a matrix. *drawK* works analogously to *drawFT* defined in *TRANS0* (see Section 2).

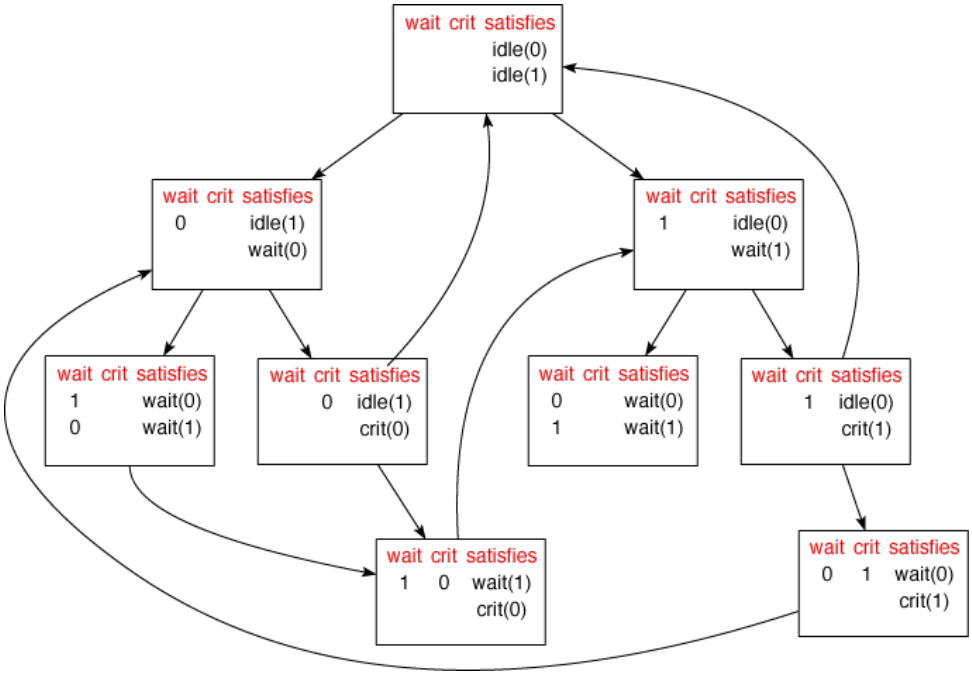

Fig. 4. The Kripke model derived from MUTEX for two processes after its transformation performed by drawK

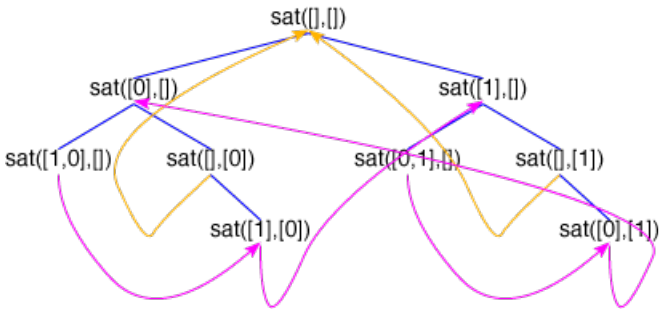

Fig. 5. The result of simplifying solsGandmap(live)[0,1]: all states satisfy live(0) and live(1).

4 Model checking by simplification

A path formula like $\forall\, pa : \varphi(pa)$ quantifies over the *infinite* set of paths of the underlying Kripke structure K and thus cannot be proved by simply evaluating it in the modal algebra over K: the

implementation of the fixpoint operators μ and ν with the functions *lfp* and *gfp* of Section 3 will not terminate. However, as fixpoint operators are ubiquitous in model design, so are the key proof rules *expansion*, *induction* and *coinduction* for properties of a—sometimes more-dimensional—fixpoint, say $a = (a_1, \ldots, a_n)$. If a solves the equation $(x_1, \ldots, x_n) = t(x_1, \ldots, x_n)$, expanding a term or formula φ means replacing occurrences of a in φ by (projections of the value of) $t(a)$. Expansion is sound for all solutions of the equation, but induction and coinduction only for the least resp. greatest one.

Expansion Let op be a fixpoint operator, $u = (t_1, \ldots, t_n)$ and $1 \leq i \leq n$.

$$\frac{op\ x_1 \ldots x_n.t}{t[\pi_i(op\ x_1 \ldots x_n.t)/x_i \mid 1 \leq i \leq n]} \qquad \frac{\pi_i(op\ x_1 \ldots x_n.u)}{t_i[\pi_j(op\ x_1 \ldots x_n.u)/x_j \mid 1 \leq j \leq n]}$$

π_i, $1 \leq i \leq n$, denotes the projection of an n-tuple on its i-th component. In the case of unary fixpoints (like the modal operators μ and ν), projections do not occur and we only need the first rule. In general, non-unary fixpoints arise from mutually recursive definitions of several functions or relations.

For reducing the danger of non-termination Expander2 applies expansion rules only to formulas that lack redices for other simplification rules. The simplifier traverses a formula tree depthfirst (leftmost-outermost) or breadthfirst (parallel-outermost) when searching for the next rule redex. The strategy of parallel-outermost simplification that postpones expansion steps as far as possible is a fixpoint strategy, i.e. terminates whenever *any* strategy terminates [17]. This suggests why the evaluation of path formulas in the modal algebra may not terminate: evaluation in an algebra always proceeds bottom-up and thus follows an *innermost* strategy!

Expansion rules are applied to the fixpoint itself. The redices of induction and coinduction, however, are implications with the fixpoint as its premise resp. conclusion:

Induction and coinduction

$$\frac{\mu x_1 \ldots x_n.\varphi \Rightarrow \psi}{\varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n] \Rightarrow \psi} \Uparrow \qquad \frac{\psi \Rightarrow \nu x_1 \ldots x_n.\varphi}{\psi \Rightarrow \varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n]} \Uparrow$$

The arrow $\Uparrow$ indicates that induction and coinduction are *backward (reasoning) rules* whose succedents imply the antecedents, but not necessarily vice versa. An important design goal of Expander2 is to emphasize the view on proofs as computation sequences. Hence our rule syntax reflects the order in which the rules are applied in a proof. Even within a backward proof, Expander2 may also apply *forward rules* (whose antecedents imply the succedents)—to subformulas with negative polarity (see Section 3). Most rules, however, are equivalence transformations, i.e., both backward and forward rules, and thus may be applied to any subformula of the current goal.

The problem with pure backward rules is their *narrowing effect*: the succedent may never reduce to *True*, although the antecedent would do so if other rules where applied to the redex. In the case of co/induction, this means that the co/induction hypothesis, which is given by ψ, is too weak resp. too strong. ψ must then be *generalized*, i.e. extended to some δ by adding a factor resp. summand. Obviously—and probably accounted for by the incompleteness of second-order logic—the candidates for δ cannot be enumerated. Just for seeing the boundaries within which

δ must be searched for one may generalize co/induction as follows:

Second-order induction and coinduction

$$\frac{\mu x_1 \ldots x_n.\varphi \Rightarrow \psi}{\exists \delta : ((\varphi[\pi_i(\delta)/x_i \mid 1 \leq i \leq n] \Rightarrow \delta) \wedge (\delta \Rightarrow \psi))} \Updownarrow$$

$$\frac{\psi \Rightarrow \nu x_1 \ldots x_n.\varphi}{\exists \delta : ((\psi \Rightarrow \delta) \wedge (\delta \Rightarrow \varphi[\pi_i(\delta)/x_i \mid 1 \leq i \leq n]))} \Updownarrow$$

The soundness of (first-order) co/induction is easy to show: $\mu x_1 \ldots x_n.\varphi$ and $\nu x_1 \ldots x_n.\varphi$ denote solutions of the equation $(x_1,\ldots,x_n) = \varphi$ in the modal algebra A (see Section 3). Since the operators of φ^A are monotone, the fixpoint theorem of Knaster and Tarski tells us that the least resp. greatest solution of $(x_1,\ldots,x_n) = \varphi$ in A is the least resp. greatest tuple $B = (B_1,\ldots,B_n)$ of sets such that (1) $\varphi[B_i/x_i \mid 1 \leq i \leq n]^A \subseteq B$ or (2) $B \subseteq \varphi[B_i/x_i \mid 1 \leq i \leq n]^A$, respectively. Since $\Rightarrow$ is interpreted in A by set inclusion, the conclusion of co/induction is valid iff (1)/(2) with B_i replaced by $\pi_i(\psi)^A$ holds true. Consequently, the rule antecedent follows from the minimality resp. maximality of B with respect to (1)/(2).

Since co/induction is part of the simplifier of Expander2, the system takes care of not destroying co/induction redices. For instance, the following simplification rules are applied only to formulas that are *not* co/induction redices:

Implication splitting Suppose that φ and ψ are simplified.

$$\frac{\varphi \;\Rightarrow\; \psi_1 \wedge \cdots \wedge \psi_n}{\varphi \Rightarrow \psi_1 \;\wedge\; \ldots \;\wedge\; \varphi \Rightarrow \psi_n} \Updownarrow \qquad \frac{\varphi_1 \vee \cdots \vee \varphi_n \;\Rightarrow\; \psi}{\varphi_1 \Rightarrow \psi \;\wedge\; \ldots \;\wedge\; \varphi_n \Rightarrow \psi} \Updownarrow$$

Since generalizing a co/induction hypothesis ψ means adding a factor resp. summand to ψ, the co/inductive provability of the antecedent of implication splitting does not imply the co/inductive provability of the succedent! On the other hand, if implication splitting does not interfere with co/induction, it *should* be applied because it brings the redex closer to its disjunctive normal form. More crucial than such Boolean trasnformations is the handling of quantified variables. Here the simplifier shifts quantifiers towards existentially quantified conjunctions of equations and universally quantified disjunctions of inequations. These are then treated separately by term replacement, atom splitting and atom removal, which often reduces the number of variables or even deletes all of them.

When simplifying a formula, Expander2 first treats it as a term, i.e., evaluates it in a suitable algebra, say A, which involves a couple of built-in types including the modal algebra over the derived Kripke structure (see Section 3)—if there is any. As to the formula's logical operators, A is a term algebra consisting of a kind of normal forms. For instance, an existential quantifier is merged with subsequent ones, distributed over subsequent implications and disjunctions and restricted to variables with free occurrences in the quantified formula.

After having been evaluated in this way, the simplifier applies rules like the ones presented here or given by the equational or equivalence axioms of user-defined specifications. In contrast to the preceding (bottom-up) evaluation these rules are applied only to outermost redices. Hence this level of simplification provides a possibility to model-check path formulas, which—due to the infinity of paths—cannot be evaluated in the modal algebra. So the following specification *LTLS* introduces the temporal operators F, G, U and *.tail* ("next") as higher-order predicates

on (a coalgebraic specification of) streams, which represent the (infinite) paths of an arbitrary Kripke structure K. K is derived from an extension of *LTLS* such as *MICROS* (see below) via the built-in predicate $\rightarrow$ in the way Kripke structures were derived from *TRANS* and *MUTEX* (see Section 2). Later proofs use the axioms of *LTLS* as simplification rules along with expansion and co/induction.

```
-- LTLS
constructs: blink                                        the stream 010101...
preds:      true false not /\ \/ `then` F G `U` P Q
fovars:     at s
hovars:     X P Q                                        higher-order variables
axioms:     head$blink == 0                              coalgebraic specification of blink
          & tail$blink == 1:blink                        dto.
          & (true$s <==> True)
          & (false$s <==> False)
          & (not(P)$s <==> Not(P$s))
          & ((P/\Q)$s <==> (P$s & Q$s))
          & ((P\/Q)$s <==> (P$s | Q$s))
          & ((P`then`Q)$s <==> (P$s ==> Q$s))
          & (F$P <==> MU X.(P\/X.tail))                  ''finally''
          & (G$P <==> NU X.(P/\X.tail))                  ''generally''
          & ((P`U`Q) <==> MU X.(Q\/(P/\X.tail)))         ''until''
```

The functions *head* and *tail*, which provide the destructors of a coalgebraic specification of streams, are defined as usually. The formula $atom(at)\$s$ checks whether the head of the path s satisfies $at \in At$ (see Section 2). The conjecture

```
s = blink | s = 1:blink ==> G(F$(=0).head)$s                    (1)
```

says that the streams $blink$ and $1:blink$ are fair insofar as they contain infinitely many zeros. By the G-axiom of *LTLS*, (1) simplifies to:

```
s = blink | s = 1:blink ==> NU X.(F((=0).head)/\X.tail)$s       (2)
```

(2) is an instance of the antecendent of coinduction (see above). Applying the rule yields:

```
All s:(s = blink | s = 1:blink ==>
       (F((=0).head)/\(rel(s,s=blink|s=1:blink).tail))$s)       (3)
```

rel is the λ-operator for predicates: $rel(s, s = blink|s = 1:blink)$ denotes the function that assigns to s the formula $s = blink|s = 1:blink$.

47 further simplification steps including three expansion steps turn (3) into *True*. The entire proof goes through automatically.

A further sample proof refers to a specification of a microwave controller [4]:

```
-- MICROS
specs:       LTLS                        imported specification
constructs:  start close heat error
```

```
preds:        Start Close Heat Error Atom
fovars:       ats
axioms:       states == [1]
            & atoms == [start,close,heat,error]
            & 1 -> branch[2,3] & 2 -> 5 & 3 -> branch[1,6]
            & 4 -> branch[1,3,4] & 5 -> branch[2,3]
            & 6 -> 7 & 7 -> 4

            & (Atom$start <==> Start)
            & (Atom$close <==> Close)
            & (Atom$heat <==> Heat)
            & (Atom$error <==> Error)

            & (Start$x <==> x `in` [2,5,6,7])
            & (Close$x <==> x `in` [3,4,5,6,7])
            & (Heat$x <==> x `in` [4,7])
            & (Error$x <==> x `in` [2,5])

            & (at `in` atoms
               ==> at -> branch$filter(Atom$at)$states)      atom valuation

            & drawK == wtree$fun(sat(x,ats),
                                 frame$matrix[x,satisfies$ats])
```

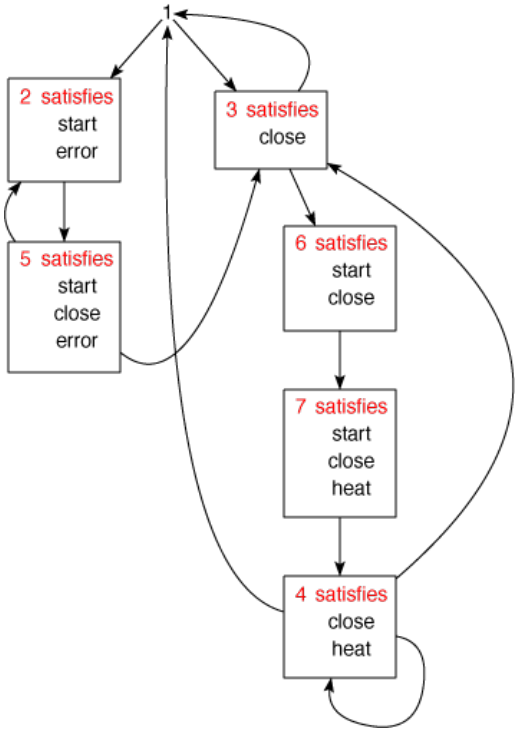

Fig. 6. The Kripke model derived from MICROS for two processes after its transformation performed by drawK

The conjecture $G(Error.head)\$s \Rightarrow G(not(Heat).head)\$s$ says that paths consisting of error states do not contain heat states. By the *G*-axiom of *LTLS*, it simplifies to:

```
NU X.((Error.head)/\(X.tail))$s ==>
NU X.((not(Heat).head)/\(X.tail))$s
```

Applying the coinduction rule yields:

```
All s:(NU X.((Error.head)/\(X.tail))$s ==>
        ((not(Heat).head)/\
         (rel(s,NU X.((Error.head)/\(X.tail))$s).tail))$s)
```

41 further simplification steps lead this formula to *True*. Three expansion steps are needed, and the entire proof goes through automatically.

5 Model checking within co/Horn logic

Both evaluation (Section 3) and simplification (Section 4) regard modal formulas as representations of data, namely (tuples of) sets. That's why we call this kind of model checking *algebraic*: the logical operators denote *functions* that create or transform data. Fixpoint operators are no exception. They map the left-hand sides of regular equations to the equations' solutions (see Section 3). First-order predicate logic as well as logic programming follow a different view. Their formulas do not denote data, but propositions or statements *about* data. Set membership takes us from the sets-as-data view to the propositional one, set comprehension back from the propositional to the data view. So where is the difference? It comes with the fixpoint property that cannot be expressed within first-order logic. Instead, we axiomatize *co/predicates* in terms of (generalized) *co/Horn clauses* and fix their interpretation as least resp. greatest relations satisfying the axioms. Details of this approach and its connection with relational and functional programming can be found in [19, 20, 23, 24]. Here we apply it to modal logics by specifying modal and temporal operators in terms of co/Horn axioms:

```
-- CTL
preds:    EX EF AF `EU` `AU` P Q      predicates
copreds:  AX EG AG                    copredicates
fovars:   st st'
hovars:   P Q
axioms:    (EX(P)$st <=== st -> st' & P(st'))
         & (AX(P)$st ===> (st -> st' ==> P(st')))
         & (EF(P)$st <=== P$st | EX(EF(P))$st)
         & (AF(P)$st <=== P$st | AX(AF(P))$st)
         & (EG(P)$st ===> P$st & EX(EG(P))$st)
         & (AG(P)$st ===> P$st & AX(AG(P))$st)
         & ((P`EU`Q)$st <=== Q$st | P$st & EX(P`EU`Q)$st)
         & ((P`AU`Q)$st <=== Q$st | P$st & AX(P`AU`Q)$st)

-- LTL
preds:    F `U` P Q                   predicates
copreds:  G                           copredicates
```

```
fovars:   s
hovars:   P Q
axioms:   (F(P)$s <=== P$s | F(P)$tail$s)
        & (G(P)$s ===> P$s & G(P)$tail$s)
        & ((P`U`Q)$s <=== Q$s | P$s & (P`U`Q)$tail$s)
```

The direction of the implication arrow (<=== or ===>) determines whether the axiom is a Horn or a co-Horn clause and whether the leading relation symbol r is a predicate or a copredicate to be interpreted as the least resp. greatest relation satisfying all axioms for r. Within a derivation, a co/Horn clause is always applied from left to right. It may start with a *guard* γ that confines redices to formulas that unify with the left-hand side (premise resp. conclusion) *and* satisfy γ (see the co/resolution rules given below). Expander2 accepts five types of formulas as axioms: Let p be a predicate (including ->), q be a copredicate and $t_1,\ldots,t_n$ be terms.

(1) γ ==> (p(t1,...,tn) <==> φ)	*equivalence used for simplification*
(2) γ ==> (t1 == t2 <==> φ)	*equation used for simplification*
(3) γ ==> (p(t1,...,tn) <=== φ)	*Horn clause used for resolution upon p*
(4) γ ==> (f(t1,...,tn) = u <=== φ)	*Horn clause used for narrowing, i.e., functional resolution, upon f*
(5) γ ==> (q(t1,...,tn) ===> φ)	*Horn clause used for coresolution upon q*

In all five cases, the axiom is applicable if the term/formula the axiom shall be applied to unifies with—in cases (1) and (2): matches—its redex and if the corresponding instance of the guard γ simplifies to *True*. Here are (simplified versions of) the main rules for processing co/predicates. For lack of space we omit those that handle functions specified in terms of Horn clauses (see above).

Parallel resolution upon the predicate p

$$\frac{p(t)}{\bigvee_{i=1}^{k} \exists Z_i : (\varphi_i\sigma_i \wedge \vec{x} = \vec{x}\sigma_i)} \Updownarrow$$

where $\gamma_1 \Rightarrow (p(t_1) \Longleftarrow \varphi_1),\ldots,\gamma_n \Rightarrow (p(t_n) \Longleftarrow \varphi_n)$ are the (Horn) axioms for p.

Parallel coresolution upon the copredicate p

$$\frac{p(t)}{\bigwedge_{i=1}^{k} \forall Z_i : (\vec{x} = \vec{x}\sigma_i \Rightarrow \varphi_i\sigma_i)} \Updownarrow$$

where $\gamma_1 \Rightarrow (p(t_1) \Longrightarrow \varphi_1),\ldots,\gamma_n \Rightarrow (p(t_n) \Longrightarrow \varphi_n)$ are the (co-Horn) axioms for p.

$\vec{x}$ is a tuple of "new" variables and for all $1 \leq i \leq n$, $Z_i = var(t_i) \cup var(\varphi_i)$, σ_i is a unifier of t and t_i and $\gamma_i\sigma_i$ simplifies to *True*.

Like co/induction as simplication (see Section 4), incremental coinduction can only be applied to implications with a predicate (the first-order analog of a variable bound by μ) in the premise or a copredicate (the first-order analog of a variable bound by ν) in the conclusion. In contrast to co/induction as simplification, we may now start a proof with the original conjecture and generalize it later—when simplification rules are no longer applicable and generalization

candidates have emerged from preceding proof steps. Incremental coinduction is also called *circular* [7, 12] and has recently been used to prove bisimilarities (behavioral equalities) induced by non-deterministic transition systems with structured states representing processes [27].

Suppose that the formulas ψ and δ do not contain the co/predicate p. AX_p denotes the set of co/Horn axioms for p.

Incremental induction upon the predicate p

$$\frac{p(x) \;\Rightarrow\; \psi(x)}{\bigwedge_{p(t)\Leftarrow\varphi\in AX_p}(\varphi[p'/p] \Rightarrow \psi(t))}\Uparrow \qquad \frac{p'(x) \;\Rightarrow\; \delta(x)}{\bigwedge_{p(t)\Leftarrow\varphi\in AX_p}(\varphi[p'/p] \Rightarrow \delta(t))}\Uparrow$$

p' is a "new" copredicate that starts with the axiom $p'(x) \Rightarrow \psi(x)$. When the second rule is applied, the co-Horn clause $p'(x) \Rightarrow \delta(x)$ becomes a further axiom for p'.

Incremental coinduction upon the copredicate p

$$\frac{\psi(x) \;\Rightarrow\; p(x)}{\bigwedge_{p(t)\Rightarrow\varphi\in AX_p}(\psi(t) \Rightarrow \varphi[p'/p])}\Uparrow \qquad \frac{\delta(x) \;\Rightarrow\; p'(x)}{\bigwedge_{p(t)\Rightarrow\varphi\in AX_p}(\delta(t) \Rightarrow \varphi[p'/p])}\Uparrow$$

p' is a "new" predicate that starts with the axioms $p'(x) \Leftarrow \psi(x)$ and—only if p is behavioral equality—Horn clauses establishing p' as an equivalence relation. When the second rule is applied, the Horn clause $p'(x) \Leftarrow \delta(x)$ becomes a further axiom for p'.

Co/resolution and co/induction complement each other in the way axioms work together with conjectures in proof: co/resolution applies axioms to conjectures and the proof proceeds with the modified conjectures, while co/induction applies conjectures to axioms and establishes the modified axioms as new conjectures. Generalizations of co/resolution and co/induction for simultaneously proving properties of several co/predicates specified by mutual recursion and thus representing more-dimensional fixpoints are straightforward. For further details an co/Horn logic and its implementation in Expander2, consult [19, 20, 23, 24, 25].

Incremental coinduction allows us to start a proof that the stream *blink* is fair (see Section 4) with the original conjecture $\psi = G(F\$(=0).head)\$blink$ and derive within the proof of ψ the additional factor $G(F\$(=0).head)\$1 : blink$ of the conjecture (1) in Section 4. Indeed, incremental coinduction of ψ returns the goal

```
All P s:(P = F((=0).head) & s = blink ===> P(s) & G0(P)$tail$s)   (1)
```

$G0$ is the "new" predicate p' of incremental coinduction (see above). Its first axiom is given by:

```
G0(z0)$z1 <=== z0 = F((=0).head) & z1 = blink                    (ax1)
```

Six simplification steps transform (1) into:

```
F((=0).head)$blink & G0(F((=0).head))$(1:blink)                  (2)
```

Parallel resolution upon F as specified in *LTL* (see above) and subsequent simplification steps remove the first factor of (2). The second factor is a redex for the second rule of incremental coinduction that turns (2) into:

```
All P s:(P = F((=0).head) & s = 1:blink ===>
         P(s) & G0(P)$tail(s))                                    (3)
```

A second axiom for $G0$ is created:

```
G0(z2)$z3 <=== z2 = F((=0) . head) & z3 = 1:blink                 (ax2)
```

Five simplification steps transform (3) into:

```
F((=0).head)$(1:blink) & G0(F((=0).head))$blink                   (4)
```

Three resolution and subsequent simplification steps turn (4) into *True*.

The conjecture $G(Error.head)\$s ==> G(not(Heat).head)\$s$ of the specification *MICROS* in Section 4 can also be proved within co/Horn logic. Once the imported specification *LTLS* of *MICROS* has been replaced with *LTL* above, incremental coinduction turns the conjecture into:

```
All s P:(G(Error.head)$s & P = (not(Heat).head) ===>
         P(s) & G0(P)$tail(s))                                    (1)
```

$G0$ is the "new" predicate p' of incremental coinduction. Its first axiom is given by:

```
G0(z0)$s <=== G(Error.head)$s & z0 = (not(Heat).head)
```

Coresolution upon G and simplification steps turn (1) into:

```
All s:(G(Error.head)$s ==> G0(not(Heat).head)$tail(s))            (2)
```

(2) admits both coresolution upon G and resolution upon $G0$. Coresolution would lead into a cycle because the only axiom for G (see *LTL*) is recursive, i.e., G occurs on both sides of the axiom. The (above) axiom for $G0$, however, is non-recursive—as axioms for the "new" predicate p' always are. Hence we resolve upon $G0$ and obtain:

```
All s:(G(Error.head)$s ==> G(Error.head)$tail(s))                 (3)
```

Coresolution upon G and subsequent simplification steps turn (3) into *True*.

6 Beyond model checking

Model checking can only be applied to individual Kripke structures. Even if built up from several transition systems communicating with each other, it is always a *single* structure one reasons about (see, e.g., [3, 4]). Systems like *MUTEX*, however, are parameterized by certain components involved. Indeed, *MUTEX* specifies many Kripke structures, one for each (finite) number of processes. Modal proofs must be carried out for each number of processes separately because Kripke structures for different numbers of processes differ considerably from each other. Among the three methods presented in this paper, it is only the last one that allows us to perform a single proof for all instances of a parameterized Kripke structure. Even then the underlying specification often needs to be revised, in order to capture all instances simultaneously. In the case of *MUTEX*, we came up with the following reformulation:

```
-- MUTEXco
specs:      CTL
preds:      Idle Wait Crit enabled safe noSeq >>
copreds:    others
constructs: c
defuncts:   request enter leave posi maxwait weight
fovars:     xs ys xs' ys'

axioms:
(st >> st' <==> weight(st) > weight(st')) &
weight(xs,ys) == (length(xs)-posi(c)(xs++ys),
                  maxwait-length(xs),length$ys) &
posi(x)$x:s = 0 &
(posi(x)$y:s = suc$posi(x)$s <=== x =/= y) &

(st -> f$st <=== enabled(f)$st) &
(enabled(request$x)(xs,ys)
            <=== Idle(x)(xs,ys) & maxwait > length$xs) &
enabled(enter)(x:xs,[]) &
enabled(leave)(xs,[x]) &
request(x)(xs,ys) == (x:xs,ys) &
enter(xs,ys) == (init$xs,[last$xs]) &
leave(xs,ys) == (xs,[]) &

(Wait(x)(xs,ys) <==> x `in` xs) &
(Crit(x)(xs,ys) <==> x `in` ys) &
(Idle(x)(xs,ys) <==> x `NOTin` xs & x `NOTin` ys) &

safe(xs,[]) & safe(xs,[x]) &
(noSeq$x
    <==> EF $ Crit(x) /\
               (Crit(x) `EU` (not(Crit$x) /\
                              (others(not.Crit)(x) `EU` Crit$x)))) &
(others(P)(x)$st ===> (x =/= y ==> P(y)$st))

conjects:
(c `in` xs | c `in` ys                  no infinite waiting
         ==> AF(Crit$c)(xs,ys))         (proof by Noetherian induction w.r.t. >>)
(Idle(x)(xs,ys) & length$xs < maxwait   no blocking
         ==> EX(Wait$x)(xs,ys)) &       (proof by resolution upon EX)
(safe$st ==> AG(safe)$st) &             safety of the critical region
                                        (proof by coinduction upon AG)
```

Again, states are pairs consisting of the lists of actually waiting resp. working processes. The transition relation is specified in terms of the "methods" *request*, *enter* and *leave* and the "attributes" *Idle*, *Wait* and *Crit*. Obviously, attributes correspond to atoms of a Kripke structure. But they are more general because they may assign a value of arbitrary type to a each state, not just a Boolean one. In the terminology of coalgebraic specifications, methods and attributes are destructors, like *head* and *tail* in *LTLS* (see Section 4).

Parameterization by a natural number (here: the number of processes) suggests verification by Noetherian induction. At least, the first conjecture of *MUTEXco*, which says that each waiting or working process c will work eventually—no matter which path the system takes—, could be proved by Noetherian induction, but only with respect to a rather sophisticated lexicographic ordering on states that takes into account the position of c in the waiting list and the lengths of the waiting and working lists, respectively.

The proof of "no blocking" is straightforward. A parameterized proof of "no sequencing" (see *MUTEX* in Section 3) has not yet been tried. We close the section with the complete Expander2 protocol of its proof of the last conjecture of *MUTEXco*: there is at most one process in the critical region. Expander2 records each interactive proof in this way and also generates a *proof term* consisting of commands whose execution repeats the proof automatically.

```
safe(st) ==> AG(safe)$st

Adding

  (AG0(z0)$st <=== safe(st) & z0 = safe)
& (notAG0(z0)$st ===> Not(safe(st)) | z0 =/= safe)

to the axioms and applying coinduction w.r.t.

  (AG(P)$st ===> P(st) & AX(AG(P))$st)

at position [] of the preceding formula leads to

All st:(safe(st) ==> AX(AG0(safe))$st)

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All st st':(safe(st) & st -> st' ==> AG0(safe)$st')

The axioms were MATCHED against their redices.
The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All st st':(safe(st) & st -> st' ==> safe(st'))

The axioms were MATCHED against their redices.
The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All st' xs:((xs,[]) -> st' ==> safe(st')) &
All st' xs x:((xs,[x]) -> st' ==> safe(st'))
```

```
The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs f:(enabled(f)(xs,[]) ==> safe(f(xs,[]))) &
All st' xs x:((xs,[x]) -> st' ==> safe(st'))

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs x0:(x0 `NOTin` xs & maxwait > length(xs) ==> safe(x0:xs,[])) &
All xs0 x0:safe(init(x0:xs0),[last(x0:xs0)]) &
All st' xs x:((xs,[x]) -> st' ==> safe(st'))

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs0 x0:safe(init(x0:xs0),[last(x0:xs0)]) &
All st' xs x:((xs,[x]) -> st' ==> safe(st'))

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All st' xs x:((xs,[x]) -> st' ==> safe(st'))

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs x f:(enabled(f)(xs,[x]) ==> safe(f(xs,[x])))

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs x x3: (x3 `NOTin` xs & x3 =/= x & maxwait > length(xs)
              ==> safe(x3:xs,[x])) &
All xs:safe(xs,[])

The reducts have been simplified.

Narrowing the preceding formula (1 step) leads to

All xs:safe(xs,[])

The reducts have been simplified.
```

```
Narrowing the preceding formula (1 step) leads to

True

The reducts have been simplified.

Number of proof steps: 12
```

7 Conclusion

We have shown a way of integrating Kripke structures into algebraic specifications, presented three methods for proving their properties and illustrated their implementation and joint use in Expander2. All three methods admit structured states represented as functional terms. Moreover, the models may involve a labelled or unlabelled transition system (also called a Kripke frame), a labelled or unlabelled atom valuation or a mixture thereof. The first method consists in evaluating modal formulas in an algebra where logical operators come as functions taking sets of states to sets of states. The evaluation procedure is part of the simplification component of Expander2. Some logical operators compute fixpoints. Hence model checking by evaluation requires models with a finite set of states.

The second technique extends the modal algebra of the first one by simplification rules, in particular by *expansion*, *induction* and *coinduction*. This allows us to prove not only state, but also path formulas and to verify Kripke models with an infinite state space. Due to the selection of only *outermost* rule redices, Expander2's strategy of applying expansion, co/induction and other simplification rules is complete: it terminates whenever *any* strategy terminates.

The third approach is based on our work on co/Horn logic [19, 20, 23, 24] where co/Horn clauses axiomatize least resp. greatest relational fixpoints and parallel co/resolution provides the counterpart of expansion in the second approach. The co/induction rules of the second approach are replaced by *incremental* co/induction that admits the automatic—and often inevitable—generalization of a conjecture. Incremental coinduction is also called *circular* [7, 12] and has recently been used to prove bisimilarities induced by non-deterministic transition systems with structured states [27]. Proof assistants offering coinduction are CLAM [5], Isabelle [6, 26] and PVS [10, 13]. We claim that the simplification version presented in Section 4 and the incremental version of Section 5 are the most general coinduction rules so far. The first one has been integrated into a formula/term simplifier, which applies it automatically. The second one can be used for checking any relation with a greatest-fixpoint semantics and not only bisimulations or other equalities. Moreover, incremental *induction* does not seem to have been mentioned anywhere else. However, the restriction of previous approaches to *co*induction and equalities only applies to the *incremental* rules. The *simplification* version of co/induction has many forerunners, also in a more general, non-logical context where an arbitrary partial order replaces the implication (see, e.g., [17, 9, 6, 26]).

Mainstream model checkers hide their logical background by translating both Kripke models and the formulas to be proved into bit representations that a deterministic proof algorithm can process efficiently without manual intervention. At least at the present stage of development,

the methods presented in this paper do not compete against established model checkers. Instead, the methods resulted from questions like: What are the characteristics, benefits and drawbacks of Kripke structures if compared with coalgebraic models in general? Which models and modal logics are adequate for describing and verifying which kind of systems? How does the complexity or generic nature of a system affects its formalization and the degree of proof automation?

Bibliography

[1] J. van Benthem, J. Bergstra, *Logic of Transition Systems*, J. Logic, Language and Information 3 (1995) 247-283

[2] C. Cirstea, A. Kurz, D. Pattinson, L. Schröder, Y. Venema, *Modal Logics are Coalgebraic*, The Computer Journal, to appear

[3] E.M. Clarke, O. Grumberg, S. Jha, *Verification of Parameterized Networks*, ACM TOPLAS 19 (1997) 726-750

[4] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press 1999

[5] L.A. Dennis, A. Bundy, I. Green, *Making a productive use of failure to generate witnesses for coinduction from divergent proof attempts*, Annals of Mathematics and Artificial Intelligence 29, Springer (2000) 99-138

[6] J. Frost, *A Case Study of Co-induction in Isabelle*, Report, Computer Laboratory, University of Cambridge 1995

[7] J. Goguen, K. Lin, G. Rosu, *Conditional Circular Coinductive Rewriting with Case Analysis*, Proc. WADT'02, Springer LNCS 2755 (2003) 216-232

[8] J. Goguen, G. Malcolm, *A Hidden Agenda*, Theoretical Computer Science 245 (2000) 55-101

[9] A.D. Gordon, *Bisimilarity as a Theory of Functional Programming*, Theoretical Computer Science 228 (1999) 5-47

[10] H. Gottliebsen, *Co-inductive Proofs for Streams in PVS*, Report, Queen Mary, University of London 2007

[11] I. Hasuo, *Modal Logics for Coalgebras - A Survey*, Report, Tokyo Institute of Technology (2003)

[12] D. Hausmann, T. Mossakowski, L. Schröder, *Iterative Circular Coinduction for CoCasl in Isabelle/HOL*, Proc. FASE'05, Springer LNCS 3442 (2005) 341-356

[13] U. Hensel, B. Jacobs, *Coalgebraic Theories of Sequences in PVS*, J. Logic and Computation 9 (1999) 463-500

[14] M. Huth, M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd Ed. Cambridge University Press 2004

[15] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259

[16] A. Kurz, *Specifying Coalgebras with Modal Logic*, Theoretical Computer Science 260 (2001) 119-138

[17] Z. Manna, Mathematical Theory of Computation, McGraw-Hill 1974

[18] M. Müller-Olm, D.A. Schmidt, B. Steffen, *Model-Checking: A Tutorial Introduction*, Proc. SAS'99, Springer LNCS 1694 (1999) 330-354

[19] P. Padawitz, *Proof in Flat Specifications*, in: Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Report, Springer (1999) 321-384

[20] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165

[21] P. Padawitz, *Dialgebraic Specification and Modeling*, in preparation, fldit-www.cs.tu-dortmund.de/∼peter/Dialg.pdf

[22] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.tu-dortmund.de/∼peter/Expander2.html

[23] P. Padawitz, *Expander2: Towards a Workbench for Interactive Formal Reasoning*, in: Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig, Springer LNCS 3393 (2005) 236-258

[24] P. Padawitz, *Expander2: Program Verification between Interaction and Automation*, Proc. 15th Workshop on Functional and (Constraint) Logic Programming, Elsevier ENTCS 177 (2007) 35-57 (more recent version: fldit-www.cs.tu-dortmund.de/∼peter/Expander2/Prover.pdf)

[25] P. Padawitz, *Algebraic Model Checking and more*, in preparation, fldit-www.cs.tu-dortmund.de/∼peter/Haskellprogs/CTL.pdf

[26] L.C. Paulson, *Mechanizing Coinduction and Corecursion in Higher-Order Logic*, J. Logic and Computation 7 (1997) 175-204

[27] A. Popescu, E.L. Gunter, *Incremental Pattern-Based Coinduction for Process Algebra and Its Isabelle Formalization*, Proc. FOSSACS 2010, Springer LNCS 6014 (2010) 109-127

[28] J. Rutten, *Universal Coalgebra: A Theory of Systems*, Theoretical Computer Science 249 (2000) 3-80

[29] C. Stirling, *Modal and Temporal Logics*, in: Handbook of Logic in Computer Science, Clarendon Press (1992) 477-563