

AMERICAN UNIVERSITY OF BEIRUT

MODEL REPAIR VIA SAT SOLVING

by

Mouhammad Issam Sakr

A thesis

submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon
September 2014

AMERICAN UNIVERSITY OF BEIRUT

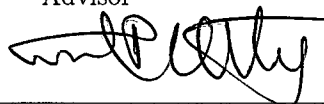
MODEL REPAIR VIA SAT SOLVING

by
Mouhammad Issam Sakr

Approved by:

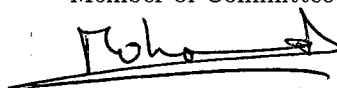
Dr. Paul Attie, Associate Professor
Computer Science

Advisor



Dr. Mohamad Jaber, Assistant Professor
Computer Science

Member of Committee



Dr. Fadi Zaraket, Assistant Professor
Electrical and Computer Engineering

Member of Committee



Date of thesis defense: September 16, 2014

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: SAKR MOUHAMMAD ISSAM
Last First Middle

Masters Thesis

Masters Project

Doctoral Dissertation

I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

I authorize the American University of Beirut, **three years after the date of submitting my thesis**, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.



01/10/2014

Signature

Date

An Abstract of the Thesis of

Mouhammad Issam Sakr for Masters of Science
Major: Computer Science

Title: Model Repair via SAT Solving

We consider the following model repair problem: given a finite Kripke structure M and a specification formula η in some modal or temporal logic, determine if M contains a sub-structure M' that satisfies η . That is, can M be repaired to satisfy the specification η by deleting some transitions? We map an instance (M, η) of model repair to a boolean formula $repair(M, \eta)$ such that (M, η) has a solution iff $repair(M, \eta)$ is satisfiable. Furthermore, a satisfying assignment determines which transitions must be removed from M to generate a model M' of η . Thus, we can use any SAT solver to repair Kripke structures. Furthermore, using a complete SAT solver yields a complete algorithm: it always finds a repair if one exists. We augment the basic method by adding *state-space reduction* methods, and also a method to repair *hierarchical* Kripke structures.

Contents

1	Introduction and Motivation	7
2	Preliminaries	10
3	The model repair problem	12
3.1	Complexity of the model repair problem	12
4	CTL model repair using SAT solvers	14
5	Repair with state-space Reductions	18
5.1	Reductions w.r.t. atomic propositions	19
5.2	Reduction w.r.t. sub-formulae	20
5.3	Inverting the reduction to repair the original structure	22
6	Repair of hierarchical Kripke structures	24
6.1	Hierarchical model checking	24
6.2	CTL decision procedure	24
7	Implementation	26
7.1	User interface and user manual	27
7.1.1	State creation	27
7.1.2	Transition creation	28
7.1.3	Kripke structures visualization	28
7.1.4	Kripke structure saving	30
7.1.5	Model checking, model repairing and state-space reductions	30
7.1.6	Hierarchical model checking	31

7.2	Software architecture	32
7.2.1	Main modules	33
7.2.2	CTL parser	34
7.2.3	User interface	38
7.2.4	Model repairer	39
7.2.5	Model optimizer	45
7.2.6	SAT solver	46
7.2.7	Decision procedure	48
8	Case studies	49
8.1	Mutual exclusion	49
8.2	Barrier synchronization	52
8.3	Phone call system	56
9	Conclusions and future work	59
10	Bibliography	60

List of Figures

4.1	The model repair algorithm.	17
5.1	Reduction by atomic propositions	20
5.2	Reduction by sub-formulae	22
7.1	Tool's main screen	26
7.2	States screen for mutex problem	27
7.3	Transitions screen for mutex problem	28
7.4	Kripke structure for mutex problem	29
7.5	State	29
7.6	Transition	29
7.7	Saved file for mutex problem	30
7.8	Model check and repair	31
7.9	Kripke sub-structure screen	32
7.10	CTL decision procedure screen	32
7.11	Tool main modules	34
7.12	User interface module	38
7.13	Model repairer module	39
7.14	The model repair algorithm.	42
7.15	Formula propagation.	43
7.16	Initializing $repair(M, \eta)$	44
7.17	Adding a conjunct to $repair(M, \eta)$	44
7.18	Kripke optimizer module	45
7.19	SAT solver module	47
7.20	Decision procedure module	48

8.1	States screen for mutex problem	50
8.2	Repaired model for mutex problem	51
8.3	Reduction w.r.t. atomic propositions	51
8.4	Repaired model using reduction w.r.t. atomic propositions	52
8.5	Barrier synchronization	53
8.6	Repaired barrier	54
8.7	Reduction w.r.t. sub-formulae	54
8.8	Repaired model using reduction w.r.t. sub-formulae	55
8.9	Phone call system	56
8.10	Phone calls	57
8.11	Hierarchical	58

Listings

5.1	Repair initial model	23
7.1	CTL BNF	36
7.2	Convert to CNF	40
7.3	Repair() method	40
7.4	Repairer utilization	41
7.5	Tarjan's strongly connected components algorithm	46

Chapter 1

Introduction and Motivation

Finite-state machines(FSM) serve as operational specifications for concurrent and distributed systems, e.g., in UML. These specifications must admit desirable behaviours and prohibit undesirable ones. Writing operational specifications that are correct is difficult and complex. We attack the problem of writing good specifications by checking an operational specification with respect to a non-operational one, in particular a temporal logic formula. Temporal logic extends propositional logic with temporal modalities, where the truth of a formula in a state depends on other states. In practice, a temporal logic specification is a long conjunction of small formulae, and hence is easier to write than a single complex finite-state machine. In order to check if a system or model does not violate temporal logic specifications, we use model checking Emerson, Clarke and Sistla [9], Quielle and Sifakis [20]. If a model fails to satisfy a specification, then a model checker typically generates a single counterexample, which is an example behavior that violates the formula being checked, and so facilitates debugging the model. However, there could be many counterexamples, and they may have to be dealt with by making different fixes manually, thus increasing debugging effort. In this thesis we deal with all counterexamples at once, by “repairing” the model: we present a method for automatically fixing Kripke structures with respect to CTL [13] specifications.

Our contribution. We first present a “subtractive” repair algorithm: fix a Kripke structure only by removing transitions and states (roughly speaking, those transitions and states that “cause” violation of the specification). If the initial state is not deleted, then the resulting structure (or program) satisfies the specification. We show that this algorithm is sound and relatively complete. An advantage of subtractive repair is that it does not introduce new behaviors, and thus any missing (i.e., not part of the formula being repaired against) conjuncts of the specification that are expressible in a universal temporal logic (no existential path quantifier) are still satisfied (if they originally were). Hence we can fix w.r.t. incomplete specifications.

We also present reduction strategies that work on minimizing Kripke structures state-space before repairing them, and we extend the repair method to deal with hierarchical

Kripke structures [1].

Formally, we consider the *model repair problem*: given a Kripke structure M and a CTL formula η , does there exist a substructure M' of M (obtained by removing transitions and states from M) such that M' satisfies η ? In this case, we say that M is *repairable* w.r.t. η , or that a repair exists.

Our algorithm computes (in deterministic time polynomial in the size of M times the size of η) a propositional formula $repair(M, \eta)$ such that $repair(M, \eta)$ is satisfiable iff M contains a substructure M' that satisfies η . Furthermore, a satisfying assignment for $repair(M, \eta)$ determines which transitions 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. Our approach leverages the research investment in SAT solvers to attack the model repair problem.

Soundness of our repair algorithm means that the resulting M' (if it exists) satisfies η . Completeness means that if the initial structure M contains a substructure that satisfies η , then our algorithm will find such a substructure, provided that a complete SAT solver is used to check satisfaction of $repair(M, \eta)$.

While our method has a worst case running time exponential in the number of global states, this occurs only if the underlying SAT solver uses exponential time. SAT-solvers have proved to be efficient in practice, as demonstrated by the success of SAT-solver based tools such as Alloy, NuSMV, and Isabelle/HOL. The success of SAT solvers in practice indicates that our method will be applicable to reasonable size models, just as, for example, Alloy [16] is.

Related work. The use of transition deletion to repair Kripke structures was suggested in [4, 5] 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, followed nonatomically by the set). In general, this may introduce new computations (corresponding to “bad interleavings”) that violate the specification. These are removed by deleting some transitions.

The use of model checking to generate counterexamples was suggested by Clarke et. al. [10] and Hojati et. al. [15]. [10] presents an algorithm for generating counterexamples for symbolic model checking. [15] presents BDD-based algorithms for generating counterexamples (“error traces”) for both language containment and fair CTL model checking. Game-based model checking [24, 21] provides a method for extracting counterexamples from a model checking run. The core idea is a *coloring algorithm* that colors nodes in the model-checking game graph that contribute to violation of the formula being checked.

The idea of generating a propositional formula from a model checking problem was presented in [6]. That paper considers LTL specifications and bounded model checking: given an LTL formula f , a propositional formula is generated that is satisfiable iff f can be verified within a fixed number k of transitions along some path (Ef). By setting f to the negation of the required property, counterexamples can be generated. Repair is not discussed.

Some authors [17, 23, 22] have considered algorithms for solving the repair problem: given a program (or circuit), and a specification, how to automatically modify the program (or

circuit), so that the specification is satisfied. There appears to be no 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.” For example, Jobstmann et. al. [17] considers only one repair at a time, and their method is complete only for invariants. In [22], the approach of [17] is extended so that multiple faults are considered at once, but at the price of exponential complexity in the number of faults.

In [7] the repair problem for CTL is considered and solved using adductive reasoning. The method generates repair suggestions that must then be verified by model checking, one at a time. In contrast, we fix all faults at once.

Antoniotti [2] has shown that the related problem of discrete event supervisory control is also NP-complete.

Ding and Zhang [11] proposed five primitive operations that capture the basic update of the CTL model, and then they define the minimal change criteria for CTL model update based on these primitive operations.

Abstract Model Repair was discussed in [8] where they present a framework for model repair that uses abstraction refinement.

The rest of this thesis is as follows. Chapter 2 provides brief technical preliminaries. Chapter 3 states the model repair problem. Chapter 4 presents our model repair method for CTL. Chapter 5 presents several state-space reduction strategies. Chapter 6 extends the repair method to hierarchical Kripke structures. Chapter 7 discusses our implementation, and a user manual. Chapter 8 presents several example applications of the method. Chapter 9 discusses our conclusions and presents future work.

Chapter 2

Preliminaries

Let \mathcal{AP} be a set of atomic propositions including the constants **true** and **false**. We use **true**, **false** as “constant” propositions whose interpretation is always the truth values tt , ff , respectively. The logic CTL [13] is given by the following grammar:

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

where $p \in \mathcal{AP}$.

The semantics of formulae are defined with respect to a Kripke structure.

Definition 1. A Kripke structure is a tuple $M = (S_0, S, R, L)$ where S is a finite state of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is a transition relation, and $L : S \mapsto 2^{\mathcal{AP}}$ is a labeling function that associates each state $s \in S$ with a subset of atomic propositions, namely those that hold in the state.

We assume that a Kripke structure $M = (S_0, S, R, L)$ is total, i.e., $\forall s \in S, \exists s' \in S : (s, s') \in R$. A path in M is a (finite or infinite) sequence of states, $\pi = s_0, s_1, \dots$ such that $\forall i \geq 0 : (s_i, s_{i+1}) \in R$. A fullpath is an infinite path.

Definition 2. $M, s \models \varphi$ means that formula φ is true in state s of structure M and $M, s \not\models \varphi$ means that formula φ is false in state s of structure M . We define \models inductively as usual:

- $M, s \models \text{true}$
- $M, s \not\models \text{false}$

- $M, s \models p$ iff $p \in L(s)$ where atomic proposition $p \in \mathcal{AP}$
- $M, s \models \neg\varphi$ iff $M, s \not\models \varphi$
- $M, s \models \varphi \wedge \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$
- $M, s \models \varphi \vee \psi$ iff $M, s \models \varphi$ or $M, s \models \psi$
- $M, s \models \text{AX}\varphi$ iff for all t such that $(s, t) \in R : (M, t) \models \varphi$
- $M, s \models \text{EX}\varphi$ iff there exists t such that $(s, t) \in R$ and $(M, t) \models \varphi$
- $M, s \models \text{A}[\varphi\text{V}\psi]$ iff for all fullpaths $\pi = s_0, s_1, \dots$ starting from $s = s_0$:
 $\forall k \geq 0 : (\forall j < k : M, s_j \not\models \varphi)$ implies $M, s_k \models \psi$
- $M, s \models \text{E}[\varphi\text{V}\psi]$ iff for some fullpath $\pi = s_0, s_1, \dots$ starting from $s = s_0$:
 $\forall k \geq 0 : (\forall j < k : M, s_j \not\models \varphi)$ implies $M, s_k \models \psi$

We use $M \models \varphi$ to abbreviate $\forall s \in S_0 : M, s \models \varphi$. We introduce the abbreviations $\text{A}[\phi\text{U}\psi]$ for $\neg\text{E}[\neg\varphi\text{V}\neg\psi]$, $\text{E}[\phi\text{U}\psi]$ for $\neg\text{A}[\neg\varphi\text{V}\neg\psi]$, $\text{AF}\varphi$ for $\text{A}[\text{trueU}\varphi]$, $\text{EF}\varphi$ for $\text{E}[\text{trueU}\varphi]$, $\text{AG}\varphi$ for $\text{A}[\text{falseV}\varphi]$, $\text{EG}\varphi$ for $\text{E}[\text{falseV}\varphi]$.

Definition 3 (Formula expansion). Given a CTL formula φ , its set of subformulae $\text{sub}(\varphi)$ is defined as follows:

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

Chapter 3

The model repair problem

Given Kripke structure M and a specification formula φ , we consider the problem of removing parts of M , resulting in a substructure M' such that $M' \models \varphi$.

Definition 4 (Substructure). *Given a Kripke structure $M = (S_0, S, R, L)$ and a structure $M' = (S'_0, S', R', L')$ we say that $M' \subseteq M$ iff $S' \subseteq S$, $S'_0 \subseteq S_0$, $R \subseteq R'$, and $L' = L \upharpoonright S'$.*

Definition 5 (Repairable). *Given Kripke structure $M = (S_0, S, R, L)$ and a formula η . M is repairable with respect to η if there exists a Kripke structure $M' = (S'_0, S', R', L')$ such that M' is total, $M' \subseteq M$, and $M', s_0 \models \eta$.*

Recall that a Kripke structure is total iff every state has at least one outgoing transition.

Definition 6 (Model Repair Problem). *Given a Kripke structure $M = (S_0, S, R, L)$, and a formula η , the repair problem is to decide if M is repairable with respect to η .*

The model repair problem is defined for any temporal or modal logic for which the \models relation is defined, e.g. μ -calculus, CTL*, CTL, etc. So, for example, we speak of the model repair problem for CTL (CTL model repair for short). An instance of model repair is then the pair (M, φ) .

3.1 Complexity of the model repair problem

Theorem 1 ([3]). *The model repair problem for CTL is NP-complete.*

Corollary 1 ([3]). *Let L be any temporal logic interpreted in Kripke structures such that (1) model checking for L is in polynomial time, and (2) there exists a polynomial time reduction from CTL model checking to L model checking. Then the model repair problem for L is NP-complete.*

See Attie and Saklawi [3] for proofs.

Chapter 4

CTL model repair using SAT solvers

Given an instance of model repair (M, η) , where $M = (S_0, S, R, L)$ and η is a CTL formula, we define a propositional formula $repair(M, \eta)$ such that $repair(M, \eta)$ is satisfiable iff (M, η) has a solution. $repair(M, \eta)$ is defined over the following propositions:

1. $E_{s,t} : (s, t) \in R$
2. $X_s : s \in S$
3. $X_{s,\psi} : s \in S, \psi \in sub(\eta)$
4. $X_{s,\psi}^n : s \in S, 0 \leq n \leq |S|$, and $\psi \in sub(\eta)$ has the form $A[\varphi V \varphi']$ or $E[\varphi V \varphi']$

The meaning of $E_{s,t}$ is that the transition (s, t) is retained in the fixed model M' iff $E_{s,t}$ is assigned *tt* (“true”) by the satisfying valuation \mathcal{V} for $repair(M, \eta)$. The meaning of X_s is that state s is retained in the fixed model M' . The meaning of $X_{s,\psi}$ is that ψ holds in state s . $X_{s,\psi}^n$ is used to propagate release formula (AV or EV) for as long as necessary to determine their truth, i.e., $|S|$ in the worst case.

A solution for satisfiability of $repair(M, \eta)$, e.g., as given by a SAT solver, gives directly a solution to model repair. Denote this solution by $model(M, \mathcal{V})$. Then $model(M, \mathcal{V}) = (S'_0, S', R', L')$, where $R' = \{(s, t) | \mathcal{V}(E_{s,t}) = tt\}$, S' consists of all states reachable from S_0 via paths of transitions in R' , and $L' = L \upharpoonright S'$. Note that $model(M, \mathcal{V})$ does not depend directly on η .

Essentially, $repair(M, \eta)$ encodes all of the usual local constraints, e.g., $AX\varphi$ holds in s iff φ holds in all successors of s . We modify these however, to take transition deletion into account. So, the local constraint for AX becomes $AX\varphi$ holds in s iff φ holds in all successors of s *after* transitions have been deleted (to effect the repair). More precisely, instead of $X_{s,AX\varphi} \equiv \bigwedge_{t|s \rightarrow t} X_{t,\varphi}$, we have $X_{s,AX\varphi} \equiv \bigwedge_{t|s \rightarrow t} (E_{s,t} \Rightarrow X_{t,\varphi})$. Here $s \rightarrow t$ abbreviates $(s, t) \in R$, i.e., t is a *successor* of s . The other modalities (EX, AV, EV) are treated similarly. We deal with AU, EU by reducing them to EV, AV using duality. We

require that the repaired structure M' be total by requiring that every state has at least one outgoing transition.

Definition 7 ($\text{repair}(M, \eta)$). Let $M = (S_0, S, R, L)$ be a Kripke structure and η a CTL formula. Let $s \rightarrow t$ abbreviate $(s, t) \in R$. $\text{repair}(M, \eta)$ 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., all undeleted initial states satisfy η

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

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

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

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

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

5. Propositional labeling

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

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

6. Propositional consistency

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

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

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

7. Nexttime formulae

$$\text{for all } \text{AX}\varphi \in \text{sub}(\eta): X_{s, \text{AX}\varphi} \equiv \bigwedge_{t|s \rightarrow t} (E_{s,t} \Rightarrow X_{t, \varphi})$$

$$\text{for all } \text{EX}\varphi \in \text{sub}(\eta): X_{s, \text{EX}\varphi} \equiv \bigvee_{t|s \rightarrow t} (E_{s,t} \wedge X_{t, \varphi})$$

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

for all $A[\varphi V\psi] \in \text{sub}(\eta)$, $m \in \{1 \dots n\}$:

$$X_{s,A[\varphi V\psi]} \equiv X_{s,A[\varphi V\psi]}^n$$

$$X_{s,A[\varphi V\psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigwedge_{t|s \rightarrow t} (E_{s,t} \Rightarrow X_{t,A[\varphi V\psi]}^{m-1}))$$

$$X_{s,A[\varphi V\psi]}^0 \equiv X_{s,\psi}$$

for all $E[\varphi V\psi] \in \text{sub}(\eta)$, $m \in \{1 \dots n\}$:

$$X_{s,E[\varphi V\psi]} \equiv X_{s,E[\varphi V\psi]}^n$$

$$X_{s,E[\varphi V\psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigvee_{t|s \rightarrow t} (E_{s,t} \wedge X_{t,E[\varphi V\psi]}^{m-1}))$$

$$\text{for all } E[\varphi V\psi] \in \text{sub}(\eta): X_{s,E[\varphi V\psi]}^0 \equiv X_{s,\psi}$$

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

Note that the above requires all states, even those rendered unreachable by transition deletion, to have some outgoing transition. This “extra” requirement on the unreachable states does not affect the method however, since there will actually remain a satisfying assignment which allows unreachable state to retain all their outgoing transitions, if some $M' \subseteq M$ exists that satisfies η . For s unreachable from s_0 in M' , assign the value to $X_{s,\varphi}$ that results from model checking $M', s \models \varphi$. This gives a consistent assignment that satisfies $\text{repair}(M, \eta)$. Clearly, $X_{s,\varphi}$ does not affect $X_{s_0,\eta}$ since s is unreachable from s_0 .

In each state $s \in S$, there are $O(|\eta| \times |S|)$ formulae to check, each of which has length $O(d)$, where d is the maximum number of successors that any state in S has. The sum of lengths of all these formulae is $O(|\eta| \times |S|^2 \times d)$. The propositional labelling formulae add $O(|S| \times |\mathcal{AP}|)$ length, and so the size of $\text{repair}(M, \varphi)$ is $O(|\eta| \times |S|^2 \times d + |S| \times |\mathcal{AP}|)$, and so is polynomial in the size of (M, η) . Clearly, $\text{repair}(M, \eta)$ can be constructed in polynomial time. Figure 4.1 presents our model repair algorithm, $\text{Repair}(M, \varphi)$, which we show is sound, and complete provided that a complete SAT-solver is used. Recall that we use $\text{model}(M, \mathcal{V})$ to denote the structure M' derived from the repair of M w.r.t. η , i.e., $M' = (S'_0, S', R', L')$, where $R' = \{(s, t) | \mathcal{V}(E_{s,t}) = tt\}$, S' consists of all states reachable from s_0 via paths of transitions in R' , and $L' = L \upharpoonright S'$.

Theorem 2 (Soundness). *Let $M = (S_0, S, R, L)$ be a Kripke structure, η a CTL formula,*

and $n = |S|$. Suppose that $\text{repair}(M, \eta)$ is satisfiable and that \mathcal{V} is a satisfying truth assignment for it. Let $M' = \text{model}(M, \mathcal{V})$, Then for all reachable states $s \in S'$ and CTL formulae $\xi \in \text{sub}(\eta)$:

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

$$\text{for } m \in \{1 \dots n\} : \mathcal{V}(X_{s,\xi}^m) = \text{tt} \text{ iff } M', s \models \xi.$$

Corollary 2 (Soundness). *If $\text{Repair}(M, \eta)$ returns a structure $M' = (S'_0, S', R', L')$, then (1) M' is total, (2) $M' \subseteq M$, (3) $M', s_0 \models \eta$, and (4) M is repairable.*

Theorem 3 (Completeness). *If M is repairable with respect to η then $\text{Repair}(M, \eta)$ returns a Kripke structure M'' such that M'' is total, $M'' \subseteq M$, and $M'', s_0 \models \eta$.*

Since M' results by removing transitions and unreachable states from M , the relation mapping each state in M' to “itself” in M is a simulation relation [14] from M' to M . Hence the following, where ACTL* [14] is the universal fragment (no existential path quantifier) of CTL*, and clause (2) follows from [14].

Proposition 1. *If $\text{Repair}(M, \eta)$ returns a structure M' , then (1) there is a simulation relation from M' to M , and (2) for all ACTL* formulae f , $M \models f$ implies $M' \models f$.*

```

Repair( $M, \eta$ ):
model check  $M, s_0 \models \eta$ ;
if successful, then return  $M$ 
else
  compute  $\text{repair}(M, \eta)$  as given in Section 3;
  submit  $\text{repair}(M, \eta)$  to a sound and complete SAT-solver;
  if the SAT-solver returns “not satisfiable” then
    return “failure”
  else
    the solver returns a satisfying assignment  $\mathcal{V}$ ;
    return  $M' = \text{model}(M, \mathcal{V})$ 

```

Figure 4.1: The model repair algorithm.

Chapter 5

Repair with state-space Reductions

Our repair algorithm faces a difficulty when repairing large Kripke structures because the generated repair formula will be huge since its length is quadratic in $|M|$. Another difficulty is that after repairing a model, the generated repaired model could lose important features and could delete important transitions that are essential in a correct model, although the best case scenario would be to delete only transitions that are the reason behind the failed model check. We will use the mutex problem as a running example in this chapter in order to clarify the reduction scenarios.

In this section we introduce four types of State-Space reduction:

1. Reduction with respect to atomic propositions in the specification formula while taking adjacency of states into consideration
2. Reduction with respect to atomic propositions in the specification formula without taking states adjacency into consideration
3. Reduction with respect to user predefined boolean expressions that are sub-formulae of the specification formulae while taking adjacency of states into consideration
4. Reduction with respect to user predefined boolean expressions that are sub-formulae of the specification formulae without taking state adjacency into consideration

We define these reductions as equivalence relations over the states of M . The reduced structure is obtained as the quotient of M by the equivalence relations. Since the resulting repair can always be model-checked at no significant algorithmic expense, we use reductions that are not necessarily correctness-preserving, since we can, in many cases, obtain larger reductions in the state-space than if we used only correctness-preserving reductions.

For any equivalence relation \equiv , we define the equivalence class of a state s as $[s] \triangleq \{t \mid s \equiv t\}$.

5.1 Reductions w.r.t. atomic propositions

In order to reduce our model with respect to atomic propositions we start by defining two equivalence relations, $\equiv_{p,a}$ and \equiv_p . Let \mathcal{AP}_η be the set of atomic propositions that occur in η .

The first equivalence relation represents a reduction strategy which takes adjacency of states into consideration.

Definition 8 (Adjacency-respecting propositional reduction, $\equiv_{p,a}$). *Given a Kripke structure $M = (S_0, S, R, L)$ and a CTL formula η , we define an equivalence relation $\equiv_{p,a}$ over S as follows:*

- $s \approx_{p,a} t \triangleq (L(s) \cap \mathcal{AP}_\eta = L(t) \cap \mathcal{AP}_\eta) \wedge ((s, t) \in R \vee (t, s) \in R)$
- $s \equiv_{p,a} t \triangleq \approx_{p,a}^*$

that is, $s \approx_{p,a} t$ iff s and t agree on all of the atomic propositions of η , and there is a transition from s to t or vice-versa, and $\equiv_{p,a}$ is the transitive closure of $\approx_{p,a}$

The next equivalence relation, \equiv_p , ignores adjacency of states. This can result in the removal of existing cycles, or the introduction of new ones.

Definition 9 (Adjacency-ignoring propositional reduction, \equiv_p). *Given a Kripke structure $M = (S_0, S, R, L)$ and a specification formula η , we define an equivalence relation \equiv_p as follows:*

- $s \equiv_p t \triangleq L(s) \cap \mathcal{AP}_\eta = L(t) \cap \mathcal{AP}_\eta$

that is, $s \equiv_p t$ iff s and t agree on all of the atomic propositions of η .

Both $\equiv_{p,a}$ and \equiv_p are equivalence relations over S , and they also preserve the values of all the atomic propositions in η . Hence, we can define a quotient of M by these relations as usual.

Definition 10 (Reduced Model). *Let $\equiv \in \{\equiv_{p,a}, \equiv_p\}$. Given a Kripke structure $M = (S_0, S, R, L)$ and a specification formula η , we define the reduced model $\overline{M} = M / \equiv$ as follows:*

1. $\bar{S} = \{[s] \mid s \in S\}$
2. $\bar{S}_0 = \{[s_0] \mid s_0 \in S_0\}$
3. $\bar{R} = \{([s], [t]) \mid (s, t) \in R\}$
4. $\bar{L} : \bar{S} \rightarrow AP$ is given by $\bar{L}([s]) = L(s) \cap \mathcal{AP}_\eta$

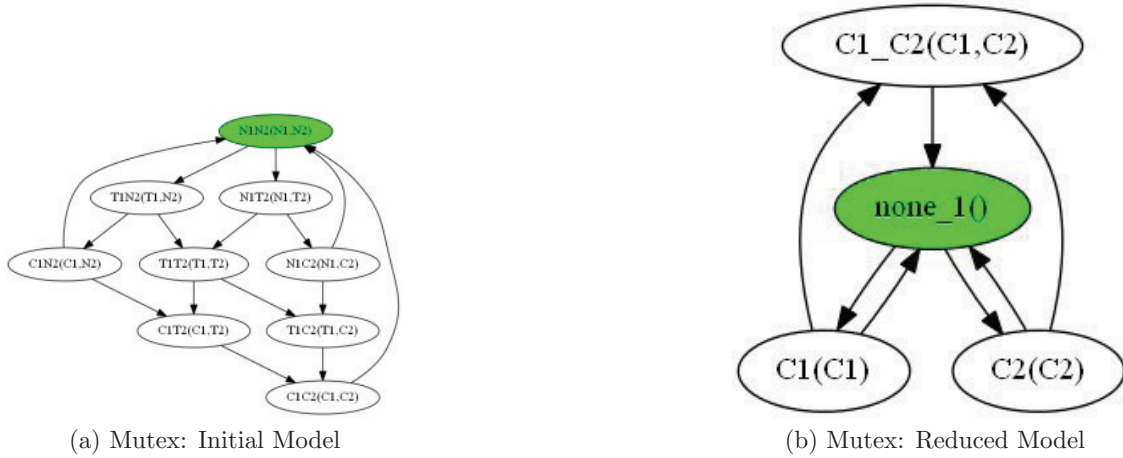


Figure 5.1: Reduction by atomic propositions

Consider the Mutex problem as an example in Figure 5.1. Text inside states represents the name of the state and its labels between brackets. Figure 5.1a depicts mutex initial model, and Figure 5.1b depicts mutex reduced model with respect to atomic propositions. As per Definition 9 the states N1N2, N1T2, T1N2, and T1T2 in Figure 5.1a are equivalent to state none.1 in 5.1b. The same applies to the other states where states C1N2 and C1T2 are equivalent to C1 state, states N1C2 and T1C2 are equivalent to state C2, and state C1C2 is equivalent to state C1_C2. Note that the number of states was reduced from nine states in the initial model to four states in the reduced model. Note that cycles are preserved in this reduction in one direction.

5.2 Reduction w.r.t. sub-formulae

We can obtain larger reductions by dropping the requirement that we preserve the values of all the atomic propositions in η . In some cases, it may be necessary only to preserve the values of some propositional subformulae in η . For example, in mutual exclusion, only the value of $C_1 \wedge C_2$ is of interest.

Let AS_η be a set of propositional sub-formulae of η that is specified by the user. Let $SUB(t) \subseteq AS_\eta$ be the set of sub-formulae in AS_η that are satisfied by the state t .

- $SUB(t) \triangleq f \mid f \in AS_\eta \text{ and } M, t \models f$

Definition 11 (Adjacency-respecting subformula reduction, $\equiv_{s,a}$). *Given a Kripke structure $M = (S_0, S, R, L)$ and a CTL formula η , we define an equivalence relation $\equiv_{s,a}$ over S as follows:*

- $s \approx_{s,a} t \triangleq (SUB(s) \cap AS_\eta = SUB(t) \cap AS_\eta) \wedge ((s, t) \in R \vee (t, s) \in R)$
- $\equiv_{s,a} \triangleq \approx_{s,a}^*$

that is, $s \equiv_{s,a} t$ iff s and t agree on all of the sub-formulae, and there is a transition from s to t or vice-versa and $\equiv_{s,a}$ is the transitive closure of $\approx_{s,a}$. We also define $[s] \triangleq \{t \mid s \equiv t\}$.

Definition 12 (Adjacency-ignoring subformula reduction, $\equiv_{s,p}$). *Given a Kripke structure $M = (S_0, S, R, L)$ and a CTL formula η , we define an equivalence relation $\equiv_{s,p}$ over S as follows:*

- $s \equiv_{s,p} t \triangleq SUB(s) \cap AS_\eta = SUB(t) \cap AS_\eta$

that is, $s \equiv_{s,p} t$ iff s and t agree on all of the sub-formulae in AS_η .

Definition 13 (Reduced model). *Let $\equiv \in \{\equiv_{s,a}, \equiv_{s,p}\}$. Given a Kripke structure $M = (S_0, S, R, L)$ and a specification formula η , we define the reduced model $\bar{M} = M / \equiv$ as follows:*

1. $\bar{S} = \{[s] \mid s \in S\}$
2. $\bar{S}_0 = \{[s_0] \mid s_0 \in S_0\}$
3. $\bar{R} = \{([s], [t]) \mid (s, t) \in R\}$
4. $\bar{L} : \bar{S} \rightarrow AP$ is given by $\bar{L}([s]) = \bigcap_{t \in [s]} L(t)$. That is, the label consists of the atomic propositions, if any, that hold in all states of $[s]$.



Figure 5.2: Reduction by sub-formulae

Consider Mutex problem as an example in Figure 5.2. Figure 5.2a depicts mutex initial model, and Figure 5.2b depicts mutex reduced model with respect to sub-formulae $(C1 \wedge C2)$. As per Definition 12 the states $N1N2$, $N1T2$, $T1N2$, $T1T2$, $C1T2$, and $T1C2$ in Figure 5.2a are equivalent to state $none_1$ in 5.2b. Note that the number of state was reduced from nine states in the initial model to 2 states in the reduced model. Note that cycles are preserved in this reduction which explains the self transition on $none_1$.

5.3 Inverting the reduction to repair the original structure

The objective is to repair the original structure M . Given a repair to a reduced structure \overline{M} , we construct a repair of M as in Listing 5.1

Listing 5.1: Repair initial model

```

1 RepairInitialModel( KripkeStructure initialModel)
2 {
3     //the reduction can implement one of the
4     //equivalence relations described earlier
5     KripkeStructure reducedModel = initialModel.Reduce();
6     List<Transition> deletedTransition = reducedModel.Repair();
7     // the transitions to be deleted in initial
8     //model in order to become repaired
9     List<Transition> transitionsTobeDeleted;
10    List<KripkeState> statesList = reducedModel.getStateList();
11    for (KripkeState state : statesList) {
12        //if the state is the result of merging multiple states
13        if (state instanceof KripkeReplacedState){
14            Map<Transition, List<Transition>> replacedTransitions =
15                state.getReplacedTransitions();
16            for (Transition trns : replacedTransitions.keySet()) {
17                if (deletedTransition.contains(trns)){
18                    transitionsTobeDeleted.addAll(replacedTransitions.get(trns));
19                }
20            }
21        }
22    }
23    initialModel.deleteTransitions(transitionsTobeDeleted);
24    return initialModel;
25 }

```

The reduction algorithm keeps a Map data structure that maps a transition in the reduced Kripke structure to a list of transitions in the initial structure, therefore after repairing the reduced structure and identifying the transitions to be deleted, using this map we can identify easily transitions to be deleted in the initial Kripke structure in order to become repaired.

Chapter 6

Repair of hierarchical Kripke structures

In this chapter we explain hierarchical model checking that is used to avoid exponential blow up in the number of states. We start by introducing hierarchical state machine and model checking of hierarchical state machine, then we introduce CTL decision procedure and then we present our implementation.

6.1 Hierarchical model checking

Model checking of hierarchical state machines was introduced in [1]. They present the model checking of sequential hierarchical (nested) systems, where these systems or models are Kripke structures that contain “boxes” that are themselves structures. This method works as follows:

- Model check a box (substructure) B w.r.t. a specification formula η_B
- Model check M (initial model) w.r.t. “coupling spec” φ that states how M uses B
- Check validity of $\varphi \wedge \eta_B \Rightarrow \eta$ using CTL decision procedure

6.2 CTL decision procedure

CTL decision Procedure is Tableau based decision procedure for satisfiability of CTL formulae. This method takes as input a CTL formula f and returns true iff f is satisfiable.

It works roughly as follows:

1. Builds tableau, a finite AND/OR graph that encodes potential models of f .
2. Test tableau for consistency by deleting inconsistent portions. If root is not deleted then f is satisfiable.

Chapter 7

Implementation

In this chapter we explain in details about the tool where we implemented our repairing algorithm and our reduction algorithms. We start by explaining the tool interface and how to use it and then we explain the software architecture by showing the different modules, the class structures, and the BNF for our CTL language.

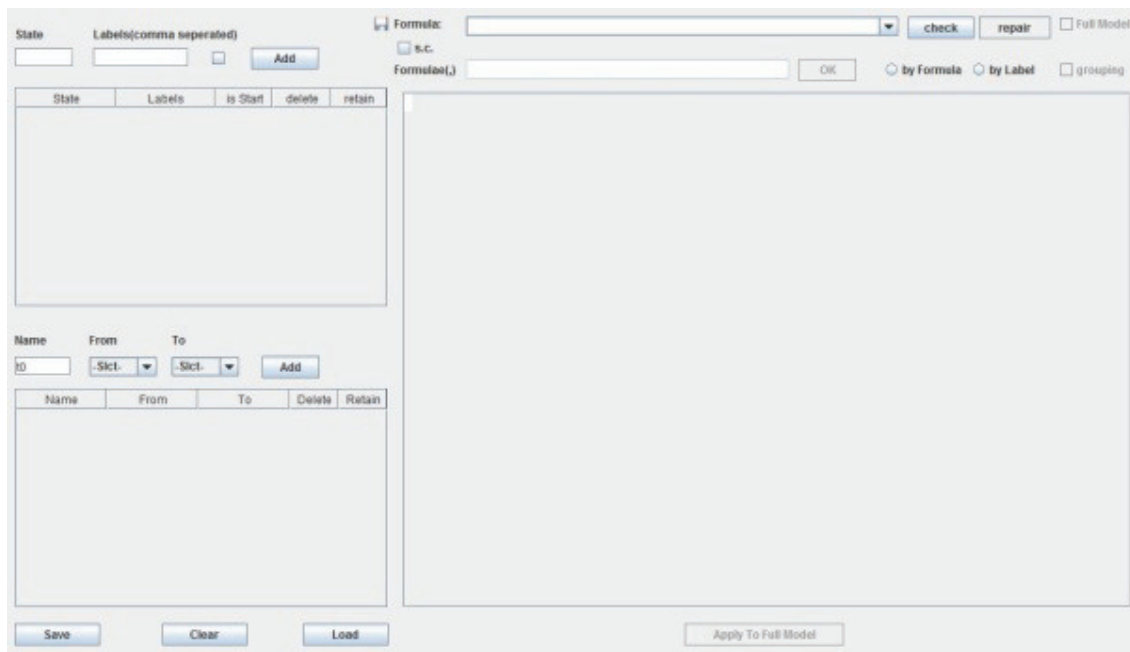


Figure 7.1: Tool's main screen

7.1 User interface and user manual

In this section we describe the tool's user interface and how to use all its features. This tool allow users to create and manage Kripke structures easily and then model check, and repair them if necessary and therefore the interface main screen is divided into five sections:

- State Creation
- Transition Creation
- Kripke Structure Visualization
- Kripke Structure Saving
- Model Checking, Model Repairing and State-space Reductions

7.1.1 State creation

The interface first section on the top left of the main screen allows users to add, edit, and delete states as well as to manage their label (atomic propositions).

State	Labels	is Start	delete	retain
State1	p,q	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
State2	r,s	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 7.2: States screen for mutex problem

To create a state, the user first enters the state name, the associated labels in a comma separated fashion (i.e. $p_1, p_2, p_3, \dots, p_n$), and specifies whether this state is initial or not. When the user clicks add the state will be displayed in the *states* table as in Figure 7.2. All the states in the table are editable, the user can edit states' name, labels, intial flag,

and can delete the state by checking the corresponding **delete** checkbox. The *retain* flag is used in the repair algorithm in order to deny the deletion of the state and to mark it as necessary for the correctness of the repaired model. State's name must be unique therefore if the user enters a name that already exists a warning message will be displayed.

7.1.2 Transition creation

The second section in our interface on the bottom left of the main screen is the Transition Creation section that allows the management of states transitions.

Name	From	To	Delete	Retain
t0	N1N2	T1N2	<input type="checkbox"/>	<input type="checkbox"/>
t1	N1N2	N1T2	<input type="checkbox"/>	<input type="checkbox"/>
t2	T1N2	C1N2	<input type="checkbox"/>	<input type="checkbox"/>
t3	N1T2	N1C2	<input type="checkbox"/>	<input type="checkbox"/>
t4	T1N2	T1T2	<input type="checkbox"/>	<input type="checkbox"/>
t5	N1T2	T1T2	<input type="checkbox"/>	<input type="checkbox"/>
t55	C1N2	C1T2	<input type="checkbox"/>	<input type="checkbox"/>
t6	N1C2	T1C2	<input type="checkbox"/>	<input type="checkbox"/>
t7	T1T2	C1T2	<input type="checkbox"/>	<input type="checkbox"/>
t8	T1T2	T1C2	<input type="checkbox"/>	<input type="checkbox"/>
t9	C1T2	C1C2	<input type="checkbox"/>	<input type="checkbox"/>
t10	T1C2	C1C2	<input type="checkbox"/>	<input type="checkbox"/>
t11	C1N2	N1N2	<input type="checkbox"/>	<input type="checkbox"/>

Figure 7.3: Transitions screen for mutex problem

To create a transition, user should specify a name, select a start state and an end state. Start states and end states are selected from dropdown lists that are bound to the already created states in the previous section. When the user clicks the **Add** button the newly created transition will be displayed in the transitions' table as in Figure 7.3. These Transitions are not editable but the user can delete them by checking the corresponding **delete** checkbox. The *retain* flag is used in the repair algorithm in order to deny the deletion of the transition and to mark it as necessary for the correctness of the repaired model. Transition's name is unique, therefore if the user enters a name that already exists a warning message will be displayed and the transition will not be created.

7.1.3 Kripke structures visualization

Created Kripke structures are displayed in the bottom right section of the tool's main screen.

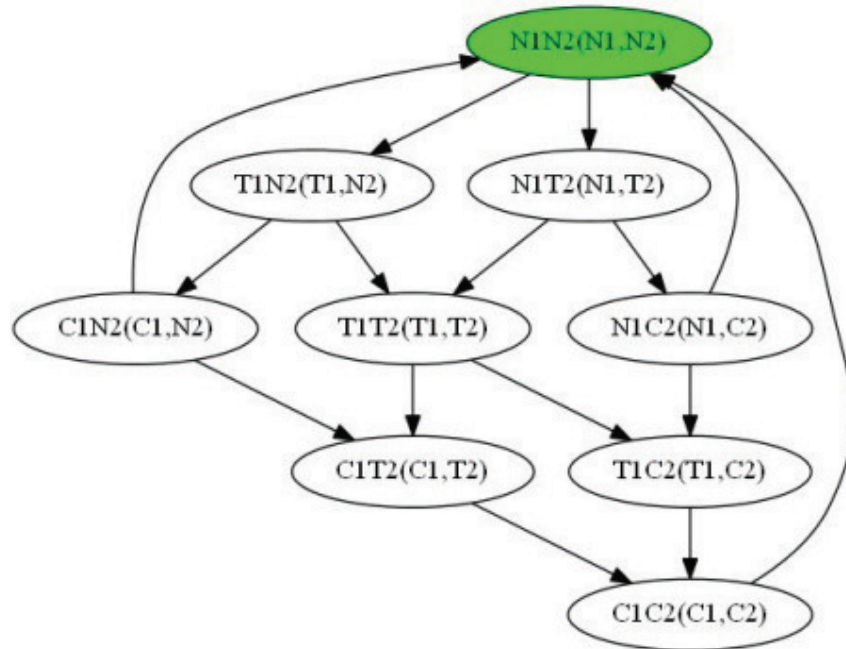


Figure 7.4: Kripke structure for mutex problem

Whenever the user makes changes to the states tables or the transitions table these changes will directly take effect on the displayed Kripke structure. For instance if the user creates two states S_1 and S_2 Figure 7.5 will be displayed. Green states are initial states. After adding a transition from S_1 to S_2 Figure 7.6 will be displayed.



Figure 7.5: State

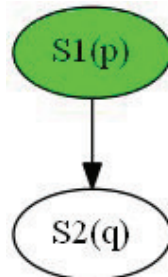
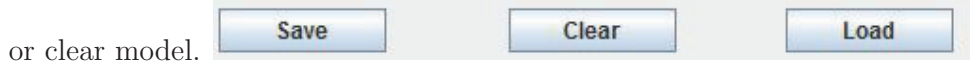


Figure 7.6: Transition

7.1.4 Kripke structure saving

The tool allows users to save their work in a text file and then reload it from that file at any time. Users might need to save their work to complete later or to keep several versions of the targeted model. Under the transitions table at the bottom left corner of the main screen there are three buttons that allow users to save their model, load a previous model



or clear model.

When a user clicks on **Save** button a pop up will be displayed to select a file location and a name for this file. If the user clicks on **Load** button a pop up will appear to allow user to browse saved files and load the saved models. **Clear** button will clear all tables in order to create a new model.

```
s0:N1,N2:true;
s1:T1,N2:false;
s2:N1,T2:false;
s3:C1,N2:false;
s4:T1,T2:false;
s5:T1,T2:false;
s6:N1,C2:false;
s7:C1,T2:false;
s8:T1,C2:false;
***
t0:s0:s1;
t1:s0:s2;
t2:s1:s3;
t3:s1:s4;
t4:s2:s5;
t5:s2:s6;
t6:s3:s0;
t7:s3:s7;
t8:s4:s7;
t9:s5:s8;
t10:s6:s0;
t11:s6:s8;
t12:s7:s2;
t13:s8:s1;
```

Figure 7.7: Saved file for mutex problem

Figure 7.7 shows the saved file structure. the file has to parts separated by *******. The first part is for states where each line describe a single state as follows: State Name : Labels comma separated : is initial state; . The second part is for transitions where each line describe a single transition as follows: Transition Name : Start State : End State; .

7.1.5 Model checking, model repairing and state-space reductions

After creating the model, the user can now model check the Kripke structure, repair it if necessary, and optimize it to reduce state space.

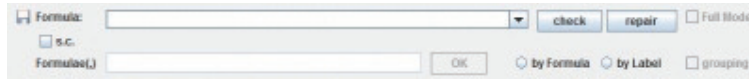


Figure 7.8: Model check and repair

Once the user completes his Kripke structure, he now can enter the CTL formula. The entered formula should respect our CTL parser rules explained in the next section. The user can choose to save his formulae by clicking the **Save** button on the top left corner of the repair section. Saved formulas can later on be selected from the CTL dropdownlist to avoid reentering complicated formulas. To model check the designed Kripke structure, the user has to click the **model check** button which implements the Clarke, Emerson, and Sistla [9] algorithm. The result of model checking will be displayed as a text message informing the user whether his model is correct or not. If the result was negative, the user can repair it by clicking on **Repair** button which implements our repair algorithm. If the model can be repaired, the displayed Kripke structure image will be changed by marking transitions to be deleted as dashed edges. On the other hand if we need to reduce the Kripke structure state space or to have a more accurate result, user can make use of our state-space feature. The **by label** checkbox implements Definition 8. Once checked a reduced Kripke structure will be displayed and the user can now model check or repair the new Kripke structure. After repairing the reduced model, the result can be projected on the initial model by clicking on the **Apply to full Model** button. The checkbox **by formula** can be used in the same way **by label** checkbox is used but they differ in the reduction algorithm they implement where **by formula** checkbox implements Definition 11. However Definition 11 needs as input a set of subformulae that are either extracted automatically from the CTL formula or extracted from the textbox where user can enter them in a comma separated fashion. The checkbox **grouping** can be checked once **by label** or **by formulae** checkbox are checked. **Grouping** checkbox with **by label** checkbox implements Definition 9 and **Grouping** checkbox with **by formulae** checkbox implements Definition 12. The checkbox **Full Model** will reload the full model that will replace all reductions. Another important and useful feature in our tool is that whenever the user chooses to reduce his model by using either the **by label** or **by formulae** checkbox, in addition to the ability to reduce or repair the reduced model, the states and transitions table on the left of the main screen will display states and transitions of the reduced model allowing users to edit these models, but then there will be no possibility to apply these changes to the initial model as the reduced model has been changed per user choice.

7.1.6 Hierarchical model checking

The tool allow the user to create substructures either starting from scratch or starting from existing initial model. If the user opt to create these substructures starting from an existing initial model, the tool provides a screen 7.9 that display all the initial model states and allow the user to select those states that form the substructure, once the states selection is done, the tool manages on its own the transitions(taken from initial model), and generates two new models: the substructure (based on user selection) and another

model that is the same as the initial models but all the selected states are now represented as a single box(machine). Moreover, the "Exist" flag is used to allow the user to keep some states in the initial model as there are important transitions going out of them. If the user opts to create the substructures starting from scratch he will need to create them as any other model described in 7.1. To make use of the CTL decision procedure, the tool provides a screen 7.10 that allows the user to enter the three formulae η_B , φ , and η and displays a message if satisfiable or not.

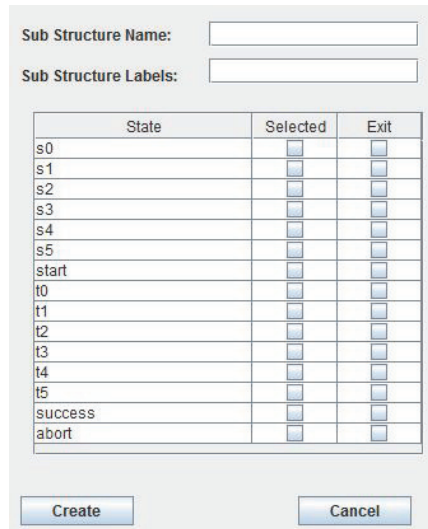


Figure 7.9: Kripke sub-structure screen

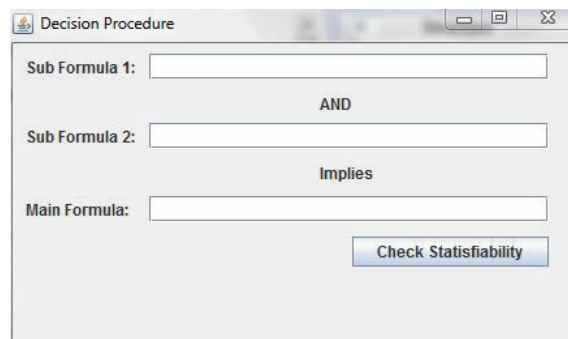


Figure 7.10: CTL decision procedure screen

7.2 Software architecture

An overall description of the tool's architecture is discussed in this section. We will start with a brief descriptions of tool's main modules and how they interact with each other. In later sections, we will discuss each module with its inputs, outputs, implemented algorithms, classes structure and interactions with other modules. The tool is

implemented using Java programming language, and it uses the library javax.swing for GUI and Graphviz [12] for Kripke structures visualization

7.2.1 Main modules

The following is a concise definition of our tool's main modules:

- CTL Parser: parses a CTL formula ϕ to generate a CTLParsedTree object which is a tree data structure representing ϕ .
- User Interface: implements GUI interface between user and the other modules.
- Model Checker: takes as input a Kripke structure $M = (S_0, S, R, L)$, and a CTL formula ϕ and verifies if M satisfies ϕ .
- Model Repairer: takes as input a Kripke structure M and a CTL formulae ϕ and return a repaired model with respect to ϕ .
- Model Optimizer: reduces the state space of created Kripke structures. It implements the state-space reduction methods in section 5.
- SAT Solver: takes as input a CNF file and return a flag that specifies whether the CNF formulae is satisfiable or not. In case it is satisfiable it also returns the satisfying valuation.
- Decision Procedure: this module implements the CTL decision procedure described in 6.2

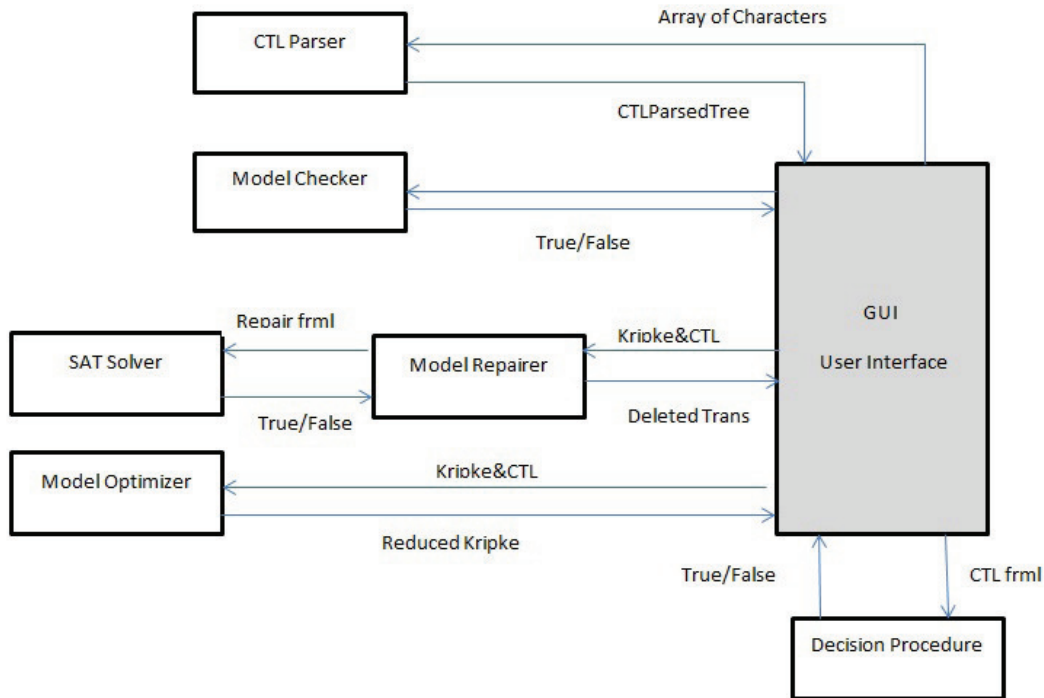


Figure 7.11: Tool main modules

Figure 7.11 depicts our main module dependency diagram and how the modules interact with each other.

7.2.2 CTL parser

Our CTL parser is implemented using ANTLR parser generator [19]. ANTLR generates code for parsers based on BNF grammar. As the Tool's BNF contains a lot of details that are specific to ANTLR which are not of our interest at this point, we will discuss the simplified BNF grammar defined in Listing 7.1. The base predicate is an atomic predicate defined at lines 67 -75. Atomic predicates are either a boolean constant, a boolean variable or arithmetic comparisons on variables and constants. A variable's syntax is defined as an arbitrary combination of lower case letters, upper case letters, `"`, and `'` (line 24). Boolean constants are either `true` or `false` as defined at line 22. Finally atomic predicates could also be arithmetic comparisons that evaluate to either true or false as defined at lines 71 - 75. Arithmetic comparisons are either between two variables or between a variable and a constant. A constant can either be numeric or a String. A numeric constant is a sequence of at least a single digit that could be prefixed by the negative sign `'-`' as defined at line 7. On the other hand, string constants are any sequence of letters surrounded by single quotes.

After defining the base atomic predicates that form a CTL formula, we will discuss CTL and propositional operators:

1. Propositional operators: are the main propositional connectives supported by our BNF grammar. These operators are divided into two groups by the arity of the operator.
 - (a) Binary operators: operate on two operands left and right
 - i. \mid : is the logical or operator.
 - ii. $\&$: is the logical and operator.
 - iii. \rightarrow : is the logical implies operator.
 - iv. \leftrightarrow : is the logical equivalent operator.
 - (b) Unary operators: operate on one operand.
 - i. $!$: is the logical negation or not operator.

2. CTL modalities: CTL modalities are divided into three types:
 - (a) CTL path quantifiers:
 - i. A: all possible paths (along all paths)
 - ii. E: at least one path (along at least one path)
 - (b) CTL binary linear time modality:
 - i. U: the until operator.
 - ii. V: the releases operator.
 - (c) CTL unary linear time modality:
 - i. X: the next time path operator.
 - ii. F: the eventually path operator.
 - iii. G: the always path operator.

The remaining of the BNF in Listing 7.1 defines the rules for building CTL formulas from the base atomic predicates and the CTL connectives.

1. CTL state formula (line 50): states that a CTL state formula is a sequence of one or more CTL state sub formula separated by propositional connectives.
2. CTL state sub formula (line 53): a CTL state sub formula is well-formed in one of the following three conditions:
 - (a) A not connective followed by a CTL state formula between opening and closing parenthesis $!(\text{CTL state formula})$.
 - (b) A CTL branch operator followed by a CTL path formula between opening and closing brackets such as $A[\text{CTL path operator}]$ or $E[\text{CTL path operator}]$.
 - (c) An atomic formula.
3. CTL path formula: a CTL path formula is valid in one of the following two conditions:

- (a) A CTL path operator followed by a CTL state formula between opening and closing brackets
- (b) A binary operation where the right and left operands are CTL state formula between opening and closing parenthesis, and the operator is CTL path connective.

Listing 7.1: CTL BNF

```
1 //propositional operators
2 PROPOSITIONAL_CONNECTIVE : '|' | '&' | '=>' | '<=>';
3
4 //arithmetic
5 COMPARISON_OPERATOR : '<' | '<=' | '>' | '>=' | '==' | '!=';
6 NEGATIVE_SIGN : '-';
7 CONSTANT : (NEGATIVE_SIGN)? (DIGIT)+ ;
8
9 NOT_CONNECTIVE : '!';
10 LEFT_PARENTHESIS : '(';
11 RIGHT_PARENTHESIS : ')';
12 LEFT_BRACKET : '[';
13 RIGHT_BRACKET : ']';
14
15 //CTL
16 CTL_BRANCH_OP : 'A' | 'E';
17 CTL_PATH_CONNECTIVE : 'U' | 'V' | 'W' ;
18 CTL_PATH_OP : ('X' | 'F' | 'G') ;
19
20 BOOLEAN : 'true' | 'false';
21 STRING : '\'' (LETTER|DIGIT)* '\'';
22
23 //variables
24 VAR : (LETTER|DIGIT|'_'|'.'|'.')+;
25
26 fragment CAPS_LETTER
27 : 'A'..'Z';
```

```
28
29 fragment SMALL_LETTER
30 : 'a'..'z';
31
32 fragment LETTER
33 : SMALL_LETTER
34 | CAPS_LETTER;
35
36 fragment DIGIT
37 : '0'..'9';
38
39 fragment NOTHING
40 : '#';
41
42 WS : (' '\r'\t'|\u000C'|\n') {_channel=99;};
43
44
45
46 ctl_formula
47 : ctl_state_formula EOF!;
48
49
50 ctl_state_formula
51 : ctl_state_sub_formula (PROPOSITIONAL_CONNECTIVE ctl_state_sub_formula)*;
52
53 ctl_state_sub_formula
54 : ctl_neg_sub_formula
55 | CTL_BRANCH_OP LEFT_BRACKET ctl_path_formula RIGHT_BRACKET
56 | atomic_formula ;
57
58
59
60 ctl_neg_sub_formula
```



```

61 : (NOT_CONNECTIVE)? LEFT_PARENTHESIS ctl_state_formula RIGHT_PARENTHESIS ;
62
63 ctl_path_formula
64 : CTL_PATH_OP LEFT_PARENTHESIS ctl_state_formula RIGHT_PARENTHESIS
65 | LEFT_PARENTHESIS ctl_state_formula RIGHT_PARENTHESIS CTL_PATH_CONNECTIVE
        LEFT_PARENTHESIS ctl_state_formula RIGHT_PARENTHESIS;
66
67 atomic_formula
68 : var_expression
69 | BOOLEAN
70
71 var_expression
72 : var1=VAR COMPARISON_OPERATOR var2=VAR
73 | VAR COMPARISON_OPERATOR CONSTANT
74 | VAR COMPARISON_OPERATOR STRING
75 | VAR;

```

7.2.3 User interface

User Interface Module is the start up module of the tool, and represents the gateway of all other modules. It contains three main classes as explained below.

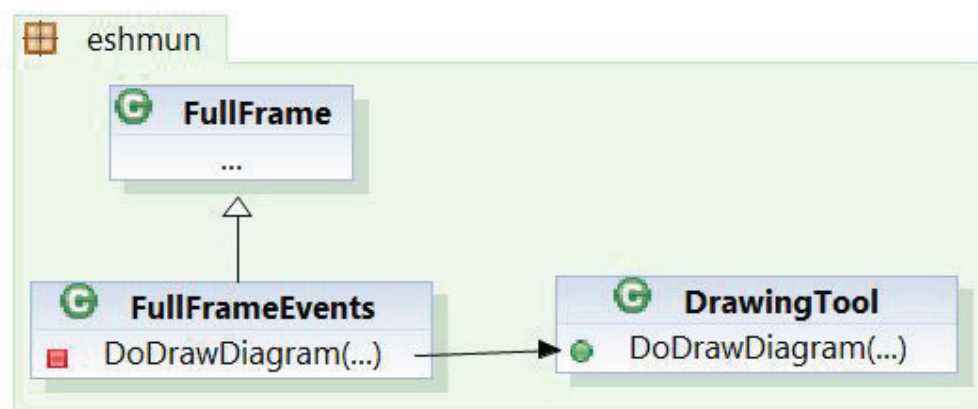


Figure 7.12: User interface module

1. FullFrame: full frame class is the main screen of the tool and contains all the

controls used by the tool

2. FullFrameEvents: this class contains all event handlers for the controls inside full frame class
3. DrawingTool: this class is responsible of drawing the Kripke structures. It uses Graphviz [12] as a drawing tool.

7.2.4 Model repairer

The Model Repairer module is responsible for repairing Kripke structures by implementing the algorithm in Figure 4.1. Figure 7.13 shows the main classes of this package.

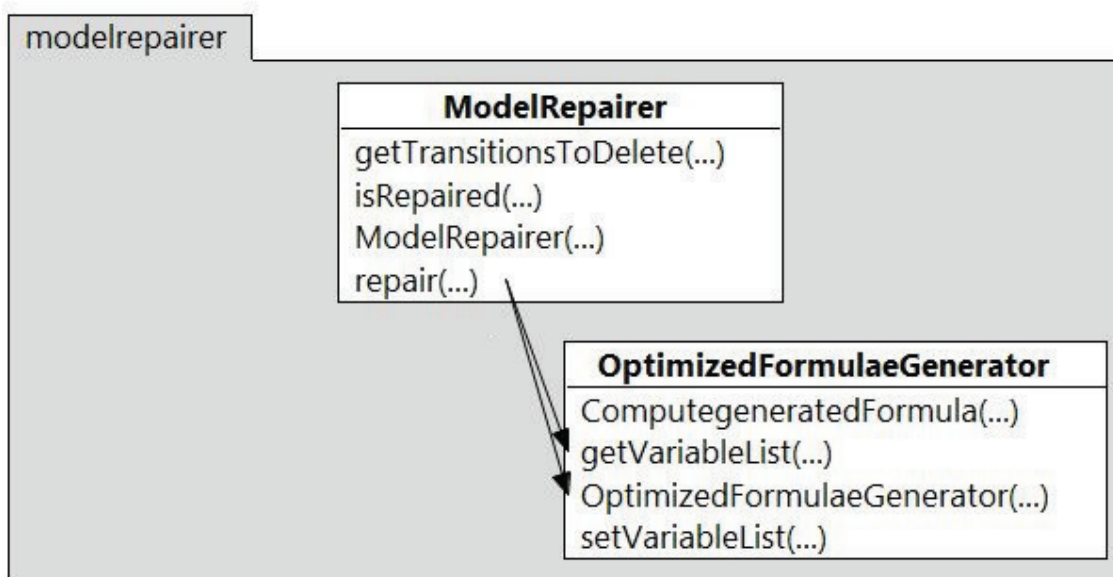


Figure 7.13: Model repairer module

ModelRepairer class has a constructor and two main methods, Repair() and GetTransitionsToDelete().

- the Constructor takes as input a Kripke structure, a CTL formulae, list of transitions to retain, and list of states to retain to initialize the repairer members.
- Repair() method (Figure 7.13) uses the repairer members to repair the Kripke structure. The method first generates the repair formula by implementing Definition 7, converts this formula into a CNF formula by implementing the algorithm in Listing 7.2, and then sends the CNF formulae to the sat solver to check if it is satisfiable or not.

- GetTransitionsToDelete method checks first if the model was repaired and returns the set of transitions to be deleted to have a correct model.

Listing 7.2: Convert to CNF

```

1 CONVERT(f)
2 {
3
4 If f is an atomic proposition then:
5     return  ;
6
7 If f has the form  $P \wedge Q$ , then:
8     return CONVERT(P)  $\wedge$  CONVERT(Q);
9
10 If f has the form  $P \vee Q$ , then:
11     let Z be a new atomic proposition;
12     CONVERT((Z  $\rightarrow$  P)  $\wedge$  ( $\sim$ Z  $\rightarrow$  Q));
13
14 If f is a negation formulae then:
15     If f has the form  $\sim A$  for some atomic proposition A, then: return f;
16     If f has the form  $\sim(\sim P)$ , then:
17         return CONVERT(P);
18     If f has the form  $\sim(P \wedge Q)$ , then:
19         return CONVERT( $\sim P \vee \sim Q$ );
20     If f has the form  $\sim(P \vee Q)$ , then:
21         return CONVERT( $\sim P \wedge \sim Q$ );
22
23 }
```

Listing 7.3: Repair() method

```

1     OptimizedFormulaeGenerator formulaGenerator = new
2         OptimizedFormulaeGenerator(modelKripke, specFormulae);
3     PredicateFormula CNFFormula =
4         formulaGenerator.ComputeGeneratedFormula(transitionsToRetain,
5         statesToRetain);
6     formulaStringList = formulaGenerator.getVariableList();
7     SAT4jSolver satSolver = new SAT4jSolver(CNFFormula, formulaStringList);
8     boolean success = satSolver.isSatisfiable ();
```

Listing 7.4: Repairer utilization

```

1 ModelRepairer repairer = new ModelRepairer(Kripke, spec, transitionsToRetain,
      statesToRetain);
2 boolean isRepaired = repairer.repair();
3 if (isRepaired)
4 {
5     List<Transition> deletedTransitions =
          GetTransitionsToBeDeleted(list);
6     DoDrawDiagram(kripke, deletedTransitions);
7 }

```

Algorithm for computing $repair(M, \eta)$

Figure 7.15 gives an algorithm that computes $repair(M, \eta)$ from $M = (S_0, S, R, L)$ and η . The algorithm operates as follows. We introduce a label $\mathcal{L}(s)$ for each state s in M . $\mathcal{L}(s)$ is a subset of $sub(\eta)$. Initially, $\mathcal{L}(s_0) = \eta$, and $\mathcal{L}(s) = \emptyset$ for all $s \in S - \{s_0\}$. The algorithm propagates formulae from the label of some state s to the labels of all successor states t of s . This propagation is performed according to Definition 7, so that if, for some CTL formula φ , $\varphi \in \mathcal{L}(s)$, and Definition 7 requires that some other CTL formula ψ (related to φ) be evaluated in every successor t , then we add ψ to $\mathcal{L}(t)$. For example, suppose $A[\varphi V \psi] \in \mathcal{L}(s)$. Then, for every successor t of s , we must add φ, ψ , and $A[\varphi V \psi]$ to $\mathcal{L}(t)$. Note that for each $\xi \in \mathcal{L}(s)$, we propagate at most one formula to the successors of s . Once $\xi \in \mathcal{L}(s)$ has been processed in this manner, we “mark” it, so that we do not repeat the propagation. We introduce a boolean array $marked(s, \varphi)$ for this purpose. When a propagation is performed, the appropriate conjunct is added to $repair(M, \eta)$. For the release modality, we include the index with the propagated formulae, so that we can “count down” properly. We summarize the data structures used:

- $repair(M, \eta)$: a string, which accumulates the repair formula which is being computed
- $\mathcal{L}(s)$: a subset of $sub(\eta)$. Contains the formulae which have been propagated to s , and whose truth in s affects the truth of η in s_0 .
- $marked(s, \xi)$: a boolean array, initially all false. An entry is set to true when formula $\xi \in \mathcal{L}(s)$ has been processed.

Figure 7.14 gives the overall algorithm **ComputeRepairFormula**. We initialize $repair(M, \eta)$ by invoking **InitializeRepairFormula(,)** which sets $repair(M, \eta)$ to the conjunction of Clause 2–5 of Definition 7. These clauses do not depend on the transitions in M , and so can be computed without traversing M . Figure 7.15 gives the propagation step **propagate**, which propagates formulae from the label $\mathcal{L}(s)$ of s to the labels of the successor states $t \in R[s]$ of s . When a propagation is performed, **propagate** invokes **conjoin**(given in Figure 7.17), which updates $repair(M, \eta)$, according to Definition 7, by conjoining the appropriate clause.

```
ComputeRepairFormula(repair( $M, \eta$ ))  
InitializeRepairFormula( $M, \eta$ );  
forall  $s_0 \in S_0$  :  $new(s) := \{\eta\}$  endfor ;           //  $\eta$  must hold in all initial states  
repeat until no change  
    select some state  $s$  in  $M$  and some  $\xi \in new(s)$ ;  
    propagate( $s, \xi$ )
```

Figure 7.14: The model repair algorithm.

propagate(s, ξ)

if $\xi \in \text{old}(s)$ **then** // ξ has already been processed

$\text{new}(s) := \text{new}(s) - \xi;$

return

//already checked for larger index

if $\xi = \mathbf{A}[\varphi\mathbf{V}\psi]^m$ and $\mathbf{A}[\varphi\mathbf{V}\psi]^{m'} \in \text{old}(s)$ for some $m' \geq m$ **then**

$\text{new}(s) := \text{new}(s) - \xi;$

return

//already checked for larger index

if $\xi = \mathbf{E}[\varphi\mathbf{V}\psi]^m$ and $\mathbf{E}[\varphi\mathbf{V}\psi]^{m'} \in \text{old}(s)$ for some $m' \geq m$ **then**

$\text{new}(s) := \text{new}(s) - \xi;$

return

case ξ : // ξ has not been processed

$\xi = \neg\varphi$:

$\text{new}(s) := \text{new}(s) \cup \{\varphi\};$

conjoin(" $X_{s,\neg\varphi} \equiv \neg X_{s,\varphi}$ ");

$\xi = \varphi \vee \psi$:

$\text{new}(s) := \text{new}(s) \cup \{\varphi, \psi\};$

conjoin(" $X_{s,\varphi\vee\psi} \equiv X_{s,\varphi} \vee X_{s,\psi}$ ");

$\xi = \varphi \wedge \psi$:

$\text{new}(s) := \text{new}(s) \cup \{\varphi, \psi\};$

conjoin(" $X_{s,\varphi\wedge\psi} \equiv X_{s,\varphi} \wedge X_{s,\psi}$ ");

$\xi = \mathbf{A}X\varphi$:

forall $t \in R[s] : \text{new}(t) := \text{new}(t) \cup \{\varphi\}$ **endfor** ;

conjoin(" $\bigwedge_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\varphi})$ ")

$\xi = \mathbf{E}X\varphi$:

forall $t \in R[s] : \text{new}(t) := \text{new}(t) \cup \{\varphi\}$ **endfor** ;

conjoin(" $\bigvee_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\varphi})$ ")

$\xi = \mathbf{A}[\varphi\mathbf{V}\psi]$:

$\text{new}(s) := \text{new}(s) \cup \{\mathbf{A}[\varphi\mathbf{V}\psi]^n\};$

conjoin(" $X_{s,\mathbf{A}[\varphi\mathbf{V}\psi]} \equiv X_{s,\mathbf{A}[\varphi\mathbf{V}\psi]}^n$ ");

$\xi = \mathbf{A}[\varphi\mathbf{V}\psi]^m, m \in \{1..n\}$:

$\text{new}(s) := \text{new}(s) \cup \{\varphi, \psi\};$

forall $t \in R[s] : \text{new}(t) := \text{new}(t) \cup \{\mathbf{A}[\varphi\mathbf{V}\psi]^{m-1}\};$

conjoin(" $X_{s,\mathbf{A}[\varphi\mathbf{V}\psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigwedge_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\mathbf{A}[\varphi\mathbf{V}\psi]}^{m-1}))$ ")

$\xi = \mathbf{A}[\varphi\mathbf{V}\psi]^0$:

$\text{new}(s) := \text{new}(s) \cup \{\psi\};$

conjoin(" $X_{s,\mathbf{A}[\varphi\mathbf{V}\psi]}^0 \equiv X_{s,\psi}$ ");

$\xi = \mathbf{E}[\varphi\mathbf{V}\psi]$:

$\text{new}(s) := \text{new}(s) \cup \{\mathbf{E}[\varphi\mathbf{V}\psi]^n\};$

conjoin(" $X_{s,\mathbf{E}[\varphi\mathbf{V}\psi]} \equiv X_{s,\mathbf{E}[\varphi\mathbf{V}\psi]}^n$ ");

$\xi = \mathbf{E}[\varphi\mathbf{V}\psi]^m, m \in \{1..n\}$:

$\text{new}(s) := \text{new}(s) \cup \{\varphi, \psi\};$

forall $t \in R[s] : \text{new}(t) := \text{new}(t) \cup \{\mathbf{A}[\varphi\mathbf{V}\psi]^{m-1}\};$

conjoin(" $X_{s,\mathbf{E}[\varphi\mathbf{V}\psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigvee_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\mathbf{E}[\varphi\mathbf{V}\psi]}^{m-1}))$ ")

$\xi = \mathbf{E}[\varphi\mathbf{V}\psi]^0$:

$\text{new}(s) := \text{new}(s) \cup \{\psi\};$

conjoin(" $X_{s,\mathbf{E}[\varphi\mathbf{V}\psi]}^0 \equiv X_{s,\psi}$ ");

endcase ;

$\text{new}(s) := \text{new}(s) - \{\xi\};$ //remove ξ from new since it has been processed

<u>InitializeRepairFormula(M, η)</u>	
$repair(M, \eta) := \text{true};$	
$\text{conjoin}(\bigvee_{s_0 \in S_0} X_{s_0});$	Clause 1
$\text{forall } s \in S_0 : \text{conjoin}(X_{s_0} \Rightarrow X_{s_0, \eta});$	Clause 2
$\text{forall } s \in S : \text{conjoin}(X_s \equiv \bigvee_{t \in R[s]} (E_{s,t} \wedge X_t));$	Clause 3
$\text{forall } (s, t) \in R : \text{conjoin}(E_{s,t} \Rightarrow (X_s \wedge X_t));$	Clause 4
$\text{forall } s \in S, p \in \mathcal{AP} \cap L(s) : \text{conjoin}(X_{s,p});$	Clause 5
$\text{forall } s \in S, p \in \mathcal{AP} - L(s) : \text{conjoin}(\neg X_{s,p});$	Clause 5

Figure 7.16: Initializing $repair(M, \eta)$

<u>conjoin(f)</u>
$g := \text{""};$
case f:
f does not contain either of $\bigwedge_{t \in R[s]}, \bigvee_{t \in R[s]}$
$g := f;$
$f = \bigwedge_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\varphi})$
$\text{forall } t \in R[s] : g := g \frown \bigwedge (E_{s,t} \Rightarrow X_{t,\varphi})$
$f = \bigvee_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,\varphi})$
$\text{forall } t \in R[s] : g := g \frown \bigvee (E_{s,t} \Rightarrow X_{t,\varphi})$
$f = X_{s,A[\varphi \vee \psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigwedge_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,A[\varphi \vee \psi]}^{m-1}))$
$\text{forall } t \in R[s] : g := g \frown \bigwedge (E_{s,t} \Rightarrow X_{t,A[\varphi \vee \psi]}^{m-1});$
$g := X_{s,A[\varphi \vee \psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \text{""} \frown g \frown \text{""})$
$f = X_{s,E[\varphi \vee \psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \bigvee_{t \in R[s]} (E_{s,t} \Rightarrow X_{t,E[\varphi \vee \psi]}^{m-1}))$
$\text{forall } t \in R[s] : g := g \frown \bigvee (E_{s,t} \Rightarrow X_{t,E[\varphi \vee \psi]}^{m-1});$
$g := X_{s,E[\varphi \vee \psi]}^m \equiv X_{s,\psi} \wedge (X_{s,\varphi} \vee \text{""} \frown g \frown \text{""})$
endcase ;
$repair(M, \eta) := repair(M, \eta) \frown g$

Figure 7.17: Adding a conjunct to $repair(M, \eta)$

7.2.5 Model optimizer

This module is responsible for generating smaller Kripke structures. Given a Kripke structure M and a CTL formula η we reduce the state space of M and generate a smaller Structure \overline{M}

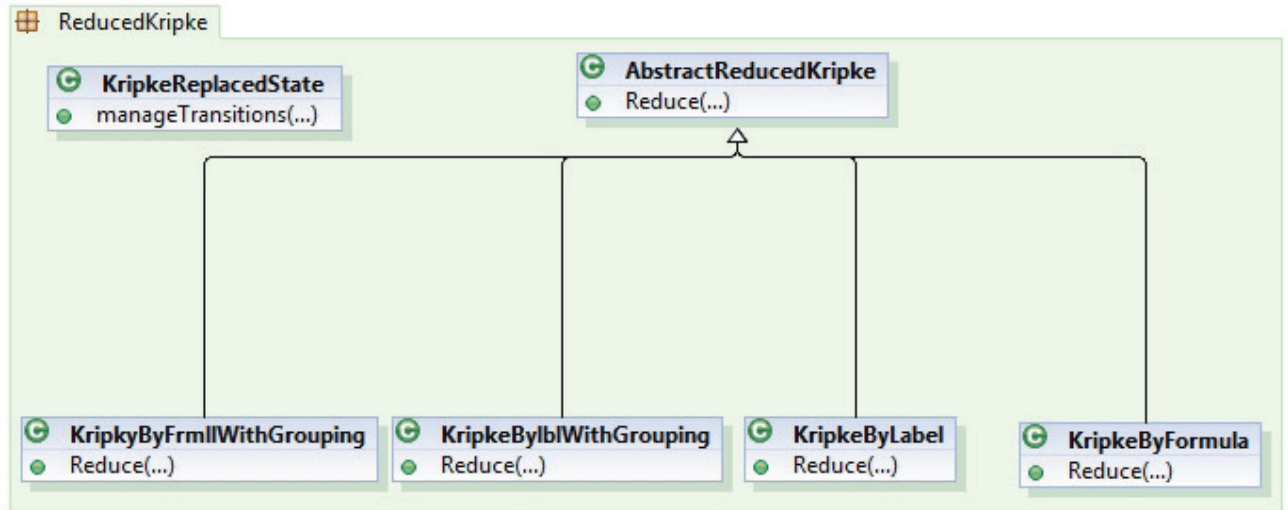


Figure 7.18: Kripke optimizer module

This module contains classes that are responsible for state space reduction of a Kripke structure given a CTL formula. Below are a brief description for each class. All the reductions use Tarjan's strongly connected components algorithm [25], Listing 7.5.

1. AbstractReduceKripke Class: abstract class for all Kripke structures optimizers
2. KripkeByLabel Class: Reduce Kripke structure state space according to atomic propositions of the CTL formula while taking the adjacency of the states into consideration. It implements Definition 8.
3. KripkeByLblWithGrouping Class: Reduce Kripke structure state space according to atomic propositions of the CTL formula without taking the adjacency of the states into consideration. It implements Definition 9.
4. KripkeByFormula Class: Reduce Kripke structure state space with respect to sub-formulae of the CTL formula while taking the adjacency of the states into consideration. It implements Definition 11.
5. KripkeByFrmlWithGrouping Class: Reduce Kripke structure state space with respect to sub-formulae of the CTL formula without taking the adjacency of the states into consideration. It implements Definition 12.

Listing 7.5: Tarjan's strongly connected components algorithm

```

1
2 function DetectCycles(G = (V, E))
3
4   index := 0
5   Stack := empty
6   for each v in V do
7     if (!v.isMarked) then
8       DFS(v)
9     end if
10  end for
11
12 function DFS(v)
13   v.isMarked := true
14   v.index := index
15   v.low := index
16   index := index + 1
17   Stack.push(v)
18
19   for each (v, w) in E do
20     if (!w.isMarked) then
21       DFS(w)
22       v.low := min(v.low, w.low)
23     else if (w is in Stack) then
24       v.low := min(v.low, w.low)
25     end if
26   end for
27
28   if (v.low = v.index) then
29     start a new strongly connected component
30     repeat
31       w := Stack.pop()
32       add w to current strongly connected component
33     until (w = v)
34     output the current strongly connected component
35   end if
36 end function

```

7.2.6 SAT solver

After a failed Model Check, A repair formula in CNF form is generated and is sent to SAT solver to specify if the model can be repaired or not. Our tool uses SAT4jSolver [18] to check formulas satisfiability.

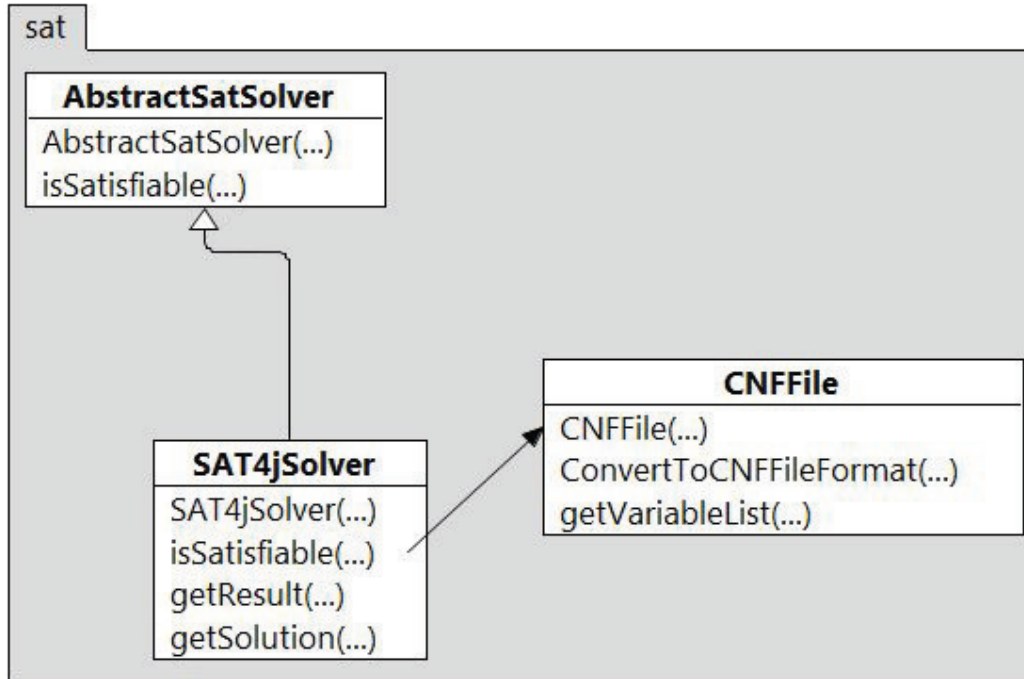


Figure 7.19: SAT solver module

To check if a formula is satisfiable the CNF formula is transformed into a CNF file which can be read by SAT4J solver to check satisfiability.

```

p cnf 167 406
1 0
-106 136 0
-81 135 0
-135 81 0
-136 106 0
-101 134 0
-89 133 0
-133 89 0
-134 101 0
-85 132 0
-108 131 0
-131 108 0
-132 85 0
  
```

(a) CNF

```

1 2 -3 4 5 6 7 8 9 -10 11 -12 13 -14 15 16 17 18 -19
20 21 22 23 24 25 -26 27 28 -29 -30 -31 -32 -33 34 -35
-36 37 38 -39 -40 -41 -42 43 -44 -45 -46 47 -48 49 -50
-51 -52 53 54 -55 56 -57 -58 -59 60 -61 -62 -63 64 -65
66 -67 68 69 -70 -71 -72 -73 -74 75 -76 77 -78 -79 80
81 82 -83 -84 85 86 -87 -88 89 90 -91 92 -93 94 -95 96
-97 -98 99 -100 101 -102 -103 104 -105 106 -107 -108
109 110 111 -112 113 114 -115 -116 117 -118 119 -120
121 -122 123 124 -125 126 -127 128 129 130 -131 132
133 134 135 136 -137 138 139 -140 -141 -142 -143 -144
-145 -146 -147 -148 -149 -150 -151 -152 -153 -154 -155
-156 -157 -158 -159 -160 -161 -162 -163 -164 -165 -166
-167 0
  
```

(b) SAT Solver Solution

The CNF file is a text file in which the first line specifies the number of variables used and the number of disjunction clauses. The remainder of the file contains lines defining the disjunction boolean expressions of the CNF formula where the number 0 specifies the end of the line. For instance, in Figure 7.20a, the first line states that it contains 167 variables and 406 disjunction clauses. The Third line is the disjunction clause $\neg x_{106} \vee x_{136}$.

If the SAT Solver identify a CNF formula as satisfiable, it returns an array of integers

where each integer represent the boolean variable index, and if the integer is negative that means that the corresponding boolean variable should be equal to false and other wise it is true. Figure 7.20b shows an example of SAT4J solution for Mutex problem. Note that each boolean variable is either $E_{s,t}$ or $X_{s,\alpha}$. If the boolean variable corresponds to an $E_{s,t} = \text{false}$ that means the transition from s to t should be deleted to repair the model.

7.2.7 Decision procedure

The Decision Procedure Module is responsible for creating substructures and for implementing the CTL decision Procedure.

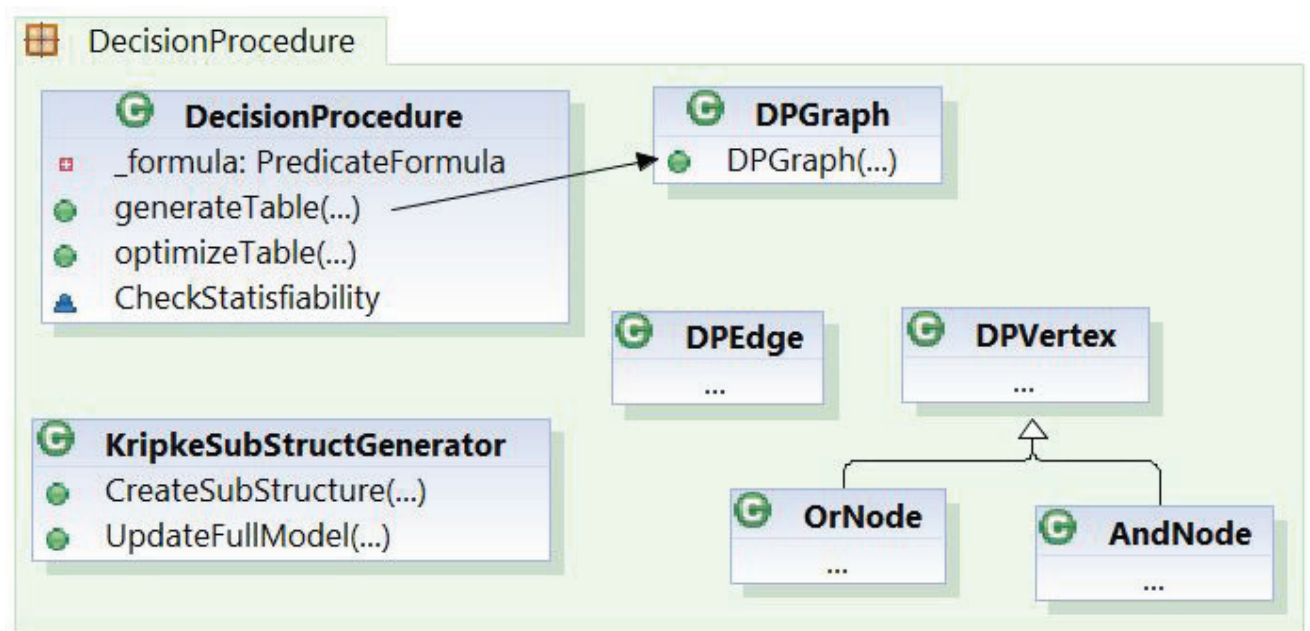


Figure 7.20: Decision procedure module

This Module contains two main classes that interact with other classes in order to accomplish their jobs. These two classes are:

- KripkeSubStructGenerator class: this class is responsible of creating sub-structures starting form an initial model, and then updates the initial model to treat all the states of the sub-structure as a single box.
- DecisionProcedure class: this class is responsible of checking the satisfiability of a CTL formula. First it generates an and/or graph (DPGraph), then it deletes all inconsistent nodes (optimizeTable) and at last it check if the initial state of the graph is preserved it return a flag that indicates that the CTL formula is satisfiable.

Chapter 8

Case studies

In this chapter, We will show three case studies that were conducted on three known problems in concurrent programming.

1. Mutual exclusion
2. Barrier synchronization
3. Phone call system

For each case study we will give a small description for the problem, display the corresponding initial model, model check it and repair it with respect to a specification formula, display the repaired model, and apply all reduction strategies explained in chapter three.

8.1 Mutual exclusion

Mutual exclusion problem is a common problem in concurrent programming when multiple processes are sharing a resource and trying to have exclusive access to this resource. A successful algorithm for this problem is to make sure that no two processes are in critical section (accessing a shared resource) at the same time.

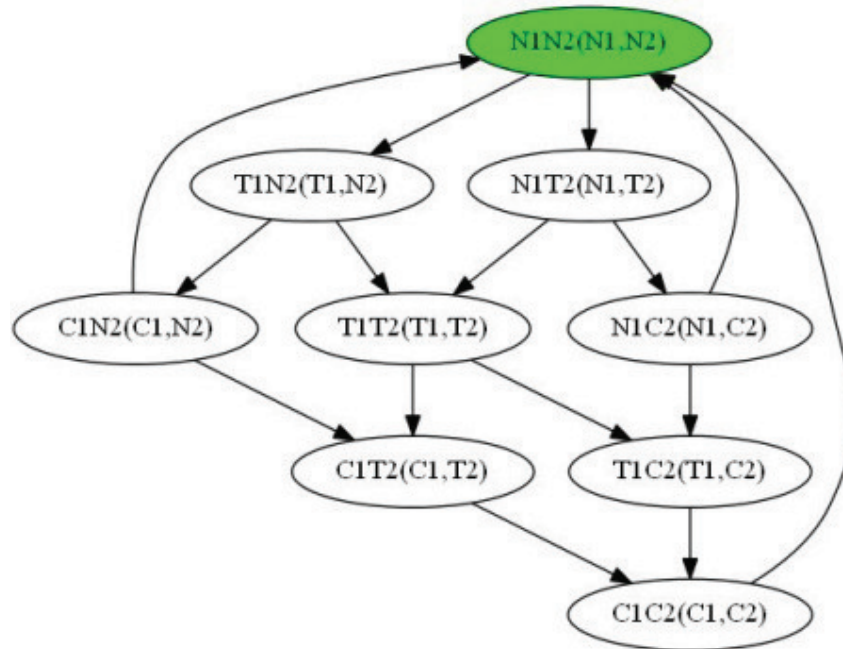


Figure 8.1: States screen for mutex problem

Figure 8.1 depicts a model where two processes P_1 and P_2 are trying to have an exclusive access to a shared resource. The model's state are named according to the labels they have. Each state has a text inside it that specifies its name and between brackets its labels. The labels are defined as follows:

- N_i states that the process P_i is in an idle state
- T_i states that the process P_i in in "Trying" state (trying to have access).
- C_i states that the process P_i is accessing the shared resource.

Therefore state " N_1T_2 " means that process P_1 is idle and process P_2 is in trying to have access and state " T_1C_2 " means that process P_1 is trying to have access and process P_2 is having access. When model checking this model with respect to the following specification formula : $AG((C1 \ \& \ C2))$, we get this message: "The model is incorrect and needs to be repaired". the model check failed as the initial model in Figure 8.1 has a state C1C2 which has the labels C1 and C2.

Figure 8.2 the result of repairing the initial model in Figure 8.1 with respect to $AG((C1 \ \& \ C2))$ CTL formula. Dashed arrows denote arrows to be deleted in order to correct the model.

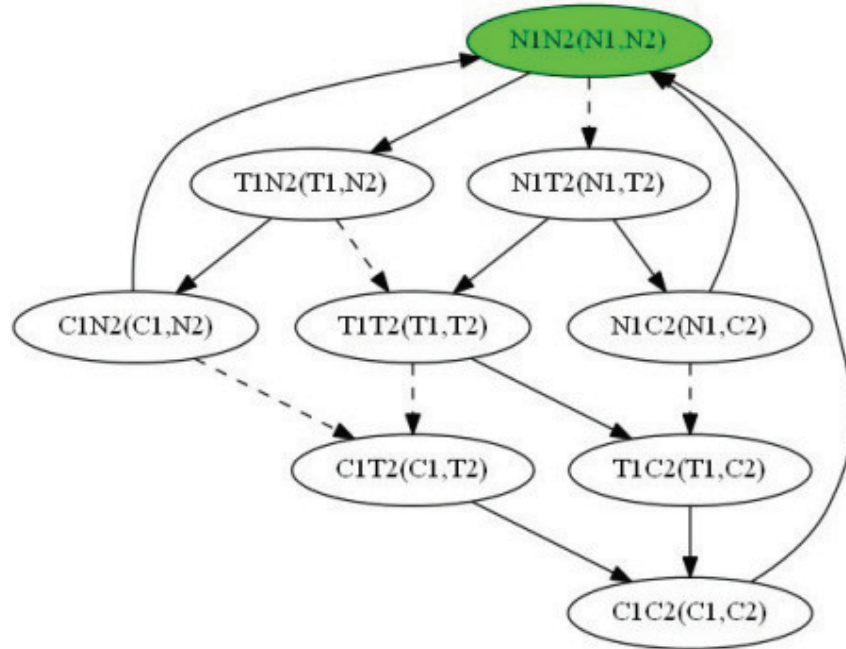


Figure 8.2: Repaired model for mutex problem

As we can see in Figure 8.2 a lot of transitions were deleted, but also a lot of other solutions exists with less deleted transitions. Reductions explained in chapter three in addition to state-space reduction, they give us better solutions. Figure 8.3a shows a reduced model with respect to atomic propositions in the specification formula($C1$ and $C2$) whereas Figure 8.3b shows a the result of repairing the reduced model. Note that in this example if we apply definition 3.1 or definition 3.2 we will have the same result as all states that have the same common atomic proposition with our specification formula are adjacent.



Figure 8.3: Reduction w.r.t. atomic propositions

Figure 8.3b shows a repaired model after reduction but we still need to repair the initial

model which can be done by reverting the reduction algorithm on the repaired reduced model as in Figure 8.4.

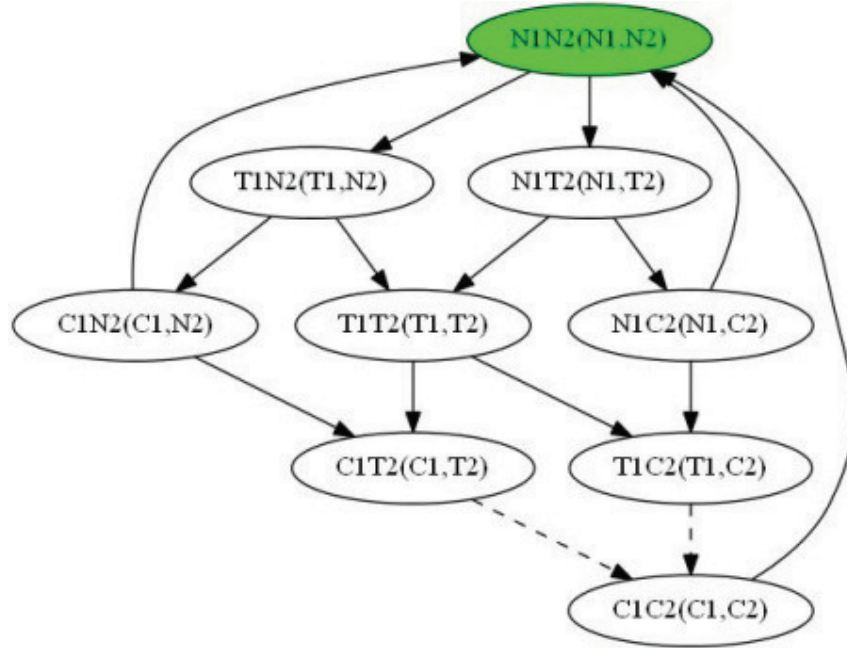


Figure 8.4: Repaired model using reduction w.r.t. atomic propositions

8.2 Barrier synchronization

Barrier is a synchronization method in parallel computing where a thread or a process in a program stops at a point and cannot continue until other threads or processes reach this point. Once all have arrived, each thread or process is allowed to proceed.

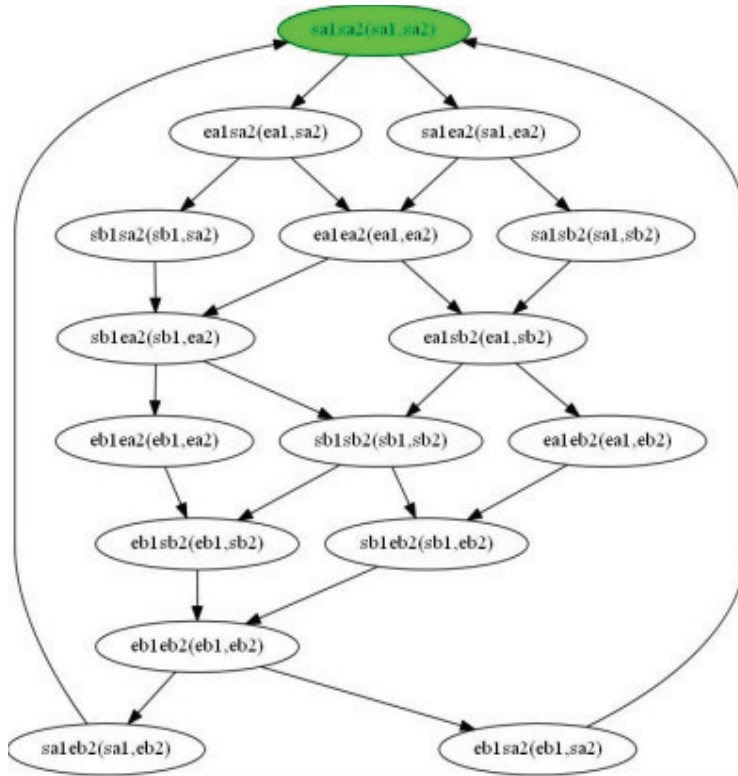


Figure 8.5: Barrier synchronization

Figure 8.6 the result of repairing the initial model in Figure 8.5 with respect to $A[G((sa1\&sb2))\&A[G((sa2\&sb1))]]\&(A[G((ea1\&eb2))]\&A[G((ea2\&eb1))])$ CTL formula. Dashed arrows denote arrows to be deleted in order to correct the model.

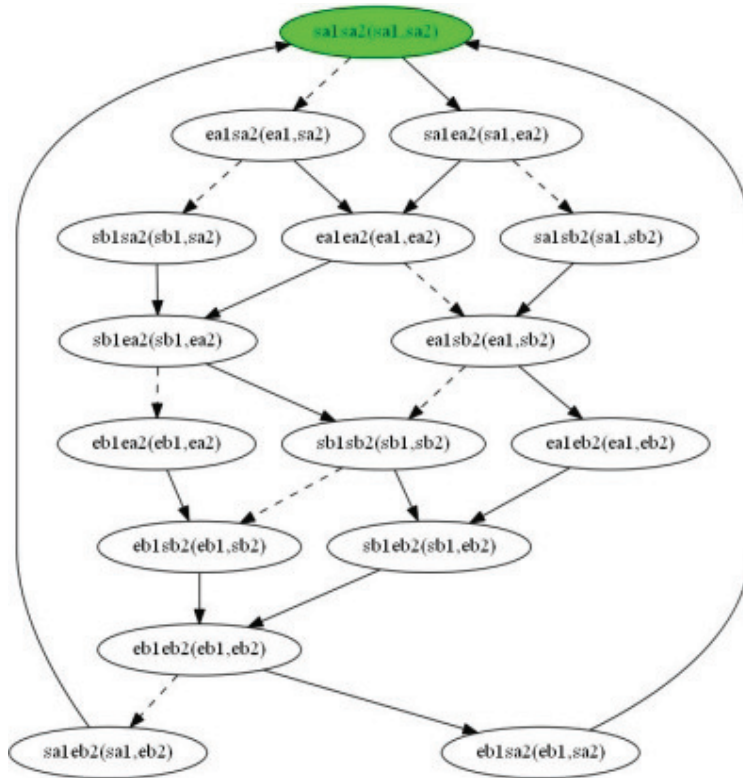


Figure 8.6: Repaired barrier

Figure 8.7a shows a reduced model with respect to Sub-Formulae in the CTL formula $AG(\neg(SA_1 \wedge SB_2)) \wedge AG(\neg(SA_2 \wedge SB_1)) \wedge AG(\neg(EA_1 \wedge EB_2)) \wedge AG(\neg(EA_2 \wedge EB_1))$ whereas Figure 8.7b shows a the result of repairing the reduced model. Note that in this example if we apply definition 3.3 or definition 3.4 we will have the same result as all states that have the same common sub-formulae with our CTL formula are adjacent.

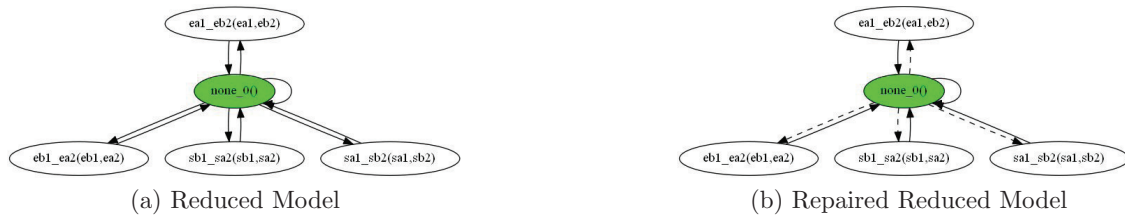


Figure 8.7: Reduction w.r.t. sub-formulae

Figure 8.8 depicts the result of projecting the result of the repaired reduced model on the initial model.

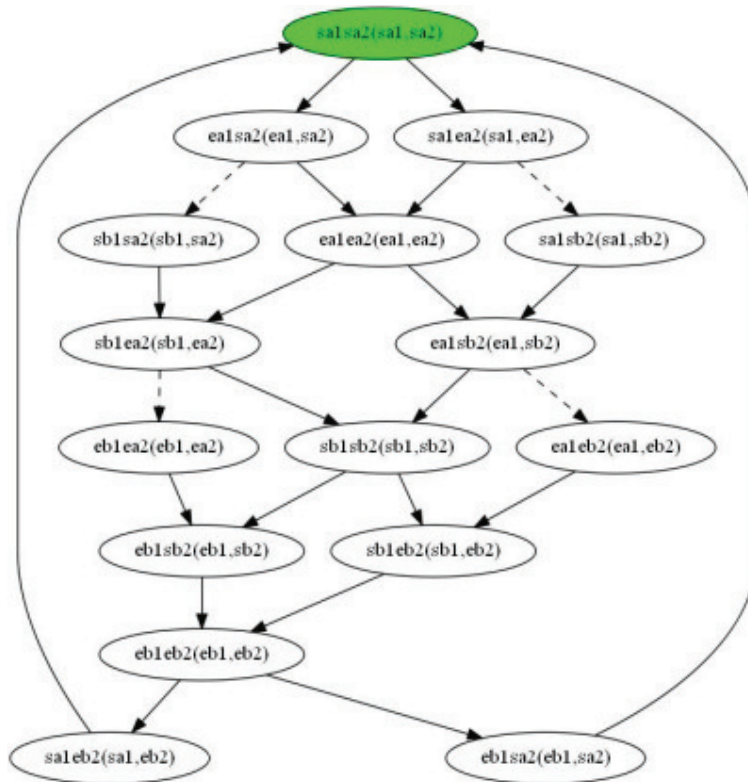


Figure 8.8: Repaired model using reduction w.r.t. sub-formulae

8.3 Phone call system

We describe here the phone call system in order to show how the tool can be used to make use of Hierarchical Model Checking and the CTL decision procedure. The phone call system model Figure 8.9 depicts two attempts to make a call, if the first attempt fails another trial is done and if both fail the phone call fails. On the other hand if any attempt succeeds then the phone call is competed successfully.

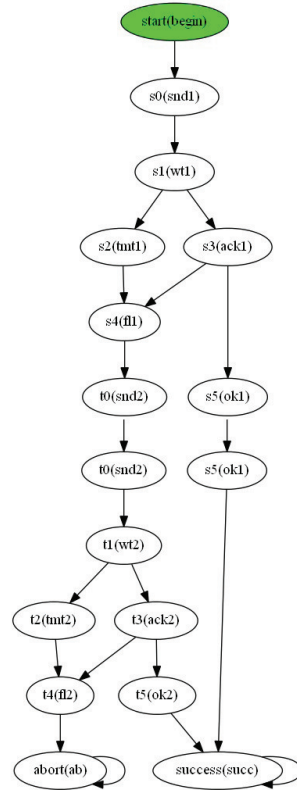


Figure 8.9: Phone call system

The states s_0 to s_5 represent the first attempt and the states t_0 to t_5 represent the second attempt as follows:

- 0: Send (s_0 and t_0)
- 1: Wait (s_1 and t_1)
- 2: Timeout (s_2 and t_2)
- 3: Acknowledge (s_3 and t_3)
- 4: Fail (s_4 and t_4)
- 5: Ok (s_5 and t_5)

Figure 8.10 depicts two substructures that represents the two phone call attempts. The tool allows the user to extract these two substructures from the initial model and then the user can check or repair them alone.

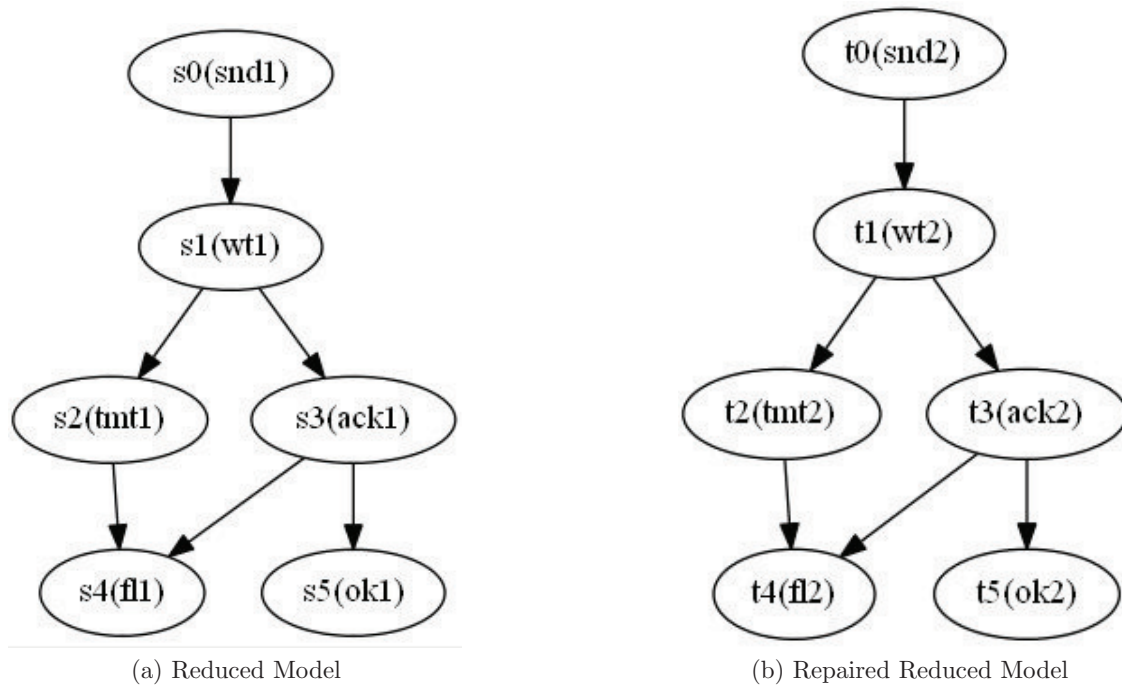


Figure 8.10: Phone calls

After extracting the two substructures, they will be replaced by a single box in the initial model with additional boe for every transition out of this substructure. Figure 8.11 depicts the resulted model after replacing the substructures of the phone call attempts. Notice that the first attempt substructure is represented in the new model by three states (Call1, s_4 , s_4). The states s_4 and s_4 remains in the new system as there are transitions out of these states that needs to remain in the resulted model as per user selection (the user chose to keep these transitions as explained in section 7.1.6).

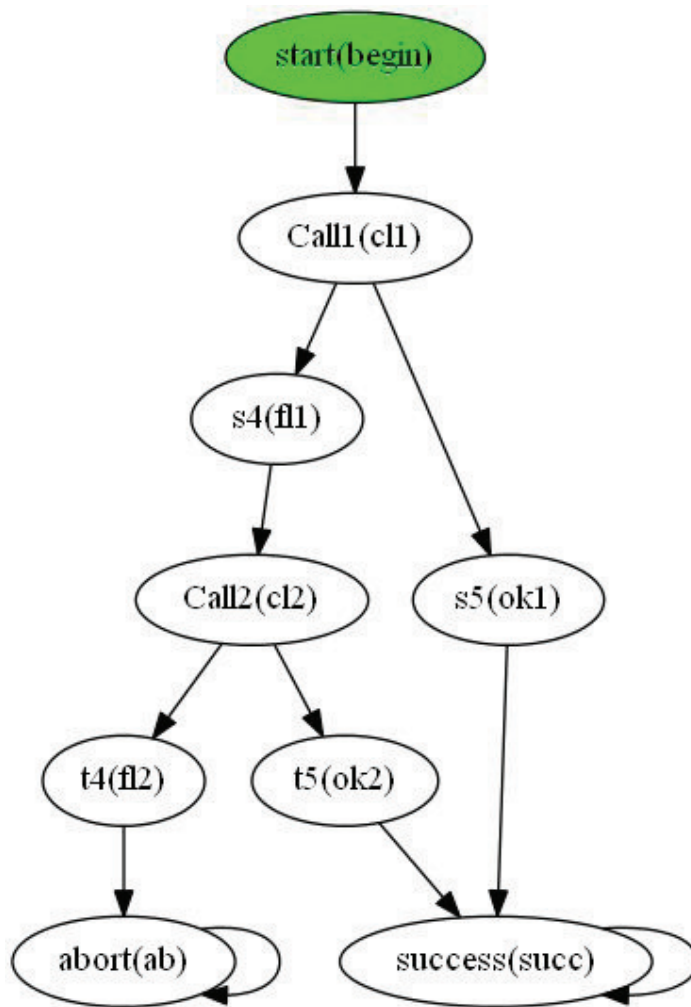


Figure 8.11: Hierarchical

Chapter 9

Conclusions and future work

In this thesis, we presented and implemented a method for the repair of Kripke structures. We demonstrated our method on several case studies: mutual exclusion, barrier synchronization, and phone call. Our implementation allows experimentation and adjustment of the repair until an optimal repair is obtained. We also presented state-space reduction techniques, which work to both reduce the complexity of the method, and to improve the quality of the repair. In our case studies, the state-space reduction methods resulted in optimal repairs. Finally, we extended the method to hierarchical Kripke structures, and demonstrated the application to a phone call system.

Future work includes (1) more ambitious case studies, (2) extending the current state-space reduction methods to make them more flexible, e.g., by using temporal sub-formulae (as opposed to propositional ones) to define the equivalence relations, and (3) extending the repair method to concurrent Kripke structures.

Chapter 10

Bibliography

- [1] Rajeev Alur and Mihalis Yannakakis. Model checking of hierarchical state machines. *ACM SIGSOFT Software Engineering Notes*, 23(6):175–188, 1998.
- [2] M. Antoniotti and B. Mishra. Np-completeness of the supervisor synthesis problem for unrestricted ctl specifications. In *Workshop on Discrete Event Systems (WODES 96)*, 1996.
- [3] Paul C. Attie and Jad Saklawi. Model and program repair via sat solvers. *CoRR*, abs/0710.3332, 2007.
- [4] P.C. Attie and E.A. Emerson. Synthesis of concurrent systems for an atomic read / atomic write model of computation (extended abstract). In *PODC*, 1996.
- [5] P.C. Attie and E.A. Emerson. Synthesis of concurrent programs for an atomic read-/write model of computation. *TOPLAS*, 23(2):187–242, 2001.
- [6] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *TACAS'99, LNCS number 1579*, 1999.
- [7] F. Buccafurri, T. Eiter, G. Gottlob, and N. Leone. Enhancing model checking in verification by AI techniques. *Artif. Intell.*, 1999.
- [8] George Chatzieftheriou, Borzoo Bonakdarpour, Scott A Smolka, and Panagiotis Katsaros. Abstract model repair. In *NASA Formal Methods*, pages 341–355. Springer, 2012.
- [9] E. M. Clarke, E. A. Emerson, and P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 1986.
- [10] 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.

- [11] Yulin Ding and Yan Zhang. Ctl model update: Semantics, computations and implementation. In *ECAI*, pages 362–366, 2006.
- [12] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.
- [13] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.
- [14] O Grumberg and D.E. Long. Model checking and modular verification. *TOPLAS*, 16(3):843–871, 1994.
- [15] R. Hojati, R. K. Brayton, and R. P. Kurshan. Bdd-based debugging of design using language containment and fair ctl. In *CAV '93*, 1993. Springer LNCS no. 697.
- [16] D. Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.
- [17] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.
- [18] Daniel Le Berre, Anne Parrain, et al. The sat4j library, release 2.2, system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010.
- [19] Terence Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.
- [20] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, pages 337–351. Springer, 1982.
- [21] S Shoham and O Grumberg. A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. In *CAV*, pages 275–287, 2003.
- [22] S. Staber, B. Jobstmann, and R. Bloem. Diagnosis is repair. In *Intl. Workshop on Principles of Diagnosis*, June 2005.
- [23] S. Staber, B. Jobstmann, and R. Bloem. Finding and fixing faults. In *CHARME '05*, 2005. Springer LNCS no. 3725.
- [24] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.*, 89(1), 1991.
- [25] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.