Automated Support for Building Behavioral Models of Event-Driven Systems

Benet Devereux and Marsha Chechik

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada
{benet,chechik}@cs.toronto.edu

Abstract. Programmers understand a piece of software by building simplified mental models of it. Aspects of these models lend themselves naturally to formalization – e.g., structural relationships can be partly captured by module dependency graphs. Automated support for generating and analyzing such structural models has proven useful. For event-driven systems, behavioral models, which capture temporal and causal relationships between events, are important and deserve similar methodological and tool support. In this paper, we describe such a technique. Our method supports building and elaboration of behavioral models, as well as maintaining such models as systems evolve. The method is based on model-checking and witness generation, using strategies to create goal-driven simulation traces. We illustrate it on a two-lift/three-floor elevator system, and describe our tool, Sawblade, which provides automated support for the method.

1 Introduction

Programs larger than a few tens of lines are generally far too complex to be understood in full by a single person. In place of complete understanding, programmers use simplified mental models [18] – a representation of some part of the program's structure or function at a high enough level of abstraction to be readily understood. One heuristic is that they should be small enough to fit on a whiteboard [25]. Mental models are informal and cannot always be completely expressed by a formal structure; however, it has often been found useful to create formal structures based upon programmers' mental models and to use them to aid construction and understanding of code. Some examples of mental models used by programmers, and their corresponding formal artifacts, are discussed below.

Modules and dependencies. Decomposition into modules that communicate via interfaces is standard software engineering practice. Module A depends on module B if any of the functions in A use a function or data defined in B. Considerable research has gone into automatically extracting a graph recording all dependencies between modules from the source code [8, 23, 24], helping programmers navigate this graph [13] or select fragments that are most relevant to a particular aspect [28].

Design patterns and architectural patterns. Design patterns describe and catalogue common relationships between groups of objects in object-oriented programs [14]. Design patterns have a formal representation as fragments of object modelling graphs; formalizing the observations has led to faster program comprehension and better communication between programmers. Work on extracting design patters from source code has also been done [21]. *Architectural patterns* relate components-and-connectors type diagrams to standard styles of decomposition, such as layers or pipes-and-filters. There is research on automated exploration of architecture and automatic comparison with a conceptual pattern [25, 13].

The models described above are mostly *structural*. However, *behavior* is as important a part of understanding a program as structure, particularly for reactive concurrent systems. In this paper, we address the problem of providing tool support for constructing behavioral models of such complex systems.

Global propositions about a system's behavior can be expressed using temporal logic [22] and automatically verified using model-checking [9]. The strength of this method is that it can be used both for assertions about *some* behavior and about *all possible* behaviors. It has two weaknesses: expressing properties can become difficult, though property patterns [12] help tame this difficulty; and, more importantly, even using property patterns, it is very hard to *guess* which properties might be both valid and useful for understanding.

Query-checking [5] is a technique for searching for interesting temporal logic properties. It enables answering user questions such as "What property P is true everywhere?" or "If event X happens, what property Q eventually becomes true sometime after X?" However, query-checking requires considerable intervention and technical knowledge. Furthermore, its output can be a large propositional formula, which is hard to interpret intuitively.

A potential candidate for behavioural models is *scenarios* [20]. Scenarios have been shown to be useful for expressing requirements and for communicating between stakeholders [16, 19]. Scenarios can capture not only sequences of events that the system allows, but also those that it prohibits, exact causal relationship between events, etc. Running the program – with the aid of a simulator or test-driver to provide inputs – generates a large number of scenarios which are certainly true, but do not yield the kind of simple, general knowledge about the system's behavior we would like in building a mental model. They lack any notion of causation between events, or of necessity or impossibility of sequences of events. These richer concepts are essential for behavioral models that support understanding and evolution of the system.

Our goal is to bridge this gap: to provide a methodology and tool for finding and validating rich scenarios that describe not just sequences of events but causal relationships between them. We need to be able to vary the level of *granularity* of scenarios – what events we distinguish – and also the *scope*, to ignore actions of parts of the system that are not considered relevant.

Furthermore, we want to go beyond simply finding and validating such scenarios: our methodology aims to help the user in elaborating scenarios by finding others which are stronger, or more detailed. Finally, since a major use of mental models is during software evolution, our methodology must also help change these scenarios along with evolving systems.

Contributions. In this paper, we propose a methodology and tool based on temporal logic, model-checking, and witness generation. Our techniques do not work directly on the program code; instead, we assume that a finite-state model of the program has been constructed, e.g., using the techniques of [11, 1], and that this model is small enough to be analyzable by existing model-checkers.

We illustrate our methodology using a two-lift/three-floor elevator system [2]. Rather than requiring direct use of temporal logic, our method uses a simple language of events and causal relationships between them. This language itself is not new: its features are drawn from property patterns and use-case maps [4]. The modeling language can be used to express scenarios, and a translation into temporal logic is used for automatic validation by a model-checker. Once scenarios have been found and validated, they can be elaborated. We describe a set of patterns for moving from a validated scenario to stronger and richer scenarios. Application of these patterns relies on generating the most useful traces of the program that help the user guess more elaborate scenarios. The notion of a useful trace is user-specified, and this specification is used by the witness-generation component of the model-checker to carry out a strategy-directed search for interesting traces.

Maintaining scenarios across change is done by representing change as an annotation of the new system, indicating how its state transitions have changed from the old. Once this is done, useful traces – in this case, those that highlight most effectively and minimally the differences in behavior, where they exist – can be searched for by the model-checker using strategies as well.

Support for this method is provided by our tool Sawblade, built on top of a model-checker XChek [6].

Structure. The rest of this paper is organized as follows: in Section 2, we give background material on temporal logic and model-checking. In Section 3, we present a language for scenario-like behavioural models and its translation into temporal logic. We also discuss the elevator system which is the running example in this paper. In Section 4, we describe witness generation and strategies for helping produce the "most interesting" witnesses. In Section 5, we describe the methodology for elaborating scenarios. In Section 6, we give our formal definition of the annotation of a changed system, and in Section 7, describe the methodology for transforming scenarios across change. We describe Sawblade, a tool supporting this methodology, in Section 8 and conclude the paper in Section 9.

2 Background

In this section, we review the basics of temporal logic model-checking, presenting the semantics of the temporal logic CTL and the definition of witnesses for existential CTL properties.

Analysis of data-driven, run-to-completion programs is predicative, examining the relation between the program's input and output. Analysis of a reactive program, however, must examine the infinite behaviours of the program, and how its behaviours are affected by input from its environment. Temporal logic is helpful for intuitively expressing properties of infinite behaviours, and, for finite-state models, *model-checking* provides a useful tool for automatically deciding satisfaction of temporal logic properties by those models.

Kripke Structures and CTL. A *Kripke structure* is an abstract model of a reactive system. Formally, it is a tuple (S, s_0, R, I, V) where S is a (finite) set of states; s_0 is

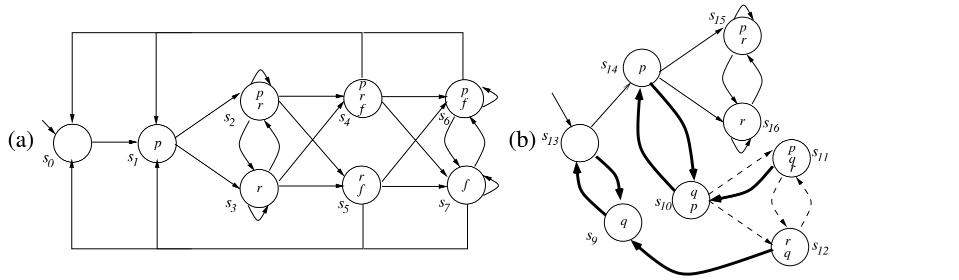


Fig. 1. (a) A Kripke structure; (b) A Kripke structure with `diff`, based on a fragment of the model in (a).

the initial state; $R \subseteq S \times S$ is the transition relation; V is a set of atomic propositions; $I : S \to 2^V$ is a labeling function that associates each state with the atomic propositions true in that state.

An example Kripke structure is shown in Figure 1(a). In this model, $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$, $V = \{p, r, f\}$, and $(s_0, s_1) \in R$. Atomic propositions not shown in a state are assumed to be false, e.g., p, r and f are false in s_0.

For each $s \in S$, the transition relation defines a *successor set* $Img(s) = \{t \mid R(s,t)\}$ of states reachable in one step from s; the *predecessor set* is $Img^{-1}(s) = \{t \mid R(t,s)\}$. A *path* p is an infinite sequence $p_0 p_1 p_2 \ldots$ of states. The set of paths $\mathcal{P}(s)$ of a state s in some Kripke structure M contains all the infinite sequences of states possible in M: $p \in \mathcal{P}(s) \Leftrightarrow p_0 = s \wedge \forall i \in \mathbb{N} \cdot p_{i+1} \in Img(p_i)$.

Computation Tree Logic (CTL) [10] is a temporal logic used to state properties of the (infinite) paths of Kripke structures. The set of CTL formulas over a set of atomic propositions (variables) V consists of the sentences defined by the following grammar:

$$C ::= p \in V \mid \neg C \mid C \wedge C \mid C \vee C \mid \mathbf{EX}\ C \mid \mathbf{AX}\ C \mid \mathbf{E}[C\ \mathbf{U}\ C] \mid \mathbf{A}[C\ \mathbf{U}\ C] \mid \mathbf{EF}\ C \mid \mathbf{AF}\ C \mid \mathbf{EG}\ C \mid \mathbf{AG}\ C$$

The symbols $\mathbf{AX}$, $\mathbf{EG}$, etc., are called temporal operators. The $\mathbf{A}$ or $\mathbf{E}$ indicates whether the following symbol is to be interpreted over all future paths, or some future paths; $\mathbf{X}$ stands for "next", $\mathbf{F}$ for "future", $\mathbf{U}$ for "until", $\mathbf{G}$ for globally; thus $\mathbf{AX}\varphi$ means "in all next states, φ holds", and $\mathbf{EF}\psi$ means "there is a future path along which, at some point, ψ holds".

The CTL satisfaction relation $\models$ is defined between states of a Kripke structure and CTL formulas. Its definition is given in Figure 2(a). Note that only $\mathbf{EX}$, $\mathbf{EU}$ and $\mathbf{EG}$ are presented. Others can be derived from these via simple identities [9]. For instance, $\mathbf{EF}\varphi \Leftrightarrow \mathbf{E}[true\ \mathbf{U}\ \varphi]$. Also note the operator $\mathbf{EU}_i$; informally, $\mathbf{E}[\varphi\ \mathbf{U}_i\ \psi]$ means that there exists a path along which ψ becomes true *no later than* at step i, and until that point, φ holds.

For instance, in the structure of Figure 1(a), we can ask whether it is possible to reach a state where f holds: $s_0 \models \mathbf{EF}\ f$. One such state is s_5, so the property holds.

Figure 2(b) shows some useful CTL identities which will be used later on. They are straightforward consequences of the semantics. Note that $\mathbf{EF}_i\varphi \Leftrightarrow \mathbf{E}[true\ \mathbf{U}_i\ \varphi]$.

The set of all states in a model M which satisfy a given property φ is denoted by $[\![\varphi]\!]^M$, or just $[\![\varphi]\!]$ when the model is implicit. The semantics of CTL can be expressed entirely in terms of $[\![\cdot]\!]$ rather than quantification over paths. For instance, $[\![\mathbf{EX}\varphi]\!] = \{Img^{-1}(s) \mid s \in [\![\varphi]\!]\}$, and $\mathbf{EF}\varphi$ is the least fixed-point of Img^{-1} applied to $[\![\varphi]\!]$.

$$
\text{(a)}\quad
\begin{aligned}
s \models p &\Leftrightarrow p \in I(s)\\
s \models \neg\varphi &\Leftrightarrow s \not\models \varphi\\
s \models \varphi \wedge \psi &\Leftrightarrow s \models \varphi \wedge s \models \psi\\
s \models \mathbf{EX}\varphi &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot p_1 \models \varphi\\
s \models \mathbf{EG}\varphi &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \forall i \cdot p_i \models \varphi\\
s \models \mathbf{E}[\varphi \mathbf{U} \psi] &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \exists i \cdot (p_i \models \psi) \wedge\\
&\qquad \forall j < i \cdot p_j \models \varphi\\
s \models \mathbf{E}[\varphi \mathbf{U}_i \psi] &\Leftrightarrow \exists p \in \mathcal{P}(s) \cdot \exists j \leq i \cdot (p_j \models \psi) \wedge\\
&\qquad \forall k < j \cdot p_k \models \varphi
\end{aligned}
\qquad
\text{(b)}\quad
\begin{aligned}
&\mathbf{EF}\varphi \Leftrightarrow \varphi \vee \mathbf{EX}\ \mathbf{EF}\varphi\\
&\mathbf{EF}_0\varphi \Leftrightarrow \varphi\\
&\mathbf{EF}_i\varphi \Leftrightarrow \mathbf{EX}\ \mathbf{EF}_{i-1}\varphi \ \text{ if } i > 0\\
&\mathbf{EF}_i\varphi \Rightarrow \mathbf{EF}\varphi \ \text{ for all } i\\
&\mathbf{EG}\ \varphi \Leftrightarrow \varphi \wedge \mathbf{EX}\ \mathbf{EG}\ \varphi
\end{aligned}
$$

Fig. 2. (a) Semantics of CTL. (b) Useful CTL identities.

Witnesses to CTL Properties. In first-order logic, an existential assertion $\exists x \cdot Q(x)$ can be proven by exhibiting a *witness* – an element in the domain of the predicate Q which makes Q true. Since CTL properties are expressed in a fragment of first-order logic, this proof method can be applied to them as well. For example, a witness for the property $\mathbf{EF}f$ in the model of Figure 1(a) is s_0, s_1, s_3, s_5.

Any CTL property whose semantics is entirely expressible using existential quantifiers where the negation is pushed to the level of atomic propositions, can be proven by exhibiting an instantiation for all the existential quantifiers over paths. Though CTL semantics is expressed over infinite paths, a witness is always made up of finite paths or finite prefixes followed by finite repeating suffixes [9]. For example, the witness for $s_0 \models \mathbf{EX}p$ for the model in Figure 1(a) is a two-step path, where the second step is a successor t such that $(s_0, t) \in R$ (state s_1 in our example). The witness for $s_1 \models \mathbf{EG}p$ is infinite, and, in the case of the model in Figure 1(a), consists of a loop $s_1, s_2, s_4, s_1, ...$ In the rest of the paper, we use "witness" to refer to *either* the necessary finite segments or to infinite paths that begin with such segments; the correct interpretation will be clear from the context. Also, we only consider properties *linear witnesses* [3], i.e., a single path through the states of the model that suffices as a proof. This restriction is for the sake of simplicity of presentation, and is not a constraint on the method discussed; our results can also be extended to witnesses with branching structure [15].

Determining whether a CTL formula has a linear witness is NP-hard [3]; a sublanguage of CTL which always has linear witnesses is given by the following grammar:

$$
\begin{aligned}
A &::= p \in V \mid \neg A \mid A \wedge A \mid A \vee A\\
T &::= A \mid \mathbf{EX}T \mid \mathbf{EF}T \mid \mathbf{E}[A\ \mathbf{U}\ T] \mid T \vee T
\end{aligned}
$$

For example, the witness to $\mathbf{E}[r\ \mathbf{U}\ \mathbf{EX}p]$ in state s_0 of the model in Figure 1(a) is linear, whereas the witness to $\mathbf{EX}p\ \wedge\ \mathbf{EX}\neg p$ in state s_1 is not.

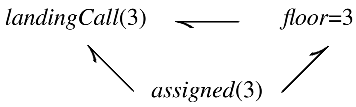
A *counterexample* is a witness to the *negation* of a property. Let φ be a formula with a witness, and let $\psi = \neg\varphi$. If ψ does not hold in some state s, then a counterexample to ψ can be computed; further, if φ has a linear witness, then ψ has a linear counterexample.

3 Scenario Language

In this section, we describe the syntax and semantics of a simple language of scenarios. The language allows expression of causal relationships between events, under qualify-

Events	
init	the elevator is started up
landingCall(3)	there is a call for an elevator on floor 3
liftCall(3)	there is a call inside an elevator for floor 3
E_1.*floor* = 1	elevator 1 arrives on floor 1
floor= 2	*either* elevator arrives on floor 2
assigned(3)	the controller assigns a call to one of the elevators.
Conditions	
E_1.*up*	elevator 1 is moving up
outstandingCall(3)	there is an unserviced call for floor 3

(a)



(b)

Fig. 3. (a) Some elevator events and conditions; (b) An example graphical expression.

ing conditions. Its semantics is a translation into CTL, so that scenarios can be automatically validated.

To illustrate the concepts in this paper, we use a simple two-lift, three-floor elevator system with a central controller. Each of the elevators, E_1 and E_2, can be standing still, or moving up or down; its door can be open or closed. It has a record of the floors it is still obliged to visit – an elevator E_i must visit a floor if either (1) its internal button for that floor was pressed, or (2) the controller received a call from a landing-button on that floor and assigned E_i to service it. The controller assigns calls to elevators based on a heuristic estimating which will arrive first.

3.1 Syntax

The basic entities of mental models of behaviour are *conditions* – the state of a program spanning some nonzero number of steps in time – and *events* – changes between one state and the next [17]. Some of the events and conditions of the elevator system are shown in Figure 3(a). Note that events do not have to be independent of each other, e.g., *floor*=2 $\Leftrightarrow$ (E_1.*floor*=2 $\vee$ E_2.*floor*=2).

The fundamental relationships between events are temporal and causal. We consider the following to be the atomic relationships between events A and B:

$A \leadsto B$ A and B can happen, and B can follow A.
$A \rightharpoonup B$ A and B can happen, and if A happens, then B must happen sometime in the future.
$A \leftharpoondown B$ A and B can happen, and if B happens, then A must have happened prior to B.

Also, $A \rightleftharpoons B$ means that both $A \rightharpoonup B$ and $A \leftharpoondown B$. Composition of relationships is transitive: writing $A \rightharpoonup B \rightharpoonup C$ means that $A \rightharpoonup B$ and $B \rightharpoonup C$. For example, the graphical expression shown in Figure 3(b) denotes (*landingCall(3)* $\leftharpoondown$ *floor*=3) $\wedge$ (*landingCall*(3) $\leftharpoondown$ *assigned*(3)) $\wedge$ (*assigned*(3) $\rightharpoonup$ *floor*=3). That is, an elevator arrives at floor 3 only because of a call to floor 3. Also, floor 3 is assigned to an elevator only because of a call; and assignment of a floor always causes its service by an elevator.

Causal relationships can be *absolute* or *conditional*: either $A \rightharpoonup B$ in any case, or $A \rightharpoonup B$ while a condition c is satisfied, that is, if A happens while condition c is true, then *either* B eventually happens *or*, before that, c becomes false. We denote this by $c[A \rightharpoonup B]$. This situation is called an *exception*. In addition, we want to allow representation of exceptions which are due to *events*. If A leads to B unless event C happens, we write $(A \rightharpoonup B) \downarrow C$. We can further generalize this by defining *scopes* as

Scenario	CTL translation	W/C
$B \rightsquigarrow C$	$\mathbf{EF}(B \wedge \mathbf{EF}\ C)$	W
$B \rightharpoonup C$	$\mathbf{AG}(B \Rightarrow \mathbf{AX}\ C)$	C
$B \leftharpoondown C$	$\mathbf{AG}(init \vee C \Rightarrow$ $(\mathbf{AG}\neg C \vee \mathbf{A}[\neg C\ \mathbf{U}\ B]))$	–
$c[B \rightharpoonup D]$	$\mathbf{AG}(B \wedge c \Rightarrow \mathbf{A}[c\ \mathbf{U}\ D \vee \neg c])$	C
$(B \rightharpoonup C) \downarrow E$	$\mathbf{AG}(B \wedge \neg E \Rightarrow \mathbf{AF}(C \vee E))$	C

Table 1. CTL semantics of atomic scenarios.

in the temporal logic patterns framework [12, 27], e.g., $A \rightharpoonup B$ between (conditions or events) P and Q.

3.2 Translation

Since Kripke structures deal only with propositions, and not events, we must explicitly encode events as changes of state. For instance, in the elevator, *floor*=1 is a state variable: the elevator *arrives* at floor 1 when this variable *becomes* true. We assume that the structure, where needed, is annotated with event variables, which become true for a single time-step whenever the event occurs, and false once it ceases to occur.

Since any scenario expression can be represented by a conjunction of atomic binary expressions, we only describe the translation of atomic expressions into CTL.

If $B \rightsquigarrow C$, then there are three properties to be checked: (1) B can occur; (2) C can occur; (3) C can follow B. These are subsumed by determining whether there exists some path from the initial state along which B occurs at some point; and whether, once B occurs, C may occur. Formally, $B \rightsquigarrow C = \mathbf{EF}(B \wedge \mathbf{EF}\ C)$. For instance, we can check *init* $\rightsquigarrow$ (*floor*=3) by asking the model-checker whether $\mathbf{EF}$(*init* $\wedge$ $\mathbf{EF}$ *floor*=3) holds. Since *init* is necessarily true of s_0, this reduces to $\mathbf{EF}$ *floor*=3.

The translation of the remaining constructs into CTL is shown in Table 1. **W/C** indicates whether the translation has a linear witness (W), a linear counterexample (C), or neither (–). If $B \rightharpoonup C$, then not only can both B and C occur, but when B occurs, then C *must* occur at some point after it. As an example, we can ask whether *call*=3 $\rightharpoonup$ *floor*=3 is a valid scenario; the model-checker determines whether for any state where *call*=3 occurs, each future path eventually reaches a state where *floor*=3. If $B \leftharpoondown C$, then either C never occurs, or on any path where C does occur, between the initial state and the first occurrence of C, and between any two occurrences of C, there is an occurrence of B. $c[B \rightharpoonup D]$ means that if B occurs *while* c is true, then along all future paths, c holds until either D occurs, or c becomes false (the exception condition). For $(B \rightharpoonup C) \downarrow E$, B must either lead to an occurrence of C or an occurrence of E; C must be possible, but E need not be.

Once relationships between events are discovered, they can be immediately validated by the model-checker. This leaves open, however, the question of how to find such relations and how to make them more precise – elaborate them. We address this issue in Section 5.

4 Witnesses and Strategies

In this section, we discuss how to define strategies for constructing "interesting" witnesses. We also discuss optimality of a witness-generation strategy with respect to objectives which may not be expressible as part of CTL.

4.1 Witness Generation

In Section 2, we defined witnesses. We now discuss their effective computation.

We start by defining *annotated witnesses*. An annotated witness is a sequence π of pairs $(\pi_0, \Phi_0), (\pi_1, \Phi_1), \ldots$ where π_i is a state and Φ_i a set of CTL formulas. The formulas Φ_i are *proof-obligations* – informally, properties which, at step π_i, still need to be demonstrated by the witness. For example, the annotated witness to $s_0 \models \mathbf{EF}\ f$ for the model in Figure 1(a) is

$$(s_0, \{\mathbf{EF}\ f\}) \rightarrow (s_1, \{\mathbf{EF}_2\ f\}) \rightarrow (s_2, \{\mathbf{EF}_1\ f\}) \rightarrow (s_5, \{f\})$$

For each state in this witness, we only show a singleton set of proof obligations, although others are possible as well (e.g., $\{\mathbf{EF}\ f, \mathbf{EF}_2\ f\}$ in state s_1). An (infinite) annotated witness to $s_1 \models \mathbf{EG\ p}$ is

$$(s_1, \{\mathbf{EG}p, p, \mathbf{EX\ EG}\ p\}) \rightarrow (s_2, \{\mathbf{EG}p, p, \mathbf{EX\ EG}\ p\}) \rightarrow s_2 \cdots$$

which moves from s_1 to s_2 and then loops infinitely on s_2. The labels are based on the identity $\mathbf{EG}\ p \Leftrightarrow p \wedge \mathbf{EX\ EG}\ p$; see Figure 2(b) for CTL identities.

An annotated witness w to $s \models \psi$ satisfies the following conditions: (1) $\pi_0 = s$, and the conjunction of the formulas in Φ_0 implies ψ: $\psi \Leftarrow \bigwedge_{\varphi_i \in \Phi_0} \varphi_i$; (2) for every state (π_i, Φ_i) in w, $\pi_i \models \varphi_i$ for each $\varphi_i \in \Phi$; (3) for every step $(\pi_i, \Phi_i) \rightarrow (\pi_{i+1}, \Phi_{i+1})$ in the witness, let Φ_i^t be the subset of Φ_i containing temporal operators. Then for every $\varphi_j \in \Phi_i^t$, there is $\varphi'_j \in \Phi_{i+1}$ such that $\mathbf{EX}\ \varphi'_j \Rightarrow \varphi_j$. If the witness is finite, as in the case of $\mathbf{EF}$, the proof obligation for the last step (π_k, Φ_k) does not include any temporal operators.

A sequence of annotated states is a *partial witness* if properties (1) and (2) of witnesses hold in it, and property (3) holds for every state except the last. Particularly, $(s, \{\varphi\})$ is always a partial witness for $s \models \varphi$ if this property holds in the model. Thus, using the model-checker's results cached from the computation of $[\![\varphi]\!]$, we can compute a complete witness starting from $(s, \{\varphi\})$, extending it one step at a time until either a final state is reached or a cycle can be closed. More precisely, given a partial witness with (π, Φ) as the last state, we compute the extension (π', Φ') so that (1) $\forall \varphi_i \in \Phi^t \cdot \exists \varphi'_i \in \Phi' \cdot \mathbf{EX}\ \varphi'_i \Rightarrow \varphi_i$, and (2) choose $\pi' \in Img(s) \cap \bigcap_{\varphi_i \in \Phi^t} [\![\varphi'_i]\!]$. That is, π must witness $\mathbf{EX}\varphi'_i$ for *every* temporal $\varphi_i \in \Phi$. The choice of a suitable π' is made by a *witness generation strategy* [7]. This is the (tableau-based) technique used by our model-checker XChek [6, 15], and it allows simple local specification of strategies. Clearly, other techniques are possible as well.

4.2 Strategies

Strategies are procedures for choosing which witness to show to the user, in case several are possible. A simple strategy, used by most model-checkers, is to compute the shortest possible witness (**Shortest**). For a finite witness property, such as $s \models \mathbf{EF}\varphi$, this strategy uses the identity $\mathbf{EF}\varphi \Leftrightarrow \exists i \cdot \mathbf{EF}_i\varphi$, and selects as the initial partial witness $(s, \{\mathbf{EF}_i\ \varphi\})$ by finding the least i such that $s \models \mathbf{EF}_i\ \varphi$. At each extension step, it chooses a successor for $(\pi, \{\mathbf{EF}_i\ \varphi\})$ by determining the *smallest* $j < i$ such that some $\pi' \in Img(\pi)$ satisfies $\mathbf{EF}_j\varphi$, and choosing any such π'. If $j = 0$, then $\mathbf{EF}_0\varphi$ is rewritten to the purely propositional φ, and the strategy halts successfully. This is how the witness to $s_0 \models \mathbf{EF}\ f$ in Section 4.1 was generated: $s_0 \models \mathbf{EF}_3\ f$; the least $i < 3$ than can be chosen is 2, and s_1 is the only choice. From $(s_1, \{\mathbf{EF}_2\ \varphi\})$, the least possible $i < 2$ is 1; s_2, s_3 are equally valid choices, so the strategy picks s_2 at random; finally the path is extended to s_5, which satisfies f ($\mathbf{EF}_0\ f$), and so the strategy halts.

For $\mathbf{EG}\varphi$, the strategy builds a (shortest) path until it reaches a state t which lies on a cycle: that is, there is a path from t which reaches t again, and φ holds continuously on this path. Since t satisfies $\varphi \wedge \mathbf{EX}\ \mathbf{E}[\varphi\ \mathbf{U}\ \{t\}]$, there is a finite path from t back to t, and once this has been constructed, the witness, consisting of a finite prefix followed by the cycle on t, is complete.

In this paper, we consider several strategies. We describe them informally here; for a more formal treatment, please see [7]. At each step, the set of possible successors is partitioned into *preferred* (P) and *avoided* (A); if the preferred set is nonempty, then the next state is chosen from it nondeterministically; otherwise, the next state is chosen from A. P and A can be the same throughout the construction of the witness, or can be updated. Clearly, this approach is *greedy*: decisions are made locally, and thus we may not generate the most interesting witness. Strategies with backtracking can also be defined, but their application is more expensive.

Avoid-Visited. This strategy uses the avoid set A that consists of previously visited states. The set is updated after the next state of the witness is chosen. For example, a sequence of states forming a witness to $\mathbf{EX}\ \mathbf{EF}r$ in state s_2 of the model in Figure 1(a) is s_2, s_3, or s_2, s_4 but not s_2, s_2.

Avoid-States. This strategy is similar to **Avoid-Visited**. However, it receives a set of states to avoid as a parameter, and does not update this set as the witness gets constructed.

Avoid-Conditions. This strategy is similar to **Avoid-States**, but its parameter-list consists of conditions on the next state to avoid.

Avoid-Events. This strategy receives a list L of events to avoid. Given the last state s of the partial witness, it tries to pick a successor t so that none of the events of L occur between s and t. The avoidance set stays the same throughout the witness generation process. We can further define a strategy that picks a successor that *minimizes* the number of events that fire on the transition between s and t.

Avoid-Vars. This is similar to **Avoid-Events**: given a set of variables L, the strategy extends the current partial witness by choosing the successor that does not change variables in L. We can further define a strategy that picks a successor that *minimizes* changes to variables in L.

Clearly, we can define **Prefer** counterparts of the above strategies. For example, **Prefer-Visited** extends the partial witness by preferring states which are already part of this witness.

5 Elaboration of Behavioral Models

In Section 3, we discussed specifying and validating simple scenarios. Since automatic extraction of interesting scenarios from the system is difficult (because scopes and events of interest need to be specified and because mental models are an abstraction of the behaviour of the system – they typically ignore exceptional cases), our methodology works by starting from simple scenarios that are guessed by the user, and elaborating them into more complex scenarios using *elaboration patterns*. Guessing simple scenarios is not hard – we can start just with determining that a certain event p is possible, without worrying about what caused it.

Pattern	Before	Strategies	After
Cause	$A \rightharpoonup B$ $\surd$	**Avoid-Events**	$A \vee C \rightleftharpoons B$
Weakening	$A \leftharpoondown B$ $\times$		
Event	$A \rightharpoonup B$ $\surd$	**Shortest**,	$c[A \rightharpoonup B_1]$,
Splitting	$B = B_1 \vee B_2$ $\surd$	**Avoid-States**	$c'[A \rightharpoonup B_2]$
	$A \leftharpoondown B_1$ $\surd$		*or*
	$A \leftharpoondown B_2$ $\surd$		$A_1 \rightharpoonup B_1$,
	$A \rightharpoonup B_1$ $\times$		$A_2 \rightharpoonup B_2$,
	$A \rightharpoonup B_2$ $\times$		$A = A_1 \vee A_2$
Intermediate	$A \rightharpoonup B$ $\surd$	**Shortest**,	$A \rightharpoonup C \rightharpoonup B$
Event		**Avoid-Vars**	
Intermediate	$A \rightharpoonup B$ $\surd$		$A \leftharpoondown D$,
Cause	$A \leftharpoondown B$ $\times$		$D \leftharpoondown B$

Table 2. Elaboration patterns.

An elaboration pattern represents a typical way in which behavioral understanding moves from a set of valid and invalid scenarios – the *base scenarios* – to stronger or richer ones – the *elaborated scenarios*. This movement usually involves enriching the current vocabulary of events of interest or strengthening the relationship between the existing events. Elaboration patterns help narrow down the focus of investigation, and determine which witnesses would be most useful for elaborating the current scenario; this in turn suggests the witness-generation strategy that should be applied. Application of an elaboration pattern does *not* guarantee the existence or the utility of an elaborated scenario of the desired form.

In this section, we describe elaboration patterns which we found useful for building behavioural models. Several of these are summarized in Table 2.

Cause Weakening. Suppose we start with a validated scenario $A \rightharpoonup B$ (A causes B), whereas $A \leftharpoondown B$ (B can only happen after A) is not valid. Thus, we cannot conclude $A \rightleftharpoons B$. Our goal is to determine an event C such that C causes A and further $C \Rightarrow A$, so that the elaborated scenario is $A \vee C \rightleftharpoons B$. To find C, we might want to examine the causes of failure of $A \leftharpoondown B$, but the counterexample to this property is non-linear and thus may not provide the necessary understanding. Instead, we propose to examine which events other than A cause B; so the useful witnesses in this case are generated by checking $init \rightsquigarrow B$ using an **Avoid-Events**($\{A\}$) strategy. Examining these witnesses helps us guess which events need to be added to C; with each successful guess, C is increased (weakened) until we can conclude $(A \vee C) \rightleftharpoons B$.

As an example of **Cause Weakening**, consider the relationship between *landingCall*(3) and *floor*=3 in the elevator system. *landingCall*(3) $\rightharpoonup$ *floor*=3, but it is not true that *landingCall*(3) $\leftharpoondown$ *floor*=3. Using the elaboration pattern, we compute a witness to *init* $\rightsquigarrow$ *floor*=3, avoiding *landingCall*(3), which results in *init*, E_1.*liftCall*(3), E_1.*assigned*(3), E_1.*doorClosed*, E_1.*floor*=2, E_1.*floor*=3. Examining this trace allows us to identify *liftCall*(3) as another possible cause of *floor*=3. We can quickly validate that *liftCall*(3) $\rightharpoonup$ *floor*=3; and furthermore, that $($*liftCall*$(3) \vee$ *landingCall*$(3))$ $\leftharpoondown$ *floor* $= 3$. The new event *liftCall*(3) $\vee$ *landingCall*(3) is called *call*(3), and *call*(3) $\rightleftharpoons$ *floor*=3.

Event-Splitting. Suppose we start with $A \rightleftharpoons B$, where B is a compound event $B \Leftrightarrow B_1 \vee B_2$. Thus, $A \rightharpoonup B_1 \vee B_2$ and $A \leftharpoondown B_1 \vee B_2$. We can prove $A \leftharpoondown B_1$ and $A \leftharpoondown B_2$, but neither $A \rightharpoonup B_1$ nor $A \rightharpoonup B_2$. We are interested in causes of B_1 and B_2. Potential elaborations can be some conditions under which $A \rightharpoonup B_i$, or perhaps splitting up A so that $A \Leftrightarrow A_1 \vee A_2$ and $A_1 \leftharpoondown B_1$ and $A_2 \leftharpoondown B_2$; finally, we may conclude that A leads to a non-deterministic choice between B_1 and B_2. We first examine counterexamples to $A \rightharpoonup B_i$, perhaps with the **Shortest** strategy. If this is not helpful, we suggest the following tactic: examining $A \leadsto B_1$, the existential counterpart of $A \rightharpoonup B_1$. A does not always result in B_1, but checking $A \leadsto B_1$ lets us examine cases where it does. Let V_1 be the set of states visited while generating a counterexample for $A \rightharpoonup B_1$. We can generate a witness to $A \leadsto B_1$ with the strategy **Avoid-States**(V_1); this avoids accidental similarities between paths from A to B_1 and those from A to B_2, and helps with the elaboration.

We show an application of **Event-Splitting** in the elevator system by studying the relationship between a call to floor 3 and the arrival of a given elevator to that floor. The event *floor=3* is composed of the events E_1.*floor*=3 and E_2.*floor*=3. *call*(3) $\leftharpoondown$ E_1.*floor*=3, and *call*(3) $\leftharpoondown$ E_2.*floor*=3; however, *call*(3) $\rightharpoonup$ E_1.*floor*=3 is not valid. A witness to φ=*call*(3) $\leadsto$ E_1.*floor*=3 shows a lift-call for E_2 (which is a sub-event of *call*(3)), followed by E_2.*floor*=3. We validate E_2.*liftCall*(3) $\rightharpoonup$ E_2.*floor*=3, and ask for another witness to φ, using the strategy **Avoid-Event**($\{E_2$.*liftCall*(3)$\}$). This yields the following trace: *init*, *landingCall*(3), E_2.*assignCall*(3), etc., until E_2 reaches floor 3. Thus, we observe that if both elevators are on floor 1, the assignment of calls appears to be nondeterministic.

To find a better reason, we use the **Avoid-States**(V_1) strategy, where V_1 is the set of states in the previous witness. This results in the following sequence of events: there is a call for floor 2; it is assigned to elevator 2, which moves to floor 2 to service it; there is a call for floor 3, and it is assigned to elevator 2. This allows us to make another guess: if E_2.*floor*=1 and E_1.*floor*=2, then *landingCall*(3) causes E_1.*floor*=3:

$$(E_1.\mathit{floor} = 1 \ \wedge \ E_2.\mathit{floor} = 2)[\mathit{landingCall}(3) \rightharpoonup E_1.\mathit{floor}{=}3]$$

Since this scenario holds, we assume that the criterion is the distance: if elevator 1 is closer to the floor called for than elevator 2, it is assigned the call. If they are equidistant, then preference is given to the elevator moving in the right direction; and otherwise the assignment is made nondeterministically by the controller. Running a sequence of witnesses using **Avoid-Visited** helps build this intuition.

Intermediate Events. Given that $A \rightharpoonup B$, is there an intermediate event C that links them, so that $A \rightharpoonup C$ and $C \rightharpoonup B$?

Intermediate Cause. This is a variant of **Cause Weakening**. Suppose we start with $A \rightharpoonup B$ valid, but $A \leftharpoondown B$ not valid. If **Cause Weakening** does not find a weaker event $A \vee C$ with $A \vee C \leftharpoondown B$, is there an intermediate event D such that D happens only because of A ($A \leftharpoondown D$), and B happens only because of D ($D \leftharpoondown B$). B can still follow A without an occurrence of D.

Variable Subset Dependence. This and the following pattern are not shown in Table 2 because they are applicable for general-purpose elaboration. The goal of **Variable Subset Dependence** is to limit the focus of the exploration. For example, we may want to study just the behaviour of the elevator E_1 by disallowing changes in variables of E_2. The pattern is to choose a subset V' of the state variables and use an **Avoid-Events**(V') strategy that attempts to avoid any changes of variables in V'.

Avoid Exception. Exceptional or error behaviour makes many systems hard to understand, but this exact understanding is usually not necessary for building mental models. For example, suppose it is possible to put elevators on service. Then most of the scenarios we attempt to validate are false: a service within the elevator would not be satisfied if the elevator is on service, scheduling of elevators to fulfill landing requests would be different, etc. This pattern allows us to exclude such behaviours from consideration. Given a failed scenario φ, we look for a condition c so that $c[\varphi]$ holds (in the elevator example, such a c is "elevator not on service"). If this fails, we can try to strengthen c by computing counterexamples to φ using **Avoid-Conditions**($\{c\}$). A similar pattern applies if the exceptional behaviour is caused by an event.

6 Evolving Models

In this section, we describe strategies for maintaining and updating mental models as the system evolves.

6.1 Formalizing Change

We start by formalizing the notion of an *evolution* of a model. We define an extension of Kripke structures which captures information about the "old" and the "new" structures; Kripke structures augmented with difference information (`diff`) are called KSDs. KSDs partition state variables into old and new and record information about changes in transitions by associating labels to pairs of states: if there is a transition between them, it is either newly-imposed (labelled by "i") or preserved from the old structure ("p"). If there is no transition, then either the transition never existed ("n") or was deleted during the evolution ("d"). KSDs also record changes of the initial state.

Formally, a *Kripke structure with diff* (KSD) is a tuple $M = (S, s_0, s_0', R, I, I', V, V')$, where S is a set of states; s_0 and s_0' are the old and the new initial states, respectively; V and V' are sets of old and newly-added atomic propositions; $R : S \times S \to \{p, i, d, n\}$ is a labelled transition relation; $I : S \to 2^V$ and $I' : S \to 2^{V'}$ are labelling functions associating each state with the set of old and new atomic propositions, respectively, that are true in that state. In addition, for $s, x \in S$, if $I(s) = I(x)$, then s and x *used to be the same state* – they are identical but for the new variables; this is an equivalence relation, and we write $s \equiv x$. If $s \equiv x$ and $t \equiv y$, then in the old structure, transitions (s, t) and (x, y) were either both present or both absent, and thus in the new one, they are either deleted or preserved: $R(s, t) \in \{p, d\} \Leftrightarrow R(x, y) \in \{p, d\}$. The equivalence class of s under $\equiv$, $\{t \mid t \equiv s\}$, is written $\hat{s}$.

For example, we augment the model in Figure 1(a) with an additional atomic proposition q ($V' = \{q\}$). If q is true, then p does not cause r to become true. If q becomes true while r is true, r becomes false in the next state. A fragment of this model is shown in Figure 1(b). In the figure, preserved transitions are regular lines, imposed ones are extra thick, and deleted ones are dashed; those never there are not shown. The initial state of the system is now s_{13}, but $s_{13} \equiv s_0$. The transition (s_{13}, s_{14}) is considered preserved because in Figure 1(a), the transition (s_0, s_1) was present, and $s_{14} \in \hat{s}_1$, $s_{13} \in \hat{s}_0$.

Our definition of KSDs enables easy extraction of the old and the new Kripke structures. Let $M = (S, s_0, s_0', R, I, I', V, V')$ be a KSD. Then the *old Kripke structure* M_o is $(S/_\equiv, s_0, R_o, I, V)$, where $S/_\equiv$ is the set of states obtained from S via the equivalence relation $\equiv$, and $R_o(\hat{s}, \hat{t}) \Leftrightarrow \forall x \in \hat{s}, y \in \hat{t} \cdot R(x, y) \in \{p, d\}$; that is, a transition between s and t exists iff for all x, y in M whose labels agree with s and t, respectively, on old variables, the transition between x and y was either preserved or deleted. The *new Kripke structure* $M_n = (S, s'_0, R_n, I \cup I', V \cup V')$ has the transition relation $R_n(s,t) \Leftrightarrow R(s,t) \in \{p, i\}$ since only the preserved and the imposed transitions are present in the new system. It is equally possible to take an old Kripke structure, and the edits (the new variables and transition changes) and compute the KSD capturing the change.

Our definition of KSDs describes the change *syntactically*. Unlike a standard simulation relation, it does not allow us to conclude anything about the *logical* relationship between the two systems; however, it does provide a way for strategies to mine the changed model for witnesses that highlight the differences induced by the change.

Note that we have not considered the *deletion* of variables. Deletion is handled by keeping the variable in V but removing dependences on this variable from the transition relation. Formally, let $M = (S, s_0, R, I, V)$ be a Kripke structure, and $x \in V$. Let s^+, s^- be states that agree on values of propositions in $V \setminus \{x\}$, but disagree on the value of x: it is true in s^+ and false in s^-. R is *independent of* x *at* s if $Img(s^+) = Img(s^-)$. R is *independent of* x if the above equality holds for all s. For example, in KSD shown in Figure 1(b), states s_{14} and s_{10} are not independent of q.

Thus, a Kripke structure can be made independent of a variable just by adding and removing transitions; this is behaviorally equivalent to removing the variable. Furthermore, removing dependence on a variable actually removes the variable from the symbolic representation of the model's transition relation.

In this paper, we do not address the problem of specifying the `diff` between the two models. However, our definition can encode many of the high-level notions of change described in the literature, e.g., the SFI feature constructs of Plath and Ryan [26].

6.2 `Diff`-Based Strategies

We define a few strategies that use change information embedded into a KSD.

Avoid-New-Variable-Events. We say that a transition (s, t) results from a *new variable event* if some proposition in V' has a different value in t as it did in s. If s is the last state of the partial witness, the preferred set consists of all successors t of s such that (s, t) does not result from new variable events. A version of this strategy that picks a transition (s, t) with the minimum *number* of new variable events can also be defined.

Avoid-New-Transitions. This strategy uses transition labels. If s is the last state of the partial witness, then t is in the preferred set if $R(s, t) = p$, and in the avoided set if $R(s, t) = i$.

Reuse-Old-Witness. The strategy is useful if the initial states of the new and the old system coincide ($s_0 \equiv s'_0$). Given a previously-generated annotated witness w for φ, with $w_i = (s_i, \Phi_i)$, this strategy prefers, at step i of generating the new witness w', states in $\hat{s}_{i+1}$.

7 Maintaining Models Under Evolution

Changes to a system can have two important effects on behavioral models: new events can be introduced and causal relationships which were established in the old system may be broken. In this section, we introduce an elaboration pattern **Exception Breaks**

Causation which helps understand change and which is supported by strategies that operate on KSDs.

In the old system, $A \rightharpoonup B$ was valid. The existential counterpart $A \rightsquigarrow B$ still holds, but $A \rightharpoonup B$ no longer does. We guess that the change has introduced an exceptional condition c under which A does not lead to B, but possibly to some other event D. Our goal is to find this c,and D if it exists, so that $\neg c[A \rightharpoonup B]$ holds, and perhaps $c[A \rightharpoonup D]$ hold. Further, we may want to check whether A is necessary for D: $A \leftharpoondown D$.

Recording the difference between the two systems in a KSD allows us to use the **Avoid-New-Transitions** strategy for the counterexample to $A \rightharpoonup B$. It minimizes the dependence on the new behavior and focuses on the essential difference between the systems to help identify potential c and D. Conversely, applying **Prefer-New-Transitions** combined with **Avoid-States** allows us to compute different witnesses to $A \rightsquigarrow B$ that focus on the new behavior and yet preserve the property.

We illustrate the use of this pattern on the elevator system, which we modify by introducing a *service* feature to each elevator: once on service, it stops servicing any of its currently-assigned landing calls and may not be assigned any other requests until it goes off service. This change breaks the scenario *landingCall*(3) $\rightharpoonup$ *floor*=3. Searching for counterexamples of this property using **Avoid-New-Transitions** yields the one where from *init*, both E_1*.service* and E_2*.service* become true, and in the next state, the landing-call for floor 3 cannot be assigned to either elevator. Further, both elevators stay on service (the last state is looping). So, we guess that the exception condition c is E_1*.service* $\wedge$ E_2*.service*.

However, c[*landingCall*(3) $\rightharpoonup$ *floor*=3] is not valid either, as the following counterexample shows: E_1 goes on service, a landing-call for floor 3 comes in, it is assigned to E_2, E_2 goes to floor 2, E_1 goes off service, and E_2 goes on service. The system may stay in this state indefinitely, without servicing the call to floor 3. Thus, we propose a weaker c, E_1*.service* $\vee$ E_2*.service*, and this guess is correct: the system behaves normally as long as neither elevator goes on service. There is no reasonable D, in this instance, with c[*landingCall*(3) $\rightharpoonup D$]; if the elevators stay on service, then *floor*=3 may never happen, but no *positive* event which does happen instead can be identified.

To build more understanding of cases where the service feature is used but a landing-call is still being serviced, we use the **Prefer-New-Transition** strategy and examine witnesses to *call*(3) $\rightsquigarrow$ *floor*=3.

8 Tool Support

Sawblade is built on top of our symbolic model-checking tool XChek [6]. Its parts are described below.

The Vocabulary Manager keeps track of variables and events currently considered to be of interest, and hierarchical relationships between them. Elements of the vocabulary can be combined (for a more abstract event), or split up (for a more concrete one).

The Pattern Tool allows users to create behavioral models from scratch using the current vocabulary. It is similar to the corresponding part of the Bandera tool [11]; the fully-realized pattern is translated into a CTL property, which is handed to the model-checker.

The Model Manager tracks validated behavioral models and the relationships between them.

The Strategy Builder allows the user to select and customize standard strategies (such as those described in this paper). Although not currently implemented, Strategy Builder will also include a scripting language for enabling users to define their own strategies.
The Interactive Witness Generator (KEGVis) [15] uses the selected strategy to produce a witness. It can either produce it immediately, or allow manual intervention at defined breakpoints.

Sawblade can maintain Kripke structures with diff as well as ordinary Kripke structures, and construct them from a specification and an edit. This information is used whenever a change-aware strategy (such as **Avoid-New-Transitions**) is used. When a new and an old model are being examined side-by-side, all parts of the tool are aware of it: the Vocabulary Manager marks old and newly-introduced elements, and the Model Manager indicates whether a behavioral model is validated in the old, new, or both systems. The Witness Generator distinguishes newly-introduced variables graphically when presenting witnesses, and also color-codes the types of transition used (preserved and imposed). When attempting to reuse an old witness, it can indicate the location where a removed transition made the reuse impossible.

9 Conclusions and Future Work

In this paper, we described a methodology for building compact behavioural models of existing event-driven systems. The methodology, supported by a tool Sawblade, is based on the use of model-checking for validating scenarios, and on strategy-augmented witness generation for helping elaborate these scenarios. We also described a methodology for storing information about the system evolution and using strategies that use the old and the new systems to help users understand the change in behavioural models. We illustrated our approach using an elevator controller.

In future work, we plan to augment the current capabilities of the tool by adding a scripting language, and expand the witness generator so that it can use strategies with backtracking. We are also interested in combining our methodology with query-checking [5]: once events of interest have been identified, query-checking may be effective in determining the exact relationship between them. We are also planning to provide a stronger empirical validation of our elaboration patterns.

References

1. T. Ball, A. Podelski, and S. Rajamani. “Boolean and Cartesian Abstraction for Model Checking C Programs”. *STTT*, 5(1):49–58, 2003.
2. G.C. Berney and S.M. dos Santos. *Elevator Analysis, Design and Control*. IEE Control Engineering Series 2. Peter Peregrinus Ltd., 1985.
3. F. Buccafurri, T. Either, G. Gottlob, and N. Leone. “On ACTL Formulas Having Linear Counterexamples”. *J. of Comp. and Sys. Sci.*, 62(3):463–515, 2001.
4. R.J.A. Buhr and R.S. Casselman. *Use Case Maps for Object-Oriented Systems*. Prentice-Hall, 1996.
5. W. Chan. “Temporal-Logic Queries”. In *CAV’00*, pp. 450–463. Springer-Verlag, 2000.
6. M. Chechik, B. Devereux, and A. Gurfinkel. “χChek: A Multi-Valued Model-Checker”. In *CAV’02*, 2002.
7. M. Chechik and A. Gurfinkel. “A Framework for Counterexample Generation and Exploration”. in *FASE’05*, pp. 217-233, Springer-Verlag, 2005.

8. Y.-F. Chen, E.R. Gansner, and E. Koutsofios. “A C++ Data Model Supporting Reachability Analysis and Dead Code Extraction”. *IEEE TSE*, 24(9), 1998.
9. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
10. E.M. Clarke, E.A. Emerson, and A.P. Sistla. “Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications”. *ACM TOPLAS*, 8(2):244–263, 1986.
11. J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. “Bandera: Extracting Finite-state Models from Java Source Code”. In *ICSE ’00*, 2000.
12. M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. “Patterns in Property Specifications for Finite-state Verification”. In *ICSE ’99*, 1999.
13. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. “The Software Bookshelf”. *IBM Sys. J.*, 36(4), 1997.
14. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
15. A. Gurfinkel and M. Chechik. “Proof-like Counterexamples”. In *TACAS’03*, pp. 160–175, 2003.
16. D. Harel and W. Damm. “LSCs: Breathing Life into Message Sequence Charts”. In *FMOODS ’99*, pp. 293–312, 1999.
17. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. “Automated Consistency Checking of Requirements Specifications”. *ACM TOSEM*, 5(3):231–261, 1996.
18. R. Holt. “Software Architecture as a Shared Mental Model”. In *IWPC ’02*, 2002.
19. G. Holzmann, D. Peled, and M.H. Redberg. “Design Tools for Requirements Engineering”. *Bell Labs Tech. J.*, 2:86–95, 1997.
20. I. Jacobson, J. Rumbaugh, and G. Booch. *The Unified Software Development Process*. Addison-Wesley, 1999.
21. R. K. Keller, R. Schauer, S. Robitaille, and B. Laguë. “Pattern-Based Design Recovery with SPOOL”. In *Advances in SE: Comprehension, Evaluation, and Evolution*, pp. 113–135, 2002.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
23. H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhi. “A Reverse Engineering Approach to Subsystem Structure Identification”. *J. of Soft. Maintenance*, 5(4):181–204, 1993.
24. G. Murphy, D. Notkin, W. Griswold, and E. Lan. “An Empirical Study of Static Call Graph Extractors”. *ACM TOSEM*, 7(2), 1998.
25. G.C. Murphy, D. Notkin, and K. J. Sullivan. “Software Reflexion Models: Bridging the Gap Between Source and High-Level Models”. In *FSE ’95*, pp. 18–28, 1995.
26. M.C. Plath and M.D. Ryan. “SFI: A Feature Integration Tool”. In *Tool Support for System Specification, Development and Verification*, Adv. in CS, pp. 201–216. 1999.
27. D. Păun and M. Chechik. “On Closure Under Stuttering”. *Formal Aspects of Computing*, 14:342–368, 2003.
28. M.P. Robillard and G. C. Murphy. “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies”. In *ICSE ’02*, pp. 406–416, 2002.