A Theory of Hints in Model Checking*

Markus Kaltenbach[1] and Jayadev Misra[2]

[1] Transmeta Corporation, Santa Clara, CA 95054
markus@transmeta.com
[2] Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX 78712
misra@cs.utexas.edu

Abstract. Model checking, in particular symbolic model checking, has proved to be extremely successful in establishing properties of finite state programs. In most cases, the proven properties are safety properties stating that the program never executes outside a specified set of states. But another important class of properties, progress (liveness) properties, which state that program execution eventually reaches some specified set of states, has been difficult to model-check as they, typically, involve doubly-nested fixpoint computations. In this paper, we propose that progress properties can be checked more efficiently if they are accompanied by *hints* as to why they hold. We develop a theory in which hints are given as regular expressions over the actions of the program. We derive a number of inference rules and algebraic properties of hints. Empirical evidence suggests that hints can significantly improve the efficiency of model checking.

0 Introduction

Model checking [CE81,CES86] has been one of the most successful recent techniques for automated verifications of finite state systems. Symbolic model checking ([McM93], [BCM91]) has made it possible to analyze systems of very large size, such as those arising in circuit and protocol designs.

The typical property that is checked for a deterministic system, such as a circuit, is that the outputs are the desired ones given the appropriate inputs, i.e., that the circuit outputs are a specified function of its inputs. In many asynchronous systems, such as communication and synchronization protocols, such properties are not adequate to specify the system behavior. Instead, we specify a set of *safety* and *progress* properties to describe the system. Informally, safety properties express that "Something will *not* happen", whereas progress properties state that "something *must* happen"([Lam77], [AS85]).

Safety properties can often be checked more efficiently than by just performing a state space exploration (which is the essence of the model checking approach), when adding some deductive reasoning by checking certain predicates

* Partially supported by NSF grant CCR–9803842.

on each state and its successor states, and supplementing this with finding suitably strong invariants. Decision procedures for progress properties have proved to be far less efficient because such properties cannot be characterized by local conditions asserting how pairs of successive states are related to each other (if this were possible then progress properties could be characterized by finite execution prefixes as well). Establishing progress properties in theorem provers requires transfinite induction over well-founded sets, whereas in model checking a doubly-nested fixpoint computation is required to establish most progress properties under any reasonable definition of fairness. Our experience confirms that progress properties are hard to check; in a model checker developed by the first author, straightforward verifications of progress properties were at least two orders of magnitude slower than the safety properties of those systems. Therefore, in practice, only safety aspects of systems are often verified; progress properties are usually dismissed or argued about informally.

In this paper, we show that progress properties can be often be checked efficiently by supplying hints to the checker. Typically, hints have been used in automated verification systems to guide the prover through a search space interactively, or in guiding the automated checking of safety properties ([BRS00]). In our work, a hint is a regular expression over the actions of the program, suggesting how a goal state is achieved. A central theorem asserts that a property established under any hint is a property of the program. We motivate our work using a small example below.

Consider a program consisting of three actions – [up], [down], [set] – that operate on two program variables, an integer n and a boolean b. The actions are as follows.

$$\begin{array}{lll} [\text{up}] & \neg b \rightarrow & n := n + 1 \\ [\text{down}] & & n := n - 1 \\ [\text{set}] & & b := \text{true} \end{array}$$

The action [up] increments n only if $\neg b$ holds, otherwise (if b holds) its execution has no effect. Actions are executed forever in arbitrary order subject to the fairness rule that every action be executed infinitely often. It follows then that eventually $n \leq 0$ holds no matter what the initial state is. If n is restricted to a finite range, a model checker can verify this property. However, even for moderate ranges of n the computation becomes prohibitively expensive.

Our approach is to provide a hint to the checker as to why the property holds. The hint embodies our intuitive understanding of how $n \leq 0$ is achieved. For this example, we realize that eventually action [set] will be executed, thus setting b to true, and from then on neither [up] nor [set] has any effect on the program state; each subsequent execution of [down] decreases n, establishing $n \leq 0$ eventually. We encode this reasoning in the hint $[\text{set}][\text{down}]^*$ which is a regular expression over the actions of the program; i.e., any execution sequence consisting of [set] and a sufficiently large number of [down] actions achieves $n \leq 0$. The key observation is that *every valid execution sequence, under our*

notion of fairness, includes every sequence described by a regular expression as a subsequence. Thus, a property proven under a hint (i.e., for specific execution sequences) holds for all execution sequences as well. A model checker can use a hint to eliminate some unnecessary fixpoint computations and to simplify others.

The precise operational meaning of progress under a given hint is surprisingly difficult to state succinctly; see Sect. 3.5. A formal definition using predicate transformers is given in Sect. 3.1 and an algebra of hints that is similar to the Kleene algebra of regular expressions is developed in Sect. 3.3. Hints have been incorporated into a model checker developed by the first author, and they seem to deliver significant improvements in performance; see Sect. 4.

We couch our discussion in terms of the UNITY logic [CM88,Mis95b,Mis95a] though the ideas can be applied to other formalisms. We discovered that the definition of the *leads-to* operator of UNITY mirrors the structure of regular expressions: transitivity of *leads-to* corresponds to concatenation, finite disjunctivity to choice, and general disjunctivity (or induction) to Kleene closure. Also, UNITY's notion of unconditional fairness in execution – every action is executed eventually – guarantees that every execution sequence includes any given finite sequence of actions as a subsequence. This observation permits us to prove a progress property for all execution sequences by proving it for a certain class of finite executions.

1 Background

We give a brief overview of the UNITY programming notation and its temporal logic, to the extent necessary for this paper. See [CM88] for the initial work and [Mis01] for some recent developments.

1.1 A Programming Notation

The computational model of UNITY is that of a deterministic, total, labeled state transition system. This model is well suited for describing many common classes of systems (e.g. hardware circuits or protocols).

A program consists of (1) a *declare* section defining the state space of the program, (2) an *initially* section specifying the initial conditions, and (3) an *assign* section specifying the actions of the program (its transition relation) as guarded multiple assignment statements, that are deterministic and terminating.

An execution of a program consists of selecting a start state satisfying the conditions of the *initially* section and then repeatedly selecting statements (from the *assign* section) and executing them (if the guard of a selected statement evaluates to false in the current state, the entire statement is equivalent to a *skip* operation, i.e., it does not change the state). The selection of statements is subject to the unconditional fairness constraint that every statement be selected infinitely often.

1.2 UNITY Logic

The UNITY logic, a fragment of linear temporal logic, has proof rules for reasoning about properties of programs. Different from many state-based computational models that reason about individual executions of programs, the UNITY operators characterize properties of programs, i.e., properties of *all* unconditionally fair execution sequences.

In the following, we introduce the UNITY operators and some rules for reasoning with them to the extent needed for this paper. Proofs of most rules are straightforward and can be found in chapters 5 and 6 of [Mis01]. In our presentation we make use of the following notation: for a program F the predicate $F.I$ denotes the initial states as characterized by the **initially** section of F, and $F.A$ denotes the set of the actions of F. For any state predicate p of F we write $[p]$ to denote the universal quantification of p over the state space of F [DS90]. We write $\mathbf{wp}\,.\alpha$, where α is any action in $F.A$, for the *weakest precondition* predicate transformer; i.e., for a state predicate p, the predicate $\mathbf{wp}\,.\alpha.p$ characterizes those states from which the execution of α terminates in a state satisfying p[3].

Safety. The fundamental safety operator of UNITY is *constrains*, or **co** for short. The **co** operator is a binary relation over state predicates and is defined as follows:

$$\frac{\langle \forall \alpha : \alpha \in F.A : [p \Rightarrow \mathbf{wp}\,.\alpha.q] \rangle,\ [p \Rightarrow q]}{p\ \mathbf{co}\ q}$$

The property p **co** q asserts that in any execution, a state satisfying p is always followed by a state satisfying q. In order to model stuttering steps p is required to imply q. Several other safety operators are expressed in terms of **co** :

$$\frac{p\ \mathbf{co}\ p}{\mathbf{stable}\ p} \qquad \frac{\langle \forall e :: \mathbf{stable}\ x = e \rangle}{\mathbf{constant}\ x}$$

$$\frac{\mathbf{stable}\ p,\ [F.I \Rightarrow p]}{\mathbf{invariant}\ p} \qquad \frac{p \wedge \neg q\ \mathbf{co}\ p \vee q}{p\ \mathbf{unless}\ q}$$

A predicate is *stable*, if it remains true once it becomes true. An expression x is *constant* if its value never changes, i.e., if for any possible value e the predicate $x = e$ is stable. A predicate is *invariant* if it is stable and holds in all initial program states. Finally, p **unless** q holds if starting in any state that satisfies p either p continues to hold forever, or holds up to (but not necessarily including) a state satisfying q.

[3] Recall that in the UNITY model all actions are terminating.

The Substitution Axiom. The operation of a program is over the reachable part of its state space. The UNITY proof rules, however, do not refer to the set of reachable states explicitly. Instead, the following *substitution axiom* is used to restrict attention to the reachable states.

$$\frac{\textbf{invariant } p}{[p]}$$

Since an invariant of a program is true over the reachable part of the state space, it is equivalent to true in any program property. Thereby the substitution axiom allows us to replace any invariant predicate of a program F by true (and vice versa) in any proof of a property of F.

Progress. The basic progress property is **transient** ; it is a unary relation on state predicates. A transient predicate is guaranteed to be falsified in any program execution: predicate p is transient in program F if there is an action in F whose execution *in any state* in which p holds establishes $\neg p$:

$$\frac{\langle \exists \alpha : \alpha \in F.A : [p \Rightarrow \textbf{wp}\,.\alpha.(\neg p)] \rangle}{\textbf{transient } p}$$

From the unconditional fairness rule, the given action α is executed eventually, falsifying p if it held before the execution of α.

The other basic progress property is **ensures** , a binary relation on state predicates:

$$\frac{p \textbf{ unless } q,\ \textbf{transient } p \wedge \neg q}{p \textbf{ ensures } q}$$

Given p **ensures** q , from the **transient** part of the definition there is an action of F that establishes $\neg p \vee q$ starting in any state in which $p \wedge \neg q$ holds; from the **unless** part, once p holds it continues to hold up to the point at which q is established. Therefore, starting in a state in which p holds, q is established eventually.

Since there is a single rule for establishing an **ensures** property of a program, we can derive from the **ensures** rule and the substitution axiom the following equivalence:

$$\begin{array}{lcl} p \textbf{ ensures } q & \equiv & [p \wedge \neg q \Rightarrow \textbf{wco}\,.(p \vee q)] \quad \wedge \\ & & [\langle \exists \alpha : \alpha \in F.A : p \wedge \neg q \Rightarrow \textbf{wp}\,.\alpha.q \rangle] \end{array}$$

where the predicate transformer **wco** is defined by $[\textbf{wco}\,.p \ \equiv \ \langle \forall \alpha : \alpha \in F.A : \textbf{wp}\,.\alpha.p \rangle]$; in other words, $\textbf{wco}\,.p$ denotes the states from which the execution of *any* action of F terminates in a state satisfying p.

In developing our theory we make use of another property, $\mathbf{ensures}_\alpha$, which resembles **ensures** but explicitly names the helpful action, α. For an action α in $F.A$ we define:

$$p \ \mathbf{ensures}_\alpha \ q \quad \equiv \quad [p \wedge \neg q \Rightarrow \mathbf{wco}\,.(p \vee q)] \wedge [p \wedge \neg q \Rightarrow \mathbf{wp}\,.\alpha.q]$$

The fundamental progress property of UNITY is the $\mapsto$ (leads-to) operator, a binary relation on state predicates. It is the transitive, disjunctive closure of the **ensures** relation, i.e., the strongest relation satisfying the following three conditions:

$$\frac{p \ \mathbf{ensures} \ q}{p \ \mapsto \ q} \qquad \textbf{(promotion)}$$

$$\frac{p \ \mapsto \ q, \ q \ \mapsto \ r}{p \ \mapsto \ r} \qquad \textbf{(transitivity)}$$

$$\frac{\langle \forall h : h \in H : p.h \ \mapsto \ q \rangle}{\langle \exists h : h \in H : p.h \rangle \ \mapsto \ q} \quad \text{for any set } H \qquad \textbf{(disjunction)}$$

There are also several derived rules for reasoning about progress properties [CM88,Mis01]. Among them is the *induction principle*: for a well-founded set $(H, \prec)$ and a function M mapping program states to H we have

$$\frac{\langle \forall h : h \in H : p \wedge M = h \ \mapsto \ (p \wedge M \prec h) \vee q \rangle}{p \ \mapsto \ q} \qquad \textbf{(induction)}$$

The Predicate Transformer wlt. In [Kna90,JKR89] a predicate transformer, **wlt**, *weakest leads-to*, was introduced for reasoning about progress properties. It is related to the $\mapsto$ relation by

$$[p \Rightarrow \mathbf{wlt}\,.q] \quad \equiv \quad p \ \mapsto \ q$$

Using this equivalence we can model check a property of the form $p \ \mapsto \ q$ for program F by first computing $\mathbf{wlt}\,.q$ and then evaluating $(p \Rightarrow \mathbf{wlt}\,.q)$ over the reachable state space of F. A fixpoint characterization suitable for computing **wlt** is given in [JKR89].

2 Hints and Generalized Progress

Progress properties are specified using the *leads-to* operator in UNITY logic, formulae of the form $\mathbf{AG}(p \Rightarrow \mathbf{AF}q)$ in CTL, or formulae like $\mathbf{G}(p \Rightarrow \mathbf{F}q)$ in linear temporal logic. These operators specify the changes to the program state, but not the way the change is effected. Usually a program designer has some (possibly partial) knowledge of how progress is achieved in the program, either as an operational argument, in the form of a high level proof sketch, or simply

based on experience with similar programs. We propose to exploit this knowledge by providing a formal foundation for it.

We elaborate on the small example introduced in Sect. 0 to illustrate how progress properties are generalized by including explicit action-based progress information. We then argue how this theory and its associated methodology can be used in program verification.

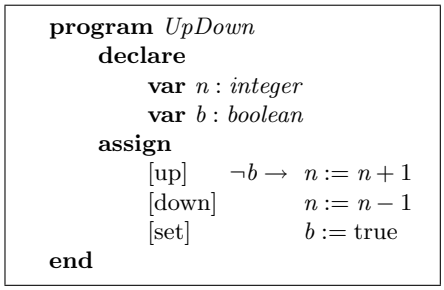

Fig. 1. Program *UpDown*

We consider the UNITY program *UpDown* shown in Fig. 1. As we have argued earlier, for any execution of this program n becomes non-positive eventually, which is expressed by the following leads-to property:

$$\text{true} \quad \mapsto \quad n \leq 0.$$

We reproduce the informal argument from Sect. 0 to justify this progress property. From unconditional fairness, action [set] will eventually be executed, thus setting b to true, and from then on neither [up] nor [set] has any effect on the program state; each subsequent execution of [down] decreases n, establishing $n \leq 0$ eventually. This argument suggests that the progress from true to $n \leq 0$ is achieved by the strategy expressed by the regular expression

$$[\text{set}][\text{down}]^*,$$

over the alphabet of actions of *UpDown*, i.e., by one [set] action followed by some finite number of [down] actions (possibly interleaved with other actions). We combine the hint with the progress property:

$$\text{true} \quad \overset{[\text{set}][\text{down}]^*}{\longmapsto} \quad n \leq 0.$$

which is a *generalized progress property* of program *UpDown*. We now show that this property is closely related to the structure of the proof of true $\mapsto$ $n \leq 0$ in the UNITY deductive system.

0.	true **ensures** b	; from program text via [set]
1.	true $\mapsto$ b	; promotion from 0
2.	$b \wedge n = k$ **ensures** $b \wedge n < k$	; from program text via [down]
3.	$b \wedge n = k \mapsto b \wedge n < k$	; promotion from 2
4.	$b \wedge \lvert n\rvert = k \mapsto (b \wedge \lvert n\rvert < k) \vee n \leq 0$	; case split and disjunction on 3
5.	$b \mapsto n \leq 0$	; leads-to induction on 4 with ; metric $\lvert n\rvert$ over the naturals
6.	true $\mapsto$ $n \leq 0$	; transitivity with 1 and 5

Steps 0 and 1 of this proof correspond to the [set] action in our proposed hint; similarly steps 2 and 3 correspond to the [down] action. The inductive argument of steps 4 and 5 establishes progress via $[\text{down}]^*$; finally, step 6, combines the subproofs through sequencing. We claim that this proof structure corresponds to the regular expression $[\text{set}][\text{down}]^*$. However, this regular expression is much less detailed than the complete proof; in particular, the state predicates needed to combine different parts of the proof are omitted. Thus, a general idea about the proof structure, or even an operational argument of a progress property can be turned into a hint.

3 Main Results about Generalized Progress

We now define the *generalized leads-to* relation, a relation of the form $p \overset{W}{\longmapsto} q$ for state predicates p and q and regular expression hints W. This definition is given inductively based on the structure of the regular expression W. In Sect. 3.2 we introduce a predicate transformer **wltr** and establish the central theorem that $[p \Rightarrow \mathbf{wltr}\,.W.q] \quad \equiv \quad p \overset{W}{\longmapsto} q$. This result permits us to prove $p \overset{W}{\longmapsto} q$ by a model checker by first computing $\mathbf{wltr}\,.W.q$, using a fixpoint characterization of $\mathbf{wltr}\,.W$, and then checking for $[p \Rightarrow \mathbf{wltr}\,.W.q]$.

Notational Conventions. Henceforth, F is a program, α is an action in $F.A$, p, p', q, r, and s are state predicates. Let U, V and W be regular expressions over the alphabet in which each symbol names an unique action of A.

3.1 Definition of Generalized Progress

First, we define a *metric* over the reachable state space of a program. In the following we use **Ord** to denote the set of ordinal numbers.

Definition 1. *A* metric M *for a program* F *is* $\{i : i \in \mathbf{Ord} : M.i\}$*, a family of state predicates which satisfy the following two conditions:*

$$[\langle \exists i : i \in \mathbf{Ord} : M.i \rangle] \qquad \textit{(MetricExh)}$$
$$\langle \forall i, j : i \in \mathbf{Ord} \wedge j \in \mathbf{Ord} : i \neq j \Rightarrow [\neg(M.i \wedge M.j)] \rangle \qquad \textit{(MetricDis).}$$

The first condition states that the predicates in M exhaust the reachable state space of F, the second asserts that any two predicates with different indices are disjoint.

The predicates in M are totally ordered by the ordering relation $\preceq$ induced by the total order relation on the ordinals:

$$\langle \forall i, j : i \in \mathbf{Ord} \wedge j \in \mathbf{Ord} : i \leq j \;\equiv\; M.i \preceq M.j \rangle$$

We proceed with the definition of the generalized leads-to relation:

Definition 2. *For a given program F, $p \overset{W}{\longmapsto} q$ is defined as follows:*

$$\begin{array}{lcll}
p \overset{\varepsilon}{\longmapsto} q & \equiv & [p \Rightarrow q] & \textbf{\textit{(AxEps)}} \\
p \overset{\alpha}{\longmapsto} q & \equiv & \langle \exists p' : [p \Rightarrow p'] : p' \;\; \mathbf{ensures}_\alpha \;\; q \rangle & \textbf{\textit{(AxAct)}} \\
p \overset{UV}{\longmapsto} q & \equiv & \langle \exists r :: (p \overset{U}{\longmapsto} r) \wedge (r \overset{V}{\longmapsto} q) \rangle & \textbf{\textit{(AxSeq)}} \\
p \overset{U+V}{\longmapsto} q & \equiv & \langle \exists r, s : [r \vee s \equiv p] : (r \overset{U}{\longmapsto} q) \wedge (s \overset{V}{\longmapsto} q) \rangle & \textbf{\textit{(AxAlt)}} \\
p \overset{U^*}{\longmapsto} q & \equiv & \langle \exists p' : [p \Rightarrow p'] : \langle \exists M : M \textit{ is a metric} : & \textbf{\textit{(AxStar)}} \\
 & & \quad \langle \forall i : i \in \mathbf{Ord} : & \\
 & & \quad\quad p' \wedge M.i \overset{U}{\longmapsto} (p' \wedge \langle \exists j : j < i : M.j \rangle) \vee q \rangle \rangle \rangle &
\end{array}$$

From these equivalences, using structural induction, we can establish that the generalized leads-to relation is well defined; see [Kal96] for a proof.

Theorem 1. *For a program F, **(AxEps)**, **(AxAct)**, **(AxSeq)**, **(AxAlt)**, and **(AxStar)** define a unique family of relations $\{W :: p \overset{W}{\longmapsto} q\}$, for all predicates p and q.*

If $p \overset{W}{\longmapsto} q$ holds in a program, this can be shown by a finite number of applications of the above proof rules (due to the finiteness of the structure of W), and, hence, there is a finite proof. In the following, we write $F \vdash p \overset{W}{\longmapsto} q$ to denote the fact that such a proof exists for F, W, p, and q.

It is worth mentioning that the use of ordinals is essential in **(AxStar)**; under unconditional fairness it is not possible, in general, to bound the number of steps required to achieve progress from any particular start state. It can only be asserted that a finite number of steps suffices; see, for example, the second program in Sect. 2. Therefore, natural numbers are not sufficient as metric; all ordinals of cardinality less than or equal to the cardinality of the size of the state space have to be considered.

3.2 Predicate Transformers for Generalized Progress

We define a family of predicate transformers $\mathbf{wltr}.W$, which is the set of states from which any execution "characterized by W" leads to a state satisfying q.

In the following we use the notation $\langle \mu Z :: f.Z \rangle$ and $\langle \nu Z :: f.Z \rangle$ to denote the least and greatest fixpoint of the predicate transformer f.

Definition 3. *For a given program F, and α an action in $F.A$,*

$$[\mathbf{wltr}\,.\varepsilon.q \quad \equiv \quad q] \qquad \textbf{\textit{(wltrEps)}}$$
$$[\mathbf{wltr}\,.\alpha.q \quad \equiv \quad \langle \nu Z :: (\mathbf{wco}\,.(Z \vee q) \wedge \mathbf{wp}\,.\alpha.q) \vee q\rangle] \qquad \textbf{\textit{(wltrAct)}}$$
$$[\mathbf{wltr}\,.(UV).q \quad \equiv \quad \mathbf{wltr}\,.U.(\mathbf{wltr}\,.V.q)] \qquad \textbf{\textit{(wltrSeq)}}$$
$$[\mathbf{wltr}\,.(U+V).q \quad \equiv \quad \mathbf{wltr}\,.U.q \;\vee\; \mathbf{wltr}\,.V.q] \qquad \textbf{\textit{(wltrAlt)}}$$
$$[\mathbf{wltr}\,.U^*.q \quad \equiv \quad \langle \mu Z :: q \;\vee\; \mathbf{wltr}\,.U.Z\rangle] \qquad \textbf{\textit{(wltrStar)}}$$

Recall that **wlt** and $\mapsto$ are related by $[p \Rightarrow \mathbf{wlt}\,.q] \quad \equiv \quad p \;\mapsto\; q$, as described in Sect. 1.2. Now, we establish the relationship between **wltr** and **wlt**. This is a result of fundamental importance that permits us to claim that a property proven under any hint is a property of the program.

Theorem 2. *For a state predicate q:*

$$[\mathbf{wltr}\,.W.q \;\Rightarrow\; \mathbf{wlt}\,.q], \textit{ for any } W \qquad \textbf{\textit{(wltrSound)}}$$
$$\langle \exists W :: [\mathbf{wlt}\,.q \;\equiv \mathbf{wltr}\,.W.q]\rangle \qquad \textbf{\textit{(wltrCompl)}}$$

The first part of the theorem can be referred to as a *soundness* result since it asserts that the states from which a q state is reached because of W are states from which a q state is reached eventually in the sense of ordinary progress. Conversely, the second part can be seen as a *completeness* result since it shows that any state from which a q state is reached eventually is a state from which a q state is reached because of some regular expression W; see [Kal96] for proofs.

Next, we establish the connection between the generalized leads-to relation and the **wltr** predicate transformers. The result is analogous to the one relating the ordinary leads-to relation to the **wlt** predicate transformer described in Sect. 1.2. The proof of the following theorem appears in [Kal96].

Theorem 3. *For state predicates p and q:*

$$[p \Rightarrow \mathbf{wltr}\,.W.q] \quad \equiv \quad p \overset{W}{\longmapsto} q$$

Finally, we state a number of rules that help us in establishing generalized leads-to properties of any program. These rules resemble the corresponding rules in UNITY ([CM88]).

Theorem 4. *For set S and mappings f and g from S to state predicates:*

$$[p \Rightarrow q] \quad \Rightarrow \quad (p \overset{W}{\longmapsto} q) \qquad \textbf{\textit{(Imply)}}$$
$$[p' \Rightarrow p] \;\wedge\; (p \overset{W}{\longmapsto} q) \quad \Rightarrow \quad (p' \overset{W}{\longmapsto} q) \qquad \textbf{\textit{(LhsStr)}}$$
$$(p \overset{W}{\longmapsto} q) \;\wedge\; [q \Rightarrow q'] \quad \Rightarrow \quad (p \overset{W}{\longmapsto} q') \qquad \textbf{\textit{(RhsWeak)}}$$
$$\langle \forall m : m \in S : f.m \overset{W}{\longmapsto} g.m\rangle \quad \Rightarrow \qquad \textbf{\textit{(GenDisj)}}$$
$$\quad (\langle \exists m : m \in S : f.m\rangle \overset{W}{\longmapsto} \langle \exists m : m \in S : g.m\rangle)$$
$$(p \overset{W}{\longmapsto} \text{false}) \quad \Rightarrow \quad [\neg p] \qquad \textbf{\textit{(Impossible)}}$$
$$(p \overset{V}{\longmapsto} q \vee b) \;\wedge\; (b \overset{W}{\longmapsto} r) \quad \Rightarrow \quad (p \overset{VW}{\longmapsto} q \vee r) \qquad \textbf{\textit{(Cancel)}}$$

Theorem 5. *For any W,*

$$(p \overset{W}{\longmapsto} q) \quad \Rightarrow \quad (p \mapsto q) \qquad \textbf{(Sound)}$$
$$(p \mapsto q) \quad \Rightarrow \quad \langle \exists W :: p \overset{W}{\longmapsto} q \rangle \qquad \textbf{(Compl)}$$

3.3 Progress Algebras

In this section we define an algebraic structure, *progress algebra*, by presenting a list of equalities and equational implications which define a congruence relation on regular expressions. For a program F we refer to the resulting algebraic structure as $\mathcal{R}_F$ from now on and call it the *progress algebra* for program F. First we define the equational Horn theory for $\mathcal{R}_F$, then show that $\mathcal{R}_F$ bears many similarities to the well known Kleene algebras and to the algebra of regular events. In the following, we use the familiar formalism and terminology of Kleene algebras.

We start with the definition of progress algebra in which the binary relation $\leq$ (pronounced *subsumed by*) is defined by $U \leq V \ \equiv \ U + V = V$.

Definition 4. *A* progress algebra $\mathcal{K}$ *is the free algebra with binary operations $\cdot$ and $+$, unary operation *, and constant ε satisfying the following equations and equational implications for all U, V, and W in $\mathcal{K}$:*

$$U + (V + W) \ = \ (U + V) + W \qquad \textbf{(PrAlg0)}$$
$$U + V \ = \ V + U \qquad \textbf{(PrAlg1)}$$
$$W + W \ = \ W \qquad \textbf{(PrAlg2)}$$
$$U(VW) \ = \ (UV)W \qquad \textbf{(PrAlg3)}$$
$$\varepsilon W \ = \ W \qquad \textbf{(PrAlg4)}$$
$$W\varepsilon \ = \ W \qquad \textbf{(PrAlg5)}$$
$$UV + UW \ \leq \ U(V + W) \qquad \textbf{(PrAlg6)}$$
$$(U + V)W \ = \ UW + VW \qquad \textbf{(PrAlg7)}$$
$$\varepsilon \ \leq \ W \qquad \textbf{(PrAlg8)}$$
$$\varepsilon + WW^* \ \leq \ W^* \qquad \textbf{(PrAlg9)}$$
$$\varepsilon + W^*W \ \leq \ W^* \qquad \textbf{(PrAlg10)}$$
$$UW \ \leq \ W \quad \Rightarrow \quad U^*W \ \leq \ W \qquad \textbf{(PrAlg11)}$$
$$WU \ \leq \ W \quad \Rightarrow \quad WU^* \ \leq \ W \qquad \textbf{(PrAlg12)}$$

A progress algebra satisfying ***(PrAlg11)*** *but not necessarily* ***(PrAlg12)*** *is called a* right-handed *progress algebra, and a progress algebra satisfying* ***(PrAlg12)*** *but not necessarily* ***(PrAlg11)*** *is called a* left-handed *progress algebra.*

There are three major differences between Kleene algebras and progress algebras:

1. Progress algebras lack the equivalent of the $\emptyset$ constant of Kleene algebras. We could introduce such a constant by defining $[\mathbf{wltr}.\emptyset.q \equiv \text{false}]$, which would actually satisfy the Kleene axioms referring to $\emptyset$. Since such a regular expression does not have a counterpart in either the operational model or the deductive system, we omit it from further consideration.

2. A progress algebra does not have to satisfy the left distributivity of $\cdot$ over $+$. Only the weaker inequality **(PrAlg6)** is required instead.
3. A progress algebra satisfies **(PrAlg8)** which is not present in Kleene algebras.

3.4 $\mathcal{R}_F$ as Progress Algebra

Next, we show that the **wltr** predicate transformers can be regarded as a progress algebra. To do so, we have to define the equational theory of **wltr**, relate the operators $\cdot$, $+$, *, and the constant ε of $\mathcal{R}_F$ to operations on predicate transformers, and show that the equations and equational implications defining progress algebras are met by **wltr**.

The equational theory and the algebraic structure of **wltr** are defined as expected: any W in $\mathcal{R}_F$ denotes the predicate transformer $\mathbf{wltr}.W$ over the state predicates of F; the meaning of the constant ε is given by **(wltrEps)** as the identity transformer; the meaning of the operators $\cdot$, $+$, and * is given by **(wltrSeq)**, **(wltrAlt)**, and **(wltrStar)** as functional composition of predicate transformers, disjunction of predicate transformers, and a least fixpoint construction respectively; the meaning of the basic elements α in $F.A$ is given by **(wltrAct)** as the $\mathbf{wltr}.\alpha$ predicate transformer. Finally, equality of regular expressions over $F.A$ (written as $=_F$) is defined as equivalence of the corresponding predicate transformers, i.e., for all U, V in $\mathcal{R}_F$:

$$U =_F V \quad \equiv \quad [\mathbf{wltr}.U \equiv \mathbf{wltr}.V] \ .$$

The induced subsumption relation $\leq_F$ on $\mathcal{R}_F$ is then given by

$$U \leq_F V \quad \equiv \quad U + V =_F V \ .$$

It follows that for all U and V in $\mathcal{R}_F$:

$$\begin{array}{ll} & U \leq_F V \\ \equiv & \quad \{\text{definition of } \leq_F\} \\ & U + V =_F V \\ \equiv & \quad \{\textbf{(wltrAlt)}, \text{ definition of } =_F\} \\ & [\mathbf{wltr}.U \vee \mathbf{wltr}.V \ \equiv \ \mathbf{wltr}.V] \\ \equiv & \quad \{\text{predicate calculus}\} \\ & [\mathbf{wltr}.U \ \Rightarrow \ \mathbf{wltr}.V] \end{array}$$

In other words, the subsumption relation $\leq_F$ on $\mathcal{R}_F$ is exactly the implication of the corresponding predicate transformers. Therefore, the algebraic structure of $\mathcal{R}_F$ is given by the following theorem; for a proof see [Kal96].

Theorem 6. *For program F, the algebra $\mathcal{R}_F$ is a right-handed progress algebra.*

3.5 On the Operational Meaning of Hints

It is natural to expect the following operational meaning of progress under hints. An interpretation of p leads-to q by hint W (which is written as $p \overset{W}{\longmapsto} q$) is: for any state s in which p holds, there exists a finite sequence of actions, w, such that (1) $w \in W$, and (2) any finite segment of an execution that starts in s and includes w as a subsequence achieves q at some point.

```
program PairReduce
    declare
        var x, y, d : integer
    assign
        [both]  x > 0 → x, y := x − 1, d
        [one]   y > 0 → y := y − 1
        [up-d]          d := d + 1
end
```

Fig. 2. Program *PairReduce*

But this interpretation is too restrictive. In general, there are states for which no predetermined action sequence can achieve the goal predicate. To see this consider the program *PairReduce* of Fig. 2.
For this program we see that $x \geq 0 \wedge y \geq 0 \mapsto (x, y = 0, 0)$ because each of [both] and [one] decreases the pair (x,y) lexicographically, and [up-d] does not affect (x,y). However, for any specific state, say $(x,y,d) = (3,5,2)$, no action sequence can be specified that corresponds to the operational interpretation given above. A more elaborate interpretation based on games is given in [Kal96].

4 Empirical Results

The algorithm for checking generalized progress properties based on Theorem 3 has been implemented as part of the *UNITY Verifier System*(UV system for short), our symbolic model checker for finite state UNITY programs and propositional UNITY properties ([Kal96,Kal94,Kal95]). The UV system can be used to verify automatically whether UNITY programs satisfy the given safety and progress properties, to obtain debugging information in the form of counterexample states that violate asserted conditions in case a verification attempt fails, and to exploit and manage design information in the form of user supplied invariants and progress hints.

The current version of the UV system employs ordered binary decision diagrams ([Bry86,BBR90]) to represent UNITY programs and properties symbolically. The input language used for writing programs and properties is a strongly typed version of the original UNITY notation that is restricted to finite data

types. Since the UV system is implemented using the Tcl/Tk package ([Ous94]) it has a customizable graphical user interface as well as a scripting interface. The current version is available for SunOS, Solaris and Linux operating systems and consists of about 35000 lines of C++ code. It has been used to model check a variety of programs, in particular the ones presented in this section.

In this section we present empirical results of applying the UV system to some practical problems. Each example has been chosen with the intention to emphasize a particular aspect of the UV system. All examples were run on a SPARC-20 workstation with about 20 MB of main memory allocated to the model checker.

4.1 A Counter

In Sect. 2 program *UpDown* was used as an illustrative example to introduce the concept of generalized progress properties and was discussed there in detail. A finite state version of the program restricts the counter variable n to the range from 0 to $N-1$ for a positive integer N.

In the following table we summarize performance measurements for different values of N for two progress properties: the ordinary leads-to property $\text{true} \mapsto n = 0$ (indicated by $\mapsto$ in the table), and the generalized leads-to property $\text{true} \overset{[set][down]*}{\longmapsto} n = 0$ (indicated by r- $\mapsto$). Three measurements are listed: *iterations* states the number of inner fixpoint iterations needed to complete the check, *ops* states the number of OBDD node lookup requests in thousands, and *time* shows the execution time in seconds. All model checker invocations establish the respective properties as correct.

N		10	20	50	100	200	500	1000	10000
itera-	$\mapsto$	107	320	1551	5608	21222	128000	506006	n/a
tions	r- $\mapsto$	41	81	201	401	801	2001	4001	40001
ops	$\mapsto$	2.5	11	100	548	2810	22283	88933	n/a
in 10^3	r- $\mapsto$	0.8	2.1	8.7	18	39	119	243	3844
time	$\mapsto$	0.3	0.3	1.1	4.8	27.3	227.9	1028.4	n/a
in s	r- $\mapsto$	0.2	0.3	0.3	0.4	0.5	1.1	2.2	38.0

It may be verified from this table that the number of iterations for the ordinary leads-to check is quadratic in N, whereas it is only linear for the generalized leads-to check.

4.2 Scheduling: Milner's Cycler

The scheduling problem known as *Milner's Cycler* ([Mil89]) has been used as a benchmark in the literature to compare different verification methods and systems. Here, we merely present the empirical results; consult[Kal96] for details.

The following table compares the ordinary progress check (indicated by $\mapsto$) for the key progress property with the generalized one (indicated by r- $\mapsto$) using

a regular expression hint. The measurements listed are the same as in Sect. 4.1, while N denotes the number of processes:

N		4	8	12	16	20
itera-	$\mapsto$	86	174	268	370	480
tions	r- $\mapsto$	12	12	12	12	12
ops	$\mapsto$	22	287	2030	8917	29334
in 10^3	r- $\mapsto$	7	24	52	87	145
time	$\mapsto$	0.4	2.3	16.5	87.7	369.8
in s	r- $\mapsto$	0.3	0.5	0.8	1.3	1.8

4.3 Elevator Control

Our final example, an elevator control for a building with N floors, has been motivated by similar programs in the literature (e.g. , [CWB94]). Again, we merely present the empirical results; consult[Kal96] for details:

N		20	50	100
states		$3.77 \cdot 10^8$	$1.01 \cdot 10^{18}$	$2.28 \cdot 10^{33}$
itera-	$\mapsto$	2264	13248	51576
tions	r- $\mapsto$	658	1798	3698
ops	$\mapsto$	2.25	18.3	659
in 10^6	r- $\mapsto$	1.17	7.6	95
time	$\mapsto$	10.8	280	>10000
in s	r- $\mapsto$	5.3	83	1150

5 Concluding Remarks

The motivation for developing the theory of generalized progress has been threefold: such a theory should (i) provide a novel way of establishing ordinary progress properties of programs by allowing the user to characterize explicitly how progress is achieved, (ii) make it possible to take advantage of design knowledge in order to verify programs more effectively, and (iii) increase the efficiency of mechanical verification procedures based on the developed theory. We have addressed these goals by treating hints as formal objects (namely as elements of a progress algebra) and by providing a calculus for reasoning about such hints, for relating them to program executions, and for combining them with state-based reasoning methods (such as proving safety properties).

References

[AS85] B. Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[BBR90] K. S. Brace, R. E. Bryant, and R. L. Rudell. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Design Automation Conference*, 1990.

[BCM91] J. R. Burch, E. M. Clarke, and K. M. McMillan. Representing circuits more efficiently in symbolic model checking. In *Proccedings of the 28th Design Automation Conference 1991*, pages 403–407, 1991.

[BRS00] R. Bloem, K. Ravi, and F. Somenzi. Symbolic guided search for CTL model checking. In *Proccedings of the 37th Design Automation Conference 2000*, pages 29–34, 2000.

[Bry86] Randy E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computing*, (6), 1986.

[CE81] Edmund M. Clarke and Ernest Allen Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop*, volume 131 of *Lecture Notes in Computer Sience*, May 1981.

[CES86] Edmund. M. Clarke, Ernest Allen Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, (2), 1986.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design, A Foundation*. Addison Wesley, 1988.

[CWB94] Jorge Cuellar, Isolde Wildgruber, and Dieter Barnard. The temporal logic of transitions. In *Formal Methods Europe (Barcelona, Spain)*, 1994.

[DS90] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Text and Monographs in Computer Science. Springer Verlag, 1990.

[JKR89] C. S. Jutla, Edgar Knapp, and J. R. Rao. A predicate transformer approach to semantics of parallel programs. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, 1989.

[Kal94] Markus Kaltenbach. Model checking for UNITY. Technical Report TR94-31, The University of Texas at Austin, December 1994.

[Kal95] Markus Kaltenbach. *The UV System, User's Manual*, February 1995. Revision 1.18.

[Kal96] Markus Kaltenbach. *Interactive Verification Exploiting Program Design Knowledge: A Model-Checker for UNITY*. PhD thesis, The University of Texas at Austin, December 1996.

[Kna90] Edgar Knapp. A predicate transformer for progress. *Information Processing Letters*, 33:323–330, 1990.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions of Software Engineering*, 3(2):125–143, 1977.

[McM93] Ken L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[Mil89] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[Mis95a] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[Mis95b] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–272, 1995.

[Mis01] Jayadev Misra. *A Discipline of Multiprogramming*. Monographs in Computer Science. Springer-Verlag New York Inc., New York, 2001. The first chapter is available at `http://www.cs.utexas.edu/users/psp/discipline.ps.gz`.

[Ous94] John K. Ousterhout. *Tcl and the Tk toolkit*. Professional Computing Series. Addison-Wesley, 1994.