Model Checking and Evidence Exploration

Yifei Dong, C.R. Ramakrishnan, and Scott A. Smolka
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400, USA
E-mail:{ydong,cram,sas}@cs.sunysb.edu

Abstract

We present an algebraic framework for evidence exploration*: the process of interpreting, manipulating, and navigating the proof structure or* evidence *produced by a model checker when attempting to verify a system specification for a temporal-logic property. Due to the sheer size of such evidence, single-step traversal is prohibitive and smarter exploration methods are required. Evidence exploration allows users to explore evidence through smaller, manageable* views*, which are definable in* relational graph algebra*, a natural extension of relational algebra to graph structures such as model-checking evidence. We illustrate the utility of our approach by applying the Evidence Explorer, our tool implementation of the evidence-exploration framework, to the Java meta-locking algorithm, a highly optimized technique deployed by the Java Virtual Machine to ensure mutually exclusive access to object monitor queues by threads.*

1. Introduction

It is widely believed that formal verification can play an essential role in the design and development of high-confidence computer-based systems. While a number of powerful formal verification techniques exist, their acceptance in the industrial sector has been limited in part by a perceived lack of usability. Research related to improving usability has targeted various stages of formal verification, including writing more understandable and less error-prone specifications, visualizing system dynamics via graphical languages such as statecharts and message sequence charts, shortening verification time (efficiency is a usability issue too), and generating meaningful error diagnostics.

In this paper, we are interested in the latter stages of model checking: manipulating and interpreting the output of a model checker, in particular its proof structure. Model checking [10, 22] can be viewed as the problem of proving or disproving that a system M satisfies a property ϕ specified in some kind of temporal logic. For a decidable model-checking problem, there exists a proof or disproof of the goal $M \models \phi$. A proof in this setting consists of a set of subgoals, including the main goal, and their interrelationships via inference rules. This notion of proof also covers disproofs since a disproof of $M \models \phi$ is a proof of $M \models \neg\phi$ if the logic is closed under negation. We refer to such a proof structure as the model checker's *evidence* [28].

Evidence is a rich medium for understanding model-checking problems. Obviously, an evidence carries much more information than a one-bit $true$ or $false$ answer. It may also provide richer diagnostic information than witnesses or counter-examples, for these tend to be linear in nature (except for the "multi-path" [8] and "tree-like" [9] counterexamples for ACTL formulas), and sometimes do not even exist [6].

On the other hand, users require advanced tool support to manipulate evidence effectively. In the case of the modal mu-calculus [19], a very expressive temporal logic, the size of a model-checking evidence is $O(|M| \cdot |\phi|)$; other temporal logics have similar complexity. Although this complexity is linear, the term $|M|$, the size of the system specification, can easily reach millions of states in practice due to the state-explosion problem. Thus, if users explore evidence by simply traversing the links between subgoals, the sheer volume of evidence can cause myriad usability problems:

- It is difficult for users to build a mental picture from such a large data set.
- Users may be unable to traverse the entire evidence.
- Even if a user traverses the evidence selectively, he may spend a lot of time trying to reach the portions of interest.
- Because evidence incorporates the dynamics of both the system specification and logical property, users have to constantly switch between these two contexts and may become fatigued quickly.

- Since a formula is typically small with respect to the size of the system state space, formula-specific patterns may repeat themselves throughout an evidence. Without the help of distinctive cues, users may quickly lose track of where they are in the evidence.

Users obviously require smarter exploration methods to make effective use of evidence. It would be desirable if users could do the following:

- View evidence from different angles.
- View evidence in different resolutions by zooming in and zooming out.
- Restrict the scope of exploration according to some criterion of interest.
- Directly locate a particular portion via an index.
- Extract a representative trace of the system from the selected portion.

Many of the above methods have been implemented in our prototype tool, the Evidence Explorer (EE), which visualizes evidences and enables users to navigate them via certain state and formula "projection" operations. Our experience with EE indicates that it can significantly enhance a user's ability to analyze the results obtained with a model checker during system verification.

EE is highly extensible and customizable, owing to the fact that users can explore evidence in smaller, manageable *views*. Such views are definable in an algebra we call *relational graph algebra*, a natural extension of the relational algebra to graph structures such as model-checking evidence. We illustrate the utility of our approach by applying the Evidence Explorer to the Java meta-locking algorithm, a highly optimized technique deployed by the Java Virtual Machine to ensure mutually exclusive access to object monitor queues by threads [2]. In particular, this case study demonstrates how EE can quickly allow a user to hone in on the portions of interest of model-checking evidence from a real-life application.

The rest of the paper is organized as follows. Section 2 defines the modal mu-calculus and its evidence. Section 3 describes the main features of the Evidence Explorer and the Java meta-locking case study. Section 4 defines relational graph algebra, while Section 5 formalizes the framework of evidence exploration in terms of this algebra. Section 6 contains our concluding remarks and a discussion of related work.

2. Evidence for the modal mu-calculus

In this section, we first present the syntax and semantics of the modal mu-calculus, and then formally define the notion of evidence in the context of model checking properties written in this logic.

We assume a system M is in the form of a Kripke structure $\langle \mathcal{S}, s_0, \rightarrow, \mathcal{L} \rangle$, where $\mathcal{S}$ is the set of states, s_0 is the initial state, $\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of transitions labeled by actions in $\mathcal{A}$, and $\mathcal{L} : \mathcal{S} \rightarrow 2^{\mathcal{P}}$ labels each state with a set of atomic propositions in $\mathcal{P}$.

2.1. The modal mu-calculus

Syntax. Following [25], the syntax of modal mu-calculus formulas over a set of variable names Var is given by the following grammar, where $p \in \mathcal{P}$, $Z \in Var$, and $K \subseteq \mathcal{A}$:

$$\begin{array}{rcl} \varphi & \rightarrow & p \mid Z \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\ & & \mid [K]\varphi \mid \langle K \rangle \varphi \mid \nu Z.\varphi \mid \mu Z.\varphi \end{array}$$

We use ρ to range over the fixed-point operators $\{\nu, \mu\}$. In a fixed-point formula $\rho Z.\varphi$, every free occurrence of Z in φ should be positive. Operators $\wedge$ and $\vee$, $[\cdot]$ (read *box*) and $\langle \cdot \rangle$ (read *diamond*), and ν and μ are three pairs of dual operators. By De Morgan's rule, a formula can be converted to equivalent positive normal form by pushing $\neg$ inward until it is applied to an atomic proposition. We hereafter assume that all formulas are automatically converted to positive normal form.

Semantics. The semantics of the modal mu-calculus is captured by the satisfaction relation between states and formulas. For a state s and a formula ϕ, the notation $s \models \phi$ indicates that s satisfies ϕ. When ϕ is an atomic proposition, $s \models \phi$ if s is labeled by ϕ. When ϕ is a complex formula, the meaning of $s \models \phi$ is derived recursively according to the syntactic structure of ϕ. For examples, s satisfies $\varphi_1 \wedge \varphi_2$ if s satisfies both φ_1 and φ_2; s satisfies $[K]\varphi$ (read *box K* φ) if all the successors of s after a transition labeled by an action in K (K-successors) satisfy φ; s satisfies $\langle K \rangle \varphi$ (read *diamond K* φ) if there exists a K-successor of s that satisfies φ.

Formally, given a Kripke structure $M = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{L} \rangle$ and an interpretation $\mathcal{V} : Var \rightarrow 2^{\mathcal{S}}$ of the variables, the set $[\![\phi]\!]^M_{\mathcal{V}}$ of states satisfying a formula ϕ is defined by the following equations, where the extended transition relation $s \xrightarrow{K} s'$ is defined as $\exists a \in \mathcal{A}.(s \xrightarrow{a} s' \wedge a \in K)$.

$$\begin{array}{rcl} [\![P]\!]^M_{\mathcal{V}} & = & \{s \in \mathcal{S} \mid P \in \mathcal{L}(s)\} \\ [\![Z]\!]^M_{\mathcal{V}} & = & \mathcal{V}(Z) \\ [\![\neg\varphi]\!]^M_{\mathcal{V}} & = & \mathcal{S} - [\![\varphi]\!]^M_{\mathcal{V}} \\ [\![\varphi_1 \wedge \varphi_2]\!]^M_{\mathcal{V}} & = & [\![\varphi_1]\!]^M_{\mathcal{V}} \cap [\![\varphi_2]\!]^M_{\mathcal{V}} \\ [\![\varphi_1 \vee \varphi_2]\!]^M_{\mathcal{V}} & = & [\![\varphi_1]\!]^M_{\mathcal{V}} \cup [\![\varphi_2]\!]^M_{\mathcal{V}} \\ [\![[K]\varphi]\!]^M_{\mathcal{V}} & = & \{s \mid \forall s' \in \mathcal{S}.\, s \xrightarrow{K} s' \Rightarrow s' \in [\![\varphi]\!]^M_{\mathcal{V}}\} \\ [\![\langle K \rangle\varphi]\!]^M_{\mathcal{V}} & = & \{s \mid \exists s' \in \mathcal{S}.\, s \xrightarrow{K} s' \wedge s' \in [\![\varphi]\!]^M_{\mathcal{V}}\} \end{array}$$

$$\begin{aligned} [\![\nu Z.\varphi]\!]^M_{\mathcal{V}} &= \bigcup\{S \subseteq \mathcal{S} \mid S \subseteq [\![\varphi]\!]_{\mathcal{V}[Z:=S]}\} \\ [\![\mu Z.\varphi]\!]^M_{\mathcal{V}} &= \bigcap\{S \subseteq \mathcal{S} \mid S \supseteq [\![\varphi]\!]_{\mathcal{V}[Z:=S]}\} \end{aligned}$$

M and $\mathcal{V}$ are usually dropped from $[\![\phi]\!]^M_{\mathcal{V}}$ when the context is clear and ϕ has no free variable. We write $s \models \phi$ if $s \in [\![\phi]\!]$, and $M \models \phi$ if $s_0 \models \phi$.

Graphical representation. We use formula graphs to graphically render modal mu-calculus formulas. Besides capturing formula syntax, formula graphs also convey certain kinds of semantic information.

Definition 1 *The* formula graph $\mathcal{G}_\phi$ *of a formula* ϕ *is the minimal directed graph such that each node is a formula, including* ϕ*, and for each node* φ*, its out-edges are labeled by its top-level operator, and the set of* φ*'s successors is*

$$next(\varphi) = \begin{cases} \emptyset & (\varphi = p \textit{ or } \neg p, \textit{ where } p \in \mathcal{P}) \\ \{\psi_1, \psi_2\} & (\varphi = \psi_1 \wedge \psi_2 \textit{ or } \psi_1 \vee \psi_2) \\ \{\psi\} & (\varphi = [K]\psi \textit{ or } \langle K\rangle\psi) \\ \{\psi[\rho Z.\psi/Z]\} & (\varphi = \rho Z.\psi) \end{cases}$$

We refer to $\mathcal{G}_\phi$'s node set as ϕ's *closure*, denoted by $cl(\phi)$. The Fischer-Ladner closure $FL(\phi)$ [13, 27] covers both ϕ and its negation, so it is equal to $cl(\phi) \cup cl(\neg\phi)$.

2.2. Evidence

An evidence or proof structure of a goal (state-formula pair) contains a set of subgoals and their interrelationships defined by logic-specific inference rules. The following definition of evidence for a modal mu-calculus goal is adopted from the notion of well-founded pre-model given in [27].

Definition 2 *Given a Kripke structure* $M = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{L}\rangle$ *and a formula* ϕ*, an* evidence *for* $M \models \phi$ *is a directed graph whose nodes are state-formula pairs* $\vdash \subseteq \mathcal{S} \times cl(\phi)$ *containing and reachable from* $s_0 \vdash \phi$*, under the following constraints*

(1) Each node and its successors match one of the rules in Figure 1, where the predecessor (above the line) and the successors (under the line) satisfy the side condition (in brackets, if it exists).

(2) There is no infinite path where a least-fixed-point formula $\mu Z.\varphi$ *is a common subformula of the formula components of all the nodes on the path.*

A node $s \vdash \varphi$ stands for the subgoal "state s satisfies formula φ". Edges stand for the derivation relation between subgoals. The meaning of $s \vdash \varphi$'s successors $s'_1 \vdash \varphi', \ldots, s'_n \vdash \varphi'$ is "$s \models \varphi$ is true because $s'_1 \models \varphi', \ldots, s'_n \models \varphi'$ are true".

The second constraint in the definition prevents circular reasoning of least fixed points. In finite-state systems, this constraint is equivalent to insisting that a least fixed point cannot be the outermost fixed point in a loop.

Note that the rules for disjunction ($s \vdash \varphi_1 \vee \varphi_2$) and diamond ($s \vdash \langle K\rangle\varphi$) are nondeterministic. [27] uses a *choice function* to determine which subgoal is derived by the predecessor. Directly adopting the proofs given in [27], we can establish evidence's soundness and completeness.

Theorem 1 $M \models \phi$ *if and only if there is an evidence for it.*

Note that multiple evidences may exist for the same goal. Also note that although the *existence* of an evidence is a sufficient and necessary condition for $M \models \phi$, actual *materialization* of an evidence may not be necessary to solve the model-checking problem $M \overset{?}{\models} \phi$. An extreme example is when the formula is a tautology, e.g. $\nu X.[-]X$. An evidence for $M \models \phi$ may need to cover M's entire state space, but the proof for the tautology does not need to deal with a concrete system at all. Whether there is a most succinct form of evidence is an interesting open topic.

To make evidence self-contained, we can further label the edge between nodes $s_1 \vdash \varphi_1$ and $s_2 \vdash \varphi_2$ by a tuple $\langle \alpha, \omega\rangle$, where ω is φ_1's top-level operator and if ω is $[\cdot]$ or $\langle\cdot\rangle$ then α is the label on the transition from s_1 to s_2; otherwise α is ϵ, indicating no transition occurred.

For the sake of simplicity, we have only defined evidence for positive goals. For negative goals, recall that $M \not\models \phi$ if and only if $M \models \neg\phi$, so the evidence for $M \not\models \phi$ can be defined as the evidence for $M \models \neg\phi$.

3. The Evidence Explorer: a tool for evidence exploration

We have developed a prototype tool, called the Evidence Explorer, to visualize evidences and help users navigate them for better understanding. In this section, we describe the main features of the Evidence Explorer and illustrate its utility by an example.

3.1. Features of evidence explorer

The main idea of the Evidence Explorer is to let users navigate an evidence through smaller views. A *view* of an evidence is either a subgraph or a projection, representing a portion of either the graph or its attributes.

The Evidence Explorer, whose architecture is depicted in Figure 2, consumes the evidence generated by a model checker. Once model checking is completed, Evidence Explorer allows users to explore the proof in the following six

$$\frac{s \vdash p}{-}(p \in \mathcal{L}(s)) \quad \frac{s \vdash \neg p}{-}(p \notin \mathcal{L}(s)) \quad \frac{s \vdash \varphi_1 \wedge \varphi_2}{s \vdash \varphi_1 \quad s \vdash \varphi_2} \quad \frac{s \vdash \varphi_1 \vee \varphi_2}{s \vdash \varphi_1} \quad \frac{s \vdash \varphi_1 \vee \varphi_2}{s \vdash \varphi_2}$$

$$\frac{s \vdash [K]\varphi}{s_1 \vdash \varphi \quad s_2 \vdash \varphi \ldots}(\{s_i\} = \{s' \mid s \xrightarrow{K} s'\}) \quad \frac{s \vdash \langle K \rangle \varphi}{s' \vdash \varphi}(s \xrightarrow{K} s') \quad \frac{s \vdash \rho Z.\varphi}{s \vdash \varphi[\rho Z.\varphi/Z]}$$

Figure 1. Predecessor-successors relation in evidence.

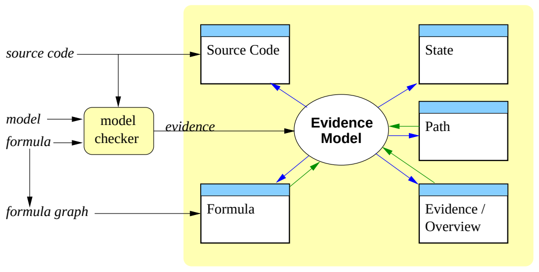


Figure 2. Architecture of the Evidence Explorer.

windows. A node in the evidence, initially the root, is chosen as the *focus node* to synchronize these windows. The state and formula components of the focus node are called the "focus state" and "focus formula", respectively.

1. The *Source Code* window contains the source program. The lines corresponding to the focus state's subprocess control points are highlighted.

2. The *Formula* window displays the formula graph of the checked formula. Formula graphs are designed to provide users with several visual cues about the nature of the formula and evidence. Each node's shape is determined by its top-level operator, and its border color reflects the validity of the formula in the proof (blue for $true$, red for $false$). Nodes within the scope of the same fixed point are painted with the same color. If a subformula does not appear in the proof (and hence is vacuous [6]), its interior is painted transparent and its border is dotted.

3. The *Overview* window depicts the whole evidence. Because of an evidence's branching nature, we transform the evidence, originally a directed graph, to tree form (the graph's spanning tree) and mark cross edges and back edges by special symbols (arrow and circle, respectively). For easy recognition, each node is painted the color its formula component assumes in the Formula window. A box encloses the portion of the proof shown in the Evidence window.

4. The *Evidence* window depicts exactly the same evidence pictured in the Overview window, but in zoomed-in scale. If the whole evidence cannot be drawn inside the window, users can scroll the view port using the scroll-bar, or click on the Overview window to jump directly to a certain portion. Users can set a new focus node by clicking on the desired node or by navigating via arrow keys.

5. The *State* window lists the details of the focus state. The hierarchy of parallel composition is depicted by a tree, where each node is labeled by the name of the corresponding subprocess. Variable bindings are listed for each sequential subprocess.

6. The *Path* window displays a message sequence chart capturing the state dynamics of the path in the evidence from the root to the focus node.

These six windows are synchronized by the focus node of the evidence, such that if the focus is changed in one window, the other windows will automatically update their display. Users can change the focus directly in the Evidence window, or indirectly by selecting a state in the Path window or a formula in the Formula window; then a new focus will be set to match the selected state or formula.

3.2. Case study: the Java meta-locking algorithm

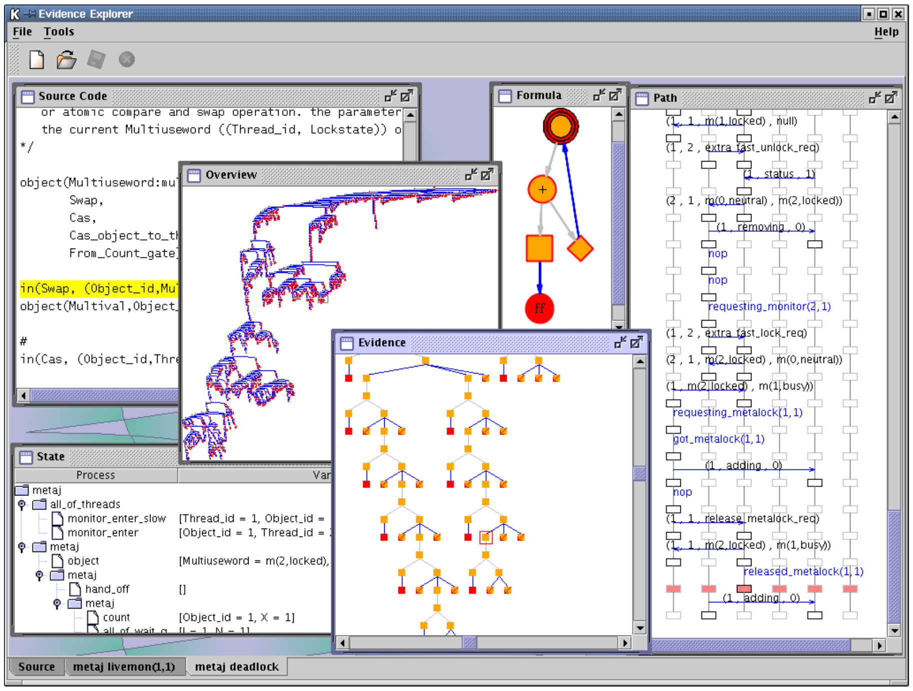
We use the Java meta-locking algorithm as an example to illustrate the utility of the Evidence Explorer. Meta-locking is a highly optimized technique deployed by the Java Virtual Machine (JVM) to ensure that threads have mutually exclusive access to object monitor queues [2]. An abstract specification of the meta-locking algorithm was verified in [5], and a more detailed version of this specification is considered in [4]. Because access to object monitor queues is not necessarily first-in-first-out, certain liveness properties may be violated.

The following two properties are used in this case study.

$$\texttt{deadlock} = \mu X.[-]\texttt{ff} \vee \langle - \rangle X$$

meaning that there exists a path to a state where no transition is possible, in other words, a deadlock; and

$$\begin{aligned} \texttt{livemon} \quad = \quad & \nu X.([\texttt{requesting\_monitor}] \\ & (\mu Y.(\langle \texttt{got\_monitor} \rangle \texttt{tt} \vee [-]Y)) \wedge [-]X) \end{aligned}$$

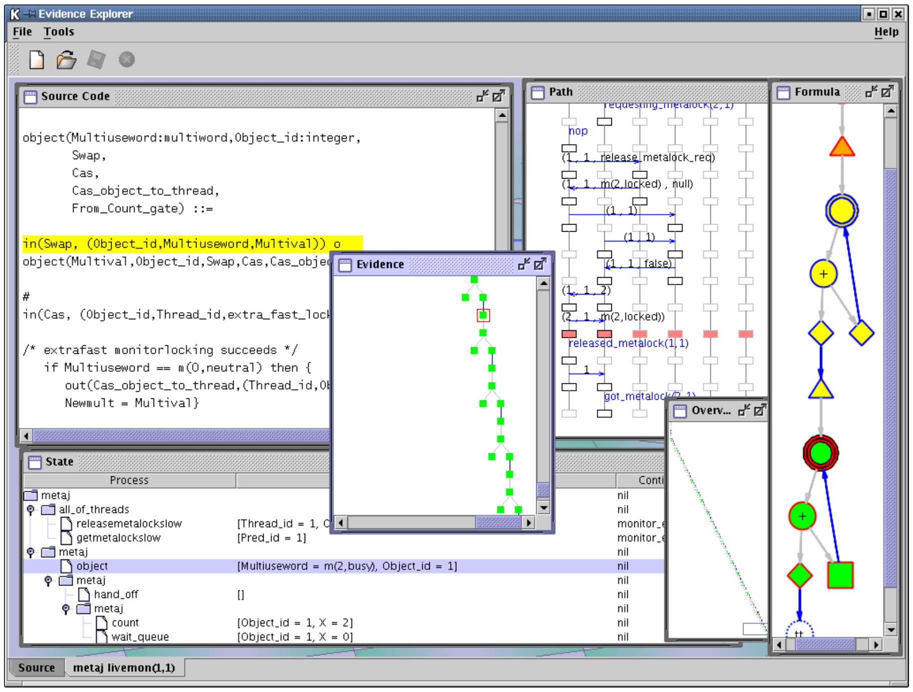


```
 or atomic compare and swap operation. the parameter
 the current Multiuseword ((Thread_id, Lockstate)) o
*/

object(Multiuseword:mu
       Swap,
       Cas,
       Cas_object_to_th
       From_Count_gate

in(Swap, (Object_id,Mu
object(Multival,Object_

#
in(Cas, (Object_id,Thre
```

Process	Var
metaj	
all_of_threads	
monitor_enter_slow	[Thread_id = 1, Object_id =
monitor_enter	[Object_id = 1, Thread_id =
metaj	
object	[Multiuseword = m(2,locked),
metaj	
hand_off	[]
metaj	
count	[Object_id = 1, X = 1]
all_of_wait_q	[I = 1, N = 1]

Evidence Explorer
File
Tools
Help
Source Code
Overview
Formula
Path
Evidence
State
ff
(1 , 1 , m(1,locked) , null)
(1 , 2 , extra_fast_unlock_req)
(1 , status , 1)
(2 , 1 , m(0,neutral) , m(2,locked))
(1 , removing , 0)
nop
nop
requesting_monitor(2, 1)
(1 , 2 , extra_fast_lock_req)
(2 , 1 , m(2,locked) , m(0,neutral))
(1 , m(2,locked) , m(1,busy))
requesting_metalock(1, 1)
got_metalock(1, 1)
(1 , adding , 0)
nop
(1 , 1 , release_metalock_req)
(1 , 1 , m(2,locked) , m(1,busy))
released_metalock(1, 1)
(1 , adding , 0)
Source
metaj livemon(1,1)
metaj deadlock

(a) deadlock



```
object(Multiuseword:multiword,Object_id:integer,
       Swap,
       Cas,
       Cas_object_to_thread,
       From_Count_gate) ::=

in(Swap, (Object_id,Multiuseword,Multival)) o
object(Multival,Object_id,Swap,Cas,Cas_objec

#
in(Cas, (Object_id,Thread_id,extra_fast_lock

/* extrafast monitorlocking succeeds */
   if Multiuseword == m(0,neutral) then {
      out(Cas_object_to_thread,(Thread_id,Ob
      Newmult = Multival}
```

Process		Conti
metaj		nil
all_of_threads		nil
releasemetalockslow	[Thread_id = 1, O	monitor_e
getmetalockslow	[Pred_id = 1]	monitor_e
metaj		nil
object	[Multiuseword = m(2,busy), Object_id = 1]	nil
metaj		nil
hand_off	[]	nil
metaj		nil
count	[Object_id = 1, X = 2]	nil
wait_queue	[Object_id = 1, X = 0]	nil

(b) livemon

Figure 3. Screen-shots of the Evidence Explorer.

meaning that it is globally true that after any `requesting_monitor` transition, along all paths eventually `got_monitor` is enabled.

Figure 3(a) is a screen-shot of the Evidence Explorer for the `deadlock` property. Because there is no deadlock in the system, all the states must be visited, leading to an very large evidence. Users can examine its details in the Evidence window.

Figure 3(b) is the screen-shot for the property `livemon`. This property is violated due to starvation, so a counterexample exists. The proof of the violation is mostly linear and consists of two parts: the upper part is a straight line whose subformulas are in the scope of the outer fixed point (*i.e.*, νX), and the lower part is a cycle (with small branches) certifying the falsity of the inner fixed point (μY). Users may click at the bottom of the Overview window to jump directly to the end of the proof, study the most critical steps of the starvation, and then go upward to see the whole trace.

Our experience in using the Evidence Explorer on the meta-locking case study shows a gain in user productivity due to shortened evidence traversal time. We had to spend nearly an hour to traverse the above proof using a standard tree browser. With the Evidence Explorer, we cut the process to only a couple of minutes and were able to experiment with the specification via more frequent modifications. Of course, we need to conduct more rigorous usability tests to fully evaluate the tool's effectiveness and to reveal potential improvements.

3.3. Extending the tool

The main idea behind the Evidence Explorer is that users explore evidence through smaller *views*, either statically generated (as in the Formula window) or dynamically updated according to the user's input (as in the Source, State, and Path windows). To extend the tool, we can simply define more views. Furthermore, we feel obliged to develop a general framework of evidence exploration to let not only us, the tool developers, but also users systematically define new views.

Views in some sense reflect the user's focus, and they vary from user to user and from system to system. As it is impossible to predict every user's focus, pre-defined views may be either too general or too specific. Allowing users define their own views will better serve their needs.

4. Queries on graphs: relational graph algebra

We want to find an appropriate data structure for evidence views. Observe that both evidences and systems are graphs whose node sets are relations: an evidence node is state-formula pair; a state in a concurrent system is a tuple of substates. We call this kind of graph a *relational graph*.

We also want to find appropriate operations on views. In databases, views are the (non-materialized) result of queries on relations. Similarly in evidence exploration, we define views as the result of queries on relational graphs: evidences and existing views. We simply lift the operators of relational algebra to relational graphs, and call the lifted algebra *relational graph algebra*.

First, we formally define relational graphs. A *domain* D is either a primitive domain such as the integers or reals, or a Cartesian product of domains $D_1, \ldots, D_n$. A domain d is said to be a *subdomain* of D ($d \sqsubseteq D$) if d is the Cartesian product of a subsequence of $(D_1, \ldots, D_n)$. A set R is a *relation* on a sequence of domains $(D_1, \ldots, D_n)$ if $R \subseteq D_1 \times \cdots \times D_n$. We assume each domain contains a special tuple ϵ representing the null tuple.

Definition 3 *A* relational graph *on node-label domain D^V and edge-label domain D^E is a directed graph $G = \langle V, E\rangle$ whose node set V is a relation on D^V, and whose set of edges E is a relation on (D^V, D^E, D^V) and $E \subseteq V \times D^E \times V$.*

The graph G cannot be represented by the edge relation E alone, because there may exist nodes not attached to an edge. The clause $E \subseteq V \times D^E \times V$ is a constraint that the source and destination nodes of an edge must be in V. We use G_{D^V, D^E} to denote the complete graph $\langle D^V, D^V \times D^E \times D^V\rangle$.

Basic operators. The basic operators of the relational graph algebra are defined in Table 1. In this definition, we assume that the inputs are relational graphs $G_i = \langle V_i, E_i\rangle$ on domains D_i^V and D_i^E for $i = 1, 2$, and the result is the relational graph $G = \langle V, E\rangle$ on domains D^V and D^E. The requirement that G_1 and G_2 are union-compatible (for the union ($\cup$), intersection ($\cap$), node-difference ($-^V$), and edge-difference ($-^E$) operators) means that they have the same node-label and edge-label domains.

A few comments about the definition are in order.

- There are two kinds of difference operators [16]: $G_1 -^V G_2$, the *node difference* of G_1 and G_2, excludes nodes of G_2 and their attached edges from G_1; $G_1 -^E G_2$, the *edge difference* of G_1 and G_2, excludes only edges of G_2 from G_1.

- Let f_v be a Boolean function on D_1^V and f_e be a Boolean function on $D_1^V \times D_1^E \times D_1^V$. Then $\sigma_{f_v, f_e}(G_1)$, the *selection* of G_1 w.r.t. f_v and f_e, is the subgraph of G_1 whose nodes satisfy the constraint represented by f_v and whose edges satisfy the constraint represented by f_e.

- Let d^v be a subdomain of D_1^V and d^e be a subdomain of D_1^E. Then $\pi_{d^e, d^e}(G_1)$, the *projection* of G_1 on

Table 1. Relational graph algebra operations

Result G	Precondition	Domains D^V, D^E	Node set V	Edge set E
$G_1 \cup G_2$	G_1 and G_2 are union-compatible	$D^V = D_1^V = D_2^V$ $D^E = D_1^E = D_2^E$	$V_1 \cup V_2$	$E_1 \cup E_2$
$G_1 \cap G_2$			$V_1 \cap V_2$	$E_1 \cap E_2$
$G_1 -^V G_2$			$V_1 - V_2$	$E_1 - V_2 \times D^E \times V_2$
$G_1 -^E G_2$			V_1	$E_1 - E_2$
$\sigma_{f_v,f_e}(G_1)$	$f_v : D_1^V \to \{0,1\}$, $f_e : D_1^E \to \{0,1\}$	$D^V = D_1^V, D^E = D_1^E$	$\sigma_{f_v}(V_1)$	$\sigma_{f_e}(E_1) \cap V \times D^E \times V$
$\pi_{d^v,d^e}(G_1)$	$d^v \sqsubseteq D_1^V, d^e \sqsubseteq D_1^E$	$D^V = d^v,\ D^E = d^e$	$V = \pi_{d^v}(V_1)$	$E = \pi_{d^v \times d^e \times d^v}(E_1)$ $- V \times \{\epsilon\} \times V$
$G_1 \times G_2$		$D^V = D_1^V \times D_2^V$ $D^E = D_1^E \times D_2^E$	$V_1 \times V_2$	$\{((v_1,v_2),(e_1,e_2),(v_1',v_2')) \mid$ $\forall i = 1,2\ ((v_i,e_i,v_i') \in E_i \vee$ $(v_i = v_i' \wedge e_i = \epsilon)$ $\wedge \exists i = 1,2\ (v_i,e_i,v_i') \in E_i\}$

d_v and d_e, is obtained by projecting G_1's nodes and edges onto the respective subdomains and removing the edges labeled by ϵ.

- $G_1 \times G_2$, the *product* of G_1 and G_2, is obtained as follows: its node set is the product of G_1 and G_2's nodes, and each edge embeds at least one edge from G_1 or G_2 and keeps the remaining idle (labeled by ϵ).

Derived operators. Derived operators such as join, extension, and grouping [12], can be defined in terms of the basic operators in a similar way as they are defined in relational algebra. Suppose d is a subdomain of a domain D. We use $D \backslash d$ to represent the remaining subdomain of D by removing d from D, and $t[d]$ to represent the d-component of a tuple t.

- Let d^v be a common subdomain of D_1^V and D_2^V, and d^e be a common subdomain of D_1^E and D_2^E. The *natural join* of G_1 and G_2 by d^v and d^e is defined as the relational graph on domains $D^V = (D_1^V \times D_2^V) \backslash d^v$ and $D^E = (D_1^E \times D_2^E) \backslash d^e$ as follows:

$$G_1 \bowtie_{d^v,d^e} G_2 = \pi_{D^V,D^E}(\sigma_{f_v,f_e}(G_1 \times G_2))$$

where

$$\begin{aligned} f_v &= \lambda v.(v[d_1^v] = v[d_2^v]) \\ f_e &= \lambda e.(e[d_1^e] = e[d_2^e]) \end{aligned}$$

- The *extension* of G_1 by functions $x_v : D^V \to d^v$ and $x_e : D^V \times D^E \times D^V \to d^e$

$$\mathcal{E}_{x_v,x_e}(G_1) = \sigma_{f_v,f_e}(G_1 \times G_{d^v,d^e})$$

where

$$\begin{aligned} f_v &= \lambda v.(x_v(v[D_1^V]) = v[d^v]) \\ f_e &= \lambda e.(x_e(e[D_1^E]) = e[d^e]) \end{aligned}$$

- Using the same parameters of extension, the *mapping* of G_1 by mapping functions x_v and x_e is defined as

$$\rho_{x_v,x_e}(G_1) = \pi_{d^v,d^e}(\mathcal{E}_{x_v,x_e}(G_1))$$

- The *grouping* of G_1 based on an equivalence function $q : D_1^V \to d$ is defined as

$$\gamma_q(G_1) = \rho_{\lambda v'.\sigma_{\lambda v.(q(v)=v')}(G_1)}(\rho_{\lambda v.q(v)}(G_1))$$

Selecting a single element. Sometimes we want to select a single element from a relation or graph, for example the nearest node from the root. We use the function $\mathbf{pick}$ to represent such a selection. Formally, let D and D' be domains and R be a relation on D. Given a weight function $w : D \to D'$ and an aggregate function $a : 2^{D'} \to D'$, selecting a single tuple from R by a and w is defined as

$$\mathbf{pick}_{a,w}(R) \quad \in \quad \sigma_{\lambda v.w(v)=a(\{w(v') \mid v' \in R\})}(R)$$

If there is more than one tuple in $\sigma_{\ldots}(R)$, $\mathbf{pick}$ nondeterministically returns one tuple. The $pick$ operator is lifted to a relational graph such that $\mathbf{pick}_{a,w}(G)$, selecting a node from a relational graph G, is defined as $\mathbf{pick}_{a,w}(V_G)$. So the nearest node from root is $\mathbf{pick}_{\min,\lambda v.dist(root,v)}(G)$, where $dist(v_1, v_2)$ computes the distance from node v_1 to v_2.

To model user interaction, we subscript $\mathbf{pick}$ with $user$ to denote a user selecting an element from a relation or a graph: $\mathbf{pick}_{user}(R)$ picks a tuple from relation R, and $\mathbf{pick}_{user}(G)$ a node from graph G. $\mathbf{pick}_{user}$ is the only interactive operator in our formalism.

Path operators. Path is an important graph concept. Hereafter, a path specifically refers to a connected subgraph where each node has exactly one successor. Under this definition, in finite graphs, although a path may contain a cycle, there are only a finite number of paths and each path is finite.

For convenience, we assume the following functions are pre-defined: $\mathbf{paths}(v_1, v_2)$ returns the set of paths from node v_1 to node v_2, and $\mathbf{reachable}(v_1, v_2)$ is boolean function that checks whether there is a path from node v_1 to node v_2.

5. An algebraic framework of evidence exploration

The concepts of system, formula graph, and evidence can be expressed as relational graphs. Suppose $M = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{L} \rangle$ is a Kripke structure, ϕ is a formula, and $\mathcal{E}$ is an evidence of $M \models \phi$. Then M is a relational graph whose node-label domain is $\mathcal{S}$, the state space, and edge-label domain is $\mathcal{A}$, the set of action labels; formula graph $\mathcal{G}_\phi$ is a relational graph whose node-label domain is $\mathcal{F}$, the set of all modal mu-calculus formulas, and edge-label domain is $\mathcal{O}$, the set of all operators; and an evidence for $M \models \phi$ is a subgraph of M and $\mathcal{G}_\phi$'s product, under the constraints in Definition 2.

5.1. Formalization of Evidence Explorer actions

In the Evidence Explorer, the Path window shows a selected trace of evidence, the Formula windows shows the formula projection of evidence, and clicking a formula in the Formula window resets a new focus evidence node. We use two variables, *root* – the node $s_0 \vdash \phi$ in $\mathcal{E}$, and *focus* – the focus node interactively selected by the user, to formalize these views and actions.

Trace extraction. The trace embedded in the shortest path from the evidence's root to the focus node is extracted by

$$\pi_{state}(\mathbf{pick}_{\min, \lambda p.length(p)}(\mathbf{paths}(root, focus)))$$

where $\pi_{state}(\mathcal{E}')$, the *state projection* of a subgraph $\mathcal{E}'$ of evidence $\mathcal{E}$ is defined as

$$\pi_{state}(\mathcal{E}') = \pi_{\mathcal{S},\mathcal{A}}(\mathcal{E}')$$

Since $\mathcal{E}$ is a subgraph of $M \times \mathcal{G}_\phi$, $\pi_{state}(\mathcal{E})$ is a subgraph of the system M.

Formula projection. The formula projection of evidence $\mathcal{E}$ is defined as

$$\pi_{formula}(\mathcal{E}) = \pi_{\mathcal{F},\mathcal{O}}(\mathcal{E})$$

Similar to state projection, $\pi_{formula}(\mathcal{E})$ is a subgraph of ϕ's formula graph $\mathcal{G}_\phi$. When the subgraph relation is proper, it implies that some subformula of ϕ does not contribute to the validity of $M \models \phi$, and therefore M vacuously satisfies ϕ [6]. Note that the formula projection of a particular $\mathcal{E}$ being a proper subgraph of $\mathcal{G}_\phi$ is only a sufficient condition of the vacuity of $M \models \phi$; the sufficient and necessary condition is that there *exists* an $\mathcal{E}$ whose formula projection is a proper subgraph of $\mathcal{G}_\phi$.

Formula graph as an index to evidence. Because ϕ is usually much smaller than M, users can use $\pi_{formula}$ as an index to directly locate a node with a particular formula, such as an inner fixed point, without going through long sequence of nodes of outer formulas. This is done in two steps: First, the user chooses the formula he wants: $\varphi = \mathbf{pick}_{user}(\pi_{formula})$; then the software selects the nearest node from node *focus* whose formula component is φ:

$$\mathbf{pick}_{\min, \lambda v.dist(focus, v)}(\sigma_{\lambda v.(formula(v) = \varphi)}(\mathcal{E})$$

The user may keep jumping to the next desired node by resetting *focus* by the above operation.

5.2. Guidelines

Views are the core of our framework for evidence exploration. We study two aspects of using views: defining single views and organizing multiple views.

Defining views. We expect views be smaller than the evidence. Three relational graph algebra operators—selection, projection, and grouping—generate smaller graphs than the input. They represent three ways to observe a big graph:

- Selection returns a subgraph. It can be used to restrict the scope of exploration, or to locate a portion of interest. Examples of usage include looking at the part of the evidence that involves a certain sub-formula, and searching for paths containing transitions with certain actions or states with certain properties.
- Projection trims the attributes. Different projections provide different angles of observation. In addition to the above-mentioned state and formula projections, we can project the evidence by subprocesses to study the dynamics of a subsystem in a concurrent system.
- Grouping partitions the graph. It can be used to provide a logical overview of the evidence, or to summarize the evidence by aggregate attributes. Navigation through the evidence by big jumps instead of single steps becomes possible.

Other operations may also be used to define views, but the above three must be the outermost operations in a definition to get smaller views.

Organizing views. Views can be used independently for orthogonal observations. Through different views generated from the same evidence or existing views, users study the evidence from different angles and focus their attention on only one dimension at a time without interference from other dimensions. As we have already seen, users can separately study system dynamics by state projection and/or formula dynamics by formula projection.

Multiple views can also be used cooperatively to accomplish complex tasks. For examples,

- Views can be defined on other views, forming a hierarchy of views. In particular when a grouping is built upon another, the hierarchy yields layers of finer to coarser logical resolutions.
- By selecting the subgraph corresponding to a node in a grouping, we do the reverse of grouping: zooming in to see the finer details. We can explore up and down a hierarchy of different logical resolutions by alternatively zooming out and zooming in.
- An entry in a projection corresponds to several entries in the view being projected. By selecting nodes or edges corresponding to an element in a projection, we effectively use the projection as an index to the base view.

6. Conclusions

We have described an algebraic framework and associated tool support for evidence exploration. Our focus has been on the exploration's *operational* aspects. For the visual aspects, we have drawn on the results of graph drawing [17] and information visualization [24] in building our tool.

Related work. There is some similarity between our approach and work in theorem proving and logic programming on proof visualization and navigation (see e.g. [1, 14]). However, our work is more concerned with proof interpretation, while the latter mostly deals with proof generation.

Various forms of evidence have appeared in the literature. One of the earliest is the tableau [11, 26, 29], which can be considered an unfolding of evidence to a tree whose leaves can point back to their ancestors. Conversely, an evidence can be obtained from a tableau by merging sequents (nodes in a tableau) that share the same state and formula components. Among others, [7] introduces a proof structure called "and-or Kripke structure"; [20] develops a symbolic proof system in terms of alternating automata and parity games; [21] defines a proof system for LTL using a computational model called "just discrete system"; [28] defines "support set" on Boolean equation systems.

We did not discuss how to generate evidence and simply assumed it is readily available as the byproduct of model checking, an assumption validated by Theorem 1 (soundness and completeness of evidence). In explicit-state model checking, evidence is materialized and can be extracted in time linear in its size. Some examples of evidence generation during or after model checking are: [18] extracts evidence from global model checking to produce witnesses and counterexamples; [23] uses logic-programming justification to regenerate evidence from memo tables; the performance of this approach is improved by [15]; and [28] shows how to modify an automaton-based model-checking algorithm for the purpose of evidence generation.

Although the exact phrase "relational graph algebra" has never been spelled out before, the concept is not new, and is inherently classical since both graph and relational algebra are classical. Graphs are naturally expressed by relations in database applications. To support applications on network structures, the operators of relational algebra have been extended to manipulate large graphs and define new graph views upon existing ones (see e.g. [16]). The terminology in the field of image pattern recognition is slightly different, where relational graph refers to encoding a relation by a graph, and the term corresponding to our relational graph is "attributed" relational graph whose nodes and edges are labeled with attributes [3].

The Evidence Explorer has enhanced the usability of evidence exploration by a few simple predefined views, but it does not yet allow users to specify their own views. When the potential of the framework is fully exploited, Evidence Explorer will make it a more pleasant experience to understand model checking by manipulating and interpreting evidences.

Acknowledgements. The authors would like to thank Stephanie White for her valuable comments. Research supported in part by NSF Grants CCR-9988155 and CCR-9705998, and ARO grants DAAD190110003 and DAAD190110019.

References

[1] *International Workshop on User Interfaces for Theorem Provers (UITP).* 1995–1998.

[2] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. *ACM SIGPLAN Notices*, 34(10):207–222, 1999.

[3] D. H. Ballard and C. M. Brown. *Computer Vision*. Prentice Hall, 1982.

[4] S. Basu and S. A. Smolka. Model checking the Java Meta-Locking algorithm. 2002. submitted.

[5] S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, pages 342–350, Edinburgh, Scotland, Apr. 2000.

[6] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–163, Mar. 2001.

[7] G. Bhat and R. Cleaveland. Efficient local model-checking for fragments of the modal mu-calculus. In *TACAS 1996*, LNCS 1055, pages 107–126, 1996.

[8] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. On ACTL formulas having linear counterexamples. *Journal of Computer and System Sciences*, 62(3):463–515, May 2001.

[9] E. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-like counterexamples in model checking. In *Symposium on Logic in Computer Science (LICS'2002)*, Copenhagen, Denmark, July 2002.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, LNCS 131, pages 52–71. Springer Verlag, 1981.

[11] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[12] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley, 7 edition, 2000.

[13] M. J. Fischer and R. E. Ladner. Propositional dynamic logic of regular programs. *Journal of Computer and System Sciences*, 18(2):194–211, 1979.

[14] J. A. Goguen. Social and semiotic analyses for theorem prover user interface design 1. *Formal Aspects of Computing*, 11(3):272–301, 1999.

[15] H.-F. Guo, C. R. Ramakrishnan, and I. V. Ramakrishnan. Speculative beats conservative justification. In *ICLP 2001*, LNCS 2237, pages 150–165, 2001.

[16] A. Gutiérrez, P. Pucheral, H. Steffen, and J.-M. Thévenin. Database Graph Views: A Practical Model to Manage Persistent Graphs. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 391–402, Santiago, Chile, 1994.

[17] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer Verlag, 2001.

[18] A. Kick. Generation of witnesses for global μ-calculus model checking. Available at http://liinwww.ira.uka.de/$\sim$kick/cavf.ps, 1995.

[19] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[20] K. S. Namjoshi. Certifying model checkers. In *CAV 2001*, LNCS 2102, pages 2–13, 2001.

[21] D. Peled, A. Pnueli, and L. Zuck. From falsification to verification. In *FSTTCS 2001*, LNCS 2245, pages 292–304, 2001.

[22] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, LNCS 137, Berlin, 1982.

[23] A. Roychoudhury, C. Ramakrishnan, and I. Ramakrishnan. Justifying proofs using memo tables. In *Proc. of Second International Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 178–189, Sept. 2000.

[24] R. Spence. *Information Visualization*. ACM Press, 2001.

[25] C. Stirling. Modal and temporal logics for processes. In F. Moller and G. Birtwistle, editors, *Logics for concurrency: structure versus automata*, LNCS 1055, pages 149–237. Springer, 1996.

[26] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1):161–177, 1991.

[27] R. S. Streett and E. A. Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–342, 1989.

[28] L. Tan and R. Cleaveland. Evidence-based model checking. In *CAV 2002*, LNCS 2404, pages 455–470, 2002.

[29] G. Winskel. A note on model checking the modal ν-calculus. *Theoretical Computer Science*, 83:157–167, 1991.