Discriminative Model Checking

Peter Niebert[1], Doron Peled[2], and Amir Pnueli[3]

[1] Laboratoire d'Informatique Fondamentale de Marseille
CMI, 39, rue Joliot Curie, 13453 Marseille Cedex 13, France
[2] Department of Computer Science, Bar Ilan University
Ramat Gan 52900, Israel
Computing Science Department, Courant Institute of Mathematical Sciences,
New York University, 251 Mercer Street,
New York, NY 10012

Abstract. Model checking typically compares a system description with a formal specification, and returns either a counterexample or an affirmation of compatibility between the two descriptions. Counterexamples provide evidence to the existence of an error, but it can still be very difficult to understand what is the cause for that error. We propose a model checking methodology which uses two levels of specification. Under this methodology, we group executions as *good* and *bad* with respect to satisfying a *base* LTL specification. We use an *analysis specification*, in CTL* style, quantifying over the good and bad executions. This specification allows checking not only *whether* the base specification holds or fails to hold in a system, but also *how* it does so. We propose a model checking algorithm in the style of the standard CTL* decision procedure. This framework can be used for comparing between good and bad executions in a system and outside it, providing assistance in locating the design or programming errors.

1 Introduction

A 20 years old debate in the model checking community exists between the use of a branching or state-centric specification (e.g., CTL [3,2,13]) or using linear or path-centric specification (e.g., LTL [11]). One can also combine the approaches (using, e.g., CTL* [4]). We promote here a view where the specification of a system is given using linear formalism, as its *primary* or *base* specification. However, unlike the traditional linear approach, we are not satisfied with an affirmation or a counter example. Instead, we embed the base specification in a branching *analysis specification* that can express *how* it fails. The use of an analysis specification helps us obtain more refined information than the yes/no and counterexample situation of traditional model checking.

The base specification φ is given in LTL. This is the property that we are interested that the system (hardware or software) will satisfy. The formalism we use for the analysis specification is termed EmCTL*, for *embedded* CTL*. This is a CTL*-based specification, where the path quantifiers can range over the executions that satisfy or do not satisfy some base property. These will also be

A. Gupta and S. Malik (Eds.): CAV 2008, LNCS 5123, pp. 504–516, 2008.

referred to as *good* or *bad* executions, respectively. The analysis specification thus has some past flavor in it; one needs to observe some information of the prefix of the current execution so far in order to decide whether it can be continued into a good or a bad execution, rather than just looking at the possible future(s) from the current state.

We show the relation of EmCTL* to existing formalisms, in particular to its closest logic mCTL*, from which we can also obtain relevant expressiveness and lower bound complexity results. In fact, one can use a linear translation into mCTL*, and from there using the transformation into alternating hesitant tree automata with satellites word automata for model checking, presented in [8]. It turns out that CTL*, mCTL* and EmCTL* have the same expressive power, although expressing the same property in the different logics may vary in form and size.

We provide an alternative model checking algorithm in the style of standard CTL* decision procedure [4]. One advantage of using our algorithm is that it can be easily programmed as a variant of the CTL* algorithm. Moreover, its complexity analysis takes into account the different components of the specification, namely the base LTL part and the EmCTL* analysis specification. Specifically, we show that the complexity of the model checking is in PSPACE complete w.r.t. the analysis specification, and in EXSPACE complete w.r.t. the base property.

Our decision procedure suggests a generic method of exploiting an ability of finitary (regular-based) encoding of the past that can be combined with a nondeterministic choice for the future part, in order to add past-related quantification to branching temporal logics. We show how to use this capability in order to extend the quantification so that one can reason not only about the executions of a system, but also on executions that are not generated by the system, and may or may not satisfy the given specification.

In the analysis specification we use $\forall^{\varphi}$, $\exists^{\varphi}$ to range over the good executions, $\forall^{\neg\varphi}$, $\exists^{\neg\varphi}$ to range over the bad executions, and $\forall$, $\exists$ to range over all the executions. We use $\Diamond$, $\Box$, U, W for *eventually*, *always*, *until* and *unless* (weak until), respectively. Some examples follow:

Every prefix of a bad execution is also a prefix of some good execution: $(\forall^{\neg\varphi}\Box\exists^{\varphi} true)$.

A prefix of a good execution always has a point where it cannot turn bad: $(\forall^{\varphi}\Diamond\neg\exists^{\neg\varphi} true)$.

There are always a possibility of a good and a bad execution: $(\forall\Box(\exists^{\neg\varphi} true \wedge \exists^{\varphi} true))$.

Before executing α, there are good and bad executions. Once α is executed for the first time, there are only bad executions. In order to assert about the execution of particular transition, we assume that the predicate $Exec_{\alpha}$ holds in a state if α is the last transition executed[1]: $\forall((\neg Exec_{\alpha} \wedge \exists^{\varphi} true) W (Exec_{\alpha} \wedge \forall^{\varphi} false))$.

[1] This is either a transition predicate, or a predicate on states, where the state information includes the last executed transition, hence may separate formerly identical states.

The paper is structured as follows: In Section 2, the syntax and semantics of EmCTL* is introduced. In Section 3, we describe the model checking procedure. More detailed applications are given in Section 4, in particular, the formalism is extended to deal with execution sequences beyond those allowed by executing the system. In Section 5, we give details on the relation between EmCTL* and mCTL*. Conclusions appear in Section 6.

2 Embedding LTL Properties in Branching Time Specification

Syntax

An EmCTL* formula is expressed with respect to an LTL formula φ called the base specification. The syntax of EmCTL* is similar to CTL*, including state formulas (ψ) and path formulas (μ). The only difference is that the path quantifiers $\forall$ and $\exists$ may be superscripted by φ, referring to the base LTL formula. (In calculating the size of an EmCTL* formula we do not count the size of the superscripting formula φ, which is provided separately.)

$$\psi ::= p|\psi \vee \psi|\neg\psi|\exists\mu|\exists^{\varphi}\mu|\exists^{\neg\varphi}\mu \tag{1}$$

where p is a propositional letter over some set of propositions $\mathcal{P}$.

$$\mu ::= \psi|\mu \vee \mu|\neg\mu|X\mu|\mu U\mu \tag{2}$$

LTL syntax is restricted to a path formula without state subformulas.

Semantics

Let M be a finite structure $(S, \{s_0\}, E, \mathcal{P}, L)$ with states S, an initial state[2] $s_0 \in S$, edges $E \subseteq S \times S$, set of propositions $\mathcal{P}$, and labeling function $L : S \mapsto 2^{\mathcal{P}}$. (We may also assume a mapping on the edges from a fixed domain of *transitions* T, e.g., in order to define predicates such as $Exec_\alpha$, used in the introduction.) For simplicity, we assume that each state in S has a successor. This can be forced by adding a self loop, marked with a special symbol ϵ that is not in T, for each node without successors. A *path* in S is a finite or infinite sequence $\langle g_0 g_1 g_2 \ldots\rangle$, where $g_0 \in S$ and for each $i \geq 0$, $g_i E g_{i+1}$. An *execution* is an infinite path, starting with $g_0 = s_0$.

We denote the ith state of a path π by π_i, the suffix of π from the ith state by π^i and the prefix of π up to the ith state by $\hat{\pi}^i$. The concatenation of two paths ρ and π, where the last state of ρ is the same as first state of π, is denoted by $\rho \frown \pi$. Note that $\langle \pi_0 \rangle \frown \pi = \pi$, and that $\rho \frown \langle s \rangle = \rho$, where s is necessarily the last state of ρ.

[2] The initial state is denoted as a singleton *set* for future compatibility when taking the product with other structures.

We first recall the LTL semantics, for the base property φ. The semantics is defined for a suffix of an execution π of M as follows:
$\pi \models p$ if $p \in L(\pi_0)$.
$\pi \models \mu_1 \vee \mu_2$ if either $\pi \models \mu_1$ or $\pi \models \mu_2$.
$\pi \models \neg\mu$ if it is not the case that $\pi \models \mu$.
$\pi \models X\mu$ if $\pi^1 \models \mu$.
$\pi \models \mu U \eta$ if there exists some i such that $\pi^i \models \eta$ and for each $0 \leq j < i$, $\pi^j \models \mu$.

The semantics given for an $EmCTL^*$ state formula is of the form $M, \rho, s \models \psi$, where ρ is a finite prefix of an execution in M, leading to (i.e., ending with) the state s. The semantics given for a path formula μ is of the form $M, \rho, \pi \models \mu$, where ρ is again finite prefix of an execution of M, leading up to a state from which an infinite path π of M starts (thus, $\rho \frown \pi$ is an infinite execution of M). A path subformula μ is then evaluated in $M, \rho, \pi \models \mu$ according to the path π.

Intuitively, in the semantic definition of in $M, \rho, s \models \psi$ and, similarly, in the definition of $M, \rho, \pi \models \mu$, we keep the path so far ρ, so that we can assert whether the base property holds from the beginning of the execution or not. As we progress with the temporal operators in time over some finite fragment of a path, this fragment is appended to ρ and removed from π. We can either talk about the existence of an execution path, with $\exists$, one that satisfies φ, with $\exists^{\varphi}$, or one that does not satisfy φ, with $\exists^{\neg\varphi}$. The formal semantics is given as follows.
$M, \rho, s \models p$ if $p \in L(s)$.
$M, \rho, s \models \psi_1 \vee \psi_2$ if either $M, \rho, s \models \psi_1$ or $M, \rho, s \models \psi_2$.
$M, \rho, s \models \neg\psi$ if it is not the case that $M, \rho, s \models \psi$.
$M, \rho, s \models \exists\mu$ if there exists a path π of M such that $\pi_0 = s$ and $M, \rho, \pi \models \mu$.
$M, \rho, s \models \exists^{\varphi}\mu$ if there exists a path π of M such that $\pi_0 = s$ and $M, \rho, \pi \models \mu$, and furthermore, $\rho \frown \pi \models \varphi$ in LTL.
$M, \rho, s \models \exists^{\neg\varphi}\mu$ if there exists a path π of M such that $\pi_0 = s$ and $M, \rho, \pi \models \mu$, and furthermore, $\rho \frown \pi \models \neg\varphi$ in LTL.
$M, \rho, \pi \models \psi$ for a state formula ψ if $M, \rho, \pi_0 \models \psi$.
$M, \rho, \pi \models \mu_1 \vee \mu_2$ if either $M, \rho, \pi \models \mu_1$ or $M, \rho, \pi \models \mu_2$.
$M, \rho, \pi \models \neg\mu$ if it is not the case that $M, \rho, \pi \models \mu$.
$M, \rho, \pi \models X\mu$ if $M, \rho \frown \langle \pi_0 \pi_1 \rangle, \pi^1 \models \mu$.
$M, \rho, \pi \models \mu U \eta$ if there exists some i such that $M, \rho \frown \hat{\pi}^i, \pi^i \models \eta$ and for each $0 \leq j < i$, $M, \rho \frown \hat{\pi}^j, \pi^j \models \mu$.
We write $M \models \psi$ when $M, \langle s_0 \rangle, s_0 \models \psi$.

Other operators can be obtained using equivalences, e.g., $true = p \vee \neg p$, $false = \neg true$, $\psi_1 \wedge \psi_2 = \neg((\neg\psi_1) \vee (\neg\psi_2))$, $\mu \rightarrow \eta = (\neg\mu) \vee \eta$, $\forall^{\varphi}\mu = \neg\exists^{\varphi}\neg\mu$, $\Diamond\mu = true U \mu$, $\Box\mu = \neg\Diamond\neg\mu$, $\mu W \eta = (\mu U \eta) \vee \Box\mu$, etc.

3 Model Checking

Büchi and Generalized Büchi Automata

A *Büchi automaton* is similar to a finite structure, with an additional component $F \subseteq S$ of *accepting states*. An *accepted execution* (or *accepted run*) must satisfy

in addition to the conditions described for an execution of a state space above, that at least one accepting state from F appears in the execution infinitely many times.

A *generalized Büchi automaton* can have *multiple* accepting states F_1, F_2, $\ldots$, $F_m \subseteq S$. An accepted execution satisfies that at least one state from *each* accepting set appears on it infinitely often. Hence, this generalizes the singleton accepting set of a Büchi automaton. The generalization does not increase the expressive power, but makes the following presentation simpler. Translation from generalized Büchi automata to (simple) Büchi automata is standard. We identify the case of having no accepting set (i.e., when $m = 0$) with the case of having one accepting set consisting of all the states.

The product of two generalized Büchi automata is obtained in a standard way, where the state space consists of pairs of states that agree on their labeling. The acceptance is defined by taking together the accepting sets of each automata. Define $Q \bowtie R$ when $Q \subseteq S^1$, $R \subseteq S^2$ as $\{(q, r) | q \in S^1 \wedge r \in S^2 \wedge L^1|_{\mathcal{P}^1 \cap \mathcal{P}^2}(q) = L^2|_{\mathcal{P}^1 \cap \mathcal{P}^2}(r)\}$ (where $L|_{\mathcal{P}}(s) = L(s) \cap \mathcal{P}$). That is, the set of pairs that agree on the labeling of mutual propositional values. Let $M^1 = (S^1, S^1_0, E^1, \mathcal{P}^1, L^1, F^1_1, \ldots F^1_m)$, $M^2 = (S^2, S^2_0, E^2, \mathcal{P}^2, L^2, F^2_1, \ldots F^2_n)$. Then $M^1 \times M^2 = (S^1 \bowtie S^2, S^1_0 \bowtie S^2_0, \hat{E}, \mathcal{P}^1 \cup \mathcal{P}^2, \hat{L}, F^1_1 \bowtie S^2, \ldots, F^1_m \bowtie S^2, S^1 \bowtie F^2_1, \ldots, S^1 \bowtie F^2_n)$. The relation $\hat{E}$ is defined so that $(s, q)\hat{E}(s', q')$ iff $(s, q), (s', q') \in S^1 \bowtie S^2$, sE^1s' and qE^2q' and $\hat{L}$ is defined such that $\hat{L}(s, q) = L^1(s) \cup L^2(q)$.

A translation from an LTL formula φ into a Büchi automaton that accepts exactly the sequences satisfying φ is quite standard (see, e.g., [10]).

Model Checking Algorithm

We use the following components:

- Let A_φ be a Büchi automaton that accepts the sequences satisfying φ.
- Let $A_{\neg\varphi}$ be a Büchi automaton that accepts the sequences satisfying $\neg\varphi$.
- Let $Det(A_\varphi)$, $Det(A_{\neg\varphi})$ be the determinized *subset construction* for A_φ, $A_{\neg\varphi}$, respectively. A state of $Det(A_\varphi)$ represents a maximal set of possible states where control can be in A_φ after some finite run. Note that all the states of A_φ that participate in one state of $Det(A_\varphi)$ agree on their labeling. Moreover, $Det(A_\varphi)$ has no accepting component (equivalently, all its states are accepting).
- Let A_δ be a Büchi automaton for a formula δ, where $\exists\delta$, $\exists^\varphi\delta$ or $\exists^{\neg\varphi}\delta$ is a subformula of the analysis specification formula and where the maximal state subformulas of δ are treated as new propositional variables. Note that δ an LTL formula.

Suppose that we want to perform model checking for an analysis specification ψ over some base LTL property φ. We perform model checking on several types of automata products:

- $P = M \times Det(A_\varphi) \times Det(A_{\neg\varphi})$. Each state of P is annotated with state subformulas of ψ in a recursive way to be described below. Thus, subformulas

can be marked based on earlier marking of their subformulas. Note that there is no acceptance component in this product (thus, all states are considered accepting). If the analysis specification does not use positive quantification ($\exists^{\varphi}$) or negative quantification ($\exists^{\neg\varphi}$) then we may omit the product component $Det(A_{\varphi})$ or $Det(A_{\neg\varphi})$, respectively[3].

- $P_{\varphi,\delta} = P \times A_{\varphi} \times A_{\delta}$. This component is designed to mark the $\exists^{\varphi}\delta$ subformulas that hold in states of P (note that a state subformula of EmCTL* depends not only on the current state of the checked structure M, but also on information depending on the prefix of the execution, as summarized in P by the $Det(A_{\varphi})$ and $Det(A_{\neg\varphi})$ components). This product requires that in each of its states, the component from A_{φ} will be a member of the subset of states in the $Det(A_{\varphi})$ component. Since both A_{φ} and A_{δ} contribute acceptance conditions, an accepted execution will have to visit infinitely often accepting states from both Büchi automata (this corresponds to the generalized Büchi automata acceptance condition).
- $P_{\varphi,\neg\delta} = P \times A_{\neg\varphi} \times A_{\delta}$. This product is similar to the previous one, but the product is with a Büchi automaton for $\neg\varphi$, and is designed to mark the $\exists^{\neg\varphi}\delta$ subformulas that hold in states of P.
- $P_{\delta} = P \times A_{\delta}$. This product is design to mark the $\exists\delta$ subformulas in P.

The subset construction components $Det(A_{\varphi})$ and $Det(A_{\neg\varphi})$ are used to represent the information that is needed in order to check whether a path that starts from some given state, together with the prefix that is leading to it from the start of the execution, satisfy or does not satisfy the base property φ. This is required due to the use of the modalities $\exists^{\varphi}$ and $\exists^{\neg\varphi}$. Intuitively, for $\exists^{\varphi}$ it is sufficient to find a state of A_{φ} that is inside the subset construction, from which one can continue the execution according to the automaton A_{φ}. Since we do not know which such state can be used to complete the execution into one that satisfies φ, the subset construction keeps all the possibilities. In this way, there is no need to keep the entire prefix executed so far in order to see if there is a continuation that satisfies φ (or $\neg\varphi$, respectively).

A preparatory step before model checking, is to translate the formula to eliminate universal quantification $\forall^{\varphi}$, $\forall^{\neg\varphi}$ or $\forall$. This can be done based on the equivalence $\forall\delta = \neg\exists\neg\delta$, which carries over also to the other two universal quantifiers. The model checking is performed now recursively on the structure of the CTL* subformula ξ of the formula ψ. In each stage, we recursively mark states of P with either ξ or, otherwise, with $\neg\xi$ (hence in the list of cases below, there is no separate case for a subformula starting with the negation symbol). We identify $\neg\neg\xi$ with ξ.

ξ is a propositional variable. Then mark each state in P with ξ if ξ is in the labeling of the M component (we will, henceforce, omit mentioning repeatedly the fact that if we deal with ξ and a state is not marked with it, we will mark it with $\neg\xi$).

[3] For example, it would be beneficial to rewrite the first example in the introduction in an equivalent form as $(\forall\Box\exists^{\varphi} true)$.

- ξ is a state formula of the form $\psi_1 \vee \psi_2$. Then recursively mark P according to ψ_1 and according to ψ_2, and then mark a state in P with $\psi_1 \vee \psi_2$ if it is marked with either ψ_1 or with ψ_2.
- $\xi = \exists^{\varphi}\delta$ where δ is a path formula, when replacing each maximal state subformula of δ by a new propositional variable (the names of these variables can be the same as the subformulas they represent). For example consider $\xi = \exists^{\varphi}(X\kappa \wedge \mu U \eta)$ with state subformulas κ, μ and η. After the replacement, δ becomes an LTL formula, with Büchi automaton A_δ. First, we recursively call the marking procedure with the maximal state subformulas. We then construct $P_{\varphi,\delta} = P \times A_\varphi \times A_\delta$, as described above. A state of P is marked with $\exists^{\varphi}\delta$ if there exists a state in $P_{\varphi,\delta}$ with first component as in P (and recall that the second component is in accordance with the subset construction of $Det(A_\varphi)$ in P, as described above), and third component an initial state of A_δ. In addition, from this state we need to be able to reach a strongly connected component of $P_{\varphi,\delta}$ where there is an accepting state for A_φ and an accepting state for A_δ (as defined in the product construction).
- $\xi = \exists^{\neg\varphi}\delta$ where δ is a path formula. The procedure is similar to the previous case, replacing $P_{\varphi,\delta}$ with $P_{\neg\varphi,\delta}$.
- $\xi = \exists\delta$. In this case, we use the product P_δ and mark a state in P with δ if there exists a state in the product with the same P component, and an initial state of A_δ, from which a strongly connected component with an accepting state of A_δ is reached.

At the end of the marking, we check whether the initial states of the product P are marked with ψ.

The last three possibilities need an additional explanation. In order to check that a subformula, say $\exists^{\varphi}\delta$, holds in a state, we first mark all the states of P recursively with the maximal state subformulas in δ, or their negation. We treat the maximal state subformulas of δ as propositional variables, which are already marked in P by our recursive procedure. The satisfaction $\exists^{\varphi}\delta$ depends not only on the current state, but also on the path so far. As explained before, that information is encapsulated in components of $Det(A_\varphi)$, which are also embedded in the states of P. In a similar way, $\exists^{\neg\varphi}\delta$ holds in a state depending on the information embedded using $Det(A_{\neg\varphi})$.

We then construct the structure $P_{\varphi,\delta}$ and look for strongly connected components that satisfy both acceptance conditions of A_φ and A_δ. We now seek the states of $P_{\varphi,\delta}$ from which such a strongly connected component is reachable with its A_δ component being at its initial state and with P being at state s. We can then conclude that from the corresponding P state s, there is a path that satisfies δ. Moreover, recall that our product construction of $P_{\varphi,\delta}$ guarantees that the A_φ component agrees with one of the possible choices of $Det(A_\varphi)$. Thus, the fact that the acceptance condition of A_φ holds on the reachable strongly connected component guarantees that the formula φ holds when prefixing this path with the finite execution that reached s, taking care of the $\exists^{\varphi}$ quantification.

Complexity

In a naive implementation, our decision procedure is exponential in the size of the EmCTL* formula, and doubly exponential in the size of the embedded LTL property φ in both time and space. The latter is because of the need to determinize the Büchi automaton constructed for φ in order to keep track of all the possibilities where the control can reside under the current prefix. Note that some LTL specifications result in deterministic Büchi automata, in which case we will not incur the additional exponential cost.

The proof of EXSPACE lower bound of mCTL* can be used here. In particular, the formula used in the reduction (from *exponential tiling*) in [8] can be expressed in EmCTL* as $\forall\Box\exists^{\varphi} true$. Encouragingly, the additional exponent of the model checking is only in the size of the LTL base property φ, and not the checked structure M or the EmCTL* specification analysis formula.

Achieving the EXSPACE complexity with our construction can be done without constructing the automaton P explicitly, nor the components $P_{\varphi,\delta}$, $P_{\neg\varphi,\delta}$, P_{δ}. Instead, we perform a binary search recursively on the structure of the formula (according to the above cases) through the state space of these components. For a path subformula, this involves searching for a lasso shape of a bounded size. One needs to represent in this way states of the construction, where the size of a state of any one of the above components being exponential in the size of the base specification (and polynomial in the size of the analysis specification). Using the standard Savitch construction [14], one needs not represent the entire state space.

Using Multiple Base Properties

A simple modification to the decision procedure that was described earlier in this paper allows us to generalize our framework and quantify over different sets of path, satisfying separate linear properties. In general, we can allow multiple path quantifiers in an *EmCTL** formula to be indexed by different LTL formulas, e.g., $\exists^{\varphi_1}\ldots\exists^{\varphi_m}$. In this case, we need in the translation to take the product of the state space with a determinized version per each indexing property φ_i that appears on a path.

We may also want to use multi-indexed quantifiers, e.g., $\forall^{\varphi_1,\varphi_2}$, which will allow for selecting sequences satisfying several LTL conditions.[4] Note that this is not the same as $\forall^{\varphi_1}\forall^{\varphi_2}$, as, according to the semantic definition, the 1st quantifier is amalgamated into the 2nd one, since the 2nd quantifier provides a state formula, rather than a linear formula for the 1st quantifier.

In this case, we need to generalize the automata products in our construction. That is, we need to take components of the subset construction $Det(A_{\varphi_1})$ and $Det(A_{\varphi_2})$ in P, and take care of the acceptance conditions of both φ_1 and φ_2

[4] Semantically, this is the same as $\forall^{\varphi_1\wedge\varphi_2}$, but it might be beneficial to separate the subformulas, e.g., when two quantifiers share some but not all the subformulas, e.g., in $\forall^{\varphi_1,\varphi_2}\psi_1 \wedge \exists^{\varphi_1,\varphi_3}\psi_2$.

(as well as δ) in $P_{\varphi_1,\varphi_2,\delta}$. The complexity will be then doubly exponential in the sum of the lengths of the formulas that appear together in such a quantifier.

We later show how to combine this extension with the ability to assert about both executions inside and outside the checked system.

4 Applications

4.1 Level of Failure

In [7], an application of model checking for the generation of correct mutual exclusion algorithms through genetic programming was described. In order to select versions (mutations) of the code that are better than others, for creating new mutations, one assigns a level of failure, based on model checking analysis. Practice shows that it is not sufficient to do simple model checking with yes/no result. In [7], a deeper analysis, based on converting the LTL property into Streett automata, was performed. We can express such levels of failures and many others using EmCTL* specification, and use the model checking algorithm described here, instead of providing ad-hoc verification procedures. Some examples for formulating such levels of failure are as follows:

All the executions satisfy the base LTL property φ. This is simply expressed as $\forall\varphi$ (or, just φ in LTL). Note that we avoid writing this as $\forall^{\neg\varphi} \mathit{false}$, since this would incur an exponential blowout in model checking.

There exists a finite prefix of an execution from which all the executions are bad. $\exists\Diamond\forall^{\varphi} \mathit{false}$.

For all bad executions, after any prefix, one can always make a decision that would complete the behavior into a good execution. $\forall^{\neg\varphi}\Box\exists^{\varphi} \mathit{true}$. (Note again that this can be written equivalently as $\forall\Box\exists^{\varphi} \mathit{true}$.) Intuitively, this means that in order to have a bad execution, one needs to schedule infinitely many bad choices.

Note that, as in the model checking procedure presented here, the decision procedure in [7] also incurs an additional exponential explosion in the size of the checked property over LTL model checking. This is due to the use of deterministic Street automata in the analysis.

4.2 Reasoning Outside the Checked System

One of the directions for trying to locate errors in a system is based on comparing related executions, e.g., similar pairs of executions such that one of them satisfies the specification and the other does not [5,16]. This is usually done in the context of the system's executions. One of the examples in the introduction already demonstrated how we can use our framework to find, e.g., the point when the execution of a transition terminates the possibility of having further good executions. A related approach [6] searches for the moves that forces the system towards the error in the sense of game theory. We propose that such

analysis for locating errors can benefit from comparing not only the executions of the system, but also executions potentially not possible in the system.

One can use multiple indexed quantification to reason within the same formula about executions of the system, as well as executions not belonging to the system.

The quantifiers $\forall, \exists, \forall^{\varphi}, \exists^{\varphi}$ now range over both, executions within the system, and executions leaving the system. To recover the possibility of reasoning within the system, let new quantifiers $\forall^{S}, \exists^{S}, \forall^{S,\varphi}, \exists^{S,\varphi}$ refer to sequences that stay in the system.

The first step for performing model checking is to expand the state space M to represent also executions that do not result from the execution of the checked system, although a prefix might be possible in the system. Intuitively, once the actual system is left, "anything is possible". An efficient coding can be done as follows:

- We assume a labeling of edges of the state space, according to the executed transition.
- The state labeling function is extended to allow a conjunction of propositional variables, either negated or non-negated, which does not necessarily includes all the propositional variables. This allows representing states in a more compact way, i.e., one state can represent all the states that satisfy such a given formula (e.g., as done in [10]).
- A new propositional variable r will mark nodes as *reached*. Its negation $\neg r$ will denote nodes that were not reached through an execution of the checked system.
- We add a new *sink* state τ, labeled with the formula $\neg r$ (with no reference to the other propositional variables). The rest of the nodes of M retain their original label, with the added conjunct r. The state τ is *not* initial if we want to perform model checking with respect to the given initial state s_0.
- From each state s of the structure M we add an edge from s to τ marked with ϵ. There is also a self loop marked with ϵ from τ to itself.

It is straightforward to adapt the constructions of Section 3 to deal with these modified quantifiers. When quantifying over executions of the system and checking an LTL subformula, we must still have a reachable strongly connected component satisfying all the Büchi conditions as in the algorithm before. But now, in addition, we need such a strongly connected component also to have a state marked with r. This is sufficient due to the monotonicity of the $\neg r$ marking.

Safety and Liveness

As an example of properties that can be expressed and checked with quantifiers going beyond the system specification, let us consider safety and liveness.

Halpern and Schneider formally defined the classification of properties in [1] based on Lamport's informal characterization as follows:

Safety. A property φ is a safety property if $\rho \models \psi$ iff for any decomposition $\rho = \rho_1 \frown \rho_2$ there exists a σ such that $\rho_1 \frown \sigma \models \varphi$.

Liveness. A property φ a liveness property iff for any ρ there exists a σ such that $\rho \frown \sigma \models \varphi$.

For properties given as Büchi automata, Alpern and Schneider moreover give constructions for splitting a temporal property φ into two parts, $safe(\varphi)$, which is the safety part related to φ, and $live(\varphi)$¡ which is the liveness part. We can express the liveness and safety parts in our setting as follows: $safe(\varphi)$ states that any finite prefix of an execution of the system can be extended (within the system or not) to satisfy φ:

$$safe(\varphi) = \forall^S \Box \exists^\varphi true$$

$live(\varphi)$ states that any system execution that does not satisfy φ passes through a prefix that has no extension (in the system or not) that satisfies φ:

$$live(\varphi) = \forall^S (\varphi \vee \Diamond \forall^\varphi false)$$

Indeed, the conjunction of $\Box \exists^\varphi true$ and $\Diamond \forall^\varphi false$ is contradictory so that the conjunction of $safe(\varphi)$ and $live(\varphi)$ is equivalent to $\forall^S \varphi$.

$safe(\varphi)$ and $live(\varphi)$ can then be directly checked using our decision procedure. Thus, when a property φ fails to hold in a system, we can check whether this is already due to its safety or liveness part failing to hold. Note, that our decision procedure (the naive version) is doubly exponential in the size of φ. The constructions of [1] is based on modifications of the property automaton, but in order to do model checking, the automaton needs to be complemented – at exponential cost. If model checking is the aim, the worst case complexity of either the original construction of [1] or the use of our algorithm is the same.

5 Related Work and Expressiveness

The closest logic to ours, as far as we know, is mCTL*, presented in [8]. In that logic, quantifying is relativized to the current prefix of execution. But as opposed to EmCTL*, path quantification is done always with respect to the beginning of the path. A special symbol, *present*, keeps track of the current state in a prefix of an execution, from which quantification forces the subformula to be interpreted from the initial state again. One can then refer to this symbol inside the subformula, in order to have the ability to assert about the continuation of that finite prefix (care should be taken, as multiple occurrences of the *present* symbol may refer to different states in the same formula).

It is shown that one can translate every CTL* property into an mCTL* property with linear blowup, where each subformula of the form $\exists\varphi$ (where universal quantification is first eliminated) is translated into $\exists\Diamond(present \wedge \varphi)$. A reverse translation is also available, but may explode the formula (inherently) in a nonelementary way [8].

We can translate each EmCTL* formula into mCTL* in the following way: each subformula of the form $\exists^\varphi \psi$ will be translated into $\exists(\varphi \wedge \Diamond(present \wedge \psi))$

and $\exists^{\neg\varphi}\psi$ will be translated into $\exists((\neg\varphi) \wedge \Diamond(present \wedge \psi))$. This allows one to use the decision procedure from [8]. The translation goes via models called *alternating hesitant tree automata with satellites word automata*. However, that construction is doubly exponential in the size of the overall property.

We suggested in this paper a generalized Büchi automata based construction, which incurs a small modification on the traditional CTL* model checking algorithm [4], and separated the complexity analysis; the additional exponent is only due to the LTL embedded part, and not related to the CTL* part (including path subformulas appearing in the embedding analysis specification formula. The main gain is in the cases where the embedded LTL part is rather small, or translates efficiently into a deterministic Büchi automaton. Of course, in general, there are cases, even for simple safety properties (take e.g., the property that p holds some fixed number of steps before q holds), where the deterministic automaton for the property is exponentially bigger than the nondeterministic version [12,9].

The collection of the following three facts:

- every CTL* formula is in particular an EmCTL* formula (when not using indexed path quantification),
- each EmCTL* formula is translatable to mCTL* (as we showed), and
- each mCTL* formula can be translated into CTL* (proved in [8]),

proves that the expressive power of the three logics is the same. Nevertheless, this does not mean that they are interchangeable for all purposes, nor that the naive approach of picking one formalism, translating the specification into it and performing the model checking, is a sensible choice. One should select the appropriate formalism according to the application (e.g., depending on ease of expressiveness, complexity of model checking with respect to the typically used formulas, etc).

6 Conclusions

We have outlined a specification and model checking methodology that allows to reason in CTL*-style about how an LTL property is satisfied (or not) by a finite state transition system. This is achieved using a formalism we named EmCTL* with path quantifiers $\forall^{\varphi}$, $\exists^{\varphi}$ limiting the quantification to system executions that meet an LTL property φ. Model checking is then in EXPSPACE-complete with respect to the indexing base formula φ.

We provided several examples of analysis specifications that can be used to gain further information about the checked system. This can be interesting in particular after model checking of the base property was performed and an error trace was found, and thence one would like to gain some additional information about the failure. Such an analysis requires an additional exponent in the size of the base specification on top of ordinary LTL model checking. However, it still in PSPACE-complete with respect to the checked system (and also in size of the analysis specification).

We propose that even a restricted subset of our logic and decision procedure can be a useful extension for model checking. In particular, we observe that our choice of examples point out to such a useful subset of formulas: the nesting of indexed quantification is at most two, and the deepest level of subformulas is followed by the trivial state subformulas *true* or *false*. This restricted form contains however the formula $\forall\Box\exists^{\varphi} true$, whose model checking is in EXSPACE complete in the size of the base property φ.

References

1. Alpern, B., Schneider, F.B.: Recognizing Safety and Liveness. Distributed Computing 2, 117–126 (1987)
2. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
3. Emerson, E.A., Clarke, E.M.: Characterizing Correctness Properties of Parallel Programs using Fixpoints. In: de Bakker, J.W., van Leeuwen, J. (eds.) ICALP 1980. LNCS, vol. 85, pp. 169–181. Springer, Heidelberg (1980)
4. Emerson, E.A., Lei, C.L.: Modalities for Model Checking. Science of Computer Programming 8, 275–306 (1987)
5. Groce, A., Chaki, S., Kroening, D., Strichman, O.: Error Explanation with Distance Metrics. STTT 8, 229–247 (2006)
6. Jin, H., Ravi, K., Somenzi, F.: Fate and Free Will in Error Traces. In: Katoen, J.-P., Stevens, P. (eds.) ETAPS 2002 and TACAS 2002. LNCS, vol. 2280, pp. 445–459. Springer, Heidelberg (2002)
7. Katz, G., Peled, D.: Model Checking Based Genetic Programming with an Application to Mutual Exclusion. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 141–156. Springer, Heidelberg (2008)
8. Kupferman, O., Vardi, M.Y.: Memoryful Branching Time Logic. In: LICS 2006, Seatle, USA, pp. 265–274 (2006)
9. Kupferman, O., Vardi, M.Y.: Model Checking Safety Properties. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999)
10. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV 1995, pp. 3–18 (1995)
11. Pnueli, A.: The Temporal Logic of Programs. In: 18th IEEE Symposium on Foundations of Computer Science, pp. 46–57 (1977)
12. Pnueli, A., Rosner, R.: On the Synthesis of Reactive Systems. In: POPL 1989, Austin, Texas, pp. 179–190 (1989)
13. Quielle, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: 5th International Symposium on Programming, pp. 337–350 (1981)
14. Savitch, W.J.: Relationships between Nondeterministic and Deterministic Tape Complexities. Journal of Computer and System Science 4, 177–192 (1970)
15. Shahar, E.: The TLV System and its Applications, M.Sc. Thesis, The Weizmann Institute of Science
16. Sharygina, N., Peled, D.: A Combined testing and Verification Approach for Software Reliability. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 611–628. Springer, Heidelberg (2001)