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: {chechik,arie}@cs.toronto.edu

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 an automated verification technique that receives a finite-state description of a system and a temporal logic property and decides whether the property holds in the system. Model-checking is rapidly becoming an accepted technique for analyzing software and hardware 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, fix the model or the property. Copty et al. [7] 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.

Counterexamples can also be used for *design exploration* [2]. A model-checker enables the user to specify scenarios of interest without specifying the exact input sequences leading to them, and can also 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 traces while checking the property.

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*.

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 an exponential algorithm that often takes hours to run even for moderately-sized models [7]. 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 evidence, and thus is costly as well. It is possible for model-checkers to generate too much evidence [9], 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 evidence [9], and users fail to notice them. It is therefore desirable to have control over just how much evidence is generated by the model-checker. This can be accomplished via *interactive explanations* – evidence generation based on user preferences. Interactive explanations can allow users to put a bound on the time that the model-checker spends computing the evidence, and continue exploring it manually; control which option is used to facilitate the generation of "interesting" evidence; 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 makes the problem of generating and understanding evidence tractable:

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

Contributions of this paper. In this paper, we propose a framework for 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 [12] without sacrificing any of the intuitiveness and close relation to the model that users have learned to expect from model-checkers. Basing the evidence 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. Application of strategies for dynamic proof generation is the major technical contribution of this paper, when compared to our previous work reported in [12].

In this paper, we *illustrate* the framework using a simple example rather than *validating* its effectiveness via a sizable case study. Here, we draw on industrial experience [7] 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.

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.[5], and was later extended to handle arbitrary universal properties [6, 12], i.e., properties that quantify over all paths of the model. An alternative approach was independently suggested by Namjoshi [13] and Tan and Cleaveland [16] with the goal to extend the counterexample generation technique to all (as opposed to just universal) branching temporal properties. The proposed methods identify what information must be stored from the intermediate run of the model-checker to reconstruct the proof of correctness of the result. A similar technique for linear properties was explored by Peled et al. [14].

The problem of the visual *presentation* of generated counterexamples was addressed by Dong et al. [9, 8]. 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. [11, 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.

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. [7] 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.

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. In Section 5, we enrich the framework with additional proof strategies that allow the user to control which counterexample gets generated. In Section 6, we discuss how to use our framework to generate several counterexamples at once. We conclude in Section 7 with the summary of the paper and discussion of future research directions. Appendix A describes KEGVis – a prototype implementation of our framework.

2 CTL Model-Checking

Model-checking is an automated verification technique that receives a system K and a temporal logic property φ and decides whether φ holds in K. In this paper, we assume that K 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 2(a).

We specify properties in *Computation Tree Logic* (CTL) [4], defined below:

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

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 2(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 $[\![\varphi]\!]^K(s)$ to indicate the value of φ in the state s of K, and $[\![\varphi]\!](s)$ when K is clear from the context. A formula φ is satisfied in a Kripke structure K if and only if it is satisfied in its initial state. The operators EX, EG, and EU form an adequate set,

$$AX\varphi \triangleq \neg EX\neg\varphi \qquad AF\varphi \triangleq A[\mathsf{true}\ U\ \varphi] \qquad E[\varphi U_0 \psi] \triangleq \psi$$
$$EF\varphi \triangleq E[\mathsf{true}\ U\ \varphi] \quad AG\varphi \triangleq \neg EF\neg\varphi \qquad E[\varphi U_i \psi] \triangleq \psi \vee (\varphi \wedge EXE[\varphi U_{i-1}\ \psi])$$
$$A[\varphi\ U\ \psi] \triangleq \neg E[\neg\psi\ U\ \neg\varphi \wedge \neg\psi]\ \wedge\ \neg EG\neg\psi$$

Fig. 1. Definitions of CTL operators.

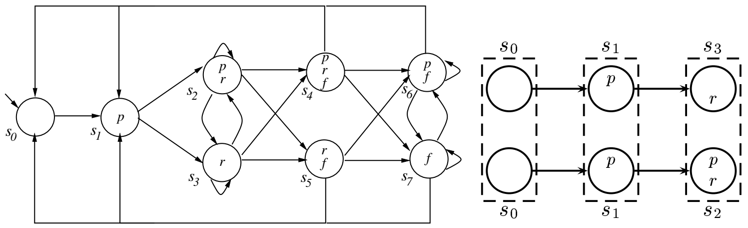


Fig. 2. (a) State machine for the module Button; (b) Two witnesses of length 3 for $[\![EFr]\!](s_0)$ for this statemachine.

i.e. all other operators can be defined from them, as shown in Figure 1. Semantics EX, EG and EU is formally in defined as follows:

$[\![EX\varphi]\!](s)$ **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, as shown in Figure 1.

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.

We use the model specified in SMV by Plath & Ryan [15]. 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 2(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 2(a), we only show true variables; thus, in state s_0, p, r, and f are false.

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

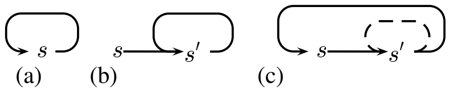


Fig. 3. Possible witnesses for EGp in state s: (a) a looping witness, (b) a path followed by a loop, and (c) two witnesses combining cases (a) and (b).

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: $EF f$, 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 $AG(r \Rightarrow AF f)$. 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 $\mathrm{notMoving}$, 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 [5], 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 3(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 3(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 3(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 2(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.

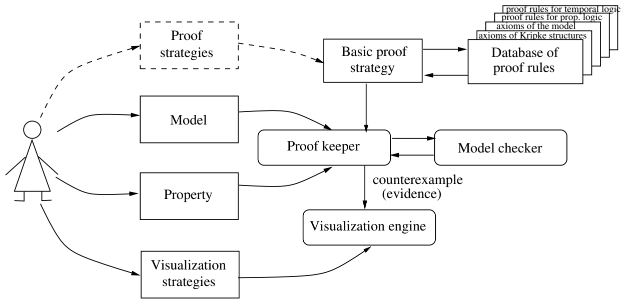


Fig. 4. Overview of the framework.

$$\frac{a \quad b}{a \wedge b}\ \wedge\text{-rule} \qquad \frac{\exists t \in S \cdot R(s,t) \wedge [\![\varphi]\!](t)}{[\![EX\varphi]\!](s)}\ EX \qquad \frac{[\![\psi]\!](s)}{[\![E[\varphi\ U_0\ \psi]]\!](s)}\ EU_0$$

$$\frac{a}{a \vee b}\ \vee\text{-rule} \qquad \frac{\exists n \cdot [\![E[\varphi\ U_n\ \psi]]\!](s)}{[\![E[\varphi\ U\ \psi]]\!](s)}\ EU \qquad \frac{f(d)}{\exists x \in D \cdot f(x)}\ \text{one-point rule}$$

$$\frac{b}{a \vee b}\ \vee\text{-rule} \qquad \frac{[\![\psi \vee (\varphi \wedge EXE[\varphi\ U_{n-1}\ \psi])]\!](s)}{[\![E[\varphi\ U_n\ \psi]]\!](s)}\ EU_i$$

Fig. 5. Some CTL proof rules.

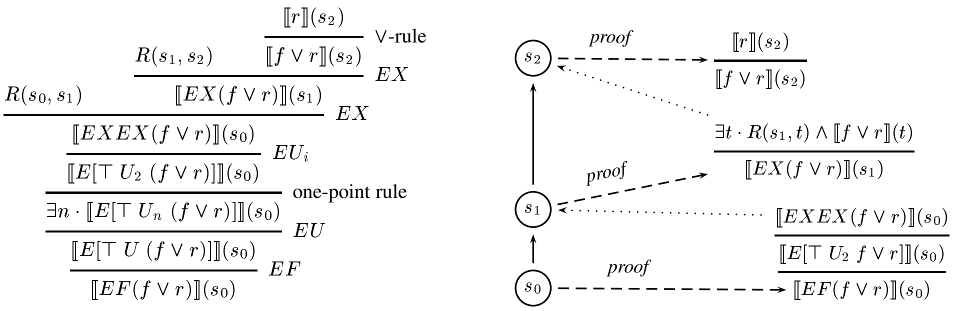
4 Framework

The framework for generating and visualizing counterexamples is shown in Figure 4. Dashed lines indicate optional inputs. The user interaction with the framework starts by providing a proof keeper with a model K 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. In this step, it uses the model-checker to decide which proof rules are applicable and to ensure the soundness of the proof. (Additional runs of the model-checker can be avoided by efficiently computing and storing evidence, as discussed in [16, 13, 12].) 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. In the rest of this section we discuss the framework in more detail. The framework augmented with additional proof strategies is described in Section 5.

4.1 Proof Rules

Several proof rules from the CTL proof system are given in Figure 5. These include all proof rules of the propositional logic that deal with negation, disjunction and conjunction, such as the $\wedge$-, $\vee$-rules, i.e., to prove $a \wedge b$, we need to prove a and b separately, and to prove $a \vee b$, we need to prove either a or b. Additionally, our proof system uses the axiomatization of the given Kripke structure K, describing its transition relation R and values of each atomic proposition in each state. For example, some of the axioms of the model in Figure 2(a) are: there is a transition between s_0 and s_1 ($R(s_0, s_1)$); there is no transition between s_0 and s_3 ($\neg R(s_0, s_3)$); p is true in s_1 ($I(s_1, p)$), etc.

The proof system is then extended with proof rules for each temporal operator. In this paper, we only show proof rules for EX and EU, and refer the reader to [12] for a



(a) (b)

Fig. 6. (a) Proof of $[\![EF(f \vee r)]\!](s_0)$; (b) Proof-like witness of $[\![EF(f \vee r)]\!](s_0)$.

complete description of the proof system for CTL and for results on its soundness and completeness. The proof rule for the EX operator follows directly from its definition, i.e., to prove $EX\varphi$ at a state s, we need to find a state t which is a successor of s and in which φ holds. Note that this proof rule introduces an existential quantifier, which is later eliminated by the application of the one-point rule. The proof rule for the EU operator follows from the fact that our models are finite. Thus, any path witnessing an EU formula has a bounded length. Finally, the proof rules for the bounded EU operator simply unroll it according to the bound, using the definitions of EU_0 and EU_i given in Section 2.

4.2 Generating Proofs

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 model-checker chooses those for which the valid proof can be constructed; (3) the rule to be applied is randomly chosen from the resulting set. In [12], we give more detail on the use of the model-checker to guide an automatic proof construction and show that the above strategy is terminating for CTL.

For example, the construction of the proof of the validity of $[\![EF(f \vee r)]\!](s_0)$ of the Button module, shown in Figure 6(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 6(a)). Finally, to prove $[\![f \vee r]\!](s_2)$, the model-checker is called once again to determine which of the two ∨-rules to apply.

4.3 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 of [12]. The proof-like 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. For example, a proof-like witness for the property $[\![EF(f \vee r)]\!](s_0)$ is shown in Figure 6(b). This property is witnessed by a 3-state path s_0, s_1 and s_2. 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.

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 6(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 6(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 6(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 useful, our visualization engine must be extended with additional visual cues, e.g., as suggested in [9, 8] (see Section 7).

5 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 6. The presented witness goes through the state s_2 of the Kripke structure in Figure 2(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 $[\![EX\varphi]\!](s_0)$, 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 2(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 `buidProof()`, shown in Figure 7(a). The basic proof strategy, described in Section 4 and shown in Figure 7(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.

5.1 Specifying Strategies

A user-specified strategy is created by implementing the `Strategy` interface shown in Figure 7(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 (see rules Q_1 Figure 8(a),(b)). Clearly, applying rule Q_1 results in a shorter proof, and therefore a shorter witness. An implementation of this strategy is shown in

(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
```

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

(e)
```
class PickExplored extends Strategy
  void init ()
    N = s_0
    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
```

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

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

(f)
```
class Sequence extends Strategy
  void init ()
    addRule( [[EXφ ∧ c_1]](s) / [[EXφ]](s)  Q_1)
    addRule( [[EXφ ∧ c_2]](s) / [[EXφ]](s)  Q_2)
    c_1_state = true
  Rule pickRule (Set rules, Node l)
    if c_1_state = true then
      if Q_1 ∈ rules then
        c_1_state = false
        return  Q_1
      end if
    else
      if Q_2 ∈ rules then
        c_1_state = true
        return  Q_2
      end if
    end if
```

Fig. 7. Proof strategies.

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

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

Fig. 8. Additional proof rules.

Figure 7(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 8) during its initialization, and then ensure that this rule is always applied whenever possible.

The strategy `PickExplored`, shown in Figure 7(e), maintains a list of all states visited by the proof in the list N, adds 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., one advocated in [2].

5.2 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 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.

6 Abstract Counterexamples

It is often convenient to see all counterexamples or witnesses to a given property at once [7]. For example, consider the property $[\![EFr]\!](s_0)$ evaluated on the Button module from Figure 2(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 2(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 2(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

$$\dfrac{\dfrac{\mathcal{R}(s_0, s_1) \qquad \dfrac{\mathcal{R}(s_1, \{s_2, s_3\}) \qquad \dfrac{r \wedge \neg f \Rightarrow r}{\llbracket r \rrbracket(\{s_2, s_3\})}}{\llbracket EXr \rrbracket(s_1)}}{\llbracket EXEXr \rrbracket(s_0)}}{\llbracket EFr \rrbracket(s_0)}$$

Fig. 9. Proof of $\llbracket EFr \rrbracket(s_0)$ for the model in Figure 2(a).

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 [7], 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 that 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 [4]. 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 $\llbracket \varphi \rrbracket(B)$ to stand for $\forall s \in B \cdot \llbracket \varphi \rrbracket(s)$, 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 $\llbracket p \rrbracket(B)$), 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. Formally, we obtain the atomic-rule, shown in Figure 8. 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 8. 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 8). 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 4.1 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 $\llbracket EFr \rrbracket(s_0)$ evaluated in the Button module, is shown in Figure 9. This proof captures all 3-step witnesses for this property.

7 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, 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 [10]. 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.

References

1. T. Ball, A. Podelski, and S. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs". *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 FME'02*, volume 2391 of *LNCS*, pages 146–162. Springer-Verlag, July 2002.
3. M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 505–509, July 2002.
4. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
5. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking". In *Proceedings of DAC 95*, pages 427–432, 1995.
6. E.M. Clarke, Y. Lu, S. Jha, and H. Veith. Tree-Like Counterexamples in Model Checking. In *Proceedings of LICS'02*, pages 19–29, July 2002.
7. F. Copty, A. Irron, O. Weissberg, N. Kropp, and G. Kamhi. "Efficient Debugging in a Formal Verification Environment". In *Proceedings of CHARME'01*, volume 2144 of *LNCS*, pages 275–292. Springer-Verlag, September 2001.
8. Y. Dong, C.R. Ramakrishnan, and S. A. Smolka. "Evidence Explorer: A Tool for Exploring Model-Checking Proofs". In *Proceedings of CAV'03*, volume 2725 of *LNCS*, pages 215–218, 2003.
9. Y. Dong, C.R. Ramakrishnan, and S. A. Smolka. "Model Checking and Evidence Exploration". In *Proceedings of ECBS'03*, pages 214–223, April 2003.
10. M. Dwyer, G. Avrunin, and J. Corbett. "Patterns in Property Specifications for Finite-State Verification". In *Proceedings of ICSE'99*, May 1999.
11. A. Groce and W. Visser. "What Went Wrong: Explaining Counterexamples". In *Proceedings of SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.
12. A. Gurfinkel and M. Chechik. "Proof-like Counterexamples". In *Proceedings of TACAS'03*, LNCS, April 2003.
13. K. Namjoshi. Certifying Model Checkers. In *Proceedings of CAV'01*, volume 2102 of *LNCS*. Springer-Verlag, 2001.
14. D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *FST&TCS*, volume 2245 of *LNCS*. Springer-Verlag, 2001.
15. 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.
16. L. Tan and R. Cleaveland. Evidence-Based Model Checking. In *Proceedings of CAV'02*, volume 2404 of *LNCS*, pages 455–470. Springer-Verlag, July 2002.

A 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 [3], 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. Abstraction counterexample generation is currently not supported in our tool.

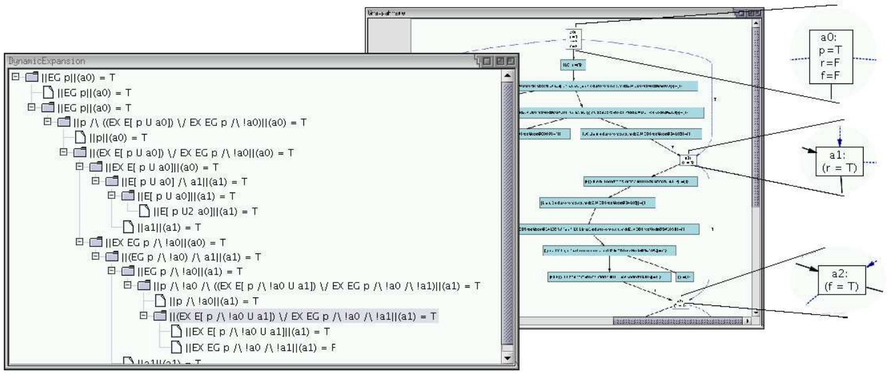


Fig. 10. A screenshot of KEGVis.

Figure 10 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. For example, 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 in Figure 10 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.