Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Framework for Counterexample Generation and Exploration

Marsha Chechik and Arie Gurfinkel

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: e-mail: {chechik,arie}@cs.toronto.edu

Received: date / Revised version: date

Abstract. Model-checking is becoming an accepted technique for debugging hardware and software systems. Debugging is based on the "Check / Analyze / Fix" loop: check the system against a desired property, producing a counterexample when the property fails to hold; analyze the generated counterexample to locate the source of the error; fix the flawed artifact – the property or the model. The success of model-checking non-trivial systems critically depends on making this Check / Analyze / Fix loop as tight as possible. In this paper, we concentrate on the Analyze part of the debugging loop. To this end, we present a framework for generating, structuring and exploring counterexamples either interactively or with the help of user-specified strategies.

1 Introduction

Model-checking is rapidly becoming an accepted method for analyzing software and hardware systems. Given a finite-state description of a system and a temporal logic property, model-checking automatically decides whether the property holds in the system. In addition to telling the user whether the desired temporal property holds, it can also generate a counterexample, explaining the *reason why* this property failed. Typically, counterexamples are given in terms of states and transitions of the model and can be effectively used for debugging. The counterexample generation ability has been one of the major advantages of model-checking when compared to other verification methods.

During debugging, a model-checker is used as a part of the Check / Analyze / Fix loop: check the model, analyze the produced counterexample, and fix the model or the property. Copty et al. [10] describe several stages of debugging: (1) the specification debugging stage, during which we fix the properties to make them trustworthy; (2) the model debugging stage, during which the actual bugs in the model are being found; and (3) the quality assurance stage which addresses the problem of "regression verification" – making sure that fixing one error does not introduce new ones.

Some version of the Check/Analyze/Fix loop frequently applies, and the goal of this work is to make this loop tighter. The Check phase involves running a model-checker, which is expensive and often takes hours to run even for moderately-sized models [10]. So, it is desirable to minimize the number of model-checking runs while maximizing the information obtained from each run.

The Analyze phase is measured in terms of the time that an engineer spends exploring the generated counterexamples, and thus is costly as well. It is possible for model-checkers to generate too many counterexamples [12], flooding the user with information, and making it hard to build a mental picture of what is going on. In this case, the user may spend too much time and energy trying to reach the portion of interest or get confused about the purpose of a given sub-trace in the overall explanation. Also, since the size of the property under analysis is typically much smaller than the size of the system, formula-specific patterns often repeat themselves throughout the counterexamples [12], and users fail to notice them. It is therefore desirable to have control over just how much information is generated by the model-checker. This can be accomplished via *interactive explanations* – counterexample generation based on user preferences. Interactive explanations can allow users to put a bound on the time that the model-checker spends computing the information, and continue exploring it manually; control which option is used to facilitate the generation of "interesting" counterexamples; and control the amount of information that is generated and presented by restricting the scope of exploration according to some criterion of interest. Clearly, interactive explanations make the problem of generating and understanding counterexamples much more tractable. Specifically,

- The amount of information generated is based on what the user is willing to understand. This helps scalability of our approach to large models.

– The amount of information displayed makes it easier to identify "interesting" cases and helps with debugging.

Since model-checking runs are expensive, it is desirable to enable users to fix as many errors as possible before repeating the verification, rather than eliminating one counterexample at a time. For example, we would like to know that $f \wedge g$ fails to hold because *both* f and g are false. To this end, providing the user with all disjoint counterexamples to a given property can significantly shorten the debugging time.

Model-checking can also be used for *design exploration* [2, 17]. For example, *query-checking* [17,3] enables users to find which temporal logic formulas hold in their system. Design exploration can also be facilitated by the counterexample generation facilities of the model-checker. This way, the primary difference between verification and exploration is their goal and thus in the meaning of the generated trace: in the former, it signifies an error; in the latter – a way to see a scenario of interest without specifying the exact input sequences leading to it. In fact, explaining why a property p fails to hold (a *counterexample*) is the same as explaining why a property $\neg p$ holds (a *witness*). In this paper, we often use witnesses and counterexamples interchangeably, referring to them collectively as *evidence*. For design exploration, it is desirable to reason about multiple executions of the system in parallel. Thus, the user can provide a set of constraints in the form of a temporal logic property that an "interesting" trace through the system should satisfy, and the model-checker computes such evidence while checking the property.

Contributions of this paper. In this paper, we propose a framework for creating, structuring and interactively exploring evidence. The framework is based on the idea that the most general type of evidence to why a property holds or fails to hold is a *proof.* Such proofs can be presented to the user in the form of *proof-like counterexamples* without sacrificing any of the intuitiveness and close relation to the model that users have learned to expect from model-checkers. Our results follow from the primary observation that counterexamples are simply proofs by example; yet they are the *coarsest* type of proof which skips all steps except those that result in the transition to the next state. Additional proof steps that explain *why* the model-checker chose this sequence of states for presentation can be added as means of annotating the counterexamples. With this approach, counterexamples remain intuitive but are easier to understand because the user can refer to the proof for the explanation. Furthermore, the proof enables us to give feedback to the user when counterexamples are not available. For example, if the model-checker determines that a property "in all next states φ holds", we (1) show the entire list of successors of the current state and (2) indicate that φ holds in these states.

Basing the evidence given to the user on proofs allows us to unify a number of existing ad-hoc approaches to exploring counterexamples. In particular, notions of forward and backward exploration as well as starting and stopping conditions are natural in the proof setting. Proofs can also be used to control what kind of evidence is being generated. The primary sources of such choices are:

1. determining which part of the property to explain (e.g., if the property is $p \vee q$, should the presented evidence be for p or for q?) and
2. determining which part of the model to use for the explanation (e.g., if the property is "there exists a next state where p holds" and the model has several such states, which should be presented?).

The above choices can be made by the user interactively or automated in the form of *strategies* (e.g., if faced with a choice of states for the explanation, always choose the one where some predicate x holds). The application of strategies is implemented in our framework by changing the proof rules used to generate evidence. The modification of proof rules can be *permanent* for the duration of the entire run of the model-checker, and thus can be facilitated by *history-free strategies*. Alternatively, the application of strategies can depend on the previously-observed behaviour of the system. For example, to see the infinite alternation between x and y, we may want to specify a strategy that oscillates between preferring a state where x holds and a state where y holds.

Finally, from the software engineering point of view, our framework provides a simple, unified way to interact with the counterexample generator. The interaction is based on defining strategies that combine property-based and model-based choices. For example, we can specify a strategy that prefers the part of the model that the user has explored previously, while attempting to satisfy a part of the property for which the witness is the shortest.

Clearly, most users cannot understand large proofs. In our framework, proofs are used in the back-end. They help generate and navigate through the evidence, without the need to be presented to the user. Instead, users see witnesses and counterexamples. Furthermore, large proofs are never computed in our approach since proof fragments are generated from the model-checking runs as part of interactive explorations to facilitate user-understanding.

In this paper, we describe the technique for generating proofs and proof-like counterexamples for CTL. This language is often used in model-checking, specifically, in an industrial-strength model-checker NuSMV [6]. We also *illustrate* the framework using a simple example rather than *validate* its effectiveness via a sizable case study. Here, we draw on industrial experience [10] that being able to limit the amount of evidence shown and generating several counterexamples at once is extremely effective in reducing the effort that engineers spend looking for a real cause of an error. Our framework unifies a number of existing approaches and allows users to create additional strategies that may further improve the debugging process. Thus, it can be used for explaining the reason why the property failed or succeeded, determining whether the property was correct ("specification debugging"), and for general model exploration.

Organization. The rest of this paper is organized as follows: We discuss CTL model-checking in Section 2 and the framework from the user perspective in Section 3. In Section 4, we discuss the internals of the framework, followed by Section 5 and Section 6 that describe aspects of the framework in more detail. In Section 7, we enrich the framework with additional proof strategies that allow the user to control which counterexamples get generated. In Section 8, we discuss how to use our framework to generate several counterexamples at once. Section 9 describes KEGVis – a prototype implementation of our framework. We compare our work with related approaches in Section 10 and conclude in Section 11 with the summary of the paper and discussion of future research directions.

2 CTL Model-Checking

Model-checking is an automated verification technique that receives a system M and a temporal logic property φ and decides whether φ holds in M. In this paper, we assume that M is a Kripke structure consisting of a finite set of states S, a designated initial state s_0, a set of atomic propositions A, a total transition relation $R \subseteq S \times S$, and a labeling function $I : S \to 2^A$ that assigns a truth value to each atomic proposition in each state. An example Kripke structure is shown in Figure 1(a).

We specify properties in *Computation Tree Logic* (CTL) [7], defined below:

$$\begin{aligned}\varphi = a \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid EX\varphi \mid AX\varphi \mid EF\varphi \mid AF\varphi \\ \mid EG\varphi \mid AG\varphi \mid E[\varphi\ U\ \varphi] \mid A[\varphi\ U\ \varphi]\end{aligned}$$

where a is an atomic proposition. The meaning of the temporal operators is: given a state and paths emanating from it, φ holds in one (EX) or all (AX) next states; φ holds in some future state along one (EF) or all (AF) paths, φ holds globally along one (EG) or all (AG) paths, and φ holds until a point where ψ holds along one (EU) or all (AU) paths. Some properties of the model in Figure 1(a) are: "it is possible to generate a request" (EFr) and "once a button is pressed, a request will be generated" ($AG(p \Rightarrow AFr)$).

We write $[\![\varphi]\!]^M(s)$ to indicate the value of φ in the state s of M, and $[\![\varphi]\!](s)$ when M is clear from the context. A formula φ is satisfied in a Kripke structure M if and only if it is satisfied in its initial state. The operators EX, EG, and EU form an adequate set, i.e., all other operators can be defined from them as follows:

$$\begin{aligned}AX\varphi &\triangleq \neg EX\neg\varphi\\ AF\varphi &\triangleq A[\mathsf{true}\ U\ \varphi]\\ EF\varphi &\triangleq E[\mathsf{true}\ U\ \varphi]\\ AG\varphi &\triangleq \neg EF\neg\varphi\\ A[\varphi\ U\ \psi] &\triangleq \neg E[\neg\psi\ U\ \neg\varphi \wedge \neg\psi]\ \wedge\ \neg EG\neg\psi\end{aligned}$$

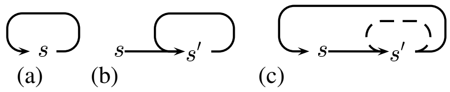

Fig. 2. Possible witnesses for EGp in state s: (a) a looping witness, (b) a path followed by a loop, and (c) two witnesses combining cases (a) and (b).

Semantics of EX, EG and EU is formally defined below.

$[\![EX\varphi]\!](s)$	**iff**	$\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t)$
$[\![E[\varphi U \psi]]\!](s)$	**iff**	there exists a path $s_0, \ldots, s_n$ such that $s = s_0$ and $[\![\psi]\!](s_n)$ and $\forall i < n \cdot [\![\varphi]\!](s_i)$
$[\![EG\varphi]\!](s)$	**iff**	there exists an infinite path $s_0, s_1, \ldots$ such that $s_0 = s$ and $\forall i \in nat \cdot [\![\varphi]\!](s_i)$

We also introduce a bounded version of the EU operator, that restricts path quantification to paths of bounded length:

$$\begin{aligned}E[\varphi U_0 \psi] &\triangleq \psi\\ E[\varphi U_i \psi] &\triangleq \psi \vee (\varphi \wedge EXE[\varphi U_{i-1}\ \psi])\end{aligned}$$

Note that

$$E[\varphi\ U\ \psi] = E[\varphi\ U_\infty\ \psi]$$

3 User View of The Framework

In this section, we illustrate the framework from the user perspective on a familiar example of an elevator controller system.

3.1 Elevator Controller System

An elevator controller system consists of a single elevator which accepts requests made by users pressing buttons at the floor landings or inside the elevator. The elevator moves up and down between floors and opens and closes its doors in response to these requests.

For our examples, we use the model specified in SMV by Plath & Ryan [23]. We do not present the state-machine model here because the purpose of our use of counterexamples is model debugging and model understanding. However, to illustrate a few concepts, we do provide a state machine for a module Button, shown in Figure 1(a). One instance of the Button module is produced for each button inside the elevator and on floor landings.

Variable f determines when the request has been fulfilled and the button can be reset. We model the latching explicitly: variable p determines the state of the button (pressed or released), whereas r determines whether the request to move to the desired floor has been generated. We further assume that a request cannot be fulfilled before it has been generated, i.e., f cannot become true if r is false. In Figure 1(a), we only show true variables; thus, in state s_0, p, r, and f are false.

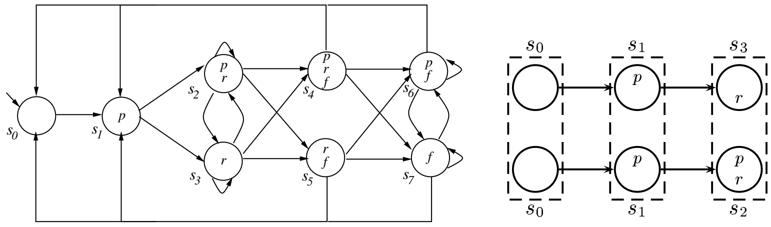

Fig. 1. (a) State machine for the module Button; (b) Two witnesses of length 3 for $[\![EFr]\!](s_0)$ for this statemachine.

3.2 Witnesses and Counterexamples

Suppose we are interested in checking the following property of the Button module: "it is never the case that a request can be fulfilled", expressed in CTL as $AG\neg f$. The counterexample to this property is a finite path that starts in the initial state (s_0) and arrives at the state where f is true, e.g., s_0, s_1, s_2, s_5. Note that this path is also a witness to the negation of the above property: EFf, i.e., "it is possible to fulfill a request".

Consider another property: "whenever a request is generated, it will eventually be fulfilled", formalized in CTL as $AG(r \Rightarrow AFf)$. The counterexample to this property, or a witness to the equivalent property $EF(r \wedge EG\neg f)$, is an infinite behavior that describes (1) how the system can reach a state where r holds and from then on (2) how it can avoid entering a state in which f holds. One such path is s_0, s_1, s_2 (reaching r), followed by a loop at s_2 (so f is always false).

Unlike traditional model checkers, our framework does not automatically generate a single counterexample. Instead, it automates the process of dynamically constructing one, or several, starting from the initial state. Further, it gives two separate views of the counterexample: the low-level view, which describes each state explicitly, naming its variables, and a high-level view that shows the complete trace and annotates each state with additional information, which we refer to as *summaries*, describing the significance of the state with respect to the overall property, and summarizing the rest of the trace.

3.3 Exploring the Elevator Controller Model

We now describe user interactions with the framework while debugging and exploring the Elevator model.

Generating several counterexamples at once. When we start verifying the system using the model-checker, it is usually the case that the property we are trying to check is wrong. Consider the property: "from any state, all paths go through a state where the elevator is on the third floor and doors are open" ($AGAF(\text{floor} = 3 \wedge \text{doors} = \text{open})$). The first counterexample tells us that it is possible to start on the first floor and stay there forever. We may conclude that the first floor is "special", and instead check that our desired configuration is reachable from any floor except the first one: $AG(\text{floor} \neq 1 \Rightarrow AF(\text{floor} = 3 \wedge \text{doors} = \text{open}))$. The counterexample we get in this case would lead us to the second floor and remain there forever, or possibly oscillate between the first and the second floor, without ever reaching the third. Seeing all three counterexamples at once would have helped us determine that the elevator never gets to the third floor unless a request for this floor has been made, and the property should have been updated to $AG(\text{btn3}.r \Rightarrow AF(\text{floor} = 3 \wedge \text{door} = \text{open}))$.

Excluding a known counterexample via strategies. Consider the above example. Instead of modifying the property to exclude our first counterexample, which is often difficult for engineers, we specify a strategy that attempts to avoid the state where $\text{floor} = 1$, if possible. A success of this strategy allows us to discover further counterexamples without modifying the property.

Preferring/avoiding the explored part of the model. Our model of the elevator controller comes with a number of desired properties, e.g., "the elevator never moves with its doors open", "every request for the elevator is eventually fulfilled", etc. When analyzing a few of these, we quickly get familiar with part of the model, e.g., we discover that the elevator can stay in a state where floor=1, doors are closed, the state of the controller is notMoving, and the direction of the elevator is up. We call this state Idle.

Strategies allow engineers to use their knowledge of "designated" states, such as Idle, to guide the counterexample generator towards them in the case where an AF property is false. In particular, using the information about the state of the doors, the direction of the movement, and the state of the controller, we define a *distance* function between the Idle state and the current state of the model, and specify a strategy that picks a state where this distance is minimized.

Note that an additional benefit of using this strategy during debugging is that we can stay within a better-understood part of the model. On the other hand, if the goal of model-checking is model exploration [2], we may instead choose to avoid the known behavior by maximizing the distance between the next state in the proof and Idle.

Choosing the "best" loop using summaries. Generating the shortest counterexample for an arbitrary temporal property is NP-hard [8], and thus conventional model-checkers apply a greedy strategy by computing the shortest counterexample to each subformula. In the case of counterexamples to AFp (or witnesses to $EG\neg p$), even this strategy is hard to implement. Instead, model-checkers consider a state s to satisfy

EGp if either there is a path on which p holds in each state that loops around s (see Figure 2(a)) or there is a successor of s in which p holds and there is a looping path of p-states around it (see Figure 2(b)). Thus, the algorithm to compute a witness to EGp in state s checks whether there is a path that leads back to s and on which p holds in each state, terminates if such a path is found, and otherwise picks a successor of s where EGp holds as the new state and continues. Such an algorithm picks the *first* loop on a path, even if it is long and hard to explore. We illustrate this scenario in Figure 2(c): the dashed loop around s' may be short and simple, whereas conventional model-checkers always return the solid loop around s, if one exists, as the witness to EGp.

Our framework allows the user to define strategies to loop around a familiar state (e.g., Idle) or use state summaries to choose the most interesting loop. Consider the witness to a property EGp in the state s_1 of the model in Figure 1(a). Clearly, there are several paths that satisfy it: s_1, followed by a loop around s_2; a loop s_1, s_2, s_4, s_1; a loop s_1, s_2, s_4, s_6, s_1, etc. The framework displays the state s_1 and indicates that EGp holds in it; this is the current "explanation" of the state s_1. Clicking on EGp produces further explanations of why EGp holds in s_1: (1) s_1 is part of a three-state loop and p holds in each state of the loop; (2) there is a successor state of s_1 from which we can explain EGp. Clicking on the second explanation tells us that the successor state of interest is s_2, and EGp holds in it. Clicking on this EGp tells us the reason: (2a): there is a self-loop (a loop of length 0) around s_2 and p holds in s_2; (2b): there is a successor state of s_2 in which EGp holds. At this point, we can either go back to the first explanation and in a few clicks reveal the three states of the loop, or decide that the self-loop around s_2 provides the better explanation. Of course, we can continue looking for other explanations until all possible p-loops have been discovered.

Alternatively, after the first explanation that tells us that s_1 is part of a three-state loop, we may choose to define a strategy that examines the model from state s_1 up to depth three to see whether there are other witnesses to EGp and how long the corresponding loops are, and then chooses the shortest such loop to explore.

4 Framework

The framework for generating and visualizing counterexamples is shown in Figure 3. Dashed lines indicate optional inputs. The user interaction with the framework starts by providing a proof keeper with a model M and a property φ. The proof keeper is the central part of the framework, responsible for generating (a fragment of) the proof and presenting it to the user. First, it calls a model-checker to find out whether φ is satisfied or violated by the model. It then uses the database of proof rules, according to a user-specified proof strategy, to prove that fact. For a given property φ, a proof of its validity is constructed by applying the basic proof strategy: (1) the database of the proof rules is consulted to find all applicable proof rules based on the syntax of the property; (2) a set of rules from which a valid proof can be constructed is chosen; (3) the rule to be applied is randomly chosen from the resulting set. Clearly, proof generation can be difficult or easy depending on who resolves part (2) of the above strategy. In our framework, this choice is made by the model-checker, and is completely automatic. The model-checker also ensures the soundness of the resulting proof. Additional runs of the model-checker are avoided by efficiently computing and storing evidence, as discussed in [25, 19]. The current proof fragment produced by the proof keeper is shown to the user via a visualization engine. The interaction of the user with this part of the framework is captured by user-supplied visualization strategies. The visualization engine converts the proof into a witness or a counterexample, and presents it to the user in a proof-like style, combining the advantages of both proofs and traditional counterexamples. In the rest of the paper, we discuss the framework in more detail. Proof generation is described in Section 5, the visualization engine – in Section 6, and the framework augmented with additional proof strategies – in Section 7.

5 Generating Proofs

We start by providing formal background on CTL model-checking, continuing to develop a proof system for CTL.

5.1 Background

We use $\top$ and $\bot$ to represent *True* and *False*, respectively. $\mathbb{B}$ and nat are the set of boolean values $\{\top, \bot\}$ and all natural numbers, respectively. An unnamed function over the domain D is denoted by $\lambda x \in D \cdot$ *F-n Body*. $\mu Z \cdot f(Z)$ and $\nu Z \cdot f(Z)$ are the least and the greatest fixpoints of a function f, respectively. These allow us to define CTL operators symbolically:

$$\begin{aligned} EG\varphi \quad &\triangleq \nu Z \cdot \varphi \ \wedge \ EXZ \\ E[\varphi \ U \ \psi] &\triangleq \mu Z \cdot \psi \vee (\varphi \ \wedge \ EXZ) \end{aligned}$$

ECTL is a fragment of CTL where every path is existentially quantified and negation is restricted to atomic propositions. ACTL is the dual of ECTL, i.e., only universal quantification is allowed.

In this paper, we use p and q to stand for arbitrary atomic propositions, s and t to represent states, and φ and ψ to represent CTL formulas. We also use the notation $\{s\}$ to express a formula that evaluates to $\top$ at state s and $\bot$ otherwise, i.e., $[\![\{s\}]\!](t) \triangleq (s = t)$. We use $\overline{\{s\}}$ for the negation of $\{s\}$.

We now aim to develop a proof system that allows us to generate proofs for CTL. We start from ECTL and then extend our framework to deal with universal quantification and negation.

5.2 Proof rules for ECTL

Our initial goal is to develop a sound and complete proof system that allows us to prove validity of sentences of the form

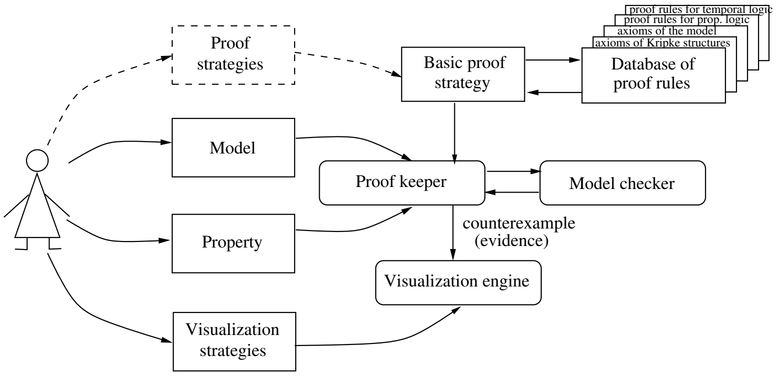

Fig. 3. Overview of the framework.

$$\frac{a}{a \vee b}\ \vee\text{-intro} \qquad \frac{a \quad b}{a \wedge b}\ \wedge\text{-intro}$$

$$\frac{b}{a \vee b}\ \vee\text{-intro} \qquad \frac{f(d)}{\exists x \in D \cdot f(x)}\ \text{one-point rule}$$

$$\frac{f(d_1) \qquad f(d_2) \qquad \ldots \qquad f(d_n)}{\forall d \in D \cdot f(d)}\ \text{finite quantification, with } \bigcup_{i=1}^{n}\{d_i\} = D$$

$$\frac{\forall x \in D_1 \cdot f(x) \wedge g(x) \qquad \forall x \in D \setminus D_1 \cdot \neg f(x)}{\forall x \in D \cdot f(x) \rightarrow g(x)}\ \text{universal case splitting}$$

Fig. 4. Some axioms of propositional and boolean logic.

$[\![\varphi]\!](s)$, where φ is an ECTL formula, and s is a state of a given Kripke structure M.

We assume that our proof system includes all axioms of boolean and propositional logic. Several of such axioms are shown in Figure 4. For example, in the one-point rule, also known as the $\exists$-introduction rule, f is a predicate and d is some element of D. Intuitively, the one-point rule states that to justify an existential statement $\exists x \in D \cdot f(x)$, one simply needs to exhibit an element $d \in D$ for which $f(d)$ holds. Note also we only consider quantification over finite domains.

In addition, we assume that all axioms of the theory of Kripke structures and the axiomatization of a particular Kripke structure M are available. The latter includes statements about M's transition relation R[1] and its labeling function I. For example, some of the axioms of the model in Figure 1(a) are: there is a transition between s_0 and s_1 ($R(s_0, s_1) = \top$); there is no transition between s_0 and s_3 ($R(s_0, s_3) = \bot$); p is true in s_1 ($I(s_1, p) = \top$), etc.

The proof rules for non-temporal operators and EX are shown in Figure 5. They follow directly from the definition of the corresponding operators. The EX-rule introduces an existential quantifier, which is typically eliminated by the one-point rule shown in Figure 4.

The proof rules for the bounded EU are given in Figure 6 and follow directly from the definition of this operator. To derive the rule for the unbounded EU, we use the monotonicity of EU_i:

Proposition 1 *Let φ, ψ be ECTL formulas and $i, j \in nat$. Then,*

$$i \geq j \Rightarrow \forall s \in S \cdot ([\![E[\varphi\ U_i\ \psi]]\!](s) \Leftarrow [\![E[\varphi\ U_j\ \psi]]\!](s))$$

Furthermore, since we assume that the state space is finite, the rule is actually bi-directional. That is, for a given Kripke structure M, there always exists a natural number n, which depends on the diameter of the directed graph induced by M, such that $E[\varphi\ U\ \psi] = E[\varphi\ U_n\ \psi]$. The proof rule for the unbounded EU is given in Figure 6.

To complete our proof system, we need to find a proof rule for EG. Unfortunately, we cannot proceed in the same manner as before and use the ECTL equivalence $EG\varphi = \varphi \wedge EXEG\varphi$. Doing so would result in a proof system which is not complete, since this proof rule can potentially be applied an infinite number of times.

Instead, note that $[\![EG\varphi]\!](s)$ is the result of evaluating $G\varphi$ on all infinite paths emanating from the state s. Moreover, since we are dealing with finite state systems, every infinite path can be decomposed into a finite (possibly empty) prefix and a finite repeating suffix. Thus, we can decompose $[\![EG\varphi]\!](s)$ into EG restricted to all non-trivial cycles around s, and EG restricted to all infinite paths that do not contain s in the future (see Figure 2).

First, we consider the restriction of $[\![EG\varphi]\!](s)$ to all non-trivial cycles around s. Essentially, this set consists of exactly

[1] We use notation $R(s,t) = \top$ and $R(s,t) = \bot$ to indicate that $(s,t) \in R$ and $(s,t) \notin R$, respectively.

$$\frac{\top}{[\![\top]\!](s)}\text{ value-rule} \qquad \frac{[\![\varphi]\!](s) \quad [\![\psi]\!](s)}{[\![\varphi \wedge \psi]\!](s)}\ \wedge\text{-rule}$$

$$\frac{I(s,p)}{[\![p]\!](s)}\text{ atomic-rule} \qquad \frac{[\![\varphi]\!](s) \vee [\![\psi]\!](s)}{[\![\varphi \vee \psi]\!](s)}\ \vee\text{-rule}$$

$$\frac{\neg I(s,p)}{[\![\neg p]\!](s)}\text{ neg-atomic-rule} \qquad \frac{\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t)}{[\![EX\varphi]\!](s)}\ EX$$

Fig. 5. Proof rules for non-temporal operators and EX.

$$\frac{[\![\psi]\!](s)}{[\![E[\varphi\ U_0\ \psi]]\!](s)}\ EU_0 \qquad \frac{\exists n \in nat \cdot [\![E[\varphi\ U_n\ \psi]]\!](s)}{[\![E[\varphi\ U\ \psi]]\!](s)}\ EU$$

$$\frac{[\![\psi \vee (\varphi \wedge EXE[\varphi\ U_{i-1}\ \psi])]\!](s)}{[\![E[\varphi\ U_i\ \psi]]\!](s)}\ EU_i$$

$$\frac{[\![(\varphi \wedge EXE[\varphi\ U\ \varphi \wedge \{s\}]) \vee (\varphi \wedge EXEG(\varphi \wedge \overline{\{s\}}))]\!](s)}{[\![EG\varphi]\!](s)}\ EG$$

Fig. 6. Proof rules for EU and EG.

those paths along which s occurs infinitely often. Furthermore, since our starting state is s, any infinite path along which s occurs infinitely often is equivalent to a finite path from s to itself. Thus, to evaluate $[\![EG\varphi]\!](s)$ restricted to cycles around s, it is sufficient to consider only finite paths from s to s. This intuition is formalized in the following theorem, the proof of which is available in [15]:

Theorem 1 *Let φ be an ECTL formula and s be a state of a Kripke structure. Then,*

$$[\![EG\varphi]\!](s) = [\![(\varphi \wedge EXE[\varphi\ U\ \varphi \wedge \{s\}]) \vee (\varphi \wedge EXEG(\varphi \wedge \overline{\{s\}}))]\!](s)$$

A proof rule for EG is given in Figure 6.

Theorem 2 *The proof system for ECTL is sound and complete.*

Proof:
The proof of soundness comes from the fact that our proof rules have been derived using definitions and equivalences between ECTL operators.

To prove completeness, we show that any valid statement of the form $[\![\varphi]\!](s)$ can be proven by a finite number of applications of our proof rules. The proof proceeds on the structure of the formula φ. A proof sketch follows:

(1) Let φ be a propositional temporal formula, that is, φ does not contain EU and EG. Each rule given in Figure 5 reduces φ to its subformulas. Thus, the rest of the proof for this case proceeds by induction on the number of subformulas of φ.

(2) If in addition to (1), φ can also contain EU_i, EU_i can be removed by expanding it using rules for EU_0 and EU_i given in Figure 6.

(3) If φ can contain bounded or unbounded EU, we note that for a given Kripke structure, there exists an equivalent formula in which all unbounded EU operators are replaced by their bounded versions, reducing the resulting formula to case (2) considered above.

(4) If φ can also contain EG, i.e., $\varphi \in$ ECTL, the EG-rule reduces a formula with EG to two formulas: (a) the one with EU, handled by the above case, and (b) a new formula containing EG, where the EG operator is restricted to a subset of the state space which does not contain the current state s. Therefore, this rule can only be applied up to $|S|$ times, ensuring that a valid statement $[\![\varphi]\!](s)$ can be proven by a finite number of applications of our proof rules. □

5.3 Automatic proof generation

Given a statement $[\![\varphi]\!](s)$, we are interested in an automated proof of its validity. We can achieve this by embedding the proof system of Section 5.2 into an automated theorem prover, such as PVS [20], and use its facilities for generating the proof. This may require backtracking because several proof rules may be applicable at a given point of the proof, generating irrelevant proof branches. We can avoid this problem and generate the proof much more efficiently if we use the model-checker as a decision procedure instead. We call this decision procedure `modelCheck` and assume that `modelCheck`(φ, s) computes $[\![\varphi]\!](s)$. In the remainder of this section, we examine proof generation of each connective that includes a choice ($\vee$) or an application of a one-point rule (EU and EX).

We start with the boolean connective $\vee$. Given a statement of the form $[\![\varphi \vee \psi]\!](s)$, we apply the $\vee$-rule:

$$\frac{[\![\varphi]\!](s) \vee [\![\psi]\!](s)}{[\![\varphi \vee \psi]\!](s)}\ \vee\text{-rule}$$

Next, we need to apply the $\vee$-introduction rule, using either φ or ψ. Since the choice depends on the validity of $[\![\varphi]\!](s)$ and $[\![\psi]\!](s)$, this suggests a simple proof strategy: if $[\![\varphi]\!](s)$ is valid, apply the $\vee$-introduction rule with φ; otherwise, if

(a)

proc `orRule`(φ, ψ, s)
  $k_\varphi :=$ `modelCheck`(φ, s)
  $k_\psi :=$ `modelCheck`(ψ, s)
  if k_φ **then**
    apply $\vee$-introduction with φ
  else if k_ψ **then**
    apply $\vee$-introduction with ψ
  else
    terminate with **invalid**
  end if
end proc

(b)

proc `euOnePoint`(φ, ψ, s, ℓ)
  $i := 0$
  $eu :=$ `modelCheck`$(E[\varphi\ U\ \psi], s)$
  $eui = \bot$
  while $eui \neq eu$ **do**
    $eui :=$ `modelCheck`$(E[\varphi\ U_i\ \psi], s)$
    $i := i + 1$
  end while
  if eui **then**
    apply the one-point rule substituting i for n
  else
    terminate with **invalid**
  end if
end proc

(c)

proc `exOnePoint`(φ, s, ℓ)
  $k :=$ `modelCheck`$(EX\varphi, s)$
  if not k **then**
    terminate with **invalid**
  end if
  $\hat{t} :=$ `exWitness`(φ, s)
  apply the one-point rule substituting $\hat{t}$ for t
end proc

Fig. 7. Algorithms for generating proof-like witnesses.

$[\![\psi]\!](s)$ is valid, apply the $\vee$-introduction rule with ψ; otherwise, terminate declaring that the statement is invalid. This proof strategy is implemented by an algorithm shown in Figure 7(a).

We now examine the case of the unbounded until (EU) operator. Given the statement $[\![E[\varphi\ U\ \psi]]\!](s)$, we first apply the EU-rule:

$$\frac{\exists n \in nat \cdot [\![E[\varphi\ U_n\ \psi]]\!](s)}{[\![E[\varphi\ U\ \psi]]\!](s)}\ EU$$

The next step is to find an instantiation of n for the one-point rule. Recall that the bounded EU_i is monotone when viewed as a function of i (by Proposition 1). Moreover, it is bounded above by the unbounded EU. Therefore, we can find the instantiation of n by a linear search, starting from $n = 0$. The algorithm for the application of the one-point rule is given in Figure 7(b). The intermediate computations performed by this algorithm are exactly the same as the ones done by a symbolic model-checking algorithm. Thus, if the results of the intermediate computations performed by `modelCheck`$(E[\varphi\ U\ \psi], s)$ are available, a more efficient binary search can be used instead of the linear one.

Finally, given a statement $[\![EX\varphi]\!](s)$, we start by applying the EX rule:

$$\frac{\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t)}{[\![EX\varphi]\!](s)}\ EX$$

Then, we need to eliminate the existential quantifier by applying the one-point rule. First, let us define a function *img* : $S \to \mathbb{B}$ as

$$img(x) \triangleq R(s,x) \wedge [\![\varphi]\!](x)$$

Clearly,

$$\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t) \Leftrightarrow \exists t \in S \cdot img(t) = \top$$

Thus, to apply the one-point rule, we need to find an element $\hat{t} \in img^{-1}(\top)$. Unfortunately, unlike the previous cases, we cannot obtain this element only by using the model-checker as a black box. Note, however, that if a path $s, \hat{t}$ is a witness to $[\![EX\varphi]\!](s)$, then the element $\hat{t} \in img^{-1}(\top)$. Thus, we assume the availability of a function `exWitness`(φ, s) which computes a witness to $[\![EX\varphi]\!](s)$. A version of this function can be easily obtained from any symbolic model-checker. The algorithm for the application of the one-point rule in the case of EX is given in Figure 7(c).

For example, the construction of the proof of the validity of $[\![EF(f \vee r)]\!](s_0)$ of the Button module, shown in Figure 8(a) (where some proof steps are skipped for clarity), proceeds as follows. First, the EF operator is expanded according to its definition into the EU operator, and the proof rule for EU is applied. This results in the subgoal

$$\exists n \cdot [\![E[\top\ U_n\ (f \vee r)]]\!](s_0)$$

The model-checker is then used to find the smallest bound on n, which is just the number of iterations required for a model-checking algorithm to converge, and in our example it is 2. Applying the EU_i rule twice, we reduce the proof to $[\![EXEX(f \vee r)]\!](s_0)$. After one application of the EX rule, we want to prove that $EX(f \vee r)$ holds in state s_1, which reduces to $\exists t \in S \cdot R(s,t) \wedge [\![EX(f \vee r)]\!](s_1)$. The model-checker is then used to find a successor s_2 of s_1 in which $f \vee r$ holds, allowing us to eliminate the existential quantifier (this step is skipped in Figure 8(a)). Finally, to prove $[\![f \vee r]\!](s_2)$, the model-checker is called once again to determine which of the two $\vee$-rules to apply.

Extension to ACTL. The ACTL subset of CTL is very similar to ECTL. The essential difference is that the EX operator in the fixpoint definition of the ECTL temporal operators is replaced by its dual AX. The proof rules are derived similarly to the ones for ECTL, and are summarized in Figure 9.

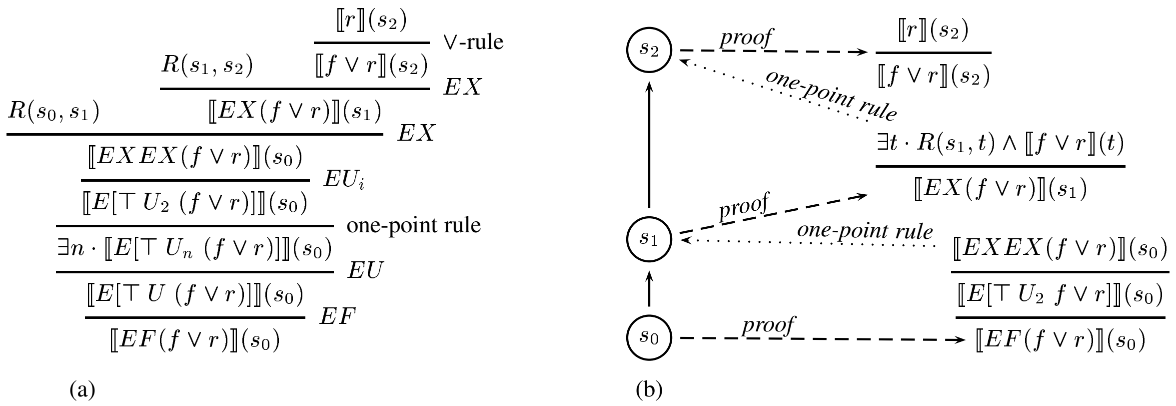

Fig. 8. (a) Proof of $[\![EF(f \vee r)]\!](s_0)$; (b) Proof-like witness of $[\![EF(f \vee r)]\!](s_0)$.

The AX-rule introduces universal quantification over the state space. Although the state-space is finite, the result is too large to present to the user explicitly and thus forms the basis of a proof obligation. Yet sometimes it is possible to reduce the complexity of such proof obligations. The application of the *universal case splitting* rule

$$\frac{\forall t \in S_1 \cdot R(s,t) \wedge [\![\varphi]\!](t) \qquad \forall t \in S \setminus S_1 \cdot \neg R(s,t)}{\forall t \in S \cdot R(s,t) \rightarrow [\![\varphi]\!](t)}$$

tells us that we can show that φ holds for all successors of s, and no other states are successors of s. If the set S_1 is relatively small, it is demonstrated to the user via the *finite quantification* rule, with $\bigcup_{i=1}^{n}\{t_i\} = S_1$:

$$\frac{R(s,t_1) \wedge [\![\varphi]\!](t_1) \qquad \ldots \qquad R(s,t_n) \wedge [\![\varphi]\!](t_n)}{\forall t \in S_1 \cdot R(s,t) \wedge [\![\varphi]\!](t)}$$

and the fact that the elements of $S \setminus S_1$ are not successors of s is retained as a proof obligation.

ACTL proofs that do not involve the use of AX, such as proving $[\![A[\varphi\ U\ \psi]]\!](s)$ when $[\![\psi]\!](s)$ holds, are generated automatically.

Extension to CTL. To extend the proof rules to full CTL, we need to extend the neg-atomic-rule to temporal operators. Using the well-known dualities between ECTL and ACTL operators [7], we obtain the negation rules summarized in Figure 10. Note that negation applied to the EU operator (neg-EU rule) is handled differently. Since any adequate set for CTL must contain the EU operator [18], it is not possible to express its negation as a combination of only ACTL operators.

6 Visualization Engine

The visualization engine converts the proof into a witness or a counterexample, and presents it to the user in a proof-like style. This presentation combines the advantages of both proofs and traditional counterexamples by highlighting the behavior of the model that is used to justify the result of the model-checker. This is achieved by extracting the set of model execution traces from the proof, and labeling each state of the trace with the part of the proof that depends on it.

There is a clear one-to-one correspondence between witnesses and proofs for CTL. Consider the proof for $[\![EX\varphi]\!](s)$ shown in Figure 11(a). Here, $\hat{t}$ is some state in S obtained by the model-checking procedure `exWitness`(φ, s) as part of the one-point rule for EX (see line 6 of the algorithm in Figure 7(c)). The witness can be visually obtained from the proof of EX via the following steps:

1. remove all nodes from the proof tree except for: (a) the root node, and (b) nodes that are the result of the application of the one-point rule to EX and that do not correspond to the axioms of the transition relation (see Figure 11(b));
2. replace horizontal bars by directed edges (see Figure 11(c));
3. relabel the root node by s and each node of the form $[\![\varphi]\!](\hat{t})$ by $\hat{t}$ (see Figure 11(d)).

The result of the application of the above procedure to the proof of $[\![EX\varphi]\!](s)$ is a path through the Kripke structure that is a witness to $[\![EX\varphi]\!](s)$. A similar procedure that uses finite quantification instead of the one-point rule also exists for AX.

Note that a sequence of steps in a Kripke structure is the only information given by the witness generators. All of such steps are results of the application of the one-point rule to EX or the finite quantification rule to AX. Other proof steps do not result in state transitions and are not shown by conventional witness generators. When extracting the witness from our proofs, we label each state in a witness with these missing proof steps. Branches in the witness result from applications of $\wedge$-intro and finite quantification rules. Loops can be identified by merging states with the same label, allowing the representation of nested loops. The resulting structures are called *proof-like witnesses* or *counterexamples*.

For example, a proof-like witness for the property $[\![EF(f \vee r)]\!](s_0)$ is shown in Figure 8(b). In this figure, the circled nodes represent states and solid arrows between them represent state transitions. This property is witnessed by a 3-state path s_0, s_1 and s_2. Each state is labeled with the part of the proof that directly depends on it. This is indicated by

$$\frac{\forall t \in S \cdot R(s,t) \rightarrow [\![\varphi]\!](t)}{[\![AX\varphi]\!](s)}\ AX \qquad \frac{[\![\psi]\!](s)}{[\![A[\varphi\ U_0\ \psi]]\!](s)}\ AU_0$$

$$\frac{[\![\varphi \wedge AXAG(\{s\} \vee \varphi)]\!](s)}{[\![AG\varphi]\!](s)}\ AG \qquad \frac{[\![\psi \vee (\varphi \wedge AXA[\varphi\ U_{i-1}\ \psi])]\!](s)}{[\![A[\varphi\ U_i\ \psi]]\!](s)}\ AU_i$$

$$\frac{\exists n \in nat \cdot [\![A[\varphi\ U_n\ \psi]]\!](s)}{[\![A[\varphi\ U\ \psi]]\!](s)}\ AU$$

Fig. 9. Proof rules for ACTL.

$$\frac{[\![AX\neg\varphi]\!](s)}{[\![\neg EX\varphi]\!](s)}\ \text{neg-}EX \qquad \frac{[\![A[\top\ U\ \neg\varphi]]\!](s)}{[\![\neg EG\varphi]\!](s)}\ \text{neg-}EG$$

$$\frac{[\![EX\neg\varphi]\!](s)}{[\![\neg AX\varphi]\!](s)}\ \text{neg-}AX \qquad \frac{[\![E[\top\ U\ \neg\varphi]]\!](s)}{[\![\neg AG\varphi]\!](s)}\ \text{neg-}AG$$

$$\frac{[\![(\neg\psi \wedge \neg\varphi) \vee (\neg\psi \wedge \neg EXE[\varphi \wedge \overline{\{s\}}\ U\ \psi \wedge \neg\overline{\{s\}}])]\!](s)}{[\![\neg E[\varphi\ U\ \psi]]\!](s)}\ \text{neg-}EU$$

$$\frac{[\![EG\neg\psi \vee E[\neg\varphi\ U\ \neg\varphi \wedge \neg\psi]]\!](s)}{[\![\neg A[\varphi\ U\ \psi]]\!](s)}\ \text{neg-}AU$$

Fig. 10. Proof rules for negation.

(a) $$\frac{\dfrac{R(s,\hat{t}) \qquad [\![\varphi]\!](\hat{t})}{\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t)}\ \text{one-point rule}}{[\![EX\varphi]\!](s)}\ EX$$

(b) $$\frac{[\![\varphi]\!](\hat{t})}{[\![EX\varphi]\!](s)}$$

(c) $$\begin{array}{c} [\![\varphi]\!](\hat{t}) \\ \uparrow \\ [\![EX\varphi]\!](s) \end{array}$$

(d) $$\begin{array}{c} \hat{t} \\ \uparrow \\ s \end{array}$$

Fig. 11. From proofs to witnesses.

the dashed *proof* arrows. Finally, each step in the proof that adds new states to the witness is labeled with the states it introduces, indicated by the dotted *one-point rule* (finite quantification rule) arrows. The proof in state s_0 tells us that a state in which $f \vee r$ holds is reachable in exactly two steps, since the EF operator is explained by an EU with the bound 2. In the last step of the proof in state s_0, the dotted arrow connecting the formula $EXEX(f \vee r)$ and the state s_1, tells us that s_1 is the witness for the outermost EX operator. The proof attached to state s_2 tells us that the formula $f \vee r$ holds in it because r is true.

As can be seen from this example, a proof-like witness is essentially a composition of two trees: the witness state tree, and the proof tree. This allows the user to either ignore the proof part and only explore the witness, or ignore the witness part and only explore the proof, or switch between the two representations at will. Moreover, the parts of the proof attached to each state can be seen as summaries that explain what is going to follow. For example, the proof attached to the state s_0 in Figure 8(b) can be summarized as "the next state is an intermediate one, and then we reach the desired state". Other types of summaries indicate whether a given state is part of a loop, which part of the property is being explained, etc.

The visual presentation of the result is controlled by the user through *visualization strategies*. A typical strategy is to restrict the scope of the explanation in order to bring forward its most useful parts. This is accomplished by specifying a starting and a stopping condition for the visualization. For example, to restrict the witness of the property $\varphi = EGEF(x \wedge EXx)$ to the EF operator, we set the starting and the stopping conditions to $EF(x \wedge EXx)$ and $x \wedge EXx$, respectively. In the proof-like witness in Figure 8(b), specifying that $f \vee r$ is the stopping condition removes the proof attached to the state s_2. If we let $f \vee r$ be the starting condition instead, s_2 would be the only displayed part of the witness.

A visualization strategy can also control how the state information is presented. For example, we can request to show all variables in each state, refer to each state by a unique name (as in Figure 8(b)), show only those variables that change between states, or always display some specific variables. Furthermore, the strategy can control the verbosity of the proof annotations, or completely replace the actual proof with a more suitable explanation. For example, we can replace the proof attached to the state s_0 with its (English) summary.

The result can be examined in a traditional forward fashion – starting from the initial state and proceeding in the direction of the trace execution to an error condition. Alternatively, the user can start the exploration at the error condition and use the proof annotations to move backwards along the trace. This corresponds to constructing the proof of the property from the basic axioms of the system.

The visualization engine that we presented in this section enables users familiar with model-checking to define strategies for counterexample generation and exploration. It also allows users who are comfortable with simple proofs to search through the counterexample effectively using the proof view. Yet, it is very simplistic – it is virtually a back-end visualizer. To be more useful, our visualization engine must be extended with additional visual cues, e.g., as suggested in [12, 11] (see Section 11).

7 Adding User-Specified Strategies

The ability of a user to understand why desired properties hold or fail in the model can be greatly enhanced if the user can control the kind of evidence that gets generated as part of the explanation. This approach also makes proof generation much more scalable: only the fragment of the proof that the user wants to see gets generated and displayed.

Consider the example in Figure 11. The presented witness goes through the state s_2 of the Kripke structure in Figure 1(a), whereas the user may have preferred it to go through s_3 instead. This is a *model-based* decision that comes from the fact that several states may satisfy $[\![EX\varphi]\!](s_1)$, for some property φ. The user can choose which of these (or whether all of these) are used in the proof.

The second decision type comes from explicit choices in *properties*, via a disjunction operator, e.g., $[\![EFp \vee EGr]\!](s_3)$. If both disjuncts are true, as in the model in Figure 1(a), the proof of which disjunct should be shown? Controlling this is especially useful during the *specification debugging* phase of the verification.

Typically, a proof proceeds by decomposing the top-level goal into simpler subgoals. For example, to prove $[\![p \wedge r]\!](s_2)$, we need to prove $[\![p]\!](s_2)$ and $[\![r]\!](s_2)$ separately. Yet, if the aim of generating the proof is *debugging*, we can often find the source of the error without expanding all of the subgoals. The choice of the order in which subgoals are to be expanded is the third type of decision that the user may want to make when generating proofs. We give the pseudocode of the proof generation in the method `buildProof`(), shown in Figure 12(a). The basic proof strategy, described in Section 4 and shown in Figure 12(c), makes all choices at random. Users can affect the proof generation by creating other strategies.

The simplest form of a strategy is to stop and ask the user to choose every time a decision needs to be made. Users can be aided in making decisions by proof summaries generated by the model-checker. For example, when choosing the part of the formula $[\![EFp \vee EGr]\!](s_3)$ to expand, the user may want to note that $[\![EFp]\!](s_3)$ converged in one iteration and $[\![EGr]\!](s_3)$ converged in two, and thus pick $[\![EFp]\!](s_3)$. Strategies can also be automated, with decisions based on summaries, observed history of the execution, or other user preferences. In this section, we discuss various types of strategies and their support in our framework.

7.1 Specifying Strategies

A user-specified strategy is created by implementing the `Strategy` interface shown in Figure 12(b). In particular, the strategy can modify the default proof system before the proof generation begins using `init`, determine which subgoal is to be expanded using `pickLeaf`, and determine which rule out of the applicable ones is to be applied using `pickRule`. Finally, after the application of any proof rule, the strategy can execute its own `ruleApplied` method. In addition, strategies have full access to the proof system: they can examine the current proof, add or remove proof rules, and examine the result of any rule application. Thus, they can affect the behaviour of the prover based on the current subgoal, proof rules that have already been applied, other historical information, subgoals yet to be proven, etc.

We now demonstrate how a few useful strategies can be specified in our framework.

Choosing the smallest subgoal. The goal of this strategy is to always pick a rule that results in a subgoal with the least number of temporal operators. For example, suppose our current subgoal is $[\![p \vee EFq]\!](s)$, and there are two applicable proof rules for disjunction (rules Q_1 and Q_2 in Figure 13). Clearly, applying rule Q_1 results in a shorter proof, and therefore a shorter witness. An implementation of this strategy is shown in Figure 12(d), and is accomplished by overriding `pickRule` to pick the rule that results in the new subgoal with the minimal number of temporal operators. The method `tryApply` allows the strategy to determine the new subgoal without modifying the proof tree. Note that this is a *greedy* strategy – choosing the subgoal with the shortest length does not guarantee the shortest witness or counterexample.

Preferring explored part of the model. This strategy attempts to guide the witness towards the part of the model that already appears in the proof. In general, a strategy can control which states of the model are used as part of a witness by introducing additional proof rules for the EX operator. For example, to ensure that all states of the witness satisfy a propositional constraint c, the strategy must add the proof rule EX_c (see Figure 13) during its initialization, and then ensure that this rule is always applied whenever possible.

The strategy `PickExplored`, shown in Figure 12(e), maintains a list of all states visited by the proof in the list N, adds

(a)

```
void buildProof (Strategy st)
  st.init ()
  repeat until  leaves ≠ ∅
    l = st.pickLeaf (leaves)
    r = st.pickRule (getRules (l), l)
    result = apply (r, l)
    st.ruleApplied (r, l, result)
  end repeat
```

(b)

```
class  Strategy
  void init ()
  Node pickLeaf (Set leaves)
  Rule pickRule (Set rules, Node l)
  void ruleApplied (Rule r, Node n, Node r)
```

(c)

```
class BasicStrategy extends Strategy
  Node pickLeaf (Set leaves)
    return randomElmnt(leaves)
  Rule pickRule (Set rules, Node l)
    return randomElmnt(rules)
```

(d)

```
class PickDisjunct extends Strategy
  Rule pickRule (Set rules, Node l)
    pick Q in rules s.t.
    size(tryApply(Q,l)) is minimal
```

(e)

```
class PickExplored extends Strategy
  void init ()
    N = s0
    addRule( ⟦EXφ ∧ N⟧(s) / ⟦EXφ⟧(s)  Q)
  Rule pickRule (Set rules, Node l)
    if Q ∈ rules then
      return Q
    end if
  void ruleApplied (Rule r, Node n, Node r)
     s =getState(r)
     N = N ∪ {s}
     update rule Q
```

(f)

```
class Sequence extends Strategy
  void init ()
    addRule( ⟦EXφ ∧ c1⟧(s) / ⟦EXφ⟧(s)  Q1)
    addRule( ⟦EXφ ∧ c2⟧(s) / ⟦EXφ⟧(s)  Q2)
    c1_state = true
  Rule pickRule (Set rules, Node l)
    if c1_state = true then
      if Q1 ∈ rules then
        c1_state = false
         return  Q1
       end if
     else
       if Q2 ∈ rules then
         c1_state = true
         return  Q2
       end if
     end if
```

Fig. 12. Proof strategies.

$$\frac{[\![p]\!](s)}{[\![p \vee EFq]\!](s)}\ Q_1 \qquad \frac{[\![EFq]\!](s)}{[\![p \vee EFq]\!](s)}\ Q_2 \qquad \frac{[\![EX\varphi \wedge c]\!](s)}{[\![EX\varphi]\!](s)}\ EX_c$$

$$\frac{B \Rightarrow p}{[\![p]\!](B)}\ \text{atomic-rule} \qquad \frac{\exists B_1, B_2 \cdot [\![\varphi]\!](B_1) \wedge [\![\psi]\!](B_2) \wedge (B \Rightarrow (B_1 \vee B_2))}{[\![\varphi \vee \psi]\!](B)}\ \vee\text{-rule} \qquad \frac{\exists B_1 \cdot \mathcal{R}(B, B_1) \wedge [\![\varphi]\!](B_1)}{[\![EX\varphi]\!](B)}\ EX$$

Fig. 13. Additional proof rules.

a new EX proof rule Q that prefers elements of N, and modifies `pickRules` so that Q is always picked when it is applicable. Finally, the strategy updates the list of visited states via the `ruleApplied` method. This is an example of a strategy that uses the proof history in order to augment its behavior.

Sequential constraint. The goal of this strategy is to ensure that states that satisfy some condition c_1 alternate with states that satisfy another condition c_2 on every path of the witness. As with the `PickExplored` strategy, it begins by adding new proof rules for the EX operator. Its `pickRule` method uses an additional boolean variable c_1`_state` to remember which of the two new rules was applied last, and augment its behavior based on that. In general, one can automatically generate such a strategy from a state machine that encodes a desired sequencing of constraints, e.g., as advocated in [2].

7.2 Discussion

The users of the framework do not have to interact with the proof engine explicitly. Instead, the interaction is based on the concepts that are already familiar to the engineers.

Some strategies are packaged for manual interaction. For example, the default implementation of the method `pickLeaf` allows the user to choose which part of the witness to extend by clicking on it. Some strategies are completely generic and serve as heuristics that are applicable to any model. For example, to ensure the shortest witness, we can combine strategies to pick the simplest subformula to explain, trying the

$$\cfrac{\mathcal{R}(s_0, s_1) \qquad \cfrac{\mathcal{R}(s_1, \{s_2, s_3\}) \qquad \cfrac{r \wedge \neg f \Rightarrow r}{[\![r]\!](\{s_2, s_3\})}}{[\![EXr]\!](s_1)}}{\cfrac{[\![EXEXr]\!](s_0)}{[\![EFr]\!](s_0)}}$$

Fig. 14. Proof of $[\![EFr]\!](s_0)$ for the model in Figure 1(a).

current state first when choosing the next state, guiding the witness through already visited states, etc. Some other strategies, e.g., the one that ensures that every path of the witness satisfies a given constraint, are parameterized. In this case, the user specifies the constraint, and the interaction with the proof engine is automated. For example, the user can provide the desired constraints in the form of a finite-state automaton, which is sufficient to generate code for the appropriate `Sequence` strategy that deduces which proof rules to add and when to apply them.

Typically, model-checkers implement some greedy strategy to generate a witness or a counterexample. However, users can specify efficient strategies that use backtracking. The complexity of these is controlled by restricting the number of applications of backtracking. For example, a strategy for generating a shortest witness for $[\![EXEFp]\!](s)$ can be specified to pick the successor of s from which EF has the shortest bound.

Strategies are also essential for producing *partial* witnesses when full witnesses are too large to be practical. For example, consider a witness to a property AFp which, in the worst case, can be of the size of the entire model. The strategy might be to expand only those paths for which the path to the state where p holds is at most x steps long. Since the size of the underlying proof is proportional to the number of steps in the witness, strategies ensure that usable proofs can be generated even for very large models.

8 Abstract Counterexamples

It is often convenient to see all counterexamples or witnesses to a given property at once [10]. For example, consider the property $[\![EFr]\!](s_0)$ evaluated on the Button module from Figure 1(a). There are two witnesses of length 3 that justify this property: leading to states s_2 and s_3, respectively, as shown in Figure 1(b). The information provided by these witnesses can be summarized using an *abstract witness* resulting from merging the states at the same depth. In Figure 1(b), these states are identified via dashed boxes. Each state in this abstract witness corresponds to one or more states of the model, and can be expressed by a propositional formula. In our example, we obtain that the first state of the witness must satisfy $\neg p \wedge \neg r \wedge \neg f$, whereas the second and the third state should satisfy $p \wedge \neg r \wedge \neg f$ and $r \wedge \neg f$, respectively. There is a disagreement on the value of p between states s_2 and s_3, and thus p is not part of the formula describing the third state.

Propositional formulas provide a very compact presentation of all of the witnesses at once, which in turn helps focus the attention of the user to the more relevant parts of the explanation. For example, by examining the constraint of the third state, we see that the value of p is irrelevant. In [10], it was shown that such a presentation can dramatically reduce the time required by the engineers to locate the real cause of an error.

In the rest of this section, we show how our framework can be used to generate abstract witnesses for reachability properties, or equivalently, abstract counterexamples for safety properties. Any reachability property can be expressed using a combination of EX and EU operators, and propositional connectives [7]. To construct a proof that captures all witnesses at once, we need to extend the corresponding proof-rules from single states to sets of states.

For notational convenience, we write $[\![\varphi]\!](B)$ to stand for $\forall s \in B \cdot [\![\varphi]\!](s)$, where φ is a temporal logic formula, and B is a set of states. Furthermore, we extend the transition relation R to sets of states and write $\mathcal{R}(B, C)$ to stand for $\forall b \in B \cdot \exists c \in C \cdot R(b, c)$. To prove that a propositional formula p holds in all states of a set B (written as $[\![p]\!](B)$), we need to show that B is a subset of the set of states defined by the formula p. That is, p is compatible with the propositional constraints imposed by B. The rule we obtain is called atomic-rule, shown in Figure 13. For example, the fact that r holds in the set of states $\{s_2, s_3\}$ of the Button module follows from the relation $(r \wedge \neg f) \Rightarrow r$. To prove that $\varphi \vee \psi$ holds in a set of states B, we need to show that there exists a partitioning of B into sets B_1 and B_2, such that φ holds in all elements of B_1, and ψ holds in all elements of B_2. The above is captured by the $\vee$-rule, shown in Figure 13. Note that user-specified strategies can influence the choice of this partitioning. For example, if a property φ is more complicated than ψ, the user may prefer B_1 to be empty, if possible.

To prove that $EX\varphi$ holds in a set of states B, we need to identify the successor states of each state in B and prove that φ holds in them (see the EX rule in Figure 13). In practice, the set B_1 can be easily computed from the intermediate results of a symbolic model-checker. For example, it can be instantiated to the set of all states that satisfy φ, and that are successors of states in B. Once again, the user can control the exact choice of B_1 using a proof-strategy, where picking the *largest* such set leads to an abstract witness capturing all possible witnesses.

Recall from Section 5 that proof-rules for the EU operator are derived by reducing it to a formula containing a disjunction, a conjunction, and an EX operator. Thus, it can be trivially extended to sets of states using the rules defined in this section.

A sample proof produced via the above proof rules for the property $[\![EFr]\!](s_0)$ evaluated in the Button module, is shown in Figure 14. This proof captures all 3-step witnesses for this property.

9 Tool Support

A prototype of the framework described in this paper has been implemented in a tool called KEGVis (Kounter-example generator and visualizer). The tool is built on top of a symbolic CTL model-checker χChek [4], but any other symbolic CTL model-checker can be used in the framework as a black-box as well. The tool provides several views of the evidence, including a high-level overview of the proof, and a detailed proof-like presentation of the resulting traces. KEGVis also implements a library of pre-specified exploration and navigation strategies and enables users to create their own. Our tool does not yet compute abstract counterexamples.

Figure 15 gives a screenshot of two views generated by the tool for the Button module described in Section 3.3. The front view shows the process of building a proof of the property EGp, described in the last example of Section 3.3. Here, the highlighted disjunction is true because the first disjunct holds, whereas the second fails to hold. Clicking on different parts of the proof in this view enables the interactive building of the proof.

At any point of the proof, users can generate the current snapshot of the proof-like witness. The back view of Figure 15 gives an example of a witness that corresponds to the proof shown in the front view. The white boxes in this view represent states of the model, and the shaded boxes contain summaries of the proof. We show an enlargement of the states of the model on the right-hand side of the view. Note that since the actual model of the elevator does not name states explicitly, the state names are generated automatically. Also, the shown configuration of the tool displays only those variables that change between states.

10 Related Work

The problem of generating and analyzing counterexamples for model-checking can be divided into three categories: generating the counterexample efficiently, obtaining a visual presentation suitable for interactive exploration, and automatically analyzing the counterexample to extract the exact source of the error.

The original counterexample generation algorithm, implemented in most symbolic model-checkers, was proposed by Clarke et al. [8] and implemented in NuSMV [6]. It produces linear counterexamples for a subset of universally- quantified CTL (ACTL), namely, ACTL $\cap$ LTL. This work was later extended to handle arbitrary ACTL properties [9] using the notion of *tree-like* counterexamples. However, navigating to "interesting" parts of the counterexamples, and thus understanding them, remains difficult.

The proof system developed in this paper is similar to the tableaux used for local model-checking [24]. In fact, our automated proof-generation technique can be seen as simulating a run of a local model-checker, where the information collected from the run of a global model-checker is used to guide the construction of the tableaux.

Several other researches have explored the idea of generating proofs from the model-checking runs [22,21,19,25]. Their common motivation is to use proofs to provide *complete* information justifying the result of the model-checker, in particular, in cases where a witness, or a counterexample, is not feasible.

The work of Namjoshi [19] concentrates on generating a proof of validity of a run of a global μ-calculus model-checker. This is accomplished by augmenting the model-checker to record the set of states satisfying each subformula and the convergence bound for each fixpoint operator. In cases where the formula has a finite witness, this information can also be used to produce proof-like witnesses, using techniques similar to the one described in Section 5.3. Tan and Cleaveland [25] extend Namjoshi's work to local model-checking.

An approach of Peled et. al. [22,21] is similar to ours in spirit. The main goal is to generate proofs that are as easy to understand as possible, and use them to communicate the reasons behind the model-checking result to the user. The technique developed in [22,21] generates proofs of satisfaction for LTL properties. Note that this is similar to providing witnesses for ACTL; thus, the approach is complimentary to ours, as it addresses the problem of automatically discharging proof obligations that we generate in the case of AX.

In this paper, we have only studied the use of proofs to aid in understanding witnesses; however, other applications of proofs are also possible. For example, Namjoshi [19] suggests that one can use proofs extracted from the model-checker to debug the model-checker itself, or use them as a basis for integration between model-checking and theorem proving. Since proof-like witnesses are effectively proofs, they can be used in any application that calls for a proof of satisfaction of a temporal property.

Visual *presentation* of generated counterexamples was also explored by Dong et al. [12,11]. The authors developed a tool that simplifies the counterexample exploration by presenting evidence through various graphical views. In particular, they found that one of the most important parts of the visualization process is highlighting the correspondence between the analyzed property and the generated counterexample.

The problem of the automatic analysis of counterexamples was addressed by many researchers but space limitations do not allow us to survey them here. Many of these techniques (e.g. [14,1]) are based on comparing all counterexamples to a safety property (i.e., a temporal property where a counterexample has a finite number of steps) to identify the common cause of the error. Barner et al. [2] concentrate on explaining reachability properties. Their main motivation is the observation that techniques used for model-checking can be used to generate *interesting* traces through the model. Thus, a model-checker can be turned into a design exploration tool. This is accomplished by letting the user interact with the witness generation engine of the model-checker. In particular, the user is allowed to provide a set of constraints that an interesting trace through the system must satisfy, request several disjoint witnesses, or witnesses of different length.

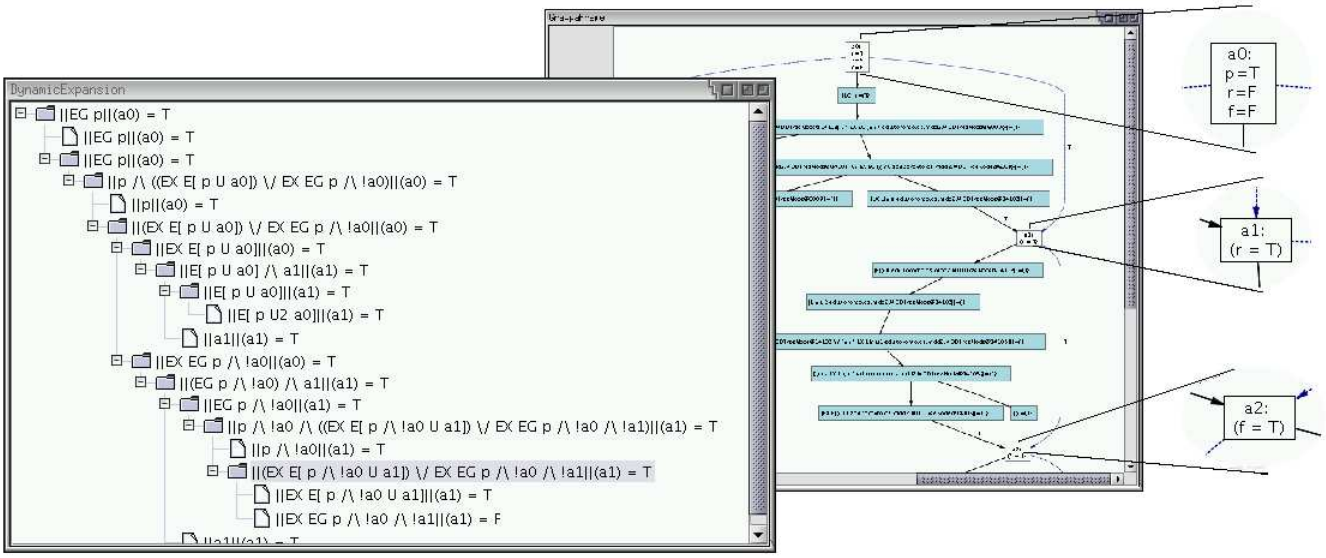

Fig. 15. A screenshot of KEGVis.

The goal of our work is to develop a unifying framework for combining various visualization and analysis techniques. In that, the work of Copty et al. [10] is the closest to ours. The authors report on a "counterexample wizard" – a tool for counterexample exploration for safety properties. The key idea of the approach is to compute and compactly store all counterexamples to a given property. Users can then visualize the result in various ways, replay several counterexamples in parallel, and apply different automatic analysis techniques.

11 Conclusion

In this paper, we presented a general framework for generating and exploring witnesses and counterexamples of temporal logic properties. The framework is based on building evidence in the form of a proof and controlling which portions of the proof are expanded and shown to the user either interactively or via user-specified strategies. Proofs also facilitate easy generation of conventional witnesses, which in our case are augmented with summaries describing which part of the property is being explained, e.g., whether a given state is part of a loop, how many steps separate a given state from the one in which a subproperty becomes true, etc. We have also created KEGVis – a prototype implementation of the framework.

We are currently looking at ways to connect exploration strategies with temporal logic property patterns [13]. Further, our preliminary experience with KEGVis indicates that users often make similar choices during their interactive exploration of witnesses. An automated strategy assistant that attempts to learn user preferences from previous interactions with the system and suggest an appropriate strategy would greatly enhance the potential usability of our framework. Finally, we are interested in how strategies can be used for understanding the impact of changing a model.

We view our current implementation as a back-end for a successful evidence exploration tool and, in its current form, it is by no means ready to be applied in an industrial setting. To enable such an application, the tool must become much more user-friendly. Most engineers find proofs too difficult, and, although proof-like witnesses bridge the gap between proofs and models, the concept of a proof is currently central to node summaries and some parts of the manual exploration.

For the sake of generality, our work has been on the level of the lowest common denominator of the interaction between the user and the model-checker. Namely, we assumed that the model of the system is given by a Kripke structure, and properties of interest are specified directly in temporal logic. This makes it possible to easily combine our approach with many of the existing model-checking tools. However, this also makes the actual technique appear more complex than it really is.

For example, in software model-checking, the user interacts with a model-checker by providing a source code of the program, and the model-checker automatically extracts a Kripke structure from it. Clearly, in this case, it is not helpful to explain the result of the model-checking run using states of this Kripke structure. Instead, such states should be converted back to what they are meant to represent, namely, line numbers of the program and values of relevant variables. Furthermore, sequences of states can be conveniently presented via interactive debug sessions. The proof part of the explanation is still useful in such cases: it can be used to annotate the debug trace, e.g., to explain why a particular branch of the program is taken next, or that the model-checker discovered a non-terminating loop in the program. Presentation of many of such proof aspects can also be tailored to a particular domain. For example, "an error in 3 steps" can become a graphical icon in the annotation of the trace.

Overall, we feel that the presented framework is flexible enough to enable creation of truly user-friendly tools that can facilitate effective model exploration and debugging using model-checking technology.

Acknowledgements

An earlier version of the material presented in this paper has appeared in TACAS'03 [16] and FASE'05 [5].

We thank Ben Wong, Kapil Shukla and Benet Devereux for their help in developing KEGVis. Financial support for this research has been provided by NSERC.

References

1. T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):49–58, November 2003.
2. S. Barner, S. Ben-David, A. Gringauze, B. Sterin, and Y. Wolfsthal. "An Algorithmic Approach to Design Exploration". In *Proceedings of Formal Methods Europe (FME'02)*, volume 2391 of *LNCS*, pages 146–162, Copenhagen, Denmark, July 2002. Springer-Verlag.
3. W. Chan. "Temporal-Logic Queries". In *Proceedings of the 12th Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 450–463, Chicago, IL, USA, July 2000. Springer.
4. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of 14th International Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 505–509, July 2002.
5. M. Chechik and A. Gurfinkel. "A Framework for Counterexample Generation and Exploration". In *Proceedings of Formal Aspects in Software Engineering (FASE'05)*, volume 3442 of *LNCS*, pages 217–233. Springer, April 2005.
6. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. "NUSMV: a new Symbolic Model Verifier". In N. Halbwachs and D. Peled, editors, *Proceedings of 11th Conference on Computer-Aided Verification (CAV'99)*, volume 1633 of *LNCS*, pages 495–499, Trento, Italy, July 1999. Springer.
7. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
8. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking". In *Proceedings of 32nd Design Automation Conference (DAC 95)*, pages 427–432, San Francisco, CA, USA, 1995.
9. E.M. Clarke, Y. Lu, S. Jha, and H. Veith. Tree-Like Counterexamples in Model Checking. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 19–29, Copenhagen, Denmark, July 2002. IEEE Computer Society.
10. F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. "Efficient Debugging in a Formal Verification Environment". In *Proceedings of 11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume 2144 of *LNCS*, pages 275–292, Scotland, UK, September 2001. Springer-Verlag.
11. Y. Dong, C.R. Ramakrishnan, and S. A. Smolka. "Evidence Explorer: A Tool for Exploring Model-Checking Proofs". In *Proceedings of the 15th International Conference on Computer-Aided Verification (CAV'03)*, volume 2725 of *LNCS*, pages 215–218, 2003.
12. Y. Dong, C.R. Ramakrishnan, and S. A. Smolka. "Model Checking and Evidence Exploration". In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS'03)*, pages 214–223, Alabama, USA, April 2003.
13. M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of 21st International Conference on Software Engineering*, May 1999.
14. A. Groce and W. Visser. "What Went Wrong: Explaining Counterexamples". In *Proceedings of SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
15. A. Gurfinkel. "Multi-Valued Symbolic Model-Checking: Fairness, Counter-Examples, Running Time". Master's thesis, University of Toronto, Department of Computer Science, October 2002.
16. A. Gurfinkel and M. Chechik. "Proof-like Counterexamples". In *Proceedings of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *Lecture Notes in Computer Science*, pages 160–175, April 2003.
17. A. Gurfinkel, M. Chechik, and B. Devereux. "Temporal Logic Query Checking: A Tool for Model Exploration". *IEEE Transactions on Software Engineering*, 29(10):898–914, October 2003.
18. F. Laroussinie. "About the Expressive Power of CTL Combinators". *Information Processing Letters*, 54:343–345, 1995.
19. K. Namjoshi. "Certifying Model Checkers". In *Proceedings of 13th International Conference on Computer-Aided Verification (CAV'01)*, volume 2102 of *LNCS*. Springer-Verlag, 2001.
20. S. Owre, N. Shankar, and J. Rushby. "User Guide for the PVS Specification and Verification System (Draft)". Technical report, Computer Science Lab, SRI International, Menlo Park, CA, 1993.
21. D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *Proceedings of 21st Conference on the Foundations of Software Technology and Theoretical Computer Science (FST&TCS'01)*, volume 2245 of *LNCS*, pages 292–304. Springer-Verlag, 2001.
22. D. Peled and L. Zuck. From model checking to a temporal proof. In *Proceedings of the 8th International SPIN Workshop (SPIN'2001)*, volume 2057 of *LNCS*, pages 1–14, Toronto, Canada, May 2001. Springer.
23. M.C. Plath and M.D. Ryan. "SFI: A Feature Integration Tool". In R. Berghammer and Y. Lakhnech, editors, *Tool Support for System Specification, Development and Verification*, Advances in Computer Science, pages 201–216. Springer, 1999.
24. C. Stirling and D. Walker. Local model-checking in the modal mu-calculus. *Theoretical Computer Science*, 89, 1991.
25. L. Tan and R. Cleaveland. Evidence-Based Model Checking. In *Proceedings of 14th Conference on Computer-Aided Verification (CAV'02)*, volume 2404 of *LNCS*, pages 455–470, Copenhagen, Denmark, July 2002. Springer-Verlag.