Model and Program Repair via SAT Solving

Paul Attie, Ali Cherri, Kinan Dak Al Bab, Mohamad Sakr, and Jad Saklawi
Department of Computer Science, American University of Beirut, Lebanon

***Abstract*—**

We consider the *subtractive model repair problem*: given a finite Kripke structure M and a CTL formula η, determine if M contains a substructure M' that satisfies η. Thus, M can be repaired to satisfy η by deleting states and/or transitions. We give a reduction to boolean satisfiability, and implement the repair method using this reduction. We also extend the basic repair method in three directions: (1) the use of abstraction, and (2) the repair of concurrent Kripke structures and concurrent programs, and (3) the repair of hierarchical Kripke structures. These last two extensions both avoid state-explosion.

I. Introduction and Motivation

We consider the following *subtractive model repair problem*: given a Kripke structure M and a specification given as a CTL formula η, does there exist a substructure M' of M (obtained by removing transitions and states from M, but leaving at least one initial state) such that M' satisfies η? Our algorithm computes (in deterministic time polynomial in the sizes of M and η) a propositional formula $repair(M, \eta)$ such that $repair(M, \eta)$ is satisfiable iff M contains a substructure M' that satisfies η. We submit $repair(M, \eta)$ to a SAT solver, and a satisfying assignment for $repair(M, \eta)$, if it exists, determines which transitions and states must be removed from M to produce M'. Thus, a single run of a complete SAT solver is sufficient to find a repair, if one exists.

We extend the method to use abstraction mappings (i.e., repair an abstract structure and then concretize to obtain a repair of the original structure), to repair hierarchical Kripke structures [1], and to repair concurrent Kripke structures and concurrent programs. This last extension uses the pairwise approach of [2]–[4], thereby avoiding state-explosion. We have implemented the repair method as a GUI-based tool, Eshmun[1], which we used to produce the examples in this paper. Eshmun displays a Kripke structure, and shows a repair by presenting the states/transitions to be deleted using dashed lines.

The rest of the paper is as follows. Sect. II provides brief technical preliminaries. Sect. III states the model repair problem and gives its complexity. Sect. IV presents our model repair method. Sect. V shows how to use abstraction in conjunction with repair. Sect. VI extends the repair method to concurrent Kripke structures and concurrent programs. Sect. VII briefly outlines our extension of the repair method to hierarchical Kripke structures. Sect. VIII discusses our implementation, Eshmun, and gives experimental results. Sect. IX discusses related work. Sect. X concludes. The extensions to concurrent and hierarchical Kripke structures do not incur state-explosion. Full proofs of all theorems, the full presentation of hierarchical repair, and more examples, can all be found in the full paper. Both Eshmun and the full paper can be downloaded from http://eshmuntool.blogspot.com/.

[1]Eshmun is the name of the Phoenician god of healing

II. Preliminaries

We assume knowledge of the temporal logic CTL [5], given by the following grammar:

$$\varphi ::= \mathsf{true} \mid \mathsf{false} \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathsf{AX}\varphi \mid \mathsf{EX}\varphi \mid \mathsf{A}[\varphi \mathsf{V} \varphi] \mid \mathsf{E}[\varphi \mathsf{V} \varphi]$$

where $p \in AP$, a set of atomic propositions. $[p\mathsf{V}q]$ (p "releases" q) means that q must hold up to and including the first state where p holds, and forever if p never holds. CTL semantics are given as usual w.r.t. a Kripke structure $M = (S_0, S, R, L, AP)$ where $S_0 \subseteq S$ is the (nonempty) set of start states, S is the set of states, $R \subseteq S \times S$ is the transition relation and $L : S \to 2^{AP}$ is the labeling function. In the sequel, we use M, M', M'' etc. for Kripke structures, and φ, ψ, η for CTL formulae. Write $M, s \models \varphi$ when φ holds in state s of M, and $M \models \varphi$ to abbreviate $\forall s \in S_0 : M, s \models \varphi$. We use the standard definition of $\models$, cf. [5], and we assume that all structures are total (every state has some outgoing transition). We use the abbreviations $\mathsf{A}[\varphi \mathsf{U} \psi]$ for $\neg\mathsf{E}[\neg\varphi \mathsf{V} \neg\psi]$, $\mathsf{E}[\varphi \mathsf{U} \psi]$ for $\neg\mathsf{A}[\neg\varphi \mathsf{V} \neg\psi]$, $\mathsf{AF}\varphi$ for $\mathsf{A}[\mathsf{true}\mathsf{U}\varphi]$, $\mathsf{EF}\varphi$ for $\mathsf{E}[\mathsf{true}\mathsf{U}\varphi]$, $\mathsf{AG}\varphi$ for $\mathsf{A}[\mathsf{false}\mathsf{V}\varphi]$, $\mathsf{EG}\varphi$ for $\mathsf{E}[\mathsf{false}\mathsf{V}\varphi]$. We also use tt, ff for semantic true, false, respectively.

Definition 1 (Formula expansion): Given a CTL formula φ, its set of subformulae $sub(\varphi)$ is defined as follows:

- $sub(p) = p$ where p is true, false, or an atomic proposition
- $sub(\neg\varphi) = \{\neg\varphi\} \cup sub(\varphi)$
- $sub(\varphi \wedge \psi) = \{\varphi \wedge \psi\} \cup sub(\varphi) \cup sub(\psi)$
- $sub(\varphi \vee \psi) = \{\varphi \vee \psi\} \cup sub(\varphi) \cup sub(\psi)$
- $sub(\mathsf{AX}\varphi) = \{\mathsf{AX}\varphi\} \cup sub(\varphi)$
- $sub(\mathsf{EX}\varphi) = \{\mathsf{EX}\varphi\} \cup sub(\varphi)$
- $sub(\mathsf{A}[\varphi \mathsf{V} \psi]) = \{\mathsf{A}[\varphi \mathsf{V} \psi]\} \cup sub(\varphi) \cup sub(\psi)$
- $sub(\mathsf{E}[\varphi \mathsf{V} \psi]) = \{\mathsf{E}[\varphi \mathsf{V} \psi]\} \cup sub(\varphi) \cup sub(\psi)$

III. The Model Repair Problem

Given Kripke structure M and CTL formula η, we want to find a *substructure* M' such that $M' \models \eta$.

Definition 2 (Substructure): Given two Kripke structures $M = (S_0, S, R, L, AP)$ and $M' = (S_0', S', R,' L', AP')$, say that M' *is a substructure of* M, denoted $M' \subseteq M$, iff $S_0' \subseteq S_0$, $S' \subseteq S$, $R' \subseteq R$, $L' = L \restriction S'$, and $AP' = AP$, where $\restriction$

denotes domain restriction, i.e., $L' : S' \to 2^{AP}$ is given by $L'(s) = L(s)$ for all $s \in S'$.

Definition 3 (Repairable): Given Kripke structure M and CTL formula η, M is *repairable* w.r.t. η if there exists a Kripke structure M' such that M' is total, $M' \subseteq M$, and $M' \models \eta$.

Definition 4 (Model Repair Problem): For Kripke structure M and CTL formula η, we use $\langle M, \eta \rangle$ for the corresponding repair problem. The decision version of repair problem $\langle M, \eta \rangle$ is to decide if M is repairable w.r.t. η. The functional version of repair problem $\langle M, \eta \rangle$ is to return an M' that satisfies Def. 3, in the case that M is repairable w.r.t. η.

Theorem 1: The decision version of the model repair problem is NP-complete.

Proof sketch: For NP-membership, given a candidate solution M', it is obvious that "M' is total" and $M' \subseteq M$ are checkable in polynomial time. $M' \models \eta$ can be checked in polynomial time using the CTL model checking algorithm of [6].

For NP-hardness, we reduce 3SAT to model repair. Consider a 3CNF formula $f = \bigwedge_{1 \le i \le n} (a_i \vee b_i \vee c_i)$, where a_i, b_i, c_i are literals over a set $\{x_1, \ldots, x_m\}$ of propositions, i.e., each of a_i, b_i, c_i is x_j or $\neg x_j$, for some $j = 1, \ldots, m$.

We map f to $\langle M, \eta \rangle$ as follows. Let $M = (\{s_0\}, S, R, L, AP)$, $S = \{s_0, s_1, \ldots, s_m, t_1, \ldots, t_m\}$, and $R = \{(s_0, s_1), \ldots, (s_0, s_m), (s_1, t_1), \ldots, (s_m, t_m)\}$, i.e., transitions from s_0 to each of $s_1, \ldots, s_m$, and a transition from each s_i to t_i for $i = 1, \ldots, m$. In addition each state has a self-loop. The set AP of atomic propositions is $\{p_1, \ldots, p_m, q_1, \ldots, q_m\}$. These propositions are distinct from the $x_1, \ldots, x_m$ used in f. L is given by $L(s_0) = \emptyset$, $L(s_j) = p_j$ and $L(t_j) = q_j$ for $j = 1, \ldots, m$. Also $\eta = \bigwedge_{1 \le i \le n} (\varphi_i^1 \vee \varphi_i^2 \vee \varphi_i^3)$ is as follows. φ_i^1 is defined by a_i: if $a_i = x_j$ then $\varphi_i^1 = \mathsf{AG}(p_j \Rightarrow \mathsf{EX} q_j)$, and if $a_i = \neg x_j$ then $\varphi_i^1 = \mathsf{AG}(p_j \Rightarrow \mathsf{AX} \neg q_j)$. Likewise b_i defines φ_i^2 and c_i defines φ_i^3.

The key idea is that x_i is assigned tt iff the edge from s_i to t_i is not deleted. Hence, if transitions can be deleted in a way such that $f = \bigwedge_{1 \le i \le n} (\varphi_i^1 \vee \varphi_i^2 \vee \varphi_i^3)$ holds, then it follows that $\eta = \bigwedge_{1 \le i \le n} (a_i \vee b_i \vee c_i)$ also holds, since each literal in η corresponds to a single φ_i^k ($k = 1, 2, 3$). Likewise a satisfying assignment to η defines a way of deleting transitions so that f holds. It is obvious that this reduction can be computed in polynomial time. □

IV. The Model Repair Algorithm

Given an instance $\langle M, \eta \rangle$ of model repair, we define a boolean formula $repair(M, \eta)$ such that $repair(M, \eta)$ is satisfiable iff $\langle M, \eta \rangle$ has a solution. $repair(M, \eta)$ contains the following propositions, which have the indicated meanings, where $\mathcal{V}$ is a satisfying boolean assignment for $repair(M, \eta)$, and M' is the repaired structure:

1) $E_{s,t}$, $(s,t) \in R$: transition (s,t) is retained in M' iff $\mathcal{V}(E_{s,t}) = tt$
2) $X_s, s \in S$: state s is retained in M' iff $\mathcal{V}(X_s) = tt$
3) $X_{s,\psi}, s \in S, \psi \in sub(\eta)$: $M', s \models \psi$, i.e., ψ holds in state s of M' iff $\mathcal{V}(X_{s,\psi}) = tt$
4) $X^n_{s,\psi} : s \in S, 0 \le n \le |S|$, and $\psi \in sub(\eta)$ has the form $\mathsf{A}[\varphi \mathsf{V} \varphi']$ or $\mathsf{E}[\varphi \mathsf{V} \varphi']$: $X^n_{s,\psi}$ is used to propagate release formulae (AV or EV) for as long as necessary to determine their truth, i.e., $|S|$ times.

The repair formula $repair(M, \eta)$ encodes the usual local constraints, e.g., $\mathsf{AX}\varphi$ holds in s iff φ holds in all successors of s, i.e., all t such that $(s,t) \in R$. We modify these however, to account for transition deletion. So, the local constraint for AX becomes $\mathsf{AX}\varphi$ holds in s iff φ holds in all successors of s *after* transition deletion, i.e., instead of $X_{s,\mathsf{AX}\varphi} \equiv \bigwedge_{t|s \to t} X_{t,\varphi}$, we have $X_{s,\mathsf{AX}\varphi} \equiv \bigwedge_{t|s \to t} (E_{s,t} \Rightarrow X_{t,\varphi})$. We use $s \to t$ to abbreviate $(s,t) \in R$. EX is treated similarly. AV, EV are dealt with using a "counting" technique, discussed below. There are also clauses for propositional consistency, propositional labeling, initial states, and to require that the repaired structure M' be total.

Definition 5 ($repair(M, \eta)$): Let η be a CTL formula and $M = (S_0, S, R, L, AP)$ be a Kripke structure. $repair(M, \eta)$ is the conjunction of all the propositional formulae listed below. These are grouped into sections, where each section deals with one issue, e.g., propositional consistency. s, t implicitly range over S. Other ranges are explicitly given.

1) some initial state s_0 is not deleted

   $\bigvee_{s_0 \in S_0} X_{s_0}$

2) M' satisfies η, i.e., undeleted initial states satisfy η

   for all $s_0 \in S_0 : X_{s_0} \Rightarrow X_{s_0,\eta}$

3) M' is total, i.e., each retained state has at least one outgoing transition, to some other retained state

   for all $s \in S : X_s \equiv \bigvee_{t|s \to t} (E_{s,t} \wedge X_t)$

4) If an edge is retained then both its source and target states are retained

   for all $(s,t) \in R : E_{s,t} \Rightarrow (X_s \wedge X_t)$

5) Propositional labeling

   for all $p \in AP \cap L(s)$: $X_{s,p}$
   for all $p \in AP - L(s)$: $\neg X_{s,p}$

6) Propositional consistency

   for all $\neg\varphi \in sub(\eta)$: $X_{s,\neg\varphi} \equiv \neg X_{s,\varphi}$
   for all $\varphi \vee \psi \in sub(\eta)$: $X_{s,\varphi\vee\psi} \equiv X_{s,\varphi} \vee X_{s,\psi}$
   for all $\varphi \wedge \psi \in sub(\eta)$: $X_{s,\varphi\wedge\psi} \equiv X_{s,\varphi} \wedge X_{s,\psi}$

7) Nexttime formulae

   for all $\mathsf{AX}\varphi \in sub(\eta)$: $X_{s,\mathsf{AX}\varphi} \equiv \bigwedge_{t|s \to t} (E_{s,t} \Rightarrow X_{t,\varphi})$
   for all $\mathsf{EX}\varphi \in sub(\eta)$: $X_{s,\mathsf{EX}\varphi} \equiv \bigvee_{t|s \to t} (E_{s,t} \wedge X_{t,\varphi})$

8) Release formulae. Let $n = |S|$, i.e., the number of states in M.

   for all $\mathsf{A}[\varphi \mathsf{V} \psi] \in sub(\eta)$, $m \in \{1 \ldots n\}$:
   $X_{s,\mathsf{A}[\varphi\mathsf{V}\psi]} \equiv X^n_{s,\mathsf{A}[\varphi\mathsf{V}\psi]}$
   $X^m_{s,\mathsf{A}[\varphi\mathsf{V}\psi]} \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigwedge_{t|s \to t} (E_{s,t} \Rightarrow X^{m-1}_{t,\mathsf{A}[\varphi\mathsf{V}\psi]}))$
   $X^0_{s,\mathsf{A}[\varphi\mathsf{V}\psi]} \equiv X_{s,\psi}$
   for all $\mathsf{E}[\varphi \mathsf{V} \psi] \in sub(\eta)$, $m \in \{1 \ldots n\}$:
   $X_{s,\mathsf{E}[\varphi\mathsf{V}\psi]} \equiv X^n_{s,\mathsf{E}[\varphi\mathsf{V}\psi]}$

$$X^m_{s,\mathsf{E}[\varphi \mathsf{V} \psi]} \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigvee_{t|s \to t}(E_{s,t} \wedge X^{m-1}_{t,\mathsf{E}[\varphi \mathsf{V} \psi]}))$$
$$X^0_{s,\mathsf{E}[\varphi \mathsf{V} \psi]} \equiv X_{s,\psi}$$

We handle the $[\varphi \mathsf{V} \psi]$ modality by "counting down", as follows. Along each path, either (1) a state is reached where $[\varphi \mathsf{V} \psi]$ is discharged ($\varphi \wedge \psi$), or (2) $[\varphi \mathsf{V} \psi]$ is shown to be false ($\neg\varphi \wedge \neg\psi$), or (3) some state eventually repeats. In case (3), we know that $[\varphi \mathsf{V} \psi]$ also holds along this path. Thus, by expanding the release modality up to $|S|$ times, we ensure that the third case holds if the first two have not yet resolved the truth of $[\varphi \mathsf{V} \psi]$ along the path in question. We therefore use a version of $X_{s,\mathsf{A}[\varphi \mathsf{V} \psi]}$ that is superscripted with an integer between 0 and $|S|$. This imposes a "well foundedness" on the $X^m_{s,\mathsf{A}[\varphi \mathsf{V} \psi]}$ propositions, and prevents for example, a cycle along which ψ holds in all states and yet the $X_{s,\mathsf{A}[\varphi \mathsf{V} \psi]}$ are assigned false in all states.

States rendered unreachable are still required to have some outgoing transition, but this does not affect the final result, since unreachable states do not affect reachable ones. Hence an unreachable state can retain its successors (recall that the initial structure M is total) without impacting the repair.

A satisfying assignment $\mathcal{V}$ of $\mathit{repair}(M, \eta)$ defines a solution to model repair, denoted by $\mathit{model}(M, \mathcal{V})$:

Definition 6 ($\mathit{model}(M, \mathcal{V})$): $\mathit{model}(M, \mathcal{V})$ is the Kripke structure $(S'_0, S', R,' L', AP')$, where $S'_0 = \{s_0 \mid s_0 \in S_0 \wedge \mathcal{V}(X_{s_0}) = tt\}$, $S' = \{s \mid s \in S \wedge \mathcal{V}(X_s) = tt\}$, $R' = \{(s,t) \mid (s,t) \in R \wedge \mathcal{V}(E_{s,t}) = tt\}$, $L' = L \restriction S'$, and $AP' = AP$.

Fig. 1 gives our model repair algorithm, which is sound, relatively complete, and implements a polynomial-time reduction of repair to satisfiability.

$\underline{\mathsf{Repair}(M, \eta)}$:
if $M, S_0 \models \eta$ (by model checking) **return** M;
submit $\mathit{repair}(M, \eta)$ to a SAT-solver;
if solver returns satisfying assignment $\mathcal{V}$ **then**
  return $M' = \mathit{model}(M, \mathcal{V})$
else return "the model cannot be repaired"

Fig. 1: The model repair algorithm.

Proposition 1: The length of $\mathit{repair}(M, \eta)$ is $O(|S|^2 \times |\eta| \times d + |S| \times |AP| + |R|)$, where $|S|$ is the number of states in S, $|R|$ is the number of transitions in R, $|\eta|$ is the length of η, $|AP|$ is the number of atomic propositions in AP, and d is the outdegree of M, i.e., the maximum of the number of successors of any state in M.

Theorem 2 (Soundness): Assume that there exists a satisfying assignment $\mathcal{V}$ for $\mathit{repair}(M, \eta)$. Let $M' = (S'_0, S', R,' L', AP') = \mathit{model}(M, \mathcal{V})$. Then for all states $s \in S'$ and CTL formulae $\xi \in \mathit{sub}(\eta)$:

$$\mathcal{V}(X_{s,\xi}) = tt \text{ iff } M', s \models \xi.$$

Proof sketch: Proof is by induction on the structure of ξ. The cases of $\neg, \wedge, \vee$ follow immediately from the induction hypothesis and the meaning of $\neg, \wedge, \vee$. We illustrate the argument for $\xi = \mathsf{AX}\varphi$: $\mathcal{V}(X_{s,\xi}) = tt$ iff $\mathcal{V}(X_{s,\mathsf{AX}\varphi}) = tt$ iff (by Def. 5) $\bigwedge_{t|s \to t} \mathcal{V}(E_{s,t} \Rightarrow X_{t,\varphi}) = tt$ iff $\bigwedge_{t|s \to t}(\mathcal{V}(E_{s,t}) = tt \Rightarrow \mathcal{V}(X_{t,\varphi}) = tt)$ iff (apply ind. hyp. and Def. 6) $\bigwedge_{t|s \to t}((s,t) \in R' \Rightarrow M', t \models \varphi)$ iff $M', s \models \mathsf{AX}\varphi$ iff $M', s \models \xi$. The argument for $\xi = \mathsf{EX}\varphi$ is similar.

For $\xi = \mathsf{A}[\varphi \mathsf{V} \psi]$ and $\xi = \mathsf{E}[\varphi \mathsf{V} \psi]$, we apply the induction hypothesis along a path starting from s, of length at most $|S|$, since this guarantees a repeated state, which terminates the "countdown" using the integer superscripts. □

Corollary 1 (Soundness): If $\mathsf{Repair}(M, \eta)$ returns a structure M', then (1) M' is total, (2) $M' \subseteq M$, (3) $M' \models \eta$, and (4) M is repairable.

Proof sketch: (1) follows from Clause. 3 (M' is total) of Def. 5. (2) holds since M' is derived from M by deleting transitions and states. For (3), let $s_0 \in S'_0$. Since s_0 was not deleted, we have $\mathcal{V}(X_{s_0}) = tt$. Hence, by Clause. 2 of Def. 5, $\mathcal{V}(X_{s_0,\eta}) = tt$. Hence, by Th. 2, $M', s_0 \models \eta$. Hence $M' \models \eta$ since s_0 was arbitrarily chosen. Finally, (4) follows from (1)–(3) and Def. 3. □

Theorem 3 (Relative completeness): If M is repairable with respect to η, and $\mathsf{Repair}(M, \eta)$ invokes a complete SAT-solver, then $\mathsf{Repair}(M, \eta)$ returns a Kripke structure M' such that M' is total, $M' \subseteq M$, and $M' \models \eta$.

Proof sketch: Assume that M is repairable with respect to η. By Def. 3, there exists a total $M'' \subseteq M$ such that $M'' \models \eta$. We define a satisfying valuation $\mathcal{V}$ for $\mathit{repair}(M, \eta)$ as follows.

Assign tt to $E_{s,t}$ for every edge (s,t) of M that is also in M'' (i.e., not deleted) and ff to every $E_{s,t}$ for every edge (s,t) of M that is not in M'' (i.e., deleted). Likewise tt to X_s for every s in M'' and ff for every s not in M''. Assign tt to $X_{s_0,\eta}$ for all initial states s''_0 of M''. Consider an execution of the CTL model checking algorithm of Clarke, Emerson, and Sistla [6] for checking $M'' \models \eta$. This algorithm will assign a value to every formula φ in $\mathit{sub}(\eta)$ in every state s of M''. Set $\mathcal{V}(X_{s,\varphi})$ to this value. Variables not given a truth value by the above procedure can be assigned an arbitrary value.

The above defines a satisfying assignment for $\mathit{repair}(M, \eta)$. Hence invoking a complete SAT solver will return some satisfying assignment for $\mathit{repair}(M, \eta)$, and so $\mathsf{Repair}(M, \eta)$ returns a structure M' (not necessarily M'') rather than "the model cannot be repaired." By Cor. 1, M' is total, $M' \subseteq M$, and $M' \models \eta$. □

A. Example: two-process mutual exclusion

Process P_i ($i = 1, 2$) has three atomic propositions, $\mathbf{Ni}, \mathbf{Ti}, \mathbf{Ci}$, and cycles through three local states: *neutral* (performing local computation, $\mathbf{Ni}$ is true and $\mathbf{Ti}, \mathbf{Ci}$ are false), *trying* (requested critical section, $\mathbf{Ti}$ is true and $\mathbf{Ni}, \mathbf{Ci}$ are false), and *critical* (inside critical section, $\mathbf{Ci}$ is true and $\mathbf{Ni}, \mathbf{Ti}$ are false). The CTL specification η is $\mathsf{AG}(\neg(\mathbf{C1} \wedge \mathbf{C2}))$, i.e., mutual exclusion: P_1 and P_2 are never simultaneously in their critical sections.

We use Eshmun to repair the Kripke structure M in Fig. 2, which violates $\mathsf{AG}(\neg(\mathbf{C1} \wedge \mathbf{C2}))$. M contains all transitions shown, and the repaired structure omits the dashed transitions. Initial states are green, transitions of each process all have the same color, and the text attached to each state consists of a name (e.g., $\mathbf{S0}, \mathbf{S1}$ etc.), and a list of the atomic propositions that hold in the state (e.g., ($\mathbf{N1}$,$\mathbf{N2}$), ($\mathbf{T1}$,$\mathbf{N2}$),

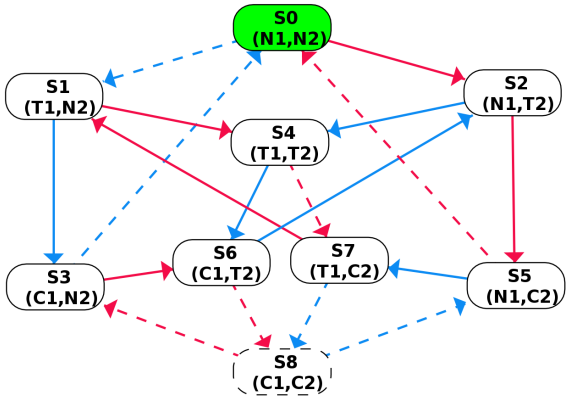


Fig. 2: Repair of mutual exclusion Kripke structure

etc.). The violating state **S8** (where $\mathbf{C1}, \mathbf{C2}$ both hold) is now unreachable, and so $\mathsf{AG}(\neg(\mathbf{C1} \wedge \mathbf{C2}))$ holds. This repair is however, overly restrictive, e.g., it prevents P_1 from initially requesting the critical section, since the transition $\mathbf{S0} \rightarrow \mathbf{S1}$ is deleted. Both processes should be able to request the critical section (move to **Ti**) whenever they are in **Ni**. We show below how to improve the quality of the repair.

V. Repair using Abstractions

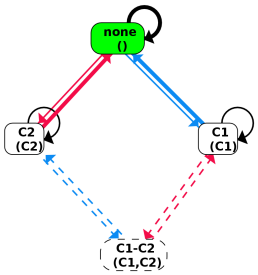
We now extend the repair method to use abstractions, i.e., repair a structure abstracted from M and then concretize the resulting repair to obtain a repair of M. The purpose of abstractions is two-fold:

1) *Reduce the size of M, and so reduce the length of $repair(M, \eta)$.* $repair(M, \eta)$ has length quadratic in $|M|$, so repairing an abstract structure significantly increases the size of structures that can be handled.
2) *"focus" the attention of the repair algorithm, which in practice produces "better" repairs.* Our repair method nondeterministically chooses a repair from those available, according to the valuation returned by the SAT solver. This repair may be undesirable, as it may result in restrictive behavior, e.g., alternate entry into the critical section. By constructing an abstract structure which tracks only the values of $\mathbf{C1}, \mathbf{C2}$, we obtain a better repair, which removes only the transitions that enter $\mathbf{C1C2}$.

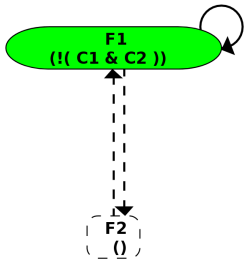
Eshmun implements two abstractions: *abstraction by label* preserves the values of all atomic propositions in η, and *abstraction by formula* preserves the values of all $\varphi \in AS_\eta$, where AS_η is a user-selected subset of the propositional subformulae of η. These abstractions are given by equivalence relations [7] over the set of states S, where equivalence is determined by the valuation of atomic propositions (abstraction by label) and subformulae (abstraction by formula) respectively [8]:

1) abstraction by label: $s \equiv_p t$ iff $L(s) \cap AP_\eta = L(t) \cap AP_\eta$
2) abstraction by formula: $s \equiv_f t$ iff $(\forall \varphi \in AS_\eta : s \models \varphi$ iff $t \models \varphi)$

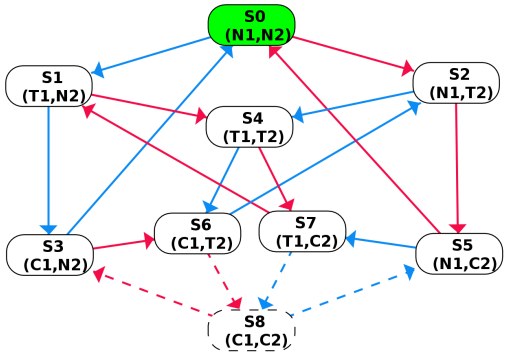
Let $\equiv$ be an equivalence relation over S, and let $[s]$ be the equivalence class of s in $\equiv$. The corresponding abstract



(a) Abstraction and repair by label



(b) Abstraction and repair by subformulae



(c) Concretized repair

Fig. 3: Abstract repair of the mutual exclusion structure

structure $\overline{M} = (\overline{S}_0, \overline{S}, \overline{R}, \overline{L}, AP_\eta)$ is obtained by taking the quotient of $M = (S_0, S, R, L, AP)$ by $\equiv$, as usual, i.e., $\overline{S} = \{[s] \mid s \in S\}$, $\overline{S_0} = \{[s_0] \mid s_0 \in S_0\}$, and $\overline{R} = \{([s],[t]) \mid (s,t) \in R\}$. For abstraction by label, $\overline{L} : \overline{S} \rightarrow AP_\eta$ is given by $\overline{L}([s]) = L(s) \cap AP_\eta$, where AP_η is the set of atomic propositions occurring in η. For abstraction by formula, $\overline{L}$ assigns truth values to the formulae being abstracted, i.e., $\overline{L} : \overline{S} \rightarrow AS_\eta$ is given by $\overline{L}([s]) = \{\varphi \mid \varphi \in AS_\eta \wedge \forall t \in [s] : t \models \varphi\}$. That is, abstract states are labeled by the set of formulae in AS_η that hold in all the corresponding concrete states. We do not need values for atomic propositions, since the values of the formulae in AS_η (in all states) determine the values of all temporal formulae. Hence Def. 5 is modified so that formulae in AS_η play the role of atomic propositions (Clauses 5, 6).

Abstract repair does not guarantee concrete repair. When we concretize the repair of $\overline{M}$, we obtain a *possible* repair of M, which we verify by model checking. We concretize as follows. The abstraction algorithm maintains a data structure that maps a transition in $\overline{M}$ to the set of corresponding transitions in M. If a transition in $\overline{M}$ is deleted by the repair of $\overline{M}$, then we delete all the corresponding transitions in M to construct the possible repair of M.

Consider structure M in Fig. 2, with $\eta = \mathsf{AG}(\neg(\mathbf{C1} \wedge \mathbf{C2}))$, and abstraction by label, i.e., equivalent states agree on both $\mathbf{C1}$ and $\mathbf{C2}$. The equivalence classes of $\equiv_p$ are: $\mathbf{none\_1} = \{\mathbf{N1N2}, \mathbf{N1T2}, \mathbf{T1N2}, \mathbf{T1T2}\}$, $\mathbf{C1} = \{\mathbf{C1N2}, \mathbf{C1T2}\}$, $\mathbf{C2} = \{\mathbf{N1C2}, \mathbf{T1C2}\}$, $\mathbf{C1\_C2} = \{\mathbf{C1C2}\}$. Fig. 3(a) shows the resulting abstract structure $\overline{M}$,

which has four states, corresponding to these equivalence classes. The repair of $\overline{M}$ is to delete the transitions shown dashed. Since a process must be able to exit its critical section, we checked **retain** for transitions $\mathbf{C1} \rightarrow \mathbf{none\_1}$, $\mathbf{C2} \rightarrow \mathbf{none\_1}$. (In Eshmun, we can require that a particular transition be retained, and the transition is then boldened, as in Fig. 3(a).) Now consider abstraction by the subformula $\mathbf{C1} \wedge \mathbf{C2}$. The equivalence classes of $\equiv_f$ are $\mathbf{none\_1} = \{\mathbf{N1N2}, \mathbf{N1T2}, \mathbf{T1N2}, \mathbf{T1T2}, \mathbf{C1N2}, \mathbf{C1T2}, \mathbf{N1C2}, \mathbf{T1C2}\}$ and $\mathbf{C1\_C2} = \{\mathbf{C1C2}\}$. Fig. 3(b) shows the resulting abstract structure $\overline{M}$, which has two states, corresponding to these equivalence classes. Figure 3(c) shows the concretized repair of the original structure. In this case, both abstractions give the same concrete repair, which is good in that it does not prevent either process from making a request (i.e., moving from neutral to trying) at any time.

VI. Repair of Concurrent Kripke Structures

We consider finite-state shared-memory concurrent programs $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ consisting of K sequential processes $P_1, \ldots, P_K$ running in parallel, together with a set St_P of starting global states. For each P_i, there is a finite set AP_i of *atomic propositions* that are *local to* P_i: only P_i can change the value of atomic propositions in AP_i. Other processes can read, but not change, these values. Local atomic propositions are not shared: $AP_i \cap AP_j = \emptyset$ when $i \neq j$. We also admit a set $SH = \{x_1, \ldots, x_m\}$ of shared variables. These can be read/written by all processes, and have values from finite domains[2]. We define the set of all atomic propositions $AP = AP_1 \cup \cdots \cup AP_K$.

Each P_i is a *synchronization skeleton* [5], that is, a directed multigraph where each node is a *local state* of P_i, which is labeled by a unique name s_i, and where each arc is labeled with a *guarded command* [9] $B_i \rightarrow A_i$ consisting of a guard B_i and corresponding action A_i. We write such an arc as the tuple $(s_i, B_i \rightarrow A_i, s_i')$, where s_i is the source node and s_i' is the target node. Each node must have at least one outgoing arc, i.e., a synchronization skeleton contains no "dead ends."

The read/write restrictions on atomic propositions are reflected in the syntax of processes: in an arc $(s_i, B_i \rightarrow A_i, s_i')$ of P_i, the guard B_i is a boolean formula over $AP - AP_i$ and SH, and the action A_i is a piece of terminating pseudocode that updates the shared variables SH.

Let S_i denote the set of local states of P_i. There is a mapping $V_i : S_i \rightarrow (AP_i \rightarrow \{\mathsf{true}, \mathsf{false}\})$ from local states of P_i to boolean valuations over AP_i: for $p_i \in AP_i$, $V_i(s_i)(p_i)$ is the value of atomic proposition p_i in s_i. Hence, as P_i executes transitions and changes its local state, the atomic propositions in AP_i are updated, since $V_i(s_i) \neq V_i(s_i')$ in general.

A *global state* is a tuple $(s_1, \ldots, s_K, v_1, \ldots, v_m)$ where s_i is the current local state of P_i and $v_1, \ldots, v_m$ is a list giving the current values of the shared variables in SH.

Let $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ be the current global state, and let P_i contain an arc from node s_i to node s_i' labeled with $B_i \rightarrow A_i$. If B_i holds in s, then a possible next state is $s' = (s_1, \ldots, s_i', \ldots, s_K, v_1', \ldots, v_m')$ where $v_1', \ldots, v_m'$ are the new values, respectively, for the shared variables $x_1, \ldots, x_m$ resulting from the execution of action A_i. The set of all (and only) such triples (s, i, s') constitutes the *next-state relation* of program P. As stated above, local atomic propositions AP_i are implicitly updated, since P_i changed its local state from s_i to s_i'.

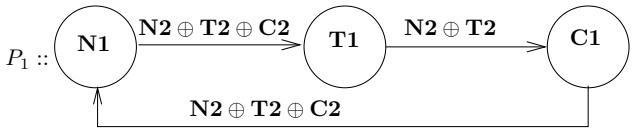
The appropriate semantic model for a concurrent program is a *multiprocess Kripke structure*, which is a Kripke structure that has its set AP of atomic propositions partitioned into $AP_1 \cup \cdots \cup AP_K$, and every transition is labeled with the index of a single process, which executes the transition. Only atomic propositions belonging to the executing process can be changed by a transition. Shared variables may also be present. The structure of Fig. 2 is a multiprocess Kripke structure. The semantics of a concurrent program $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ is given by its global state transition digram (GSTD): the smallest multiprocess Kripke structure M such that (1) the start states of M are St_P, and (2) M is closed under the next state relation of P. Effectively, M is obtained by "simulating" all possible executions of P from its start states St_P. A program satisfies a CTL formula η iff its GSTD does.

Conversely, given a multiprocess Kripke structure M, we can extract a concurrent program by projecting onto the individual process indices [5]. If M contains a transition from $s = (s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$ to $s' = (s_1, \ldots, s_i', \ldots, s_K, v_1', \ldots, v_m')$, then we can project this onto P_i as the arc $(s_i, B_i \rightarrow A_i, s_i')$, where B_i checks that the current global state is $(s_1, \ldots, s_i, \ldots, s_K, v_1, \ldots, v_m)$, and A_i is the multiple assignment $x_1, \ldots, x_m := v_1', \ldots, v_m'$, i.e., it assigns v_ℓ' to x_ℓ, $\ell = 1, \ldots, m$. For example, the concurrent program given in Fig. 4 is extracted from the multiprocess Kripke structure of Fig. 2. Each local state is shown labeled with the atomic propositions that it evaluates to true. Take for example the transition in Fig. 2 from **S2** to **S4**: this is a transition by P_1 from **N1** to **T1** which can be taken only when **T2** holds. It contributes an arc $(\mathbf{N1}, \mathbf{T2} \rightarrow \epsilon, \mathbf{T1})$ to P_1, where ϵ denotes the empty action, which changes nothing. Likewise the transition from **S0** to **S1** contributes $(\mathbf{N1}, \mathbf{N2} \rightarrow \epsilon, \mathbf{T1})$, and the transition from **S5** to **S7** contributes $(\mathbf{N1}, \mathbf{C2} \rightarrow \epsilon, \mathbf{T1})$. We group all these arcs into a single arc, whose label is $(\mathbf{N2} \rightarrow \epsilon) \oplus (\mathbf{T2} \rightarrow \epsilon) \oplus (\mathbf{C2} \rightarrow \epsilon)$. The $\oplus$ operator [2] is a "disjunction" of guarded commands: $(B_i \rightarrow A_i) \oplus (B_i' \rightarrow A_i')$ means nondeterministically select one of the two guarded commands whose guard holds, and execute the corresponding action. Using $\oplus$ means we have at most one arc, in each direction, between any pair of local states. To avoid clutter, we replace $B \rightarrow \epsilon$ by just B in the sequel, i.e., we omit empty actions.

In principle then, we can repair a concurrent program by (1) generating its GSTD M, (2) repairing M w.r.t. η to produce M', and (3) extracting a repaired program from M'. In practice, however, this quickly runs up against the *state explosion problem*: the size of M is exponential in the number of processes K. We avoid state explosion by using the approach of [2]–[4] for the synthesis and verification of concurrent programs based on *pairwise composition*, which we summarize in the next three paragraphs.

For each pair of processes P_i, P_j that interact directly, we provide as inputs a *pair-structure* $M_{ij} = (S_0^{ij}, S_{ij}, R_{ij}, L_{ij}, AP_{ij})$, which is a multiprocess Kripke

[2]In Eshmun all shared variables are boolean, i.e., atomic propositions that are not local to any process.



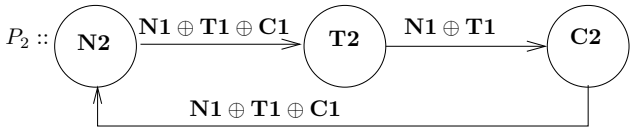


Fig. 4: Concurrent program extracted from Fig. 2

structure over P_i and P_j. M_{ij} defines the direct interaction between processes P_i and P_j. Hence, if P_i interacts directly with a third process P_k, then a second pair structure, $M_{ik} = (S_0^{ik}, S_{ik}, R_{ik}, L_{ik}, AP_{ik})$, over P_i, P_k, defines this interaction. So, M_{ij} and M_{ik} have the atomic propositions AP_i in common. Their shared variables are disjoint. The pairs of directly interacting processes are given by an *interaction relation* I, a symmetric relation over the set $\{1, \ldots, K\}$ of process indices. We extract from each pair-structure M_{ij} a corresponding *pair-program* $P_i^j \parallel P_j^i$, which consists of two *pair-processes* P_i^j and P_j^i. We then compose all of the pair-programs to produce the final concurrent program P. Composition is syntactic: the process P_i in P is the result of composing all the pair-processes $P_i^j, P_i^k, \ldots$. For example, 3 process mutual exclusion can be synthesized by having 3 pair-programs $P_1^2 \parallel P_2^1$, $P_1^3 \parallel P_3^1$, $P_2^3 \parallel P_3^2$, which implement mutual exclusion between each respective pair of processes. These are all isomorphic to Fig. 4, modulo index substitution. Then, the 3 process solution $P = P_1 \parallel P_2 \parallel P_3$ is given in Fig. 5, where only P_1 is shown, P_2 and P_3 being isomorphic to P_1 modulo index substitution. P_1 results from composing P_1^2 and P_1^3. In P_1, the arc from $\mathbf{T1}$ to $\mathbf{C1}$ is the composition (using the $\otimes$ operator [2]) of "corresponding" arcs in P_1^2 and P_1^3. For example, the arc from $\mathbf{T1}$ to $\mathbf{C1}$ in P_1^2, namely $(\mathbf{T1}, \mathbf{N2} \oplus \mathbf{T2}, \mathbf{C1})$, and the arc from $\mathbf{T1}$ to $\mathbf{C1}$ in P_1^3, namely $(\mathbf{T1}, \mathbf{N3} \oplus \mathbf{T3}, \mathbf{C1})$ are corresponding arcs, since they have the same source and target local states, namely $\mathbf{T1}$ and $\mathbf{C1}$. Their composition is given by the arc $(\mathbf{T1}, (\mathbf{N2} \oplus \mathbf{T2}) \otimes (\mathbf{N3} \oplus \mathbf{T3}), \mathbf{C1})$. The meaning of $(B_i \rightarrow A_i) \otimes (B_i' \rightarrow A_i')$ is that both B_i and B_i' must hold, and then A_i and A_i' must be executed concurrently. It is a "conjunction" of guarded commands. Hence, the meaning of $(\mathbf{T1}, (\mathbf{N2} \oplus \mathbf{T2}) \otimes (\mathbf{N3} \oplus \mathbf{T3}), \mathbf{C1})$ is that P_1 can enter its critical state $\mathbf{C1}$ iff P_2 is in either its neutral state $\mathbf{N2}$ or it trying state $\mathbf{T2}$, and also P_3 is in either its neutral state $\mathbf{N3}$ or its trying state $\mathbf{T3}$. Recall that the actions are omitted since they are all ϵ. The other two arcs of P_1 are similarly constructed. We use $\otimes$ only when the actions of the guarded commands update disjoint sets of variables, so that the result of execution is always well-defined.

Let $I(i) = \{j \mid (i,j) \in I\}$, so that $I(i)$ is the set of proceses that P_i interacts directly with. For each $j \in I(i)$, we create and repair pair-structure M_{ij} w.r.t. a *pair-specification* η_{ij}. From M_{ij} we extract pair-program $P_i^j \parallel P_j^i$. Process P_i in the overall large program is obtained by composing the pair-processes P_i^j for all $j \in I(i)$, as follows, [4, Def. 15, pairwise synthesis]:

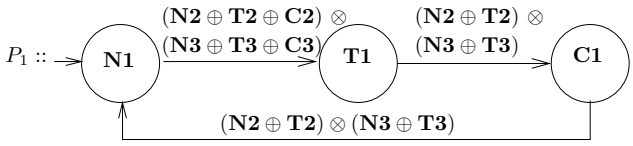


Fig. 5: Concurrent program extracted from Fig. 2

P_i contains an arc from s_i to t_i with label $\bigotimes_{j \in I(i)} \bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \rightarrow A_{i,\ell}^j$

iff

for $j \in I(i)$: P_i^j contains an arc from s_i to t_i with label $\bigoplus_{\ell \in [1:n_j]} B_{i,\ell}^j \rightarrow A_{i,\ell}^j$.

Hence the inputs to our method are the pair-specifications η_{ij}, the pair-structures M_{ij}, and the interaction relation I.

Recall that two arcs in P_i^j, P_i^k correspond iff they have the same source and target nodes. An arc in P_i is then a composition of corresponding arcs in all the $P_i^j, j \in I(i)$. For this composition to be possible, it must be that the corresponding arcs all exist. We must therefore check that, for all $j, k \in I(i)$, that if M_{ij} contains some transition by P_i^j in which it changes its local state from s_i to t_i, then M_{ik} also contains some transition by P_i^k in which it changes its local state from s_i to t_i. This ensures that every set of corresponding arcs contains a representative from each pair-program P_i^j for all $j \in I(i)$, and so our definition of pairwise synthesis produces a well-defined result. We call this the *process graph consistency constraint*, since it states, in effect, that P_i^j, P_i^k have the same graph: the result of removing all arc labels. Consider again the three process mutual exclusion example, and suppose that M_{12} contains a transition in which P_1^2 moves from $\mathbf{T1}$ to $\mathbf{C1}$, but that M_{13} does not contain a transition in which P_1^3 moves from $\mathbf{T1}$ to $\mathbf{C1}$. Then, it would not be possible to compose arcs from P_1^2 and P_1^3 to produce an arc in which P_1 moves from $\mathbf{T1}$ to $\mathbf{C1}$.

To enforce this constraint in our repair, we define a boolean formula over transition variables $E_{s,t}$ whose satisfaction implies it. Consider just two structures M_{ij}, M_{ik}. For M_{ij} and local states s_i, t_i of P_i^j, let $(s_{ij}^1, i, t_{ij}^1), \ldots, (s_{ij}^n, i, t_{ij}^n)$ be all the transitions in M_{ij} that (1) are executed by P_i^j, (2) start with P_i^j in s_i, and (3) end with P_i^j in t_i. Likewise, for pair-structure M_{ik} and the same local states s_i, t_i, let $(s_{ik}^1, i, t_{ik}^1), \ldots, (s_{ik}^m, i, t_{ik}^m)$ be all the transitions in M_{ik} that (1) are executed by P_i^k, (2) start with P_i^k in s_i, and (3) end with P_i^k in t_i. Then, define $grCon(i, j, k, s_i, t_i,) =$

$$E[s_{ij}^1, t_{ij}^1] \vee \cdots \vee E[s_{ij}^n, t_{ij}^n] \equiv E[s_{ik}^1, t_{ik}^1] \vee \cdots \vee E[s_{ik}^m, t_{ik}^m]$$

For clarity of sub/superscripts, we use $E[s,t]$ rather than $E_{s,t}$ here. Now define $grCon$ to be the conjunction of the $grCon(i, j, k, s_i, t_i,)$, taken over all $i \in \{1, \ldots, K\}$, $j, k \in I(i), j \neq k$, and all pairs s_i, t_i of local states of P_i. Then, our overall repair formula is

$$grCon \wedge (\bigwedge_{(i,j) \in I} repair(M_{ij}, \eta_{ij}))$$

Eshmun generates this repair formula from the pair-structures M_{ij} and their respective specifications η_{ij}. In particular, it computes $grCon$ automatically and conjoins it into the repair formula. A satisfying assignment of this formula gives a repair

for each M_{ij} w.r.t. η_{ij} and also ensures that the repaired structures satisfy the process graph consistency constraint. A concurrent program $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ can then be extracted as outlined above. By the large model theorems of [2]–[4], P satisfies $\bigwedge_{(i,j)\in I} \eta_{ij}$. Let $n_i = |I(i)|$ i.e., n_i is the number of pairs that P_i is involved in. The length of $grCon$ is then linear in $\Sigma_{i\in\{1,\ldots,K\}} n_i^2$, since we take overlapping pair-structures (those with a common process P_i) two at a time ($j, k \in I(i), j \neq k$), and we repeat this for every P_i. It is also linear in the size of the pair-structures. By Prop. 1, the length of each $repair(M_{ij}, \eta_{ij})$ is quadratic in the size of M_{ij} and linear in the length of η_{ij}. So overall the length of the repair formula is quadratic in the n_i, quadratic in the size of the M_{ij}, and linear in the length of the η_{ij}.

Proposition 2: $grCon \wedge (\bigwedge_{(i,j)\in I} repair(M_{ij}, \eta_{ij}))$ has length in $O(|I| \times |M|^2 \times |\eta| + |M|^2 \times \Sigma_{i\in\{1,\ldots,K\}} n_i^2)$ where $|I|$ is the number of pairs, $|M|$ is the size (states + transitions) of the largest pair-structure M_{ij}, and $|\eta|$ is the length of the largest pair-specification η_{ij}.

Hence, provided that the SAT solver remains efficient, we can repair a concurrent program $P = (St_P, P_1 \parallel \cdots \parallel P_K)$ without incurring state explosion—complexity exponential in K.

A. Example: eventually serializable data service

The eventually-serializable data service (ESDS) of Fekete et. al. [10] and Ladin et. al. [11] is a replicated, distributed data service that trades off immediate consistency for improved efficiency. A shared data object is replicated, and the response to an operation at a particular replica may be out of date, i.e., not reflecting the effects of other operations which have not yet been received by that replica. Operations may be reordered *after* the response is issued. Replicas communicate to each other the operations they receive, so that eventually every operation "stabilizes," i.e., its ordering is fixed w.r.t. all other operations. Clients may require an operation to be *strict*, i.e., stable at the time of response (and so it cannot be reordered after the response is issued). Clients may also specify, in an operation x, a set $x.prev$ of operations that must precede x (client-specified constraints, CSC). We let $\mathcal{O}$ be the (countable) set of all operations, $\mathcal{R}$ the set of all replicas, $client(x)$ be the client issuing operation x, $replica(x)$ be the replica that handles operation x. We use x to index over operations, c to index over clients, and r, r' to index over replicas. For each operation x, we define a client process C_c^x and a replica process R_r^x, where $c = client(x)$, $r = replica(x)$. Thus, a client consists of many processes, one for each operation that it issues. As the client issues operations, these processes are created dynamically. Likewise a replica consists of many processes, one for each operation that it handles. Thus, we can use dynamic process creation and finite-state processes to model an infinite-state system, such as the one here, which in general handles an unbounded number of operations with time [4].

We use Eshmun to repair a simple instance of ESDS with one strict operation x, one client $\mathbf{Cx}$ that issues x, a replica $\mathbf{R1x}$ that processes x, another replica $\mathbf{R2x}$ that receives gossip of x, and another replica $\mathbf{R2y}$ that processes an operation $y \in x.prev$. There are three pairs, $\mathbf{Cx} \parallel \mathbf{R1x}$, $\mathbf{R1x} \parallel \mathbf{R2x}$, and $\mathbf{R1x} \parallel \mathbf{R2y}$. $\mathbf{Cx}$ moves through three local states in sequence: initial state $\mathbf{IN\_Cx}$, then state $\mathbf{WT\_Cx}$ after $\mathbf{Cx}$ submits x, and then state $\mathbf{DN\_Cx}$ after $\mathbf{Cx}$ receives the result of x from $\mathbf{R1x}$. $\mathbf{R1x}$ moves through five local states: initial state $\mathbf{IN\_R1x}$, then state $\mathbf{WT\_R1x}$ after it receives x from $\mathbf{Cx}$, then state $\mathbf{DN\_R1x}$ after it performs x, then state $\mathbf{ST\_R1x}$ when it stabilizes x, and finally state $\mathbf{SNT\_R1x}$ when it sends the result of x to $\mathbf{Cx}$. $\mathbf{R2x}$ moves through four local states: initial state $\mathbf{IN\_R2x}$, then state $\mathbf{WT\_R2x}$ after it receives x from $\mathbf{R1x}$, then state $\mathbf{DN\_R2x}$ after it performs x, and finally state $\mathbf{ST\_R2x}$ when it stabilizes x. $\mathbf{R2y}$ moves through four local states: initial state $\mathbf{IN\_R2y}$, then state $\mathbf{WT\_R2y}$ after it receives y from its client (which is not shown), then state $\mathbf{DN\_R2y}$ after it performs y, then state $\mathbf{ST\_R2y}$ when it stabilizes y. For each pair, we start with a "naive" pair-structure in which all possible transitions are present, modulo the above sequences. We then repair w.r.t. these pair-specifications:

Client-replica interaction, pair-specification for $\mathbf{Cx} \parallel \mathbf{R1x}$
where $x \in \mathcal{O}$, $\mathbf{Cx} = client(x)$, $\mathbf{R1x} = replica(x)$

- $\mathsf{AG}(\mathbf{WT\_R1x} \Rightarrow \mathbf{WT\_Cx})$: x is not received by $\mathbf{R1x}$ before it is submitted by $\mathbf{Cx}$
- $\mathsf{A}[\mathbf{WT\_R1x}\,\mathsf{V}\,(\neg(\mathbf{WT\_Cx} \wedge \mathsf{EG}(\neg\mathbf{WT\_R1x})))]$ if x is submitted by $\mathbf{Cx}$ then it is received by $\mathbf{R1x}$
- $\mathsf{AG}(\mathbf{WT\_Cx} \Rightarrow \mathsf{AF}\,\mathbf{DN\_Cx})$: if $\mathbf{Cx}$ submits x then it eventually receives a result
- $\mathsf{AG}(\mathbf{DN\_Cx} \Rightarrow \mathbf{SNT\_R1x})$: $\mathbf{Cx}$ does not receive a result before it is sent by $\mathbf{R1x}$

The repaired pair-structure is given in Fig. 6.

Pair-specification for $\mathbf{R1x} \parallel \mathbf{R2x}$, where $x \in \mathcal{O}$, $x.strict$, $\mathbf{R1x} = replica(x)$

- $\mathsf{AG}(\mathbf{SNT\_R1x} \Rightarrow \mathbf{ST\_R2x})$: the result of a strict operation is not sent to the client until it is stable at all replicas

The repaired pair-structure is given in Fig. 8.

CSC constraints, pair-specification for $\mathbf{R1x} \parallel \mathbf{R2y}$, where $x \in \mathcal{O}$, $y \in x.prev$, $\mathbf{R1x} = replica(x)$, $\mathbf{R2y} = replica(y)$

- $\mathsf{AG}(\mathbf{DN\_R1x} \Rightarrow \mathbf{DN\_R2y})$: operation y in $x.prev$ is performed before x is

The repaired pair-structure is given in Fig. 9.

Fig. 7 gives a concurrent program P that is extracted from the repaired pair-structures as discussed above. By the large model theorem [2]–[4], P satisfies all the above pair-specifications.

VII. Repair of Hierarchical Kripke Structures

We outline briefly how to repair hierarchical Kripke structures [1]. Details are in the full paper. Roughly a hierarchical Kripke structure contains "boxes" , which have a single entry state and several exit states. A box can be instantiated in several places in a Kripke structure, and a box can itself contain boxes, etc.

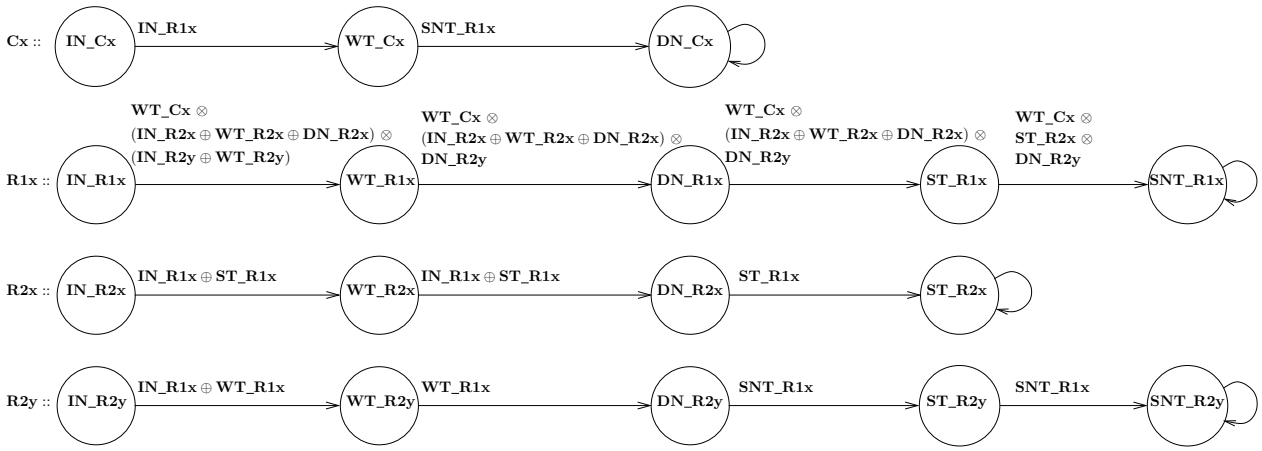


Fig. 7: ESDS Program

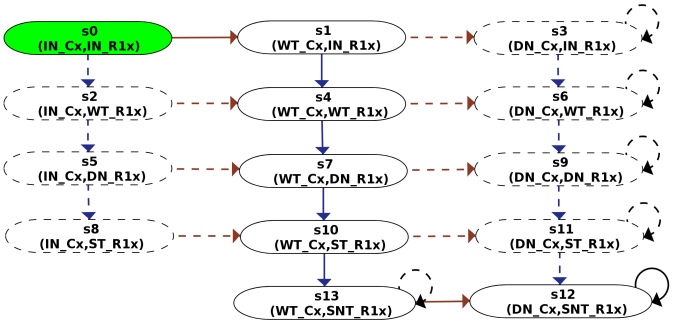


Fig. 6: Repaired pair-structure for $\mathbf{Cx} \| \mathbf{R1x}$

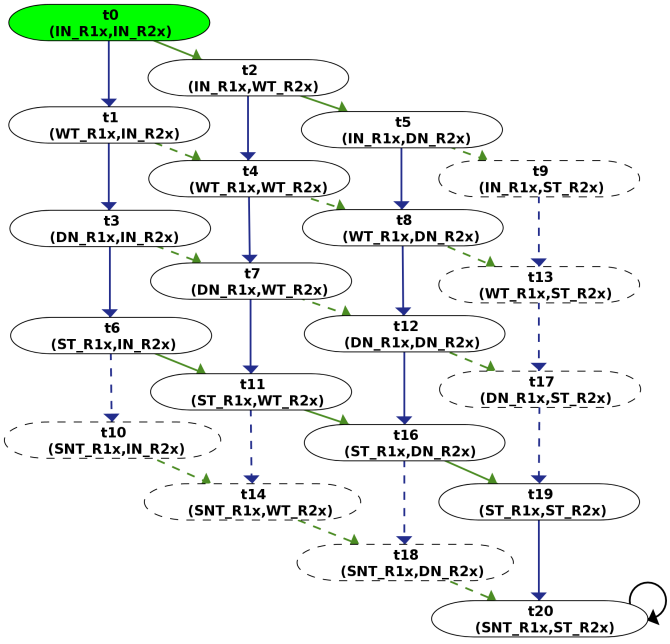


Fig. 8: Repaired pair-structure for $\mathbf{R1x} \| \mathbf{R2x}$

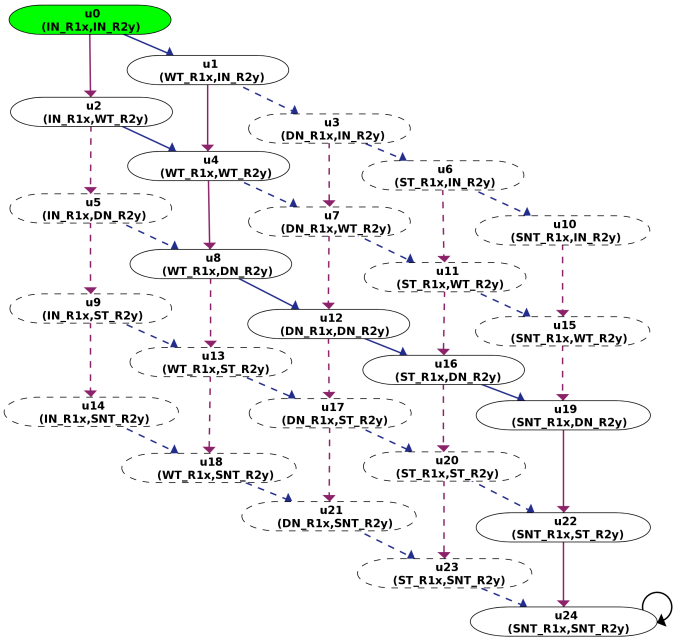


Fig. 9: Repaired pair-structure for $\mathbf{R1x} \| \mathbf{R2y}$

For simplicity of discussion, let M be a hierarchical Kripke structure containing a single instance of box B. To repair M w.r.t. η, we proceed as follows. **(1)** Write a specification formula η_B for B and repair B_M w.r.t. η_B. B_M is a modification of B which reflects the effect on B of being placed inside M, e.g., a loop in M may cause B to be invoked infinitely often. **(2)** Write a "coupling" formula φ which relates the behavior of B to that of M. Repair M_A w.r.t. φ, where M_A is the result of replacing B by B_A in M, where B_A is an abstraction of B. Typically, φ relates output states of B to subsequent reachable states of M. **(3)** Prove $\models \eta_B \wedge \varphi \Rightarrow \eta$, i.e., that $\eta_B \wedge \varphi \Rightarrow \eta$ is a CTL validity. We can then deduce $M \models \eta$. The full paper gives an example repair for a faulty version of the phone call example of [1].

VIII. The Eshmun Tool

Eshmun is an interactive GUI-based tool written in Java, it uses the `javax.swing` library for GUI functionality, and

SAT4jSolver [12] to check satisfiability. Eshmun allows users to create a Kripke structure M by adding states and transitions. Eshmun displays these structures nicely, by providing a wide range of functionalities, including auto-formatting, manual-formatting (through drag-and-drop), zoom, and search. Users enter a CTL formula η and proceed to model check and repair M w.r.t. η. Users can mark a state or transition as non-deletable by checking the **retain** option in the appropriate menus. Eshmun supports concurrent Kripke structures; each pair-structure is entered with its specification in a separate tab. Eshmun automatically computes $grCon$ (see Sect. VI) and conjoins it into the repair formula. Furthermore, users can traverse different repairs through previous and next repair buttons, each repair corresponding to a different satisfying assignment returned by the SAT solver. The panels responsible for input logic have IDE-like functionalities, like syntax error detection, and coloring.

Eshmun implements abstraction, and will show the abstract structure, the repair on the abstract structure, and the concretization of this repair back to the original structure. The CTL decision procedure [5] has been implemented, for checking validity of CTL formulae (useful in hierarchical repair). Finally, Eshmun provides direct access to the X_s and $E_{s,t}$ variables, so that a user can write boolean "structure" formulae, which, when conjoined to the repair formula, impose additional restrictions. Suppose for example that a set T of transitions are related, and we wish to restrict repair so that these are all deleted or all retained. This is achieved by conjoining $\bigwedge_{(s,t),(s',t')\in T}(E_{s,t} \equiv E_{s',t'})$ to $repair(M,\eta)$. Users can add such structural formula through the corresponding panel. Details about the use of Eshmun and its capabilities can be found in the user manual in the help section inside Eshmun.

The point of Eshmun is not to handle large state spaces, but to provide an interactive environment, with *semantic feedback* for the design and construction of small/medium sized structures. These are to be composed concurrently and/or hierarchically to yield a complex structure which would be exponentially large if "flattened out".

Table I gives experimental results for repairing mutual exclusion (w.r.t. safety) and also barrier synchronization. The structures used were generated by a Python program. N gives the number of processes in mutual exclusion and the number of barriers in barrier synchronization. Table II gives experimental results for different concurrent programs expressed as concurrent Kripke structures. The experiments were conducted on a linux computer using openJDK 7 with 4GB of ram and 3.3GHz CPU. These agree closely with Prop. 2. For mutex quadratic growth is expected since the number of pairs is $N(N-1)/2$, and the number of pairs that any process P_i is involved in is $N-1$, and so we expect quadratic growth with N. For dining philosophers (in a ring), the number of pairs is N and the number of pairs that P_i is involved in is 2, and so we expect linear growth with N.

IX. Related Work

The use of transition deletion to repair Kripke structures was suggested by Attie & Emerson [13] in the context of atomicity refinement: a large grain concurrent program is refined naively (e.g., by replacing a test and set by the test,

Program Name	N	Basic repair	Abst. by label	Abst. by subformulae
Mutex	2	0.083	0.026	0.036
Mutex	3	0.51	0.25	0.038
Mutex	4	2.34	0.84	0.17
Mutex	5	89.08	2.48	1.59
Barrier	2	0.31	0.11	0.017
Barrier	3	0.67	0.21	0.047
Barrier	4	1.03	0.76	0.11
Barrier	5	1.81	1.04	0.26

TABLE I: Times (in seconds) taken to repair the given programs

Processes	5	10	15	20	25	50
Mutex	0.33	0.64	0.84	1.35	1.86	7.25
Dining Phil.	0.48	0.59	0.67	0.74	0.81	1.20

TABLE II: Times (in seconds) to repair given concurrent programs

followed nonatomically by the set). In general, this introduces new computations ("bad interleavings") that violate the specification. These are removed by deleting some transitions. The model repair problem for CTL was formally stated by Buccafurri et. al. [14]. Their method uses abductive reasoning and generates repair suggestions that are then verified by model checking, one at a time. In contrast, we fix all faults at once. Generating counterexamples from model checking was suggested by Clarke et. al. [15] and Hojati et. al. [16]. Clarke et. al. [15] presents an algorithm for generating counterexamples for symbolic model checking. Hojati et. al. [16] presents BDD-based algorithms for generating counterexamples ("error traces") for both language containment and fair CTL model checking. Game-based model checking [17], [18] provides a method for extracting counterexamples from a model checking run. This works by coloring the nodes in the model-checking game graph which contribute to violation of the formula.

The idea of generating a propositional formula from a model checking problem was presented in Biere et. al. [19], which considers LTL formulae and bounded model checking: given a formula f, a boolean formula is generated that is satisfiable iff f can be verified within a fixed number k of

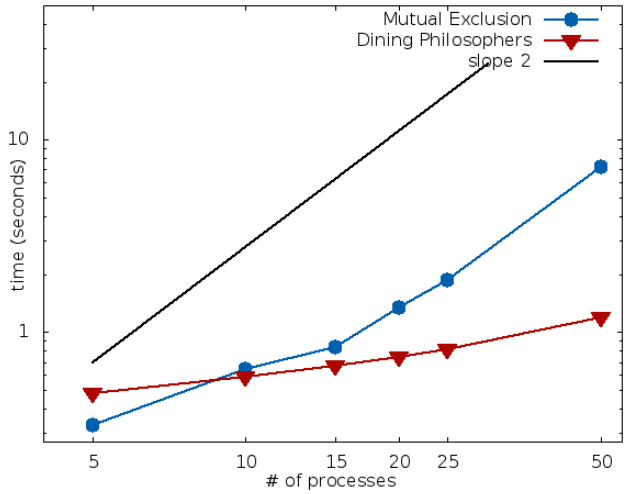


Fig. 10: Repair time for given programs

transitions along some path ($\mathsf{E}f$). By setting f to the negation of the required property, counterexamples can be generated. Repair is not discussed.

Methods for repairing a program (or circuit) w.r.t. a specification are given in [20]–[22]. These papers do not present an automatic repair method that is (1) complete (i.e., if a repair exists, then find a repair) for a full temporal logic (e.g., CTL, LTL), and (2) repairs all faults in a single run, i.e., deals implicitly with all counterexamples "at once", and (3) is reasonably efficient. Jobstmann et. al. [20] considers only one repair at a time, and their method is complete only for invariants. In Staber et. al. [22], the approach of [20] is extended so that multiple faults are considered at once, but at the price of complexity exponential in the number of faults.

Zhang and Ding [23] present a repair method based on five primitive update operations: add a transition, remove a transition, change the propositional labeling of a state, add a state, and remove an isolated state (no incident transitions). They present a "minimum change principle": the repaired model results from the original by the minimum number of primitive update operations. Their repair algorithm runs in time exponential in $|\eta|$ and quadratic in $|M|$, and appears to be highly nondeterministic, with many steps involving a nondeterministic choice of action, e.g., "do one of (a), (b), and (c)". They do not discuss how this nondeterminism is resolved. They also claim that the method has been implemented, but no link to a downloadable implementation is given. Chatzieleftheriou et. al. [24] presents an approach to repairing abstract structures, using Kripke modal transition systems and 3-valued semantics for CTL. They also aim to minimize the number of changes made to effect a repair. They use a set of basic repair operations: add/remove a may/must transition, change the propositional label of a state, and add/remove a state. Their repair algorithm is recursive CTL-syntax-directed.

In contrast to the above two approaches, we do not minimize the number of changes, but rather provide the ability to customize the repair by using the **retain** button, and also by writing a boolean structure formula to constrain the repair. Also, Eshmun facilitates the interactive design of Kripke structures, using the semantic feedback from failed and successful repair attempts.

X. Conclusions

We presented a method for repairing a Kripke structure M w.r.t. a CTL formula η by deleting transitions and states, and implemented it as an interactive graphical tool, Eshmun. This allows the gradual design and construction of Kripke structures, assisted by immediate semantic feedback. Eshmun allows further customization of the repair by preventing deletion of particular transitions, and by conjoining a "structural formula" to the repair formula. Our method can handle concurrent and hierarchical Kripke structures, which can be exponentially more succinct than the equivalent flat structure.

References

[1] R. Alur and M. Yannakakis, "Model checking of hierarchical state machines," *ACM TOPLAS*, vol. 23, no. 3, pp. 273–303, 2001.

[2] P. C. Attie and E. A. Emerson, "Synthesis of concurrent systems with many similar processes," *ACM TOPLAS*, vol. 20, no. 1, pp. 51–115, Jan. 1998.

[3] P. C. Attie, "Synthesis of large concurrent programs via pairwise composition," in *CONCUR*, 1999, Springer LNCS no. 1664.

[4] ——, "Synthesis of large dynamic concurrent programs from dynamic specifications," American University of Beirut, Beirut, Lebanon, Tech. Rep., 2015, to appear in *Formal Methods in System Design*.

[5] E. A. Emerson and E. M. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming*, vol. 2, no. 3, pp. 241–266, 1982.

[6] E. M. Clarke, E. A. Emerson, and P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM TOPLAS*, 1986.

[7] E. M. Clarke, O. Grumberg, and D. E. Long, "Model checking and abstraction," *ACM TOPLAS*, vol. 16, no. 5, pp. 1512–1542, Sept. 1994.

[8] S. Graf and H. Saidi, "Construction of abstract state graphs with pvs," in *CAV*, ser. LNCS. Springer, 1997, vol. 1254, pp. 72–83.

[9] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, N.J.: Prentice-Hall Inc., 1976.

[10] A. Fekete, D. Gupta, V. Luchango, N. Lynch, and A. Shvartsman, "Eventually-serializable data services," *Theoretical Computer Science*, vol. 220, pp. 113–156, 1999.

[11] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[12] D. L. Berre, A. Parrain *et al.*, "The sat4j library, release 2.2, system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, pp. 59–64, 2010.

[13] P. Attie and E. Emerson, "Synthesis of concurrent programs for an atomic read/write model of computation," *ACM TOPLAS*, vol. 23, no. 2, pp. 187–242, 2001.

[14] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone, "Enhancing model checking in verification by AI techniques," *Artif. Intell.*, 1999.

[15] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao, "Efficient generation of counterexamples and witnesses in symbolic model checking," in *Design Automation Conference*. ACM Press, 1995.

[16] R. Hojati, R. K. Brayton, and R. P. Kurshan, "Bdd-based debugging of design using language containment and fair CTL," in *CAV*, 1993, Springer LNCS no. 697.

[17] C. Stirling and D. Walker, "Local model checking in the modal mu-calculus," *Theor. Comput. Sci.*, vol. 89, no. 1, 1991.

[18] S. Shoham and O. Grumberg, "A game-based framework for ctl counterexamples and 3-valued abstraction-refinement," in *CAV*, 2003, pp. 275–287.

[19] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *TACAS, LNCS no. 1579*, 1999.

[20] B. Jobstmann, A. Griesmayer, and R. Bloem, "Program repair as a game," in *CAV*, 2005, pp. 226–238.

[21] S. Staber, B. Jobstmann, and R. Bloem, "Finding and fixing faults," in *CHARME '05*, 2005, springer LNCS no. 3725.

[22] ——, "Diagnosis is repair," in *Intl. Workshop on Principles of Diagnosis*, June 2005.

[23] Y. Zhang and Y. Ding, "CTL model update for system modifications," *J. Artif. Int. Res.*, vol. 31, no. 1, pp. 113–155, Jan. 2008.

[24] G. Chatzieleftheriou, B. Bonakdarpour, S. Smolka, and P. Katsaros, "Abstract model repair," in *NASA Formal Methods*, 2012, Springer LNCS no. 7226.