Incremental Construction of Counterexamples in Model Checking Web Documents

Franz Weitl and Shin Nakajima
National Institute of Informatics (NII)
Tokyo, Japan
Email: {weitl,nkjm}@nii.ac.jp

Abstract

A new algorithm for incrementally generating counterexamples for the temporal description logic *ALC*CTL is presented. *ALC*CTL is a decidable combination of the description logic *ALC* and computation tree logic CTL that is expressive for content- and structure-related properties of web documents being verified by model checking. In the case of a specification violation, existing model checkers provide a single counterexample which may be large and complex. We extend existing algorithms for generating counterexamples in two ways. First, a coarse counterexample is generated initially that can be refined subsequently to the desired level of detail in an incremental manner. Second, the user can choose where and in which way a counterexample is refined. This enables the interactive step-by-step analysis of error scenarios according to the user's interest.

We demonstrate in a case study on a web-based training document that the proposed approach reveals more errors and explains the cause of errors more precisely than the counterexamples of existing model checkers. In addition, we demonstrate that the proposed algorithm is sufficiently fast to enable smooth interaction even in the case of large documents.

1 Introduction

Model checking is a powerful technique for automatically detecting errors in hard- and software design artifacts that has also been applied to verify business processes [19], web services [15], and web documents [26]. A remaining problem of model checking is its limited usability for non-experts. In the domain of document management, formal verification methods cannot be applied without appropriate user support. In previous work, we proposed means of supporting the user in model generation [23, 22] and property specification [21]. In this paper, we introduce the formal structures and algorithms to support incremental and interactive analysis of model checking results.

A model checker determines if a given finite state transition system M is a model of a temporal formula p. If M violates p, a counterexample is provided. An ideal counterexample should take a form that demonstrates in a *complete* yet *concise and comprehensible* way [6] why M violates p. In addition, it should precisely isolate those parts of the model M and the formula p that contribute to a violation. Counterexamples provided by current state-of-the-art model checkers such as NuSMV [5] or SAL [8] consist of finite paths in the state transition system M. These "linear" counterexamples are, in general, not complete, i.e., they may demonstrate the cause of the property violation just partially [7]. Even so, they tend to be large and difficult to understand [4, 9]. These problems become worse if temporal formulae contain first order predicates and quantified variables as required, for instance, to express properties for web services [15] and documents [25]. To address the problems of incomplete and hard to understand counterexamples, various extensions to linear counterexamples have been proposed [6, 24, 4] but quantified expressions have not been considered.

We propose a new algorithm for generating counterexamples for first order quantified temporal properties expressible in the temporal description logic *ALC*CTL [25], a decidable combination of the description logic *ALC* [2] and CTL [10]. *ALC*CTL has been applied for verifying properties of web documents [26] and technical manuals [21, 23]. The proposed algorithm builds upon the concept of *evidence tree* introduced in [27]. It extends the algorithm presented in [27] in the following two aspects:

1. Support of *incremental* generation of evidence trees. A *coarse* counterexample is provided initially which can be *refined* step-by-step to the desired level of detail. This prevents overwhelming the user with details of complex error scenarios and increases the responsiveness of the system.

2. Support of user *interaction*. The user may choose where and in which way a given counterexample is refined. This supports the successive exploration of different error scenarios according to the user's interest, as opposed to existing model checkers which provide just a single, arbitrarily chosen counterexample.

In addition, we extend the case study in [27] as follows: 1) we demonstrate that the *response time* of the algorithm is sufficiently low to provide smooth user interaction even for web documents with several thousands of web pages; 2) we summarize the results of a new study on the *scalability* of the approach for documents up to 4000 pages.

The rest of the paper is organized as follows: first, the research issues of this paper are presented. After that, some technical preliminaries on *ALC*CTL and model checking are summarized, followed by the description of the proposed structures and algorithms for counterexample generation. In the sequel, experimental results are presented before discussing related work and concluding the paper with a brief summary and outlook.

2 Issues on Counterexample Generation

Example 1 (*Linear Counterexample*)

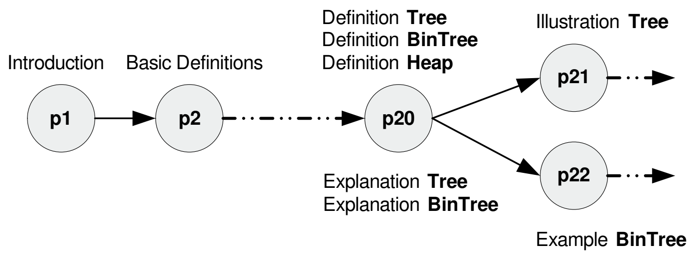


Figure 1: Web document on data structures (dotted arrows indicate omitted pages)

As an example, let us consider a web-based learning document. It deals with basic data structures such as binary trees and heaps that are presented in terms of a web document hypertext (Figure 1). For simplicity, we assume that the *web pages* of the document form the finite set of *states* S and *hyperlinks* between web pages form the set of *transitions* $T \subseteq S \times S$ of the Kripke structure M to be checked.

The document contains formal definitions, explanations, illustrations, and examples of the basic terminology. For instance in Figure 1), page **p20** contains definitions of the terms "Tree", "Binary Tree", and "Heap", as well as the explanations for "Tree" and "Binary Tree". After that, the reader can either see an illustration of "Tree" on page 21 or an example of "Binary Tree" on page 22 (**p21** and **p22** in Figure 1). Let us assume that each defined term needs to be explained on the same page and illustrated by a pertinent example on one of the next pages. This property can be represented in *ALC*CTL as

$$\mathsf{AG}(\mathit{defined} \sqsubseteq \mathit{explained} \sqcap \mathsf{EX}\ \mathit{exemplified}) \tag{1}$$

which is equivalent to the quantified CTL formula

$$\mathsf{AG}(\forall t \in \mathit{Term} : \mathit{defined}(t) \rightarrow \mathit{explained}(t) \wedge \mathsf{EX}\ \mathit{exemplified}(t)) \tag{2}$$

"On all paths it generally holds (AG) for each term t ($\forall t \in \mathit{Term}:$) that if t is defined ($\mathit{defined}(t)$) then it is explained ($\rightarrow \mathit{explained}(t)$) and it is exemplified on a certain next page ($\wedge \mathsf{EX}\ \mathit{exemplified}(t)$)".

If the set *Term* in formula (2) is finite, the formula can be reduced to propositional CTL [10] and verified by model checkers such as NuSMV [5]. Actually, the property is not satisfied in Figure 1 because, for instance, there is no explanation for "Heap" defined on page **p20**. Counterexamples provided by the current model checkers contain a trace $(p1, p2, ..., p20)$ from the initial page **p1** to page **p20**. □

Although such a trace often becomes long, it does not give much information on why the property is violated. The following questions arise:

Q1) *Where* on a counterexample trace is the property violated? As for Example 1, the property is violated on page **p20**, but also pages **p21** and **p22** may be involved in certain error scenarios.

Q2) *Which objects* violate the property? As for Example 1, the property is violated for terms "Tree" and "Heap" but not for term "Binary Tree".

Q3) *Why* is the property violated? As for Example 1, more than one reason can be suggested. The property is violated for term "Tree" on page **p20**, because none of the next pages **p21** and **p22** contains an example of "Tree". The property is violated by term "Heap", because no explanation is given for "Heap" on page **p20** or, alternatively, because none of the next pages **p21** and **p22** contains an example of "Heap".

Questions Q1) and Q2) refer to two dimensions of error *localization*. For answering Q1), the states in the state transition system M, which are involved in a specification violation, are determined. Q2) corresponds to bindings of the quantified variable t in the formula (2) that invalidate the formula w.r.t. a given model M. In previous approaches on counterexamples, quantified variables have not been considered which leaves this important dimension of error localization unexploited. To answer Q2), we extend counterexamples towards subset expressions of *ALC*CTL.

Question Q3) refers to *explaining* why a property is violated. We observe, that even in simple scenarios the cause of a property violation can be diverse and complex. Complete explanations of property violations may be time consuming to generate and may overwhelm the user with detail. To address these issues, we 1) provide differently detailed views on an error scenario by structuring counterexamples as trees, and 2) support the incremental refinement of counterexample trees down to the desired level of detail in interaction with the user.

3 The Temporal Description Logic *ALC*CTL

$$p,q \rightarrow C \sqsubseteq D \mid \neg p \mid p \wedge q \mid \mathsf{EX}\ p \mid \mathsf{AF}\ p \mid \mathsf{E}(p\ \mathsf{U}\ q)$$
$$C,D \rightarrow A \mid \neg C \mid C \sqcap D \mid \exists R.C \mid \mathsf{EX}\ C \mid \mathsf{AF}\ C \mid \mathsf{E}(C\ \mathsf{U}\ D)$$

Table 1: *ALC*CTL syntax definition

Table 1 shows the syntax definition of a base of *ALC*CTL connectives over the symbols $\mathscr{C} \cup \mathscr{R}$ where $\mathscr{C}$ is a set of unary predicates (*atomic concepts*) representing sets and $\mathscr{R}$ is a set of binary predicates (*atomic roles*) representing relations. In Formula (1), *defined*, *explained*, and *exemplified* are atomic concepts. Basic *ALC*CTL formulae are of type $C \sqsubseteq D$ (C is a subset of D) where C and D are concept expressions. According to the second row of Table 1, concepts can be formed by *ALC* connectives such as A ($A \in \mathscr{C}$ atomic concept), $\neg C$ (complement), $C \sqcap D$ (intersection), and $\exists R.C$ (quantified role $R \in \mathscr{R}$). In addition, CTL temporal connectives can be applied to form "temporal concepts": $\mathsf{EX}\ C$ represents the set of objects which are elements of C in *some next* state; $\mathsf{AF}\ C$ represents the set of objects which are on *all paths eventually* elements of C; $\mathsf{E}(C\ \mathsf{U}\ D)$ represents the set of objects which are on *some path* element of C *until* they are element of D. In Formula (1), *explained* $\sqcap$ EX *exemplified* is a (non-atomic) concept.

Any of the usual Boolean, *ALC*, or CTL connectives such as $p \vee q$ (disjunction), $\mathsf{AG}\ p$ (all paths generally p), $C \sqcup D$ (union), or $\forall R.C$ (universal quantification on roles), can be expressed in the connectives of Table 1. The CTL **fragment** of *ALC*CTL is *ALC*CTL without concept constructors (second row of Table 1) and with concept subsumption $C \sqsubseteq D$ being replaced by atomic propositions. In this way, *ALC*CTL subsumes CTL.

The **semantics** of *ALC*CTL is defined w.r.t. structures $M = (S, T, \Delta, I)$ where S is a set of *states*, $T \subseteq S \times S$ is a left-total *transition relation*, Δ is a set of objects of interest called *interpretation domain*, and I is a state-dependent *interpretation* of atomic concepts and roles in such a way that for each $s \in S, A \in \mathscr{C}$, and $R \in \mathscr{R}$ it holds: $A^{I(s)} \subseteq \Delta$ and $R^{I(s)} \subseteq \Delta \times \Delta$. In this paper, we **assume both S and Δ to be finite and non-empty**.

Example 2 (*ALC*CTL *Structure*)
As an example of an *ALC*CTL structure, consider $M = (S, T, \Delta, I)$ were

$S = \{s0, s1, s2\}$	
$T = \{(s0, s1), (s0, s2), (s1, s2), (s2, s2)\}$	
$\Delta = \{tree, heap\}$	
$Task^{I(s0)} = \{heap\}$	there is a task on the topic *heap* in $s0$
$Solution^{I(s1)} = \{heap\}$	there is a solution on *heap* in $s1$
$Test^{I(s2)} = \{tree, heap\}$	there is a test on *tree* and *heap* in $s2$

Figure 2 depicts the state transition graph (S, T). The states are annotated with non-empty interpretations of concepts. □

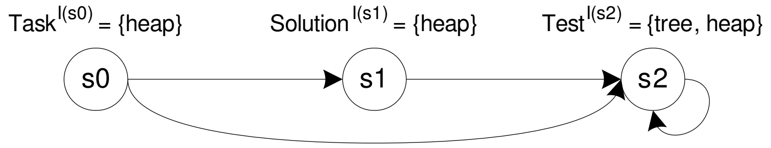


Figure 2: *ALC*CTL structure

The semantics of *ALC*CTL defines when a formula f holds in M at a state s, denoted as $M, s \models f$ or $s \models f$ if M is understood. It extends the interpretation I to non-atomic concepts. For instance, $(\neg C)^{I(s)} = \Delta \backslash C^{I(s)}$, $(C \sqcap D)^{I(s)} = C^{I(s)} \cap D^{I(s)}$, and $(\mathsf{EX}\ C)^{I(s)} = \bigcup_{s' \in T(s)} C^{I(s')}$ where $T(s)$ denotes the T-image $\{s' \in S \mid (s, s') \in T\}$ of s. Further,

$$\begin{array}{ll} s \models C \sqsubseteq D & \text{iff } C^{I(s)} \subseteq D^{I(s)} \\ s \models \neg p & \text{iff } s \not\models p \\ s \models p \wedge q & \text{iff } s \models p \text{ and } s \models q \\ s \models \mathsf{EX}\ p & \text{iff } \exists s' \in T(s) : s' \models p \end{array}$$

$s \models \mathsf{AF}\ p$ iff in each infinite path $(s_0, s_1, ...)$ in (S, T) starting from s there is a state s_i such that $s_i \models p$. $s \models \mathsf{E}(p\ \mathsf{U}\ q)$ iff there is such a path $(s_0, s_1, ..., s_n)$ in (S, T) starting from s that $s_n \models q$ and for each $i \in \{0..n-1\} : s_i \models p$.

In this paper, we discuss counterexamples for $\mathcal{ALC}\mathsf{CTL}^{+a}_{-R}$ which is $\mathcal{ALC}\mathsf{CTL}$ without quantified roles $\exists R.C$ but extended with concept assertions $C(a)$ where C is a concept and $a \in \Delta$ is a domain object. The semantics of concept assertions is $s \models C(a)$ iff $a \in C^{I(s)}$. We disregard quantified roles in this paper merely because of space limitations. The algorithm presented in this paper can be extended to handle quantified roles by integrating the respective parts of the algorithm in [27].

Example 3 (*$\mathcal{ALC}\mathsf{CTL}$ Semantics*)
Let $M = (S, T, \Delta, I)$ as in Example 2. Then

$s0 \not\models \mathit{Solution}(\mathit{heap})$	because $\mathit{heap} \notin \mathit{Solution}^{I(s0)}$
$s1 \models \mathit{Solution}(\mathit{heap})$	because $\mathit{heap} \in \mathit{Solution}^{I(s1)}$
$(\mathsf{EX}\ \mathit{Solution})^{I(s0)} = \{\mathit{heap}\}$	because $T(s0) = \{s1, s2\}$, $\mathit{Solution}^{I(s1)} = \{\mathit{heap}\}$, and $\mathit{Solution}^{I(s2)} = \emptyset$
$(\mathsf{EX}\ \mathit{Solution})^{I(s1)} = \emptyset$	because $T(s1) = \{s2\}$ and $\mathit{Solution}^{I(s2)} = \emptyset$
$s0 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$	because $\mathit{Task}^{I(s0)} \subseteq (\mathsf{EX}\ \mathit{Solution})^{I(s0)}$
$s1 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$	because $\mathit{Task}^{I(s1)} \subseteq (\mathsf{EX}\ \mathit{Solution})^{I(s1)}$

□

Definition 4 (*Model Checking Problem of $\mathcal{ALC}\mathsf{CTL}$*)
The model checking problem of $\mathcal{ALC}\mathsf{CTL}$ is to decide if $M, s \models f$ for a given structure $M = (S, T, \Delta, I)$, state $s \in S$, and $\mathcal{ALC}\mathsf{CTL}$ formula f. □

A detailed description of the syntax and semantics as well as a polynomial model checking algorithm for $\mathcal{ALC}\mathsf{CTL}$ is given in [25].

Example 5 (*Model Checking $\mathcal{ALC}\mathsf{CTL}$*)
Consider the $\mathcal{ALC}\mathsf{CTL}$ formula

$$f = \mathsf{E}((\mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution})\ \mathsf{U}\ \neg(\mathit{Test} \sqsubseteq \bot))$$

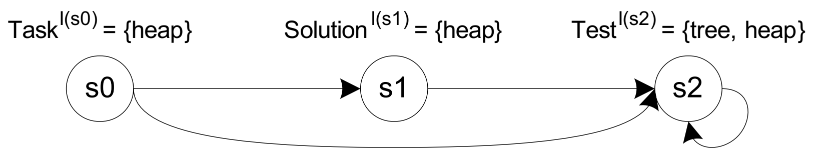
"There is a path (E) with the following properties: for each exercise task ($\mathit{Task} \sqsubseteq$), a solution is reachable in one step ($\mathsf{EX}\ \mathit{Solution}$) until ($\mathsf{U}$) there is a test ($\neg(\mathit{Test} \sqsubseteq \bot)$)." $\bot$ is an abbreviation for the empty concept $A \sqcap \neg A$. Hence, $s \models \neg(\mathit{Test} \sqsubseteq \bot)$ iff $\mathit{Test}^{I(s)} \neq \emptyset$.

Let $M = (S, T, \Delta, I)$ be the $\mathcal{ALC}\mathsf{CTL}$ structure of Example 2. Then $M, s0 \models f$. This is because there is a path $(s0, s1, s2)$ in (S, T) starting from $s0$ such that $s0 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ and $s1 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ (cf. Example 3) and $s2 \models \neg(\mathit{Test} \sqsubseteq \bot)$.

The model checking algorithm given in [25] calculates the interpretation of concepts and sub-formulae in f as depicted in Figure 3. In step 1), the interpretation of the non-atomic concept $\mathsf{EX}\ \mathit{Solution}$ is calculated for each state. In steps 2) through 4), it is determined for each state $s \in S$ and sub-formula f' of f, whether $s \models f'$. The results in bold face are used to obtain the final result of $s0 \models f$ in step 5). The intermediate results of steps 1) through 4) provide the basis for incremental and interactive counterexample generation as proposed subsequently. □

4 Generating Counterexamples

Consider an $\mathcal{ALC}\mathsf{CTL}$ structure $M = (S, T, \Delta, I)$, a state $s \in S$, and such an $\mathcal{ALC}\mathsf{CTL}$ formula f that $M, s \not\models f$. Our aim is to generate counterexamples to $M, s \models f$ that isolate (Q1) the *states* in S and (Q2) the *objects* in Δ involved in some error scenario, and explain (Q3) *why* a given property is violated for these objects and states (cf. section 2). To avoid information overload by bulky counterexamples, we



1)	(EX Solution)$^{I(s0)}$ = {heap}	(EX Solution)$^{I(s1)}$ = {}	(EX Solution)$^{I(s2)}$ = {}
2)	**s0 |= Task ⊑ EX Solution**	**s1 |= Task ⊑ EX Solution**	s2 |= Task ⊑ EX Solution
3)	s0 |= Test ⊑ ⊥	s1 |= Test ⊑ ⊥	
4)			**s2 |= ¬(Test ⊑ ⊥)**
5)	**s0 |= E((Task ⊑ EX Solution) U ¬(Test ⊑ ⊥))**		

Figure 3: Intermediate results of *ALC*CTL model checking

structure counterexamples hierarchically w.r.t. the expression tree of the verified formula and build them incrementally in interaction with the user. This way, the correspondence between parts of counterexamples and parts of the violated formula is revealed and the step-by-step analysis of complex error scenarios is supported.

4.1 Representation of Evidence

A counterexample for a formula f may contain witnesses for subexpressions of f. To generalize from witnesses and counterexamples, we use the term *evidence* (cf. [4]). A counterexample is an *evidence* for $M,s \not\models f$ while a witness is an *evidence* for $M,s \models f$. We model evidence as an ordered tree obtained from the propositional reduction of an *ALC*CTL formula f w.r.t. M and s [25].

Example 6 (*Structure of Evidence*)

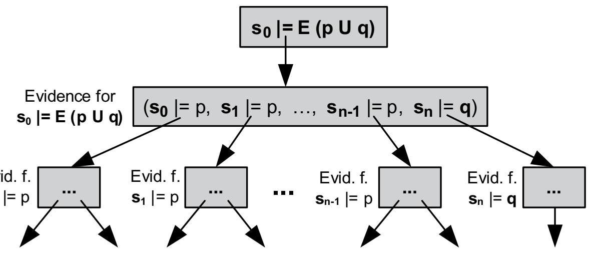


Figure 4: General structure of evidence for $s_0 \models \mathsf{E}(p \mathbin{\mathsf{U}} q)$. Read: "$\mathsf{E}(p \mathbin{\mathsf{U}} q)$ holds in s_0 *because* there is a path $(s_0, s_1, ..., s_n)$ in M such that p holds in s_0 and p holds in s_1 and ... and p holds in s_{n-1} and q holds in s_n as demonstrated by the following sub-evidences: p holds in s_0 *because* ..."

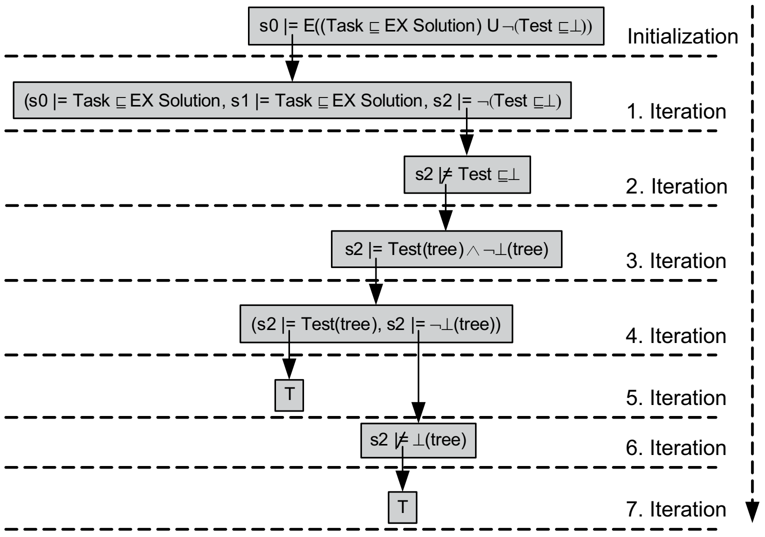
Consider the *ALC*CTL formula f of Example 5. Let $p = \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ and $q = \neg(\mathit{Test} \sqsubseteq \bot)$, i.e., $f = \mathsf{E}(p \mathbin{\mathsf{U}} q)$. Assume such a structure $M = (S, T, \Delta, I)$ and state $s_0 \in S$ that $s_0 \models f$.

A suitable *evidence* for $s_0 \models f$ is a path $(s_0, s_1, ..., s_n)$ in (S, T) on which it holds: $s_0 \models p$, $s_1 \models p$, ..., $s_{n-1} \models p$, and $s_n \models q$. Hence, the evidence for $s \models f$ should include the evidences for $s_i \models p$ ($i \in \{0..n-1\}$) and $s_n \models q$ as sub-evidences. Figure 4 depicts the structure of evidence for $s_0 \models f$. □

The *edges* of an evidence tree can be read as "because" (cf. caption of Figure 4): they associate a *state expression* $s \models f$ with a finite sequence of state expressions $(s_0 \models f_0, s_1 \models f_1, ..., s_n \models f_n)$ being the reason for $s \models f$ (note that f_i may be equal to f_j for $i \neq j$). Recursively generating evidences for each of the obtained state expressions $s_i \models f_i$ results in a tree that can be built top-down based on intermediate model checking results.

Example 7 (*Top-Down Construction of Evidence Tree*)

Let f be the $\mathcal{ALC}$CTL formula and $M = (S, T, \Delta, I)$ the $\mathcal{ALC}$CTL structure of Example 5. Figure 5 illustrates how a branch of the evidence tree for $s0 \models f$ is built in seven iterations. In an **initialization** step, the root node of the evidence tree is set to the state expression for which an evidence should be provided (Figure 5 top).



s0 |= E((Task ⊑ EX Solution) U ¬(Test ⊑⊥))
Initialization
(s0 |= Task ⊑ EX Solution, s1 |= Task ⊑ EX Solution, s2 |= ¬(Test ⊑⊥)
1. Iteration
s2 |≠ Test ⊑⊥
2. Iteration
s2 |= Test(tree) ∧ ¬⊥(tree)
3. Iteration
(s2 |= Test(tree), s2 |= ¬⊥(tree))
4. Iteration
T
5. Iteration
s2 |≠ ⊥(tree)
6. Iteration
T
7. Iteration

Figure 5: Part of the evidence tree for $f = \mathsf{E}(Task \sqsubseteq \mathsf{EX}\ Solution\ \mathsf{U}\ \neg(Test \sqsubseteq \bot))$ after seven iterations .

1. Iteration: f is of type $\mathsf{E}(p\ \mathsf{U}\ q)$ where $p = Task \sqsubseteq \mathsf{EX}\ Solution$ and $q = \neg(Test \sqsubseteq \bot)$. According to Example 6), the first step in providing evidence for $s0 \models \mathsf{E}(p\ \mathsf{U}\ q)$ is to find a sequence $(s_0 \models p, ..., s_{n-1} \models p, s_n \models q)$ where $(s_0, ..., s_n)$ is a path in (S, T) starting from $s0$. By analyzing the intermediate model checking results in Figure 3, we find the path $(s0, s1, s2)$ where

$$\begin{array}{ll} s0 \models Task \sqsubseteq \mathsf{EX}\ Solution & \text{(row 2) in Figure 3)} \\ s1 \models Task \sqsubseteq \mathsf{EX}\ Solution & \text{(row 2) in Figure 3)} \\ s2 \models \neg(Test \sqsubseteq \bot) & \text{(row 4) in Figure 3)} \end{array}$$

The resulting sequence of state expressions is added as a child node to the root node of the evidence tree (Figure 5, 1. Iteration).

2. Iteration: Let us assume that the user is interested now why $s2 \models \neg(Test \sqsubseteq \bot)$ (Figure 5, rhs of node obtained in the 1. Iteration). By semantics of negation "$\neg$" we get: $s2 \models \neg(Test \sqsubseteq \bot)$ because $s2 \not\models Test \sqsubseteq \bot$ (Figure 5, 2. Iteration).

3. Iteration: The user may now want to know why $s2 \not\models Test \sqsubseteq \bot$ which is equivalent to $Test^{I(s2)} \not\subseteq \emptyset$. Each element of $Test^{I(s2)} = \{tree, heap\}$ (Figure 3 rhs top) provides evidence for $s2 \not\models Test \sqsubseteq \bot$. As for the general case $s \not\models C \sqsubseteq D$, the set of evidence objects is $C^{I(s)} \backslash D^{I(s)}$, and the set of evidences is $\{s \models C(a) \wedge \neg D(a) \mid a \in C^{I(s)} \backslash D^{I(s)}\}$. In the given example, we get $\{s2 \models Test(tree) \wedge \neg\bot(tree)$, $s2 \models Test(heap) \wedge \neg\bot(heap)\}$ as the set of alternative evidences for $s2 \not\models Test \sqsubseteq \bot$. Let us assume that the user selects $s2 \models Test(test) \wedge \neg\bot(test)$ for further analysis. This extends the evidence tree to the level of the 3. Iteration in Figure 5.

4. Iteration: $s2 \models \mathit{Test}(\mathit{test}) \wedge \neg\bot(\mathit{test})$ because $s2 \models \mathit{Test}(\mathit{test})$ and $s2 \models \neg\bot(\mathit{test})$. Hence, the latter two state expressions in combination provide evidence to $s2 \models \mathit{Test}(\mathit{test}) \wedge \neg\bot(\mathit{test})$ which is represented by the pair of state expressions as depicted in Figure 5, 4. Iteration.

5. Iteration: Assume that the user requests evidence for $s2 \models \mathit{Test}(\mathit{tree})$ obtained in the 4. Iteration. Since $\mathit{Test}(\mathit{tree})$ is an atomic expression, it holds by definition of M and no further explanation can be provided. This is represented by the *terminal* node "$\top$" in the evidence tree of Figure 5, 5. Iteration.

6. Iteration: An evidence for $s2 \models \neg\bot(\mathit{tree})$ is provided in the same way as in the 2. Iteration.

7. Iteration: Similar to the 5. Iteration, no further explanation for $s2 \not\models \bot(\mathit{tree})$ can be provided which is represented by the terminal node "$\top$".

□

The remaining branches for $s0 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ and $s1 \models \mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ (Figure 5, 1. Iteration) can be expanded in a similar way. The evidence tree is *complete* when all leaves nodes are terminal nodes $\top$. The evidence provided by current model checkers for the given scenario consists of the path (s_0, s_1, s_2), leaving most of the analysis work to the user.

Remark 8 (*Interpretation of Evidence Tree*)
The evidence tree of Example 7 contains the information for answering the questions Q1) through Q3) in section 2. It clarifies

(Q1) in which *states* which properties hold or do not hold. For instance, $\mathit{Task} \sqsubseteq \mathsf{EX}\ \mathit{Solution}$ holds in states $s0$ and $s1$, and $\neg(\mathit{Test} \sqsubseteq \bot)$ holds in state $s2$ as indicated by the node of the 1. Iteration in Figure 5.

(Q2) for which *objects* a property holds or does not hold. For instance, $\mathit{Test} \sqsubseteq \bot$ is violated in state $s2$ by term *tree* as demonstrated by the evidence for $s2 \not\models \mathit{Test} \sqsubseteq \bot$ (Figure 5, 3. Iteration).

(Q3) *why* a property holds or does not hold. The cause of a property satisfaction or violation can be drilled down by successively expanding the nodes of the evidence tree until a terminal node is reached.

For interaction with users not acquainted in temporal logic, the evidence tree is translated into a structured error report which refers to application level objects (cf. [21, 22]). In the given case, states are mapped onto web pages, domain objects onto important terms used throughout the document, and $\mathcal{ALC}\mathsf{CTL}$ formula onto high-level properties derived from specification patterns [17]. □

Remark 9 (*Optimizations*)
The *amount of user interaction* may be reduced by clustering subexpressions of the formula and automatically expanding branches of the tree in cases without choices. As for Example 7, just the second and third iteration include choices. In the second iteration, the user has to decide for which state expression of the sequence obtained in the first iteration further evidence should be provided. In the third iteration, the user has to choose an evidence for further analysis from a set of alternative options. The Iterations 1 and 4 – 7 can be completed without involving the user.

The *size of the evidence tree* may be reduced by making use of semantic equivalences. For instance, $s \models \mathit{Test}(\mathit{tree}) \wedge \neg\bot(\mathit{tree})$ could be simplified to $s \models \mathit{Test}(\mathit{tree})$ in Iteration 3 of Example 7 because $\neg\bot(\mathit{tree}) \equiv \mathit{true}$. This would remove branches that contain just trivial information. On the other hand, applying (non-trivial) semantic optimizations may result in evidence trees that are hard to understand. Further research is necessary to find a practical approach to semantic optimization. □

4.2 Generation of Evidence

We now generalize the approach sketched in Examples 7 to an algorithm. First, we introduce the basic structures for representing evidence trees as depicted in Figure 5.

$$StateExpression = S \times \{\models, \not\models\} \times ALC\mathsf{CTL}^{+a}_{-R} \tag{3}$$

$$Node \subseteq \{\top\} \cup (StateExpression \times ChildNode) \tag{4}$$

$$ChildNode \subseteq \bigcup_{n \in \mathbb{N}} Node^n \tag{5}$$

A *state expression* (Equation (3)) is a triple (s, v, f) where $s \in S$ is a state, $v \in \{\models, \not\models\}$ a validity indicator, and f an $ALC\mathsf{CTL}^{+a}_{-R}$ formula. A *node* in the evidence tree (Equation (4)) is either a terminal node $\top$ or a state expression e which has a finite sequence of nodes as *child node* (Equation (5)) that provides evidence to e. We use the following abbreviations:

- State expressions $(s, \models, f)$ and $(s, \not\models, f)$ are denoted as $s \models f$ and $s \not\models f$, respectively.
- ε denotes the empty sequence. $\langle s \models f \rangle$ and $\langle s \not\models f \rangle$ denote evidence nodes $(s \models f, \varepsilon)$ and $(s \not\models f, \varepsilon)$ without a child node. For instance, $\langle s2 \not\models \bot(tree) \rangle$ denotes the node obtained in the 6. Iteration of Figure 5. $(s2 \not\models \bot(tree), \top)$ denotes the same node after the 7. Iteration.
- Let n be a node and $s = (n_0, ..., n_k) = (n_i)_{i \in \{0..k\}}$ a sequence of nodes. Then $s \circ n$ denotes the sequence $(n_0, ..., n_k, n)$ obtained by appending node n to s.
- Let $node = (e, c)$ be a non-terminal evidence node. Then $node.expr$ denotes the state expression e and $node.child$ denotes the child node c of $node$.

The subsequent algorithm for generating evidence consists of three parts:

1. The main function GETEVIDENCE(s, f) returns an evidence for $M, s \models f$ or $M, s \not\models f$, respectively. It calls the model checking algorithm CHECK as defined in [25] to determine whether $M, s \models f$. After that, INTERACTIVEEXPAND is called to incrementally generate the evidence tree for s and f on the lines of Example 7.
2. INTERACTIVEEXPAND$(EvTree)$ expands user selected branches of an evidence tree *EvTree*. It calls GETEVSET to calculate the set of possible child nodes of a chosen node in *EvTree*.
3. GETEVSET$(expr)$ returns the set of options for the child node of a given state expression *expr*.

Algorithm 10 (*Evidence Generation*)
In the subsequent algorithm, the structure $M = (S, T, \Delta, I)$ is assumed to be available as a global variable.

```
function GETEVIDENCE($s, f$)
    if CHECK($M, s \models f$) then return INTERACTIVEEXPAND($\langle s \models f \rangle$);
    else return INTERACTIVEEXPAND($\langle s \not\models f \rangle$);
end function

function INTERACTIVEEXPAND($EvTree$)
    $node \leftarrow$ USERSELECTNODE($EvTree$);
    while $node \neq \top$ do
        $EvSet \leftarrow$ GETEVSET($node.expr$);
        if $|EvSet| > 1$ then $node.child \leftarrow$ USERSELECTELEM($EvSet$);
```

```
            else $node.child \leftarrow$ ELEMENTOF($EvSet$);
            $node \leftarrow$ USERSELECTNODE($EvTree$);
        end while
        return $EvTree$;
15: end function

    function GETEVSET($expr$)
        case($expr$)
            $(s \models A(a) \mid s \not\models A(a)) : EvSet \leftarrow \{\top\}$;
20:         $(s \models \mid s \not\models)(\neg C \mid C \sqcap D \mid \mathsf{EX}\, C \mid \mathsf{AF}\, C \mid \mathsf{E}(C \mathbin{\mathsf{U}} D))(a) : EvSet \leftarrow$ GETEVSET($reduce(expr)$);
            $s \models C \sqsubseteq D : EvSet \leftarrow \{(\langle s \not\models C(a) \wedge \neg D(a)\rangle)_{a \in \Delta}\}$;
            $s \not\models C \sqsubseteq D : EvSet \leftarrow \{\langle s \models C(a) \wedge \neg D(a)\rangle \mid a \in C^{I(s)} \setminus D^{I(s)}\}$;
            $s \models \neg p : EvSet \leftarrow \{\langle s \not\models p\rangle\}$;
            $s \not\models \neg p : EvSet \leftarrow \{\langle s \models p\rangle\}$;
25:         $s \models p \wedge q : EvSet \leftarrow \{(\langle s \models p\rangle, \langle s \models q\rangle)\}$;
            $s \not\models p \wedge q : EvSet \leftarrow \{\langle s \not\models f\rangle \mid f \in \{p,q\}$ and $s \not\models f\}$;
            $s \models \mathsf{EX}\, p : EvSet \leftarrow \{\langle s' \models p\rangle \mid s' \in T(s)$ and $s' \models p\}$;
            $s \not\models \mathsf{EX}\, p : EvSet \leftarrow \{(\langle s' \not\models p\rangle)_{s' \in T(s)}\}$;
            $s \models \mathsf{AF}\, p : EvSet \leftarrow \{\top\}$;
30:         $s \not\models \mathsf{AF}\, p : EvSet \leftarrow \{(\langle s_i \not\models p\rangle)_{i \in \{0..n\}} \mid (s_i)_{i \in \{0..n\}} \in$ FINDLOOP$(M,s,p)$ $\}$;
            $s \models \mathsf{E}(p \mathbin{\mathsf{U}} q) : EvSet \leftarrow \{(\langle s_0 \models p\rangle)_{i \in \{0..n-1\}} \circ \langle s_n \models q\rangle \mid (s_i)_{i \in \{0..n\}} \in$ FINDPATH$(M,s,p,q)\}$;
            $s \not\models \mathsf{E}(p \mathbin{\mathsf{U}} q) : EvSet \leftarrow \{\top\}$;
        end case
        return $EvSet$;
35: end function
```

□

User interaction is involved in the following functions:

- USERSELECTNODE(*EvTree*), called in lines 7 and 12, returns the node of the evidence tree selected by the user to be expanded, e.g. $\langle s2 \models \neg(\mathit{Test} \sqsubseteq \bot)\rangle$ in the 2. Iteration of Example 7.
- USERSELECTELEM(*EvSet*), called in line 10, returns the child node selected by the user from a set of options, e.g. $\langle s2 \models \mathit{Test}(\mathit{tree}) \wedge \neg\bot(\mathit{tree})\rangle$ in the 3. Iteration of Example 7.

Function GETEVSET calculates the set of evidences for a state expression $expr = (s, v, f)$ by matching it against a list of possible cases. In GETEVSET, $s \in S$ is a state, C, D are $\mathcal{ALC}\mathsf{CTL}^{+a}_{-R}$ concepts, A is an atomic concept, $a \in \Delta$ is a domain object, and p, q are $\mathcal{ALC}\mathsf{CTL}^{+a}_{-R}$ formulae.

If the parameter *expr* is a state expression of type $s \models A(a)$ or $s \not\models A(a)$ (line 19), a terminal evidence $\top$ is returned (cf. 5. and 7. Iteration in Example 7). In the case of other concept assertions $(\neg C)(a)$, $(C \sqcap D)(a)$,...,$\mathsf{E}(C \mathbin{\mathsf{U}} D)(a)$ (line 20), the argument a is pushed down to inner expressions by function *reduce* and GETEVSET is called with the reduced expression. For instance, $reduce(s \models (\neg C)(a))$ returns $s \models \neg(C(a))$, $reduce(s \models (C \sqcap D)(a))$ yields $s \models C(a) \wedge D(a)$, and $reduce(s \models \mathsf{E}(C \mathbin{\mathsf{U}} D)(a))$ results in $s \models \mathsf{E}(C(a) \mathbin{\mathsf{U}} D(a))$.

As for the other connectives, witnesses ($\models$) and counterexamples ($\not\models$) are distinguished. In line 21, a witness for $s \models C \sqsubseteq D$ is generated. Such a witness demonstrates that for each $a \in \Delta$: $s \models \neg C(a)$ or $s \models D(a)$ which is equivalent to $s \not\models C(a) \wedge \neg D(a)$. Hence, the tuple $(\langle s \not\models C(a) \wedge \neg D(a)\rangle)_{a \in \Delta}$ is returned as an evidence for $s \models C \sqsubseteq D$ in line 21.

In line 22, a counterexample for $s \models C \sqsubseteq D$ is generated along the lines of Iteration 3 in Example 7. Line 23 corresponds to Iterations 2 and 6 in Example 7. The nodes obtained in Iterations 1 and 4 of Example 7 correspond to the cases of lines 31 and 25, respectively. FINDPATH(M,s,p,q) in line 31 applies breath-first-search to find the set of shortest paths $(s_i)_{i\in\{0..n\}}$ in M starting from s such that $s_0 \models p,\ldots,s_{n-1} \models p$ and $s_n \models q$. Such sequences are witnesses for $s \models \mathsf{E}(p \mathbin{\mathsf{U}} q)$. Similarly, FINDLOOP$(M,s,p)$ in line 30 searches for shortest paths $(s_i)_{i\in\{0..n\}}$ in M from s such that $s_n = s_j$ for some $j \in \{0..n-1\}$ (loop property) and $s_i \not\models p$ for each $i \in \{0..n\}$. Such sequences are counterexamples for $s \models \mathsf{AF}\ p$.

Note that witnesses for $s \models \mathsf{AF}\ p$ would have to demonstrate that on all paths starting from s eventually p holds. Since, in general, there are infinitely many such paths, a compact evidence for $s \models \mathsf{AF}\ p$ cannot be provided. The same holds in the case of $s \not\models \mathsf{E}(p \mathbin{\mathsf{U}} q)$. As a consequence, a terminal evidence $\top$ is returned in lines 29 and 32.

Remark 11 (*Soundness, Completeness, and Termination of the Algorithm*)

A preliminary, non-incremental version of Algorithm 10 has been proven to be *sound* for *ALC*CTL and *complete* for a larger fragment of *ALC*CTL [27] than previous counterexample algorithms [18,7,6]. Function GETEVSET terminates because each calculated set is finite. Function INTERACTIVEEXPAND is terminated by the user. □

5 Experimental Results

The proposed algorithm has been implemented in Java and integrated in our *ALC*CTL model checker for document verification [26]. The runtime results have been acquired on a notebook computer with Intel Core 2 Duo processor at 2.93 GHz, 4 GB RAM, 64 GB SSD, running Windows 7 (32 Bit) and Java 6 update 19. The proposed algorithm has been compared with the CTL model checker NuSMV 2.4.3 [5].

5.1 Evaluation Case

As an evaluation case, we used an XML-based training document on industrial robots which is implemented in the SCORM [1] standard for web-based e-learning content. The document consists of 90 web pages, 79 of them being represented as "states" in the *ALC*CTL model M that is generated from the XML markup by a software component. The document has been checked against 25 criteria each represented both as an *ALC*CTL formula and as a CTL formula for comparison with NuSMV. 5 formulae were found violated. For each satisfied and violated formula, a complete evidence tree has been calculated using Algorithm 10. User interactions were simulated by a depth-first expansion of nodes in lines 7 and 12, and a random selection in the case of alternatives in line 10 of Algorithm 10.

5.2 Results

The rows in the center of Table 2 summarize the sizes of generated evidence as compared to counterexamples generated by NuSMV. Since the document structure M and the verified properties have been chosen in such a way that they can be represented equally well in *ALC*CTL and CTL, the counterexamples of *ALC*CTL and of NuSMV are quite similar in size and range between 1 and 80 states in the case of NuSMV, and between 3 and 85 nodes in the case of *ALC*CTL. However, the *ALC*CTL counterexamples are structured as trees, breaking down large error scenarios into comprehensible units. The largest evidence tree (height 17, 85 nodes) was constructed for a formula with as many as 42 subexpressions. It is almost impossible to manually analyze the linear counterexamples returned by NuSMV for such complex properties. The evidence sets for each node contained between 1 and 9 elements, i.e., the user could choose one of at most nine options in line 10 of Algorithm 10 to expand the current node.

The counterexamples provided by both *ALC*CTL and CTL lead to 6 "error states", i.e., states that correspond to defective web pages. However, while in the case of *ALC*CTL these states are clearly identified in the evidence tree, it requires a considerable amount of manual effort to find the relevant states in

	$ALC\mathsf{CTL}$	CTL (NuSMV)
# web pages / states	90 / 79	
# violated formulae	5 of 25	
evidence trees for satisfied formulae		
height	2 – 7	– (no witnesses)
# nodes	2 – 146	–
# elements of evidence set per node	1 – 8	–
evidence trees for violated formulae		
height	3 – 17	1 (linear counterexamples)
# nodes (CTL: # states)	3 – 85	1 – 80
# elements of evidence set per node	1 – 9	– (no alternatives provided)
# error locations (states) found	6	6 (required manual analysis)
# error objects found	5	–
total runtime	280 ms	660 ms
runtime of model checking	31 ms	30 ms
runtime of evidence generation	30 ms	310 ms
rest (doc. analysis, model generation)	219 ms	320 ms

Table 2: Size and results of the case study on XML documents

counterexample sequences returned by NuSMV. In addition to error locations, the $ALC\mathsf{CTL}$ counterexamples identified a total of 5 "error objects". Error objects are bindings of quantified variables in violated subexpressions (cf. Example 1 and Q2) in section 2). In the given case, they represent incorrect properties of parts of web pages. For instance, a "test solution", which has been tagged as an "information unit" by mistake, has been detected as an error object but has not been reported in the counterexamples provided by NuSMV.

The lower part of Table 2 summarizes the runtime results of the experiment. Although providing more structured and accurate evidence, the proposed algorithm performed better than NuSMV. We assume that it is easier to extract evidence in the case of an explicit representation of the state space as applied in $ALC\mathsf{CTL}$ model checking than in the case of a symbolic representation used by NuSMV.

5.3 Performance

For determining the scaling of runtime in the document size, a series of 8 documents consisting of 16 through 128 chapters of 32 pages each has been synthesized. Each of these documents have been checked against 10 formulae. Seven of them were satisfied and three of them were violated. A complete evidence tree for each satisfied and violated formula was generated by simulating user choices as described in section 5.1.

Table 3 shows the results for four cases of the experiment. The rows in the center of Table 3 report on the size of the largest evidence tree for each document. The height of an evidence tree merely depends on the corresponding formula. Since the same set of formulae were checked on each document, the heights of the resulting evidence trees do not vary across different documents. In contrast, the number of nodes of the largest evidence tree and the size of the largest evidence set per node grow proportionally in the size of the document. When evidence sets grow beyond 50 elements, the interactive exploration of each case becomes infeasible. The height of evidence trees grows linearly in the size of the formula and thus may become large for very complex properties. Promising strategies to reduce the height and width of evidence trees are: 1) clustering larger parts of formulae into *macro operators*, based on specification

# web pages / states	512	1024	2048	4096
# violated formulae	3 of 10			
size of largest evidence tree				
height	7	7	7	7
# nodes	900	1796	3588	7172
# elements of the largest evidence set	64	128	256	512
total runtime	0.7 s	1.3 s	2.1 s	3.6 s
runtime of model checking	31 ms	78 ms	140 ms	280 ms
runtime of evidence generation	16 ms	32 ms	47 ms	172 ms
worst case user interaction response time	<10 ms	15 ms	32 ms	93 ms
rest (doc. analysis, model generation)	0.65 s	1.2 s	1.9 s	3.1 s

Table 3: Results on larger documents

patters proposed in [17]; 2) providing *partial* evidence dependent on the user's focus; 3) generating a compact *symbolic* representation of evidence using variables (cf. [27]).

The rows on the bottom of Table 3 summarize the runtime results. Even for large documents, the total runtime remains below 5 seconds. The runtime is dominated by the time for document analysis and model generation and scales approximately linearly in the document size. Evidence generation takes less than 5% of the total runtime. Most important for smooth user interaction is the response time of the system when expanding the evidence tree. The response time is dominated by the runtime for calculating the evidence set for a given state expression (line 9 in Algorithm 10). Even in the case of very large documents, the worst case response time remains below 100 ms, sufficiently low for smooth user interaction.

6 Related Work

[18] describes the basic method of generating linear counterexamples for CTL which is still adopted in state-of-the-art model checkers such as NuSMV [5] or SAL [8]. [7] suggests a method for generating richer "tree-like" counterexamples. The tree structure of the counterexamples corresponds with computation trees of the verified model. We structure counterexamples along the expression tree of the verified formula. This clarifies the correspondence between parts of the counterexample and parts of the violated formula and supports tracking the cause of a property violation down to the desired level of detail. Further, a higher level of completeness and detail is obtained than in [7] because we also consider Boolean connectives, the subset operator of *ALC*, and both witnesses and counterexamples for EX.

There have been a number of efforts for addressing the problem of bulky counterexamples. [16] suggests efficient algorithms, based on transitions shuffling, for approximating the smallest counterexample in on-the-fly model checking. [20] and [24] define a method for minimizing variable assignments in CTL counterexample traces which makes it simpler for the user to find areas of interest. [12, 13, 11] and [3] localize errors in C programs based on model checking results by comparing incorrect and correct runs of the program. Incremental generation of counterexample and witnesses w.r.t. the user's interest, however, has not been considered.

[4] proposes a framework for counterexample generation and exploration, based on "proof-like" counterexamples [14]. Counterexamples and witnesses provided by a symbolic CTL model checker are annotated with proofs that explain why a property holds in a given state of a model. While these proofs support experts in analyzing counterexamples they may be difficult to understand for users not acquainted

in proof systems and proof rules.

A first method for finding counterexamples for the CTL fragment of *ALC*CTL has been described in our previous work in [25] and demonstrated in case studies on checking the consistency of technical documentations [21,23]. [27] proposes a formal definition and analysis of evidence trees as well as a first algorithm for generating them. This paper extends previous work towards incremental and interactive generation of evidence trees based on intermediate model checking results. In addition, the case study in [27] is extended towards larger and more complex documents and towards the analysis of the system's response time in interactive evidence generation.

7 Conclusion

We have presented a new algorithm for the *incremental* generation of tree-structured counterexamples and witnesses for properties of web documents expressed in the temporal description logic *ALC*CTL. The algorithm supports exploring alternative error scenarios according to the user's interest, instead of providing just a single, arbitrarily chosen counterexample as existing model checkers do. The generated counterexamples identify both the parts of the model and the parts of the formula involved in some error scenario, and support the step-by-step analysis of the cause of a property violation. The runtime performance of the algorithm scales up to application-relevant problem sizes. The worst case response time remains below 100 ms even for documents with several thousands of web pages. The presented approach thus provides a solid basis for generating structured, precise, and user adaptable error reports. Issues of **future work** include the *optimization* of the algorithm to minimize the size of evidence and the amount of user interaction, the *visualization* of application level reports generated from evidence trees, and the *evaluation* of the usefulness of the approach for differently experienced users.

8 Acknowledgements

This work is funded by the program "Research at International Science and Technology Centers" of the German Academic Exchange Service (DAAD). The case study reported in section 5 originated from the Verdikt project [26]. We thank Burkhard Freitag for continuing support.

References

[1] Advanced Distributed Learning. *Sharable Content Object Reference Model (SCORM) 2004 2nd Ed., Overview*, 2004.

[2] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider, editors. *The Description Logic Handbook - Theory, Implementation and Applications*. Cambridge Univ. Press, 2003.

[3] Thomas Ball, Mayur Naik, and Sriram K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 97–105, 2003.

[4] Marsha Chechik and Arie Gurfinkel. A framework for counterexample generation and exploration. *International Journal on Software Tools for Technology Transfer*, 9(5):429–445, 2007.

[5] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Proceedings of Computer Aided Verification (CAV 02)*, volume 2404 of *LNCS*. Springer, 2002.

[6] Edmund Clarke and Helmut Veith. *Verification: Theory and Practice*, volume 2772 of *LNCS*, chapter Counterexamples Revisited: Principles, Algorithms, Applications, pages 41–43. Springer-Verlag, 2004.

[7] Edmund M. Clarke, Somesh Jha, Yuan Lu, and Helmut Veith. Tree-like counterexamples in model checking. In *Proc. of the IEEE Symposium on Logic in Computer Science (LICS 2002)*, Copenhagen, Denmark, 2002.

[8] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. Tool description presented at CAV 2004. volume 3114 of *LNCS*, pages 496–500. Springer, 2004.

[9] Yifei Dong, C. R. Ramakrishnan, and Scott A. Smolka. Model checking and evidence exploration. In *Proceedings of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03)*, pages 214–223, Huntsville Alabama, USA, 2003.

[10] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Formal Models and Semantics*, pages 996–1072. Elsevier, 1990.

[11] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135, 2003.

[12] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *Software Tools for Technology Transfer (STTT)*, 2005.

[13] Alex Groce, Daniel Kroening, and Flavio Lerda. Understanding counterexamples with explain. In R. Alur and D. A. Peled, editors, *Proceedings of 16th International Conference on Computer Aided Verification 2004*, volume 3114 of *LNCS*, pages 453–456. Springer, 2004.

[14] Arie Gurfinkel and Marsha Chechik. Proof-like counterexamples. In *Proceedings of the 9th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 160–175. Springer-Verlag, 2003.

[15] Sylvain Halle, Roger Villemaire, and Omar Cherkaoui. Specifying and validating data-aware temporal web service properties. *IEEE Transactions on Software Engineering*, 35(5):669–683, 2009.

[16] Henri Hansen and Jaco Geldenhuys. Cheap and small counterexamples. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM '08)*, pages 53–62, Washington, DC, USA, 2008. IEEE Computer Society.

[17] Mirjana Jakšić and Burkhard Freitag. Temporal patterns for document verification. In *Proc. of the 6th Intl. Workshop on Automated Specification and Verification of Web Systems (WWV'10)*, Vienna, Austria, 2010.

[18] K. L. McMillan, O. Grumberg, X. Zhao, and E. M. Clarke. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd ACM/IEEE Conference on Design Automation Conference (DAC'95)*, pages 427–432, San Francisco, CA, USA, 1995.

[19] Shin Nakajima. Model-checking behavioral specification of BPEL applications. *Electronic Notes in Theoretical Computer Science*, 151:89–105, 2006.

[20] Kavita Ravi and Fabio Somenzi. Minimal assignments for bounded model checking. In *Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *LNCS*, pages 31–45, Barcelona, Spain, 2004. Springer-Verlag.

[21] Christian Schönberg, Mirjana Jakšić, Franz Weitl, and Burkhard Freitag. Verification of web-content: A case study on technical documentation. In *Proceedings of the 5th International Workshop on Automated Specification and Verification of Web Systems (WWV'09)*, Linz, Austria, 2009.

[22] Christian Schönberg, Franz Weitl, and Burkhard Freitag. Verifying the consistency of web-based technical documentations. *Journal of Symbolic Computation, Special Issue on Automated Specification and Verification of Web Systems*, 2010.

[23] Christian Schönberg, Franz Weitl, Mirjana Jakšić, and Burkhard Freitag. Logic-based verification of technical documentation. In *Proceedings of the 9th ACM Symposium on Document Engineering (DocEng 09)*, pages 251–252, Munich, Germany, 2009. ACM.

[24] ShengYu Shen, Ying Qin, and SiKun Li. Minimizing counterexample with unit core extraction and incremental SAT. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, volume 3385 of *LNCS*, pages 298–312, Paris, France, 2005. Springer-Verlag.

[25] Franz Weitl. *Document Verification with Temporal Description Logics*. PhD thesis, Univ. of Passau, 2008.

[26] Franz Weitl, Mirjana Jakšić, and Burkhard Freitag. Towards the automated verification of semi-structured documents. *Journal of Data & Knowledge Engineering*, 68:292–317, 2009.

[27] Franz Weitl, Shin Nakajima, and Burkhard Freitag. Structured counterexamples for the temporal description logic $\mathscr{ALC}\mathsf{CTL}$. In *Proceedings of the 8th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2010)*, Pisa, Italy, 2010.