



No \volumetitle defined!

Algebraic Model Checking

Peter Padawitz

22 pages

Algebraic Model Checking

Peter Padawitz

TU Dortmund, Germany

Abstract: Several more or less algebraic approaches to model checking are presented and compared with each other with respect to their range of applications and their degree of automation. All of them have been implemented and tested in our Haskell-based formal-reasoning system Expander2. Besides realizing and integrating state-of-the-art proof and computation rules the system admits rarely restricted specifications 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. Indeed, this paper does not present purely theoretical work. Due to the increasing abstraction potential of programming languages like Haskell the boundaries between developing a formal system and implementing it or making it ‘user-friendly’ as well as between systems developed in different communities become more and more obsolete. The individual topics discussed in the paper reflect this observation.

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. [5, 14, 27]). 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. [2, 19]), but did not affect very much the direction of research in model checking. With the invention of *coalgebraic* logics (see e.g. [15, 26, 16, 9, 3]) 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 (usually called its *global* semantics): their formulas denote relations; the logical operators (including fixpoint operators!) are functions building relations from relations. The underlying data are either states (elements of a destructor-based type) or paths (which also form a destructor-based type).

We have investigated and implemented in our proof assistant Expander2 [21, 22, 23] four approaches to model checking. The first one may be called purely algebraic because proving a formula boils down to its complete *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*. All model representations and proof records given here were generated by Expander2. 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-Jrg 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** $K = (Q, At, Lab, \rightarrow, val, valL)$ consists of a set Q of **states**, a set At of **atoms**, a set Lab of **labels** (actions, input, output, etc.), a **transition relation** $\rightarrow \subseteq Q \times Q$ or $\rightarrow \subseteq Q \times Lab \times Q$ and **state valuations** $val \subseteq At \times Q$ and $valL \subseteq At \times Lab \times Q$. If Lab is nonempty, \rightarrow denotes $\cup \{ \overset{lab}{\rightarrow} \mid lab \in Lab \}$.

Let $s \in Q$, $lab \in Lab$ and $qs \cup qs' \subseteq Q$. $sucs(s) = \{s' \in Q \mid s \rightarrow s'\}$ and $sucsL(s, lab) = \{s' \in Q \mid s \overset{lab}{\rightarrow} s'\}$ denote the sets of all (direct) successors of s resp. all successors of s after input/execution of lab .

$$path(K) = \{p \in Q^{\mathbb{N}} \mid \forall i \in \mathbb{N} : p_i \rightarrow p_{i+1} \vee (sucs(p_i) = \emptyset \wedge p_i = p_{i+1})\}$$

denotes the set of *paths* of K . Given a function $f : Q \rightarrow \mathcal{P}(Q)$,

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

denote the sets of states $s \in qs$ such that at least one resp. all f -images of s are in qs' . Expander2 admits the specification of Kripke structures in terms of rewrite rules (axioms for \rightarrow) as in the following example. It is small and not very practical, but involves a couple of frequently used functional or logical operators.

```

-- TRANS
constructs: less SAT                                     -- constructors
defuncts:   inits states atoms drawSF                 -- defined functions
fovars:     n x y                                       -- first-order variables
axioms:     states == [0] &                             -- initial states
            atoms == map(less) [0..10] &
            (n < 6 & n `mod` 2 = 0 ==> n -> n<+>n+1) &
            (n < 6 & n `mod` 2 /= 0 ==> n -> n+1) &
            6 -> branch$[1..5]++[7..10] &
            7 -> 14 &
            less$x -> branch $ filter(rel(y,y<x)) $ states &
            drawSF == wtree $ fun (SAT$x,x,rframe$text$x,x,x)
    
```

After the specification has been entered and the button *build Kripke model* has been pushed, Expander2 constructs the set `states` of states, the transition relation and the state valuation of the model from the axioms for the binary predicate `->`. For instance, `states` is the set of terms that are reachable from the initial ones `states` via the transitive closure of `->`. `&` and `|` denote conjunction resp. disjunction. Equational axioms involving `==` are used as simplification rules (see below). `<+>` and `branch` are constructors for building sets of successor states. The apply-operator `$` and the list functions `map`, `++` and `filter` are interpreted as in Haskell. `fun` and `rel` are the λ -abstraction operators for functions resp. relations. For instance, `fun (p, t, q, u)` denotes the function that, when applied to `v`, yields the corresponding instance of `t` resp. `u` if `v` matches `p` resp. `q`.

`wtree(f) (t)` turns each node `n` of the term `t` into a graphical widget by applying `f` to the term representation (!) of `n`. In the `drawSF` axiom of `TRANS`, nodes matching `SAT (x)` are framed by a rectangle, others are (re-)turned into their string representation. The result of simplifying `wtree(f) (t)` is interpreted by the painter module of Expander2 (see, e.g., Fig. 3). Another graphical interpreter of Expander2 turns binary or ternary relations into matrices:

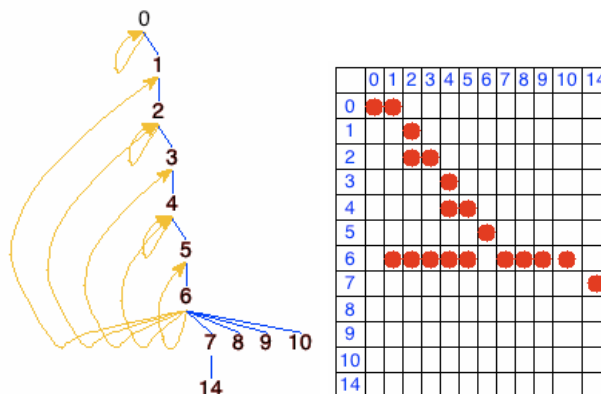


Fig. 1. The term (graph) representing the transition relation of `TRANS` and its interpretation by the matrix interpreter of Expander2

The solver module of Expander2 always produces or transforms term graphs like the one on the left-hand side of Fig. 1. Basically, term graphs are trees, but they may involve additional

edges (those with tips). The solver module computes further term representations of a binary or ternary relation: a list of pairs resp. triples and a conjunction of regular equations (see Fig. 2).

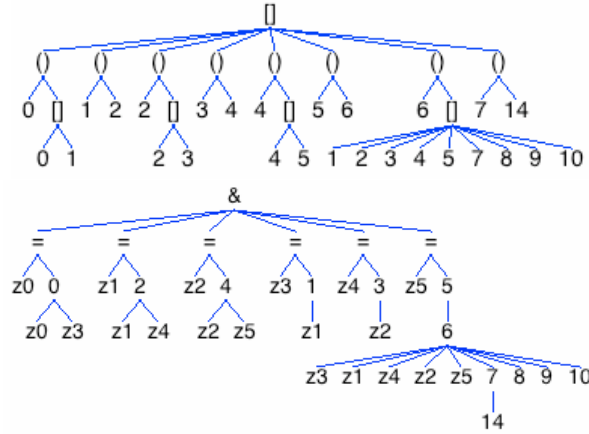


Fig. 2. A list of pairs and a conjunction of regular equations representing the transition relation of TRANS

3 Modal logic and algebra

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 $x \in Var$.

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

Some of the above operators are subsumed by others. This is intended because we favor *natural* deduction where the user is 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 reflects what is often called the a *global semantics* of modal logic:

$$\begin{aligned} sf^A &= (Var \rightarrow \mathcal{P}(Q)) \rightarrow \mathcal{P}(Q) \\ pf^A &= (Var \rightarrow \mathcal{P}(path(K))) \rightarrow \mathcal{P}(path(K)) \end{aligned}$$

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 $Var \times sf \rightarrow sf$ and $Var \times pf \rightarrow pf$. Analogous *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 $s \in Q$ and $lab \in Lab$, $sucs(s)$ resp. $sucsL(lab)(s)$ is finite. Hence, if Q is finite, the global semantics of a modal formula is stepwise computable if all free variable occurrences in φ have positive polarity.

Given a Kripke structure K , the above interpretations of sf and pf extend to a Σ -algebra, called the **modal algebra over K** : Let $s \in Q$, $lab \in Lab$, $\varphi, \psi \in sf^A \cup pf^A$, $b : Var \rightarrow \mathcal{P}(Q)$ and $c : Var \rightarrow \mathcal{P}(path(K))$.

$$\begin{aligned} at^A(b) &=_{def} val(at) \\ at^A(c) &=_{def} \{p \in path(K) \mid p_0 \in val(at)\} \\ true^A(b) &=_{def} Q \\ false^A(b) &=_{def} \emptyset \\ \neg^A(\varphi)(b) &=_{def} Q \setminus \varphi(b) \\ (\varphi \vee^A \psi)(b) &=_{def} \varphi(b) \cup \psi(b) \\ (\varphi \wedge^A \psi)(b) &=_{def} \varphi(b) \cap \psi(b) \\ \varphi \Rightarrow^A \psi &=_{def} \neg^A(\varphi) \vee^A \psi \end{aligned}$$

$$\begin{aligned}
EX^A(\varphi) &=_{def} \text{imgsShares}(Q)(\text{sucs}) \circ \varphi \\
AX^A(\varphi) &=_{def} \text{imgsSubset}(Q)(\text{sucs}) \circ \varphi \\
\langle lab \rangle^A(\varphi) &=_{def} \text{imgsShares}(Q)(\text{sucsL}(lab)) \circ \varphi \\
[lab]^A(\varphi) &=_{def} \text{imgsSubset}(Q)(\text{sucsL}(lab)) \circ \varphi \\
x^A(b) &=_{def} b(x) \\
next^A(\varphi)(c) &=_{def} \{p \in \text{path}(K) \mid \lambda i. p_{i+1} \in \varphi(c)\} \\
(\mu x)^A(\varphi)(b) &=_{def} \text{lfp}(\varphi(\lambda y. b[y/x]))(\emptyset) \\
(\nu x)^A(\varphi)(b) &=_{def} \text{gfp}(\varphi(\lambda y. b[y/x]))(Q)
\end{aligned}$$

$f[a/x]$ 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 synonymous operators on path formulas are interpreted analogously: just replace the state valuation b by the path valuation c . The functions lfp and gfp (“least” and “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 applied to $s = \emptyset$ resp. $s = Q$ and provided that Q is finite, the iteration terminates and—by Kleene’s fixpoint theorem—return the least resp. greatest solution of the equation $x = \varphi$ in $\mathcal{P}(Q)$.

All operators of Σ that are not interpreted directly in the modal algebra over K can be reduced to fixpoints:

$$\begin{aligned}
EF(\varphi) &= \mu x(\varphi \vee EX(x)) && \text{finally} \\
AF(\varphi) &= \mu x(\varphi \vee (EX(\text{true}) \wedge AX(x))) \\
EG(\varphi) &= \nu x(\varphi \wedge (AX(\text{false}) \vee EX(x))) && \text{generally} \\
AG(\varphi) &= \nu x(\varphi \wedge AX(x)) \\
\varphi EU \psi &= \mu x(\psi \vee (\varphi \wedge EX(x))) && \text{until} \\
\varphi AU \psi &= \mu x(\psi \vee (\varphi \wedge AX(x))) \\
F(\varphi) &= \mu x(\varphi \vee next(x)) && \text{finally} \\
G(\varphi) &= \nu x(\varphi \wedge next(x)) && \text{generally} \\
\varphi U \psi &= \mu x(\psi \vee (\varphi \wedge next(x))) && \text{until}
\end{aligned}$$

Provided that the Kripke structure K has only finitely many states, each state formula φ can be completely evaluated in the modal algebra over K . For this purpose Expander2 derives K from a specification like TRANS and thus makes the following simplification rules applicable to state formulas φ , states q and state sets qs , respectively:

State formula evaluation

$$\begin{aligned}
&\frac{\varphi(q)}{\text{True}} \quad q \in \varphi^A && \frac{\varphi(q)}{\text{False}} \quad q \in Q \setminus \varphi^A \\
(1) \quad \frac{\text{sols}(\varphi)}{\varphi^A} && (2) \quad \frac{\text{embed}(qs)}{\text{transition graph of } K \text{ with each state } q \in qs \text{ replaced by SAT}(q)}
\end{aligned}$$

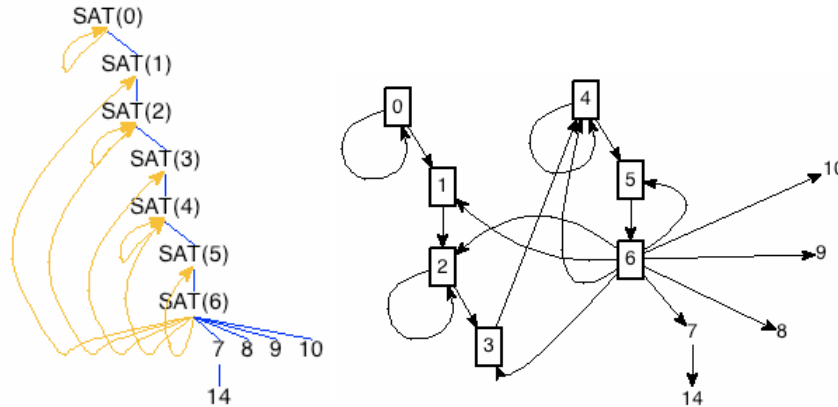


Fig. 3. Results of applying rules (1) and (2) and, on the right, the function `drawSF` of `TRANS` to `solsEFatom$less$4`

The following Kripke structure models a system of n processes (`procs`) that have access to a critical region. The model is specified along the lines of [14], Example 3.3.1, where the system is described for two processes. The predicates `live` (*liveness*), `nonBlock` (*non-blocking*) and `noSeq` (*no strict sequencing*) have also been taken from there. States are pairs (xs, ys) consisting of the list `xs` of waiting processes and the list `ys` of processes in the critical region.

```
-- MUTEX
preds:      Idle Wait Crit live nonBlock noSeq true false atom
            not \ / \ \ `then` or and EX AX EF AF EG AG `EU`
                                                    -- predicates

constructs:  idle wait crit
defunctors:  procs start states atoms drawM frame
fovvars:     xs ys ats

axioms:
start == ([], []) & states == [start] &
(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

atoms == mapprodL($) [[idle,wait,crit],procs] &
            -- mapprodL(f) [s1,...,sn] = f(s1 x ... x sn)
idle$x -> branch $ filter(rel((xs,ys),x `in` procs-xs-ys)) $ states &
wait$x -> branch $ filter(rel((xs,ys),x `in` xs)) $ states &
crit$x -> branch $ filter(rel((xs,ys),x `in` ys)) $ states &

(Idle <==> atom . idle) &
(Wait <==> atom . wait) &
(Crit <==> atom . crit) &
(live$x <==> AG $ Wait(x) `then` AF$Crit$x) &
(nonBlock$x <==> AG $ Idle(x) `then` EX$Wait$x) &
(noSeq$x <==> AG $ EF $ Crit(x) /\
                (Crit(x) `EU`
```

```

(not(Crit(x)) /\
 (and(map(not.Crit)$procs-[x]) `EU`
  Crit$x)))) &

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

```

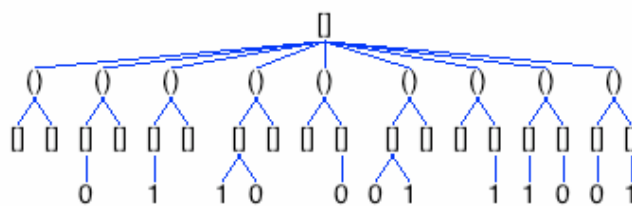


Fig. 4. The result of applying rule (1) to $\text{sols}\$and\$map(\text{live})\$\{0,1\}$

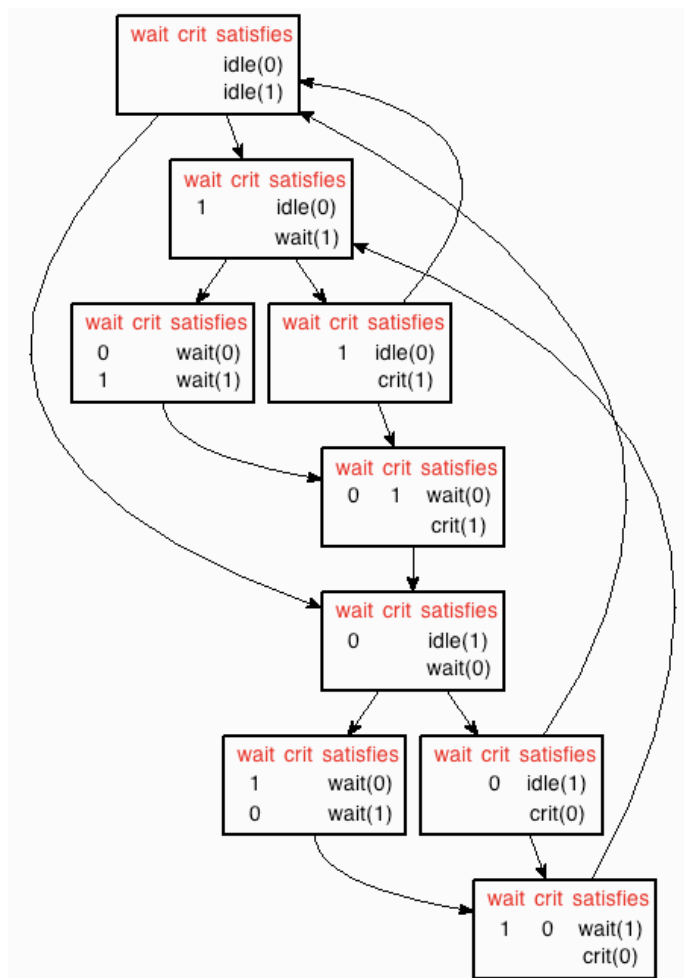


Fig. 5. The Kripke model that Expander2 derives from MUTEX and the function drawM

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 `upWith` and `downWith` will not terminate. However, as fixpoint operators are ubiquitous in model design, so are the key proof rules *induction*, *coinduction* and *expansion* for properties of a fixpoint, say $a = (a_1, \dots, a_n)$. If a solves the equation $(x_1, \dots, x_n) = t(x_1, \dots, x_n)$, expanding a term or formula φ means replacing all occurrences of a (or components thereof) in φ by (the corresponding projections on) $t(a)$. Expansion is sound for all solutions of the equation, induction and coinduction only for the least resp. greatest one.

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

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

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 (or a component thereof). The redices of induction and coinduction, however, are implications with the fixpoint as its premise resp. conclusion:

Induction and coinduction

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

The arrow \uparrow indicates that the succedent of the rule, i.e., the formula below the horizontal line, implies the antecedent, but not necessarily vice versa. Anyhow, we write the conclusion of the implication above the line because the rule syntax should reflect the order in which the rules are applied in a proof.

Hence it may happen that induction or coinduction is applicable to a valid formula, but the rule succedent does not hold true. Then the co/induction hypothesis, which is given by ψ , was too weak (resp. too strong). ψ must then be *generalized*, i.e. extended to some δ by adding a factor (resp. summand). It follows from the incompleteness of second-order logic that the candidates for δ cannot be enumerated. The following rule shows the boundaries within which δ must be

searched for:

Second-order induction and coinduction

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

The soundness of (first-order) co/induction is easy to show: $\mu x_1 \dots x_n. \varphi$ and $\nu x_1 \dots x_n. \varphi$ denote solutions of the equation $(x_1, \dots, x_n) = \varphi$ 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 $(x_1, \dots, x_n) = \varphi$ in A is the least resp. greatest tuple $B = (B_1, \dots, B_n)$ 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 the co/induction rule 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* such redices:

Implication splitting Suppose that φ and ψ are simplified.

$$\frac{\varphi \Rightarrow \psi_1 \wedge \dots \wedge \psi_n}{\varphi \Rightarrow \psi_1 \wedge \dots \wedge \varphi \Rightarrow \psi_n} \Downarrow \frac{\varphi_1 \vee \dots \vee \varphi_n \Rightarrow \psi}{\varphi_1 \Rightarrow \psi \wedge \dots \wedge \varphi_n \Rightarrow \psi} \Downarrow$$

The above statements on (the necessity of) generalizations should convince the reader that the co/inductive provability of the premise of a splitting rule does not imply the co/inductive provability of its conclusion! On the other hand, if implication splitting does not interfere with co/induction, it *should* be applied because, as a hidden distribution of \wedge over \vee , it is a step towards a disjunctive normal form. More crucial than Boolean simplifications is the simplifier's handling of quantified variables. Here the aim is to move quantifiers such that most of them occur in existentially quantified conjunctions of equations or, dually, universally quantified disjunctions of inequations. Such subformulas are then treated separately by term replacement, atom splitting and atom removal, which often reduces the number of variables or even eliminates all of them.

At first, the simplifier of Expander2 treats a formula as a term to be evaluated bottom-up by applying interpretations of the involved operators in a suitable algebra, say B . In contrast to the modal algebra, B is a term algebra, i.e. it consists of formulas, but usually smaller ones than the original equivalent ones. For instance, an existential quantifier is (1) merged with directly following ones, (2) distributed over a subsequent implication or disjunction and (3) restricted to those variables that have free occurrences in the quantified formula.

If a formula has been evaluated in this way, the simplifier applies rules (including the ones presented in this and the previous section) only to outermost redices as described above. Since path formulas cannot be evaluated in the modal algebra, we specify temporal operators in terms of further simplification rules that will be used in proofs together with expansion and co/induction.

-- LTLs

```
preds:      P Q true false hatom not \ / \ `then` F G `U`
constructs: blink                               -- the stream 010101...
```

```

fovars:      at s
hovars:      X P Q      -- higher-order variables
axioms:
  (true$s <==> True)
  & (false$s <==> False)
  & (hatom(at)$s <==> at -> head$s)
  & (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$P <==> NU X.(P\X.tail))      -- generally
  & ((P`U`Q) <==> MU X.(Q\/(P\X.tail)))  -- until
  & head$blink == 0
  & tail$blink == 1:blink      -- coalgebraic specification of blink

```

Except for the fixpoint operators, the axioms directly implement the interpretation of temporal operators in the modal algebra. State operators could be axiomatized analogously. However, this is not needed if the underlying Kripke structure is finite. The formula `hatom(at)$s` checks whether the head of the path `s` satisfies $at \in At$ (see section 2). The functions `head` and `tail` are defined as in Haskell. They provide the destructors of the data type of paths and are used here for specifying the stream `010101...`. A point in terms denotes function composition. The conjecture

$$s = \text{blink} \mid s = 1:\text{blink} \implies G(F\$(=0).\text{head})\$s \quad (1)$$

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:

$$s = \text{blink} \mid s = 1:\text{blink} \implies \text{NU } X.(F((=0).\text{head})\backslash X.\text{tail})\$s \quad (2)$$

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

$$\text{All } s:(s = \text{blink} \mid s = 1:\text{blink} \implies F((=0).\text{head})\backslash (\text{rel}(s, s=\text{blink} \mid s=1:\text{blink}).\text{tail}))\$s) \quad (3)$$

47 further simplification steps including three expansion steps turn (3) into *True*. The entire proof goes through automatically. The second sample proof is based on a model of a microwave controller [5]:

```

-- MICROS
specs:      LTLS      -- imported specification
constructs: start close heat error SAT
defuncts:   inits states atoms drawK
fovars:     ats
axioms:
inits == [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
& start -> branch[2,5,6,7]
& close -> branch[3,4,5,6,7]
& heat -> branch[4,7]
& error -> branch[2,5]
& drawK == wtree$fun(x`sat`ats,rframe$matrix[x,satisfies(ats)],x,x)

```

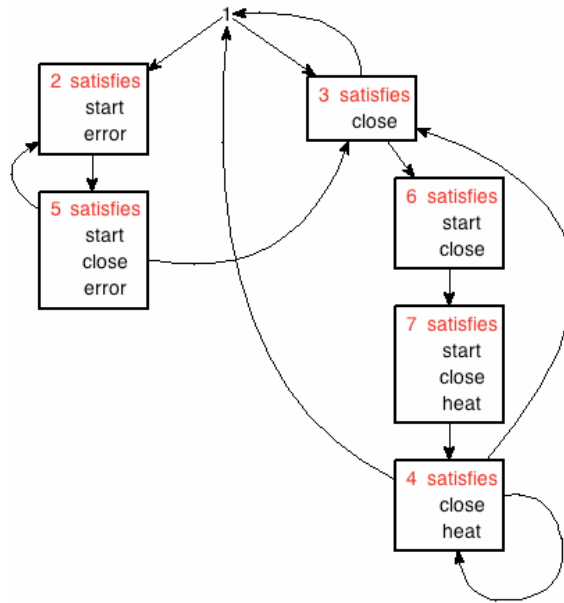


Fig. 6. The Kripke model that Expander2 derives from MICROS and the function drawK

The conjecture

$$G(\text{hatom}\$error)\$s \implies G(\text{not}\$\text{hatom}\$heat)\$s \tag{1}$$

says that a path consisting of error states never contains heat states. By the G-axiom of LTL, (1) simplifies to:

$$\begin{aligned} & \text{NU } X. (\text{hatom}(error) / \backslash X.tail) \$s \implies \\ & \text{NU } X. (\text{not}(\text{hatom}\$heat) / \backslash X.tail) \$s \end{aligned} \tag{2}$$

Applying the coinduction rule yields:

$$\begin{aligned} & \text{All } pa: (\text{NU } X. (\text{hatom}(error) / \backslash X.tail) \$s \implies \\ & \quad (\text{not}(\text{hatom}\$heat) / \backslash \\ & \quad \quad (\text{rel}(s, \text{NU } X. (\text{hatom}(error) / \backslash X.tail) \$s).tail)) \$s) \end{aligned} \tag{3}$$

41 further simplification steps lead (3) to *True*. Three expansion steps are needed, and the entire proof goes through automatically.

5 Model checking by co/resolution and co/induction

Both evaluation and simplification regard modal formulas as representations of data, namely (tuples of) sets. This is the actual reason for the algebraic flavor of modal logics: their operators

denote *functions* that generate 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 the least resp. greatest relations (on a given data model) that satisfy the axioms. Details of this approach and its connection with relational and functional programming can be found in [19, 20].

For checking Kripke structures with co/Horn logic we use co/Horn axioms in addition to the modal algebra of section 3 and the specification LTLs of section 4. For lack of space we only present a part of the corresponding specifications that includes the axioms needed in the proofs given later.

```
-- CTL
preds:      P Q true false atom not \/ /\ `then` EX EF AF `EU` `AU`
copreds:    AX EG AG                                -- copredicates
fovvars:    at st st'
hovvars:    P Q
axioms:
  (EX(P)$st <=== st -> st' & P(st'))                -- next
  & (AX(P)$st ==> (st -> st' ==> P(st')))
  & (EF(P)$st <=== P$st | EX(EF(P))$st)             -- finally
  & (AF(P)$st <=== P$st | AX(AF(P))$st)
  & (EG(P)$st ==> P$st & EX(EG(P))$st)             -- generally
  & (AG(P)$st ==> P$st & AX(AG(P))$st)
  & ((P`EU`Q)$st <=== Q$st | P$st & EX(P`EU`Q)$st) -- until
  & ((P`AU`Q)$st <=== Q$st | P$st & AX(P`AU`Q)$st)

-- LTL
preds:      P Q true false hatom not \/ /\ `then` F `U`
copreds:    G
fovvars:    s
hovvars:    P Q
axioms:
  (F(P)$s <=== P$s | F(P)$tail$s)                  -- finally
  & (G(P)$s ==> P$s & G(P)$tail$s)                 -- generally
  & ((P`U`Q)$s <=== Q$s | P$s & (P`U`Q)$tail$s)   -- until
```

The direction of the implication arrow (<=== or ==>) indicates whether the axiom is called a Horn or a co-Horn clause and the relational expression on its left-hand side a predicate or a co-predicate and thus interpreted as the least or greatest relation satisfying the axiom(s). When applied in a logical derivation, a co/Horn clause is always applied from left to right. Besides premise and/or conclusion a clause may contain a *guard* that confines redices to formulas that unify with the left-hand side (premise resp. conclusion) *and* satisfy the guard (see the co/resolution rules given below).

An expansion step is replaced by the simultaneous application of all axioms with the same relational expression on the left-hand side:

Parallel resolution upon the predicate p

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

where $\gamma_1 \Rightarrow (p(t_1) \Leftarrow \varphi_1), \dots, \gamma_n \Rightarrow (p(t_n) \Leftarrow \varphi_n)$ are the (Horn) axioms for p (with guards $\gamma_1, \dots, \gamma_n$).

Parallel coresolution upon the copredicate p

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

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

In both rules, \vec{x} is a vector of ‘new’ variables, for all $1 \leq i \leq k$, $t \sigma_i = t_i \sigma_i$, $\gamma_i \sigma_i \vdash \text{True}$ and $Z_i = \text{var}(t_i, \varphi_i)$, and for all $k < i \leq n$, t is not unifiable with t_i .

As in section 4, co/induction 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 a simplification rule, 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. Restricted to the proof of bisimilarities (relations describing behavioral equality), this incremental procedure is also known as *circular coinduction* [8, 12].

Incremental induction upon the predicate p

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

AX_p denotes the set of axioms for p . When the first rule is applied, p' is stored as a new copredicate with the axiom $p'(x) \Rightarrow \psi(x)$. When the second rule is applied, the axiom $p'(x) \Rightarrow \delta(x)$ is added.

Incremental coinduction upon the copredicate p

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

When the first rule is applied, p' is stored as a new predicate with the axiom(s) $p'(x) \Leftarrow \psi(x)$ and, if p is behavioral equality, Horn clauses that establish p' as an equivalence relation. When the second rule is applied, the axiom $p'(x) \Leftarrow \delta(x)$ is added.

Co/resolution and co/induction complement each other in the way axioms work together with conjectures in proofs. Roughly said, co/resolution applies axioms to conjectures and the proof proceeds with the modified conjectures. Conversely, co/induction applies conjectures to axioms and establishes the modified axioms as new conjectures.

The generalization of the above rules to several co/predicates (the first-order analog of a fix-point formula with several bound variables) is straightforward.

In contrast to section 4, incremental coinduction allows us to start a proof that the stream `blink` is fair with the conjecture $\psi = G(F\$(=0) . head) \$blink$ and derive the factor

$$\delta = G(F\$(=0) . head) \$1:blink$$

of the generalized conjecture $\psi \wedge \delta$ within the proof of ψ . Indeed, applying incremental coinduction to ψ yields the new conjecture

$$\text{All } P \text{ } s: (P = F((=0) . head) \ \& \ s = blink \implies) \\ P(s) \ \& \ G0(P)\$tail\$s \tag{1}$$

$G0$ is the predicate p' created during rule application (see above). Its axiom is:

$$G0(z0)\$z1 \iff z0 = F((=0) . head) \ \& \ z1 = blink \tag{ax1}$$

Six simplification steps transform (1) into:

$$F((=0) . head) \$blink \ \& \ G0(F((=0) . head)) \$ (1:blink) \tag{2}$$

Parallel resolution upon F and subsequent simplification steps remove the first factor of (2). The second factor is a redex for the second rule of incremental coinduction. Hence (2) is turned into:

$$\text{All } P \text{ } s: (P = F((=0) . head) \ \& \ s = 1:blink \implies) \\ P(s) \ \& \ G0(P)\$tail(s) \tag{3}$$

and a further axiom for $G0$ is created:

$$G0(z2)\$z3 \iff z2 = F((=0) . head) \ \& \ z3 = 1:blink \tag{ax2}$$

Five simplification steps transform (3) into:

$$F((=0) . head) \$ (1:blink) \ \& \ G0(F((=0) . head)) \$blink \tag{4}$$

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

$$G(\text{hatom}\$error)\$s \implies G(\text{not}\$\text{hatom}\$heat)\$s \tag{1}$$

(see MICROS in section 4) can also be proved by incremental coinduction and co/resolution. The coinduction rule adds

$$G0(z0)\$s \iff G(\text{hatom}\$error)\$s \ \& \ z0 = \text{not}\$\text{hatom}\$heat$$

to the set of axioms. Subsequent coresolution and simplification steps automatically lead to:

$$\text{All } s: (G(\text{hatom}\$error)\$s \implies G0(\text{not}\$\text{hatom}\$heat)\$tail\$s) \tag{2}$$

(2) admits both coresolution upon G and resolution upon $G0$. The first step would lead the proof into a cycle because the only axiom for G (see LTL) is recursive (G occurs on both sides of the axiom). The axiom for $G0$, however, is non-recursive—as axioms introduced by co/induction steps always are. Hence we choose the resolution step and obtain after simplification:

$$\text{All } s: (G(\text{hatom}\$error)\$s \implies G(\text{hatom}\$error)\$tail\$s) \tag{3}$$

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

6 Beyond model checking

We are also about to integrate inductive techniques such as those for reasoning about systems communicating between a varying number of processes [4, 5].

```
-- MUTEXco

specs:          CTL
preds:          Idle Wait Crit enabled safe noSeq
copreds:        others
constructs:     c
defuncts:       request enter leave posi maxwait weight
fovvars:        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) &

init(s)++[last$s] == s &

(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]) &

(others(P)(x)$st ==> (x /= y ==> P(y)$st)) &

(noSeq$x <==> EF $ Crit(x) /\ (Crit(x) `EU`
                               (not(Crit$x) /\
                                (others(not.Crit)(x) `EU` Crit$x))))

theorems:
```

```

(posi(x)$s++s' = posi(x)$s <=== x `in` s) &
(length$init$s = length(s)-1 <=== x `in` s) &
(x-y > x-suc(y) <=== x > y) &
(x-z > y-z <=== x > y) &
((xs,ys) >> st <=== c `in` xs & (xs,ys) -> st) &           -- transDesc
(x `in` xs' | x `in` ys' <=== x `in` xs & (xs,ys) -> (xs',ys')) &
                                                                    -- transIn
(True ==> Any xs ys: st = (xs,ys)) &
(x `in` init$y:s | x = last$y:s <=== x = y | x `in` s)

```

conjects:

```

([],[]) -> st &
(safe$st ==> AG(safe)$st) &           -- AGsafe: proof by coinduction on AG
(Idle(x) (xs,ys) & length$xs < maxwait ==> EX(Wait$x) (xs,ys)) &
                                                                    -- EXwait: proof by resolution on EX
(c `in` xs | c `in` ys ==> AF(Crit$c) (xs,ys)) &
                                                                    -- AFcrit: proof by Noetherian induction wrt >>

```

Derivation of

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':

Algebraic Model Checking

```
(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 presented three approaches to the verification of Kripke structures based on a labelled or unlabelled transition system (also called Kripke frame) or a mixture thereof. The first method consists in evaluating modal formulas in an algebra of sets of states or paths. For state formulas, the evaluation procedure is part of the simplification component of Expander2. Since fixpoint computations are involved, model checking by evaluation is restricted to models with a finite set of states.

The second technique uses simplification rules, which extend the modal algebra of the first approach by expansion, induction and coinduction. This allows us to prove also path formulas and to verify Kripke models with infinitely many states. We have described and illustrated a strategy of applying expansion, coinduction and other simplification rules that is complete: it terminates whenever any other strategy would also terminate.

The third approach is based on our previous work [18, 19] on co/Horn logic where co/Horn clauses axiomatize least resp. greatest relational fixpoints and parallel co/resolution provides the counterpart of expansion in pure simplification proofs. Coinduction as used in the second approach is replaced by incremental coinduction, a proof rule that admits the automatic—and often inevitable—generalization of the respective conjecture. Incremental coinduction was in-

spired by the method of circular coinduction [8, 12] that, however, is tailored to the proof of equations.

The first method may be compared with other model checkers, which also hide all logical inference involved from the user by turning both the Kripke structure and the formula to be proved into some efficiently processible internal representation and then running a deterministic algorithm that checks the formula in a single visible step. The second and the third method work on both the Kripke structure's internal representation—if there any—and its specification given by rewrite rules (Horn clause axioms for \rightarrow) and thus admit the treatment of infinite-state systems. The formula to be proved, however, is processed in its original form. With the second method, the proof goes through automatically—provided that co/inductive subconjectures appear as suitable implications and generalizations are not needed (see section 4). Similar co/induction rules were implemented in Isabelle [7, 25], but their use needs manual control. PVS [11, 13] and CLAM [6] also admit coinduction, but—like circular coinduction (see section 5)—only for proving bisimilarities. Manual control is needed for our third method, but this is offset by more general co/induction redices and the possibility to generalize co/inductive conjectures during proof construction.

If proof assistants for Kripke structures were put on a line, starting from the most efficient to the most powerful ones, model checkers would occupy one end and established theorem provers the other. Our methods distribute over the whole line and their integration in Expander2 shows that model checking and (modal-)theorem proving can be performed simultaneously.

More and greater examples can be found in [24] and the *Examples* directory of Expander2. The examples of sections 5 and 6 reveal the goal of our research: to integrate model checking, which is restricted to transition systems with mostly finite sets of unstructured states, - via the algebraic approaches presented in this paper - into the verification of co/algebraic data types with respect which those transition systems form not more that a simple, particular case.

We are also about to integrate inductive techniques such as those for reasoning about systems communicating between a varying number of processes [4, 5].

Bibliography

- [1] J.C.M. Baeten, W.P. Weijland, *Process Algebra*, Cambridge University Press 1990
- [2] J. van Benthem, J. Bergstra, *Logic of Transition Systems*, J. Logic, Language and Information 3 (1995) 247-283
- [3] C. Cirstea, A. Kurz, D. Pattinson, L. Schröder, Y. Venema, *Modal Logics are Coalgebraic*, The Computer Journal, to appear
- [4] E.M. Clarke, O. Grumberg, S. Jha, *Verification of Parameterized Networks*, ACM TOPLAS 19 (1997) 726-750
- [5] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press 1999
- [6] 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

- [7] J. Frost, *A Case Study of Co-induction in Isabelle*, Report, Computer Laboratory, University of Cambridge 1995
- [8] J. Goguen, K. Lin, G. Rosu, *Conditional Circular Coinductive Rewriting with Case Analysis*, Proc. WADT'02, Springer LNCS 2755 (2003) 216-232
- [9] H. P. Gumm, *Universal Coalgebras and their Logics*, AJSE-Mathematics, to appear
- [10] J. Goguen, G. Malcolm, *A Hidden Agenda*, Theoretical Computer Science 245 (2000) 55-101
- [11] H. Gottlieb, *Co-inductive Proofs for Streams in PVS*, Report, Queen Mary, University of London 2007
- [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] P. Padawitz, *Proof in Flat Specifications*, in: Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Report, Springer (1999) 321-384
- [19] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165
- [20] P. Padawitz, *Dialgebraic Specification and Modeling*, draft, fdit-www.cs.tu-dortmund.de/~peter/Dialg.pdf
- [21] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fdit-www.cs.tu-dortmund.de/~peter/Expander2.html
- [22] 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
- [23] P. Padawitz, *Expander2: Program verification between interaction and automation*, Proc. 15th Workshop on Functional and (Constraint) Logic Programming, Elsevier ENTCS 177 (2007) 35-57

- [24] P. Padawitz, *Algebraic Model Checking and more*, fdit-www.cs.tu-dortmund.de/~peter/Haskellprogs/CTL.pdf
- [25] L. C. Paulson, *Mechanizing Coinduction and Corecursion in Higher-Order Logic*, *J. Logic and Computation* 7 (1997) 175-204
- [26] J. Rutten, *Universal Coalgebra: A Theory of Systems*, *Theoretical Computer Science* 249 (2000) 3-80
- [27] C. Stirling, *Modal and Temporal Logics*, in: *Handbook of Logic in Computer Science*, Clarendon Press (1992) 477-563