Generation of Reduced Models for Checking Fragments of CTL

Dennis Dams[1], Orna Grumberg[2], Rob Gerth[1]*

[1] Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, E-mail: {wsindd,robg}@win.tue.nl
[2] AT&T Bell Labs, 600 Mountain Ave., Murray Hill, NJ 07974, USA, E-mail: orna@research.att.com

Abstract. In the first part of the paper, we present an algorithm for the construction of a quotient model under the equivalence induced by ACTL, the universal fragment of CTL. As this equivalence coincides with simulation equivalence, the achieved reduction is better than reductions achieved by methods based on bisimulation. Simulation equivalence is also the equivalence that is induced by ACTL*, hence, the reduced model allows model checking of any ACTL* formula. Furthermore it provides diagnostic information. In the second part, we restict our attention to single ACTL formulae. An algorithm is given that constructs a model which is reduced w.r.t. a specific formula.
The reduced models are constructed directly, without intermediate need for the full models. The approach, splitting of states w.r.t. formulae, yields a sequence of successively more refined models which at any stage preserve truth of formulae.

1 Introduction

In the *model checking* approach to program verification [6, 22, 28], a model of the program is constructed over which formulae are checked for satisfaction. The model reflects the possible behaviours of the program, the formulae express certain required properties of such behaviours. The *state explosion problem* refers to the inherently large size of the model relative to the size of the program, limiting the feasability of model checking. From a variety of solutions proposed in order to overcome this limitation [16, 19, 15, 13, 27, 20, 30, 23, 18, 4, 7, 12, 17], we concentrate here on *abstraction methods*. The aim of such methods is to abstract the model to a smaller one, in such a way that if some property holds for the abstracted model $\mathcal{A}$, it also holds for the concrete model $\mathcal{C}$. The implication $\mathcal{A} \models \varphi \Rightarrow \mathcal{C} \models \varphi$ for all φ in some property set Φ is referred to as the *(weak) preservation* of Φ. The investigation of abstraction methods has resulted in a variety of preservation results, relating types of abstraction to sets of properties preserved. In [1], the considered abstraction type is the *simulation* preorder [24],

* Currently working in ESPRIT project P6021: "Building Correct Reactive Systems (REACT)".

which is shown to preserve fragments of the μ-calculus. A direction which is closely related makes use of *homomorphisms* between processes [21, 5]. Other preservation results have been presented in, among others, [16, 9]. However, the drawback of weak preservation is that when a property is *not* satisfied in $\mathcal{A}$, it may still hold in $\mathcal{C}$.

In order to be able to effectively verify all the properties in Φ, we require Φ to be *strongly preserved*: $\mathcal{A} \models \varphi \Leftrightarrow \mathcal{C} \models \varphi$ for all $\varphi \in \Phi$. This requirement limits the degree of reduction that may be attained since only those states that cannot be distinguished by any formula in Φ may be collapsed in $\mathcal{A}$. In other words, the maximal reduction possible is the quotient of $\mathcal{C}$ under the equivalence induced by Φ. A commonly applied abstraction is based on bisimulation [26], the equivalence induced by CTL [6, 3] and also by CTL* [10]. Because bisimulation is a fine-grained equivalence, the resulting degree of reduction is restricted. When we consider smaller property sets, e.g., subsets of CTL, the induced equivalences are in general coarser and hence better reductions may be expected.

The success of applying abstractions depends on the possibility to *directly* construct abstract models, i.e., without intermediate construction of the complete concrete models. In [2], an algorithm is developed that directly constructs, for the case of bisimulation, the minimal abstract graph of accessible states. This construction is based on the *partition refinement* algorithm described in [25], which requires that the *preimage sets* of sets of states are effectively computable. [5] and [14] describe how abstract models may be constructed by *abstract interpretations* of the program text, where the elementary operations of the programming language are re-interpreted over a domain of *abstract values*. However, this approach requires that an abstraction relation or function, linking concrete and abstract values, be provided.

In this paper, we present a method for the direct construction of abstract models that strongly preserve a given set of properties. In particular, we give algorithms for ACTL—the same algorithm may be used for ACTL*—and for single properties specified in ACTL. The logics ACTL and ACTL* are fragments of CTL and CTL* resp. where the existential path quantifier is forbidden (see Section 2 for a formal definition). The method is based on successive refinement of a model by splitting states w.r.t. formulae. This approach is different from the partition refinement algorithm which yields a model which is minimal w.r.t. bisimulation, as will be illustrated by an introductory example.

1.1 Example

As models we consider transition systems in which states are labelled with propositions or their negations (denoted by overlining). Bisimulation equivalence of states can be defined in a concise way as follows. Let $\approx$ be an equivalence relation on states, and consider its equivalence classes. Define the *successor classes* of a state to be the classes of its successors. Then $\approx$ is a bisimulation if every class S is *stable*: all states in S have the same labels and the same successor classes. This simple notion of equivalence forms the basis of the elegant algorithm in [25] that computes the coarsest partitioning of a transition system in which all

classes are stable. The algorithm starts from an initial partitioning such that in every class, all states have the same labels. Then, every class that is not stable is split into stable subclasses. This splitting is repeated until a fixpoint is reached, which is proven to be the coarsest partitioning for which the classes are bisimulation equivalence classes. The algorithm of [2] integrates this with a computation of accessible classes. As an example, consider the transition diagram in Fig. 1. The s_i and t_i are names of states; p is a proposition. The splitting algorithm described above will start from the initial partitioning with classes $S_1 = \{s_1, s_2, s_3, t_1, t_2\}$ and $S_2 = \{t_3\}$ — the states where p holds and the states where $\overline{p}$ holds respectively. In the second step, $\{t_2\}$ is split off S_1 because it has both S_1 and S_2 as successor classes, whereas the other elements of S_1 only have S_1 as successor class. The resulting new classes are $S_{11} = \{s_1, s_2, s_3, t_1\}$ and $S_{12} = \{t_2\}$. Next, S_{11} is split into $S_{111} = \{s_2, s_3\}$, $S_{112} = \{s_1\}$ and $S_{113} = \{t_1\}$. In the resulting partitioning, indicated by the light grey areas, all classes are stable, hence, the coarsest bisimulation classes have been identified. All states within one such class are CTL equivalent. In fact, a class which is not stable contains states which are not CTL equivalent. The refinement algorithm may be seen as a process which splits classes w.r.t. CTL formulae: as long as there is a formula which can distinguish among the states within a class, the class should be split accordingly. Because the equivalence induced by CTL has bisimulation as its structural equivalence, the splitting algorithm can be based on the simple inductive definition of bisimulation given above, so that the way a class is being split can be determined solely by computing the preimage sets of its successor classes. In the algorithm described in [2], the classes of concrete states are represented implicitly as predicates; in order to split classes it then suffices to be able to compute the predicate transformers which correspond to the preimage sets of classes.

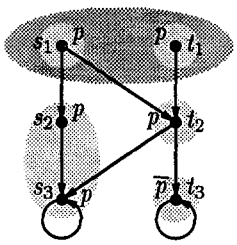

Fig. 1. Concrete transition system.

However, the equivalence induced by an arbitrary property set does not have in general a usable structural characterization. If, for such an equivalence, we want to obtain the equivalence classes by a similar method of partition refinement, a different criterion for splitting states than just preimage sets of successors will be needed. The method we suggest is based on the set of formulae to be preserved, therefore it is easily extended to also work for an arbitrary ACTL formula, as is shown in the second part of this paper. To give an idea of the proposed approach, we illustrate it on Fig. 1. The criterion for splitting a state is now an ACTL formula. Thus, state t_3 is distinguished from the other states by formula p, resulting in a split into the same classes S_1 and S_2 as above. $\mathsf{AX}p$ distinguishes $\{t_2\}$ from $\{s_1, s_2, s_3, t_1\}$, effecting a split of S_1 into S_{11} and S_{12}, again as above. Another ACTL formula is $\mathsf{AXAX}p$; splitting for it causes S_{11}

to split into $S'_{111} = \{s_1, t_1\}$ and $S'_{112} = \{s_2, s_3\}$, this time different from above. Now, there is no ACTL formula φ such that splitting for φ will cause any more changes. In particular, class S'_{111} (dark grey) remains stable; indeed s_1 and t_1, although not bisimilar, cannot be distinguished by any ACTL formula. The algorithm terminates when no ACTL formula can distinguish between the states of any class. In Section 3, we will identify a subset of formulae for which it suffices to split and give a termination condition which is practically computable.

1.2 Overview

After providing some preliminary background in Section 2, we present in Section 3 an algorithm for the direct construction of an abstract model by splitting for ACTL. As the equivalence induced by ACTL coincides with simulation equivalence, the reduction will be better than that obtained by bisimulation reduction, while ACTL* is preserved. The construction process is integrated with a reachability analysis. We state correctness of the algorithm and minimality w.r.t. ACTL equivalence of the resulting model. In Section 4, we give an algorithm that constructs an abstract model which strongly preserves a given ACTL formula. In this case, an even more drastic reduction may be expected. Section 5 places the proposed approach in a broader perspective, draws some conclusions, and points to future work. Proofs have been omitted in this abridged version.

2 Preliminaries

Temporal Logic. We assume that the reader has some familiarity with temporal logic. The logic ACTL is the fragment of CTL [6] that only allows the universal path quantifier A. We assume given a set AP of *atomic predicates* (or *atoms*). ACTL is the set of formulae given by the following inductive definition.

1. for $p \in AP$, p and $\overline{p}$ are formulae;
2. if φ and ψ are formulae, then so are $\varphi \wedge \psi$, $\varphi \vee \psi$, $\mathsf{AX}\varphi$, $\mathsf{AU}(\varphi, \psi)$ and $\mathsf{AV}(\varphi, \psi)$.

The fragment ACTL* of CTL* [10] is defined similarly, see for example [16]. Note that negation, denoted by overlining, is only allowed on atoms. $\mathsf{AU}(\varphi, \psi)$ is the universally quantified *until*, for which our notation deviates slightly from the more common $\mathsf{A}[\varphi \mathsf{U} \psi]$. $\mathsf{AV}(\varphi, \psi)$ is its dual, i.e., $\mathsf{AV}(\varphi, \psi) \equiv \overline{\mathsf{AU}(\overline{\varphi}, \overline{\psi})}$; AW is used to denote one of AU and AV. We assume the usual abbreviations, e.g., *true*, *false*, $\mathsf{AF}\varphi$, $\mathsf{AG}\varphi$. For ease of notation, we assume that AP is closed under negation, i.e., for every $p \in AP$ there is $q \in AP$ such that $q \equiv \overline{p}$. The *approximants* $\mathsf{AW}_i(\varphi, \psi)$ are defined as follows:

$$\mathsf{AU}_0(\varphi, \psi) \equiv \mathit{false} \qquad \mathsf{AU}_{i+1}(\varphi, \psi) \equiv \psi \vee (\varphi \wedge \mathsf{AXAU}_i(\varphi, \psi))$$
$$\mathsf{AV}_0(\varphi, \psi) \equiv \mathit{true} \qquad \mathsf{AV}_{i+1}(\varphi, \psi) \equiv \psi \wedge (\varphi \vee \mathsf{AXAV}_i(\varphi, \psi))$$

The *level* of $\varphi \in$ ACTL is the maximal number of nested AX operators in φ if there are no AU and AV operators in φ, and ω otherwise. L_i is the set of all ACTL formulae of level $\leq i$.

Transition Systems. Formulae are interpreted over transition systems $\mathcal{T} = (\Sigma, R)$ where Σ is a set of *states* over which the predicates in AP are evaluated, and R a *total* transition relation. Satisfaction ($\models$) of formulae in states is defined as usual, see for example [6]. We define $\mathcal{T} \models \varphi \Leftrightarrow \forall_{s \in \Sigma}\, s \models \varphi$. A *path* is any infinite sequence of states related by R. For $s \in \Sigma$, $pre_R(s)$ and $succ_R(s)$ denote the sets of predecessors and successors of s in $\mathcal{T}$ respectively. These functions are extended pointwise to sets of states. We fix a transition system $\mathcal{C} = (\Sigma_\mathcal{C}, R_\mathcal{C})$, called the *concrete* model, and assume $\Sigma_\mathcal{C}$ is finite. *pre* abbreviates $pre_{R_\mathcal{C}}$.

Pred is a set of predicates on $\Sigma_\mathcal{C}$. When writing $pre(p)$ for $p \in Pred$, we interpret the function *pre* as a predicate transformer by identifying predicates with their characteristic sets. Define $\widetilde{pre}(p) \equiv \overline{pre(\overline{p})}$; it characterizes the set of all states for which every outgoing transition leads to a p-state (a state where p holds). We assume that *Pred* includes AP and that it is closed under $\wedge$, $\vee$ and $\widetilde{pre}$. The function $\|\cdot\|$: ACTL $\rightarrow$ *Pred*, which maps every formulae to its *characteristic predicate*, is defined by:

$$\|p\| \equiv p, \quad \|\varphi \wedge \psi\| \equiv \|\varphi\| \wedge \|\psi\|, \quad \|\varphi \vee \psi\| \equiv \|\varphi\| \vee \|\psi\|, \quad \|\mathsf{AX}\varphi\| \equiv \widetilde{pre}(\|\varphi\|)$$
$$\|\mathsf{AU}(\varphi, \psi)\| \equiv \bigvee_i \|\mathsf{AU}_i(\varphi, \psi)\|, \quad \|\mathsf{AV}(\varphi, \psi)\| \equiv \bigwedge_i \|\mathsf{AV}_i(\varphi, \psi)\|$$

Because $\mathcal{C}$ is finite, the definitions of $\|\mathsf{AU}(\varphi, \psi)\|$ and $\|\mathsf{AV}(\varphi, \psi)\|$ agree with their standard interpretations.

Abstractions. An *abstraction* of $\mathcal{C}$ is a transition system $\mathcal{A} = (\Sigma_\mathcal{A}, R_\mathcal{A})$ such that $\Sigma_\mathcal{A} \subseteq Pred$ is finite and $R_\mathcal{A}$ is defined by $R_\mathcal{A}(a, b) \Leftrightarrow \exists_{c \in a}\, \exists_{d \in b}\, R_\mathcal{C}(c, d)$. Satisfaction of atoms is defined $a \models p \Leftrightarrow a \Rightarrow p$; satisfaction of other formulae is then defined as usual. Note that $a \not\models p$ does not imply $a \models \overline{p}$, so the satisfaction of formulae in $\mathcal{A}$ is based on a non-classical logic. In [8], we prove the following preservation result, which was presented before in [16, 29] and follows from the more general results in [1].

Lemma 1. *If $\mathcal{A}$ is an abstraction of $\mathcal{C}$, then $\mathcal{A} \models \varphi \Rightarrow \mathcal{C} \models \varphi$.*

Intuitively, the reason for this is that $\mathcal{A}$ *preserves paths*: if $c_1 c_2 \cdots$ is a path in $\mathcal{C}$, then any sequence $a_1 a_2 \cdots$ in $\Sigma_\mathcal{A}$ such that $c_i \in a_i$ is a path in $\mathcal{A}$; hence, if some property holds along all paths in $\mathcal{A}$, then it holds along all paths in $\mathcal{C}$.

Our goal is to obtain *strongly* preserving abstractions. This is a question that was left open by the works mentioned above. In the sequel of this paper, the symbol $\mathcal{A}$ denotes an abstraction $(\Sigma_\mathcal{A}, R_\mathcal{A})$ of $\mathcal{C}$.

3 Splitting for ACTL

Formula φ is *determined* in abstract state a if either $a \Rightarrow \|\varphi\|$ or $a \Rightarrow \|\overline{\varphi}\|$; this notion is extended pointwise for sets of formulae and for sets of states. Determinedness implies strong preservation:

Lemma 2. *If* ACTL *is determined in $\Sigma_\mathcal{A}$, then $\forall_{\varphi \in \mathrm{ACTL}}\, \mathcal{A} \models \varphi \Leftrightarrow \mathcal{C} \models \varphi$.*

Determinedness of φ in a is achieved by *splitting* it into $a \wedge \|\varphi\|$ and $a \wedge \|\overline{\varphi}\|$.

Definition 3. Let A and $B = \{b_1, \ldots, b_k\}$ be subsets of *Pred*. The *splitting of* A *for* B, $split(A, B) \subseteq Pred$, is defined by

$$split(A, B) = \{a \wedge b'_1 \wedge \cdots \wedge b'_k \mid a \in A,\ \forall_i\ (b'_i \equiv b_i \text{ or } b'_i \equiv \overline{b_i})\}$$

If A or B are singletons, set brackets will be omitted. For $\Phi \subseteq$ ACTL, we will abusively write $split(A, \Phi)$ for $split(A, \{\|\varphi\| \mid \varphi \in \Phi\})$.

Lemma 4. *For* $A \subseteq Pred$, $\varphi, \psi \in$ ACTL, $\Phi \subseteq$ ACTL*:*

1. $split(A, \Phi) = A \Leftrightarrow \Phi$ *is determined in* A*;*
2. $split(A, \{\varphi, \psi\}) = A \Rightarrow split(A, \{\varphi \wedge \psi\}) = A \wedge split(A, \{\varphi \vee \psi\}) = A$

The smallest abstraction in which ACTL is determined can be obtained by starting from a model consisting of one abstract state *true* (describing all concrete states), and splitting it for every $\varphi \in$ ACTL. In the following, this algorithm is integrated with a reachability analysis, which leads to a significant optimization. Computation of the characteristic predicates is done inductively in line with their definition on the previous page. Furthermore, a practical termination condition is provided.

3.1 Algorithm

The algorithm, shown in Figure 2, inductively computes the characteristic predicates of ACTL formulae and at the same time splits for them. Thus, in the ith iteration of the repeat-loop, the algorithm splits for formulae of level i. The observations of Lemma 4(2) limit the number of formulae that has to be split for: a model that has been split for both φ and ψ, has been split for both $\varphi \wedge \psi$ and $\varphi \vee \psi$ as well. Furthermore, a formula of the form $\mathsf{AX}(\varphi \wedge \psi)$ is equivalent to $\mathsf{AX}\varphi \wedge \mathsf{AX}\psi$, so, by the same observation, splitting for $\mathsf{AX}(\varphi \wedge \psi)$ is not needed once we have split for $\mathsf{AX}\varphi$ and $\mathsf{AX}\psi$. Because AW formulae have level ω, it is not immediately clear what is an appropriate termination condition. Below, we will prove that the algorithm may terminate if splitting for some level leaves the model unchanged (line 16). The variable P keeps track of the set of all predicates that has been split for. Σ contains the set of abstract states, which are being split at every iteration of the repeat-loop; in R, the transitions between abstract states are kept. They are assumed to be computed from Σ by a function *rel* which is left unspecified (see Section 5.1). In every iteration, the "new" predicates that have to be split for are computed in Q, and added to P. The assertions on lines 2 and 16 as well as the auxiliary variable i are used to argue correctness. Lines 4–14 update reachability information while splitting states, as explained below.

Technically, the difference between this algorithm and the refinement algorithms for bisimulation of [25, 2] is that here in every iteration we split w.r.t. P, the set of previously computed formulae, whereas the algorithms for bisimulation split w.r.t. Σ, the set of previously computed states.

```
(1)  i:=0 ; Σ:=split({true},AP) ; P:=AP ; R:=rel(Σ) ; changed:=false;
(2)  repeat  {(Σ,R) = A_i}
(3)      i:=i+1 ; Q:=pre~({⋁ S | S ⊆ P}) ; P:=P ∪ Q;
(4)      Reach:=∅ ; Σ_n:=Σ ; Σ_s:=∅ ; Front:={a ∈ Σ_n | a ⊨ ι};
(5)      while Front ≠ ∅ do
(6)          choose a ∈ Front ; Front:=Front \ {a} ; B:=split(a,Q) ;
(7)          if B ≠ {a} then changed:=true fi;
(8)          Σ_n:=Σ_n \ {a} ; Σ_s:=Σ_s ∪ B ; R:=rel(Σ_n ∪ Σ_s);
(9)          for each b ∈ B do
(10)             if b ⊨ ι or b ∈ succ_R(Reach)
(11)             then Reach:=Reach ∪ {b} ; Front:=(Front ∪ succ_R(b)) \ Reach
(12)             fi
(13)         od
(14)     od;
(15)     Σ:=Reach ; R:=rel(Σ)
(16) until not changed  {A_{i-1} = A_i}
```

Fig. 2. Splitting algorithm for ACTL.

Reachability. In practice, there will be a notion of *initial state(s)* in a system, and properties to be verified are only required to be satisfied by the initial states. In our formalism, where satisfaction of a property is defined as satisfaction in *each* state, this can be expressed by assuming that properties are always of the form $\iota \Rightarrow \varphi$, where $\iota \in AP$ is an atom that holds in initial states only. This assumption gives rise to a significant optimization (see e.g. [2]).

We define the notion of reachability in the obvious way: a state s is *reachable* if there is a path from an ι-state to s. For abstract models, the definition of reachability may only be applied if ι is determined. Note that the type of abstractions that are considered in this paper do not preserve reachability: a state a can be reachable in $\mathcal{A}$ without any state $c \in a$ being reachable in $\mathcal{C}$. However, *unreachability* is preserved, and furthermore it is stable under state-splitting:

Lemma 5. *Let $a \in \Sigma_{\mathcal{A}}$. If a is unreachable in $\mathcal{A}$, then:*

1. *for all $c \in a$, c is unreachable in $\mathcal{C}$, and*
2. *for any $Q \subseteq Pred$, if $\mathcal{A}' = (\Sigma'_{\mathcal{A}}, rel(\Sigma'_{\mathcal{A}}))$ where $\Sigma'_{\mathcal{A}} = split(\Sigma_{\mathcal{A}}, Q)$, then every $a' \in split(a, Q)$ is unreachable in $\mathcal{A}'$.*

The validity of a temporal formula of the form $\iota \Rightarrow \varphi$ can only depend on properties that hold in reachable states. Lemma 5(1) now implies that unreachable abstract states may be ignored in the verification of a property and hence they may be removed from the constructed abstract model. Lemma 5(2) says that this removal may take place at any point during the construction by the splitting algorithm. This provides us with a powerful method to optimize the expected running-time and memory demands of the splitting algorithm: the unreachable states do not have to be kept in memory, and no splitting has to be performed

on them. Although the worst-case complexity will remain the same, in practice we may expect a considerable gain.

In the algorithm of Fig. 2, reachability of states is updated while splitting them. In the variable *Reach*, the set of all reachable states of $split(\Sigma, Q)$ is computed. The set of all states in the model is kept in two variables: Σ_n keeps track of those states which have not been split yet, and Σ_s contains the resulting substates of states that have been split already. *Front* contains the "frontier" of states that will be considered for splitting and/or reachability: every state in *Front* is an initial state from Σ_n or a successor in $\Sigma_n \cup \Sigma_s$ of a state in *Reach*. Note that a state from *Front* may already be in Σ_s; in this case it was the result of some previous split, but by then not yet a successor of *Reach*. Several optimizations are possible, which we will not describe here.

3.2 Correctness and Minimality; Simulation Equivalence

The following lemmas underly the correctness of the algorithm. Define $\mathcal{A}_i$ to be the transition system $(\Sigma_i, rel(\Sigma_i))$, where $\Sigma_i = split(\{true\}, L_i)$, i.e., the result of splitting $\{true\}$ for all formulae up till (and including) level i. Note that because C is finite, the algorithm will terminate. Below, $\mathcal{A}$ denotes the result after termination of the algorithm.

Lemma 6.

1. *The assertions on lines 2 and 16 in Fig. 2 are valid.*
2. $\mathcal{A}_{i-1} = \mathcal{A}_i \Rightarrow \forall_{j \geq i}\, \mathcal{A}_j = \mathcal{A}_{j+1}$
3. $\mathcal{A}$ *is the minimal abstract model in which* ACTL *is determined.*

Theorem 7. $\mathcal{A}$ *is the quotient of* C *under* ACTL *equivalence.*

Two transition systems are simulation equivalent when they simulate each other [24]; this equivalence notion is strictly weaker than bisimulation, which in addition requires the simulation relation to be the same one in both directions. We have the following result.

Lemma 8. $\mathcal{A}$ *is the quotient of* C *under simulation equivalence.*

As simulation equivalence is also the notion of equivalence which is induced by ACTL*, we have

Corollary 9. $\mathcal{A}$ *is the quotient of* C *under* ACTL* *equivalence.*

4 Splitting for One ACTL Formula

In this section we consider a different scenario. We assume that it is known beforehand what is the property φ to be verified, and present an algorithm that constructs an abstract model which is reduced with respect to φ. Obviously, a better reduction may be expected in this case. The approach to construct the model will be the same as before: we split states according to ACTL formulae. The following lemma suggests which formulae to split for.

Lemma 10. *If the subformulae of φ and the approximants of subformulae of the form* $\mathsf{AW}(\zeta,\xi)$ *are all determined in $\Sigma_{\mathcal{A}}$, then $\mathcal{A} \models \varphi \Leftrightarrow \mathcal{C} \models \varphi$.*

A single formula will usually induce a coarser equivalence on states than the set of all ACTL formulae. Hence, the model which is constructed by the algorithm given below may be expected to be significantly smaller (although in general it will not be minimal). Secondly, it may be used to model check φ even before the splitting algorithm has terminated. For, at every stage during the splitting, the abstract model weakly preserves ACTL (even ACTL*). Thus, compared to a symbolic model checking algorithm in which no model is constructed, our algorithm will allow earlier termination in some cases. Furthermore, the constructed model instantly provides diagnostic information.

The algorithm is shown in Fig. 3. L is the length, i.e., the number of atoms and operators, of φ. The first split is for all atoms in φ (l=1); in every next iteration of the while-loop, more complex subformulae of φ are considered. Con- and disjunctions may be ignored for the same reason as before (Lemma 4(2)). If the subformula considered is of the form $\mathsf{AW}(\zeta,\xi)$, then $\mathcal{A}$ is split for all approximants of it. After having split for all subformulae (and their approximants) of length l, a model check is performed; the algorithm may then terminate when either this check returns *yes*, or when l=L, implying that $\mathcal{A}$ has been split for all subformulae of φ. In the latter case, the final result of the model check indicates, by Lemma 10, whether or not φ is satisfied in $\mathcal{C}$. For reasons of limited space, several possible optimizations are not described here.

```
(1)  Σ:={true} ; R:=rel(Σ) ; answer:=no ; l:=1;
(2)  while l ≤ L and answer=no do
(3)      for each subformula ψ of φ with length(ψ)=l do
(4)          if ψ ∈ AP → Σ:=split(Σ,ψ)
(5)          [] ψ is of the form AXζ → Σ:=split(Σ, pre~(||ζ||))
(6)          [] ψ is of the form AW(ζ,ξ) →
(7)              i:=0;
(8)              repeat
(9)                  Σ:=split(Σ, ||AW_i(ζ,ξ)||) ; i:=i+1;
(10)             until no change
(11)         [] otherwise → skip
(12)         fi
(13)     od;
(14)     A:=(Σ, rel(Σ)) ; model_check(φ, A, answer) ; l:=l+1
(15) od
```

Fig. 3. Splitting algorithm for $\varphi \in$ ACTL.

5 Discussion

We have presented two algorithms for the automatic and direct construction of reduced models strongly preserving ACTL* and any given ACTL formula respectively. The algorithms are both based on an approach in which abstract states are being refined in successive steps by state splitting. In contrast to algorithms for bisimulation based reduction, the criteria for splitting a state in our approach are the characteristic predicates of some subset of ACTL. Conditions were given under which full automation of the method may be realized. An interesting aspect is that when the construction process is aborted at any point, the obtained abstract model still preserves ACTL*.

The conditions required in order to automate the method are basically effective computability of $\widetilde{pre}$ and *rel*; this calls for efficient implementations of predicates and operations on them. Although BDD's form a promising representation, our approach is aimed at avoiding the state explosion even in case of verification of very large systems. In such cases, also BDD's are often still too large. We have left open the problem of effective computation of $\widetilde{pre}$ and *rel*. In the following, we indicate a possible solution using algebraic reasoning, thereby avoiding the state explosion.

5.1 Symbolic Computation

To handle predicates representing arbitrarily large or infinite state sets, representations which allow algebraic manipulations will be needed. Such representations may be seen as being yet more symbolic than BDD's, and operations on them are defined in terms of "symbol-pushing". For example, the $\widetilde{pre}$ function for a program S can be defined in terms of substitution of expressions for variables in predicates. To keep the representations of predicates as short as possible, simplifications based on rewrite rules are applied. Such algebraic computations may benefit from techniques developed in the field of abstract interpretation, originally developed as a unifying theory for the description and design of *program analyses*. There, the idea is that the operations of a programming language, which operate on concrete values, are *mimicked* by corresponding abstract operators defined over abstract values, describing sets of concrete values. A program analysis can then be viewed as an abstract execution of the program, either forwards or backwards, in which data and operations are abstractly interpreted, yielding a description of any concrete execution. Abstract interpretations are normally based on a notion of approximation, reflecting the fact that some precision of the result is being traded for efficient computability of abstract operations. In our case, $\widetilde{pre}$'s may be approximated by subsets, while $rel(\Sigma_{\mathcal{A}})$ may be approximated by a superset. Both approximations are safe in the sense that the resulting abstract model still preserves ACTL*, although strong preservation may be lost.

5.2 Further Work

The approach can easily be adapted to ECTL, the existential fragment of CTL, by splitting for characteristic predicates of ECTL formulae—which may be inductively computed using the predicate transformer *pre*—and choosing a different notion of abstraction. This abstraction is called *conservative* in [8] and shown to preserve ECTL. However, it is not immediately clear how the integrated reachability analysis may be adapted, as conservative abstractions do not preserve unreachability (Lemma 5).

The same approach of developing splitting algorithms may be applied to different logics, preorders and equivalences, such as fragments of ACTL and ECTL, LTL, SCTL (see [11] for an overview and bibliography of branching time temporal logics) and trace-equivalences and preorders. It may be expected that the degree of reduction of models will grow as the logic gets less expressive; it would be interesting to investigate the complexity of the resulting algorithms.

An obvious next step is implementation; in fact, implementing the algorithms is the only way to evaluate their usability.

Acknowledgements. We thank Ed Clarke for inviting the first two authors to CMU, where cooperation started. The first author thanks Joseph Sifakis and his group for the visit to VERIMAG that they made possible, their hospitality and the inspiring atmosphere. In particular, Susanne Graf is thanked for several discussions. Philippe Schnoebelen contributed the proof for Lemma 8. We thank the anonymous referees for their suggestions. The second author was partially supported by the U.S.-Israeli Binational Science Foundation.

References

1. S. Bensalem, A. Bouajjani, C. Loiseaux, and J. Sifakis. Property preserving simulations. In *CAV'92*.
2. A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18(3):247–271, 1992.
3. M.C. Brown, E.M. Clarke, and O. Grumberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131, 1988.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proc. 5th LICS*, 1990.
5. E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. In *Proc. 19th POPL*, 1992.
6. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
7. O. Coudert, J.C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagram. In *CAV'90*.
8. D.R. Dams and O. Grumberg. Abstract interpretation of reactive systems: Abstractions preserving ACTL*, ECTL* and CTL*. Technical report, Eindhoven University of Technology, Department of Computer Science, 1992.

9. D.L. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations. MIT Press, 1989.
10. E.A. Emerson and J.Y. Halpern. 'Sometimes' and 'Not Never' revisited: on branching time versus linear time temporal logic. *J. of the ACM*, 33(1):151–178, 1986.
11. E.A. Emerson and J. Srinivasan. Branching time temporal logic. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, LNCS 354, Springer-Verlag, 1988.
12. R. Enders, F. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *CAV'91.*
13. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *CAV'91.*
14. S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In these proceedings.
15. S. Graf and B. Steffen. Compositional minimization of finite state processes. In *CAV'90.*
16. O. Grumberg and D.E. Long. Model checking and modular verification. In *CONCUR'91*, LNCS 527, Springer-Verlag, 1991.
17. A.J. Hu, D.L. Dill, A.J. Drexler, and C. Han Yang. Higher-level specification and verification with BDDs. In *CAV'92.*
18. R. Janicki and M. Koutny. On some implementation of optimal simulations. In *CAV'90.*
19. R. Kaivola and A. Valmari. The weakest compositional semantic equivalence preserving nexttime-less linear temporal logic. In *CONCUR'92*, LNCS 630, Springer-Verlag, 1992.
20. S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, 6:107–120, 1992.
21. R.P. Kurshan. Analysis of discrete event coordination. In *Proc. Workshop on Stepwise Refinement of Distributed Systems: Models, Formalisms, Correctness*, LNCS 430, Springer-Verlag, 1989.
22. O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th POPL*, 1985.
23. K. McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *CAV'92.*
24. R. Milner. An algebraic definition of simulation between programs. In *IJCAI'71.*
25. R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
26. D. Park. Concurrency and automata on infinite sequences. In *Proc. 5th GI-Conf. on Theoretical Computer Science*, LNCS 104, Springer-Verlag, 1981.
27. D.K. Probst and H.F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In *CAV'90.*
28. J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symp. on Programming*, LNCS 137, Springer-Verlag, 1981.
29. G. Shurek and O. Grumberg. The modular framework of computer-aided verification: Motivation, solutions and evaluation criteria. In *CAV'90.*
30. A. Valmari. A stubborn attack on state explosion. In *CAV'90.*