Combining Abduction and Model Checking Techniques for Repair of Concurrent Programs* (Extended Abstract)

Francesco Buccafurri† **Thomas Eiter**‡ **Georg Gottlob**§ **Nicola Leone**¶,‡

Keywords: abduction, model checking, diagnosis and repair

Summary

Model checking is an approach to automated verification of finite-state concurrent systems such as circuit designs and communication protocols. In this approach, specifications are expressed in a temporal logic, and the concurrent system is modeled as a state transition graph which amounts to a Kripke structure for this logic. The most relevant advantage of model checking over other methods for verification of circuits and protocols is that it is efficient and highly automatic. Recent advances by using special data structures and algorithms, known as symbolic model checking, make it possible to verify systems with tremendously many states. On the other hand, various techniques for diagnostic reasoning on systems have been developed in the field of AI, including logic-based approaches like model-based diagnosis and repair. Our work approaches a new and promising field for application of AI techniques by proposing the enhancement of model checking by abductive reasoning, which is a major technique in AI and knowledge representation. We study the integration of model checking and AI principles by introducing the system repair problem in the context of Computational Tree Logic. The system repair problem amounts to an interesting abductive model revision problem: Determine by abductive reasoning a suitable change of the system (i.e., of its Kripke model) such that the specification is satisfied upon this change. We address to the issue of searching a program repair by developing optimization techniques which sensibly reduce the search space of solutions. Moreover, we formally extend the notion of counterexample which is a heuristically selected computation path from a conceptual counterexample tree (i.e., an evolving branching computation) that gives a hint at the failure of the system.

*This work was supported by Austrian Science Fund Project N. Z29-INF.

†DIMET, University of Reggio Calabria, loc. Feo di Vito, I-89100 Reggio Calabria, Italy. E-mail: bucca@ns.ing.unirc.it

‡Institut and Ludwig Wittgenstein Labor für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Wien, Austria. E-mail: eiter@kr.tuwien.ac.at

§Institut and Ludwig Wittgenstein Labor für Informationssysteme, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Wien, Austria. E-mail: gottlob@dbai.tuwien.ac.at

¶Dept. of Mathematics, University of Calabria, Arcavacata di Rende, I-87030, Italy. E-mail: leone@unical.it

Introduction Model checking (Clarke, Grumberg, & Peled 2000) is an approach to automated verification of finite-state concurrent systems such as circuit designs and communication protocols. In this approach, specifications are expressed in a temporal logic, and the concurrent system is modeled as a state transition graph which amounts to a Kripke structure for this logic. Checking whether the system satisfies its specification, given by a logical formula, reduces then to test whether the Kripke structure is a model of the formula.

Model checking has several important advantages over other methods for verification of circuits and protocols, like mechanical theorem provers or proof checkers. The most relevant is that it is efficient and highly automatic. Recent advances by using special data structures and algorithms, known as *symbolic model checking*, make it possible to verify systems with tremendously many states (Burch *et al.* 1992). Major companies including Intel, Motorola, Fujitsu, and AT&T have started using symbolic model checkers to verify actual circuits and protocols. Thus, (symbolic) model checking is nowadays considered as one of the most fruitful and promising applications of computational logic.

On the other hand, various techniques for diagnostic reasoning on systems have been developed in the field of AI, including logic-based approaches like model-based diagnosis and repair (Hamscher 1992). These approaches utilize general AI principles and are successfully used in different application domains. Our work approaches a new and promising field for application of AI techniques, which is in particular attractive for knowledge representation and reasoning methods, and thus adds to the application perspective of this field (Aiello & Nardi 1991). In particular, we propose the enhancement of model checking by abductive reasoning, which is a major technique in AI and knowledge representation, cf. (Poole 1988; Console, Theseider Dupré, & Torasso 1991; Konolige 1992; Eiter & Gottlob 1995; Inoue & Sakama 1995; Selman & Levesque 1996; Brewka, Dix, & Konolige 1997). The work presented is a first step towards an integration of model checking with AI techniques, and may stimulate other work in this direction.

Main contributions The main contributions of our work, which is described in more detail in (Buccafurri *et al.* 1999a), can be summarized as follows.

- We study the integration of model checking and AI principles. In particular, we introduce the *system repair problem* in the context of Computational Tree Logic (*CTL*). The latter is a propositional temporal logic which may be used to express specifications of concurrent systems modeled as Kripke structures in this logic. The system repair problem formalizes the repair of a concurrent system at the semantical level. Notice that in a different context, repair was introduced in (Friedrich, Gottlob, & Nejdl 1990; 1994; Provan & Poole 1991).

  The system repair problem amounts to an interesting *abductive model revision* problem: Determine by abductive reasoning a suitable change of the system (i.e., of its Kripke model) such that the specification is satisfied upon this change. Interestingly, this problem is an intermingled abductive reasoning and theory revision problem, which is best understood as an abductive theory revision problem. In fact, the system repair problem can be modeled as an abductive theory revision problem in the frameworks of (Lobo & Uzcátegui 1996; Inoue & Sakama 1995). Note that the close relationship between abduction and revision is well-recognized, and its investigation received increasing interest more recently, e.g., (Boutilier & Becher 1995; Inoue & Sakama 1995; Lobo & Uzcátegui 1996; 1997).

- We show that the proposed framework for system repair can be profitably used, by providing an application to the repair of concurrent programs and protocols. In particular, we describe a program repair problem, in which repair of a concurrent program in terms of changes at the syntactical level (i.e., modifications of the program code) is mapped to changes at the semantical level. As dealing with all possible modifications is clearly infeasible, we restrict in our approach to some types of modifications which seem to be relevant in practice. A *correction* is sequence $\overline{\alpha} = \alpha_1 \cdots \alpha_q$ of *single corrections* α_i on the program; it is a *repair*, if the modified program satisfies the specification φ. Applying AI principles and, in particular, Occam's principle of parsimony, we provide also a notion of minimal repair, which prunes repairs which are not free of redundancy.

- We address to the issue of searching a program repair. In general, the search space for this problem is large. It might still contain a high number of candidates, even if only elementary corrections such as inverting the value of an expression, or exchanging the name of a variable in a Boolean assignment statement are allowed.

  In order to tackle this problem, we develop optimization techniques which sensibly reduce the search space for a repair by exploiting structural information about the failure of the system provided by a counterexample. In particular, we formulate two pruning criteria referred to as *correction execution* and *correction exploitation*, which can be evaluated efficiently. In fact, given the program, a collection of candidate repairs, and a counterexample, the candidates violating these principles can be discarded in linear time. As we demonstrate, this may yield considerable savings, and thus applying correction execution and exploitation is an effective pruning method which comes at low computational cost.

- We formally extend the notion of counterexample from (Clarke, Grumberg, & Long 1994), which is a heuristically selected computation path from a conceptual counterexample tree (i.e., an evolving branching computation) that gives a hint at the failure of the system. As shown by examples, there are cases in which no single path is a counterexample. We therefore introduce the concept of a multi-path, which enables representation of nested paths. Multi-paths are a suitable tool for expressing full counterexample trees, which is needed for our purposes.

Technical development We provide the above results by the following steps.

(1) First of all, we recall the syntax and semantics of *CTL*. In addition to standard Boolean connectives, CTL has linear-time and branching-time operators. The linear-time operators are used to express statements involving the temporal order of events. The basic linear-time operators are $\mathbf{X}\varphi$ (*next time*), $\varphi\mathbf{U}\psi$ (*until*), and $\varphi\mathbf{V}\psi$ (*unless, releases*); further operators $\mathbf{G}\varphi$ (*always*) and $\mathbf{F}\varphi$ (*eventually*) are derived from them. The branching-time operators allow to take the existence of multiple possible future scenarios into account, starting from a given point of a computation. That is, the temporal order defines a tree which branches toward the future. Thus, every point of time has a unique past, but, in general, more than one future. Using the operators $\mathbf{E}\varphi$ and $\mathbf{A}\varphi$, one can express properties φ that are *possible* (true in some computation branch, i.e., evolution of the system) and *necessary* (true in every computation branch), respectively.

The semantics of *CTL* is formally defined using *Kripke structures*. Informally, a Kripke structure consists of a finite transition graph where each state is labeled with the atomic propositions which are true in it.

(2) After these preliminary steps, we introduce the notion of a *system repair problem*, which provides a general framework for the problem of properly changing a system at the semantical level in order to meet a formal specification in *CTL*. A system repair problem constitutes a kind of abductive model revision problem, which can be represented in the frameworks for abductive theory change of (Lobo & Uzcátegui 1996; Inoue & Sakama 1995).

(3) We then consider the corresponding problem at the syntactical level, in particular the one of correcting concurrent programs and protocols. The *program repair problem*, which is addressed there, resorts at the semantical level to a system repair problem.

At this point, we focus on the problem of finding repairs. In order to prune the search space, we try to exploit the information contained in counterexamples that witness the violation of the system specification.

For this purpose, we suitably generalize the notion of counterexample in (Clarke, Grumberg, & Long 1994), and show that counterexamples characterize errors. We model counterexamples by multi-paths. Informally, a *multi-path* represents an infinite tree T, by using a vertical axis rather

Process P_A

```
1:  flag1A := true;
2:  turn1B := false;
3:  if flag1B and turn1B then
4:     goto 3;
5:  x := x and y;
6:  flag1A := false;
7:  if turn1B then
8:  begin  flag2A := true;
9:         turn2B := true;
10:        if flag2B and turn2B then
11:           goto 10;
12:        y := false;
13:        flag2A := false;
    end;
14: goto 1;
```

Process P_B

```
1:  flag1B := true;
2:  turn1B := false;
3:  if flag1A and turn1B then
4:     goto 3;
5:  x := x and y;
6:  flag2B := true;
7:  turn2B := false;
8:  if flag2A and turn2B then
9:     goto 8;
10: y := not y;
11: x := x or y;
12: flag2B := false;
13: flag1B := false;
14: goto 1;
```

Table 1: A concurrent program $\mathcal{P}$ and its specification φ

than the usual recursion from a node to its descendants. The branches of T are infinite paths; the axis is a distinguished main branch of the tree, from which other paths spring off. These paths are main paths of subtrees of T. The main advantage of this concept is preservation of the nesting of paths, which is lost in the usual tree definition. Moreover, for a class of multi-paths which is sufficient for our purposes, effective finite representations exist. It is worthwhile to note that even if counterexamples are trees in general, the branching is often purely syntactical and the multi-path actually corresponds semantically to a path (in this case, we talk about *linear counterexamples*). This issue is thoroughly investigated in (Buccafurri *et al.* 1999b), where the largest class of generic formulas is characterized whose instances are guaranteed to have linear counterexamples.

(4) After the formalization of the notion of counterexample, we design the counterexample-based technique for optimize the repair search. This technique is based on the following two observations:

- Suppose we have a counterexample Π. If a repair modifies a statement not executed by Π, then such a repair can be discarded, as Π would be a counterexample in the modified system too.
- Every correction $\overline{\alpha}$ to the program code involves a set of variables, denoted by $V(\overline{\alpha})$. Only computations that evaluate some variable in $V(\overline{\alpha})$ can be influenced by $\overline{\alpha}$. Indeed, the values of variables not in $V(\overline{\alpha})$ and guards of labels in the process graphs can change only if variables are referenced whose values are affected by $\overline{\alpha}$. A path that does not evaluate variables of $V(\overline{\alpha})$ will be transformed by the correction into an equivalent path on $A \setminus V(\overline{\alpha})$; the same happens to a multi-path. Hence, a further property that any correction $\overline{\alpha}$ must satisfy for being a repair is that all counterexamples must evaluate some variable in $V(\overline{\alpha})$.

The optimization techniques using counterexamples may allow to considerably reduce the number of possible repairs that have to be considered. In particular, we formulate correction execution and correction exploitation, investigate their computational feasibility, and show the effectiveness of the techniques on an example.

Example: Monitor To give the flavor of theory and application that we developed, we discuss a motivating example. Consider the concurrent program $\mathcal{P}$ in Table 1. It consists of processes P_A and P_B, which share two common boolean variables x and y. To ensure mutual exclusion of the assignments to x and y, some control variables, *flags* and *turns*, are introduced, following the classical Peterson scheme (Peterson 1981), in which each critical section is executed obeying an *entry* and *exit protocol*. There are then two critical sections in each process, one for the assignments to x (statements 5 in P_A and statements 5–11 in P_B), and another one for the assignments to y (statements 12 in P_A and 10 in P_B, respectively); notice that in P_B, the critical section for y is nested into the critical section for y. Each variable $flagiV$ indicates the request of process V to enter critical section i, and $turniB$ tells whether such a request by process B in case of simultaneous requests should be granted. The six control variables $flagiA$ and $flagiB$, and $turniB$, for $i \in \{1,2\}$ are shared among the two processes.

$$\begin{aligned}
\varphi_{spec} &= \varphi_{flags} \wedge \varphi_{crit}, \quad \text{where} \\
\varphi_{flags} &= \textstyle\bigwedge_{i\in\{1,2\},\, V\in\{A,B\}} \mathbf{AG}(flagiV \rightarrow \mathbf{AF}\neg flagiV), \\
\varphi_{crit} &= \mathbf{AG}(\neg(b_A^{12} \wedge b_B^{10})) \wedge \mathbf{AG}(\neg(b_A^5 \wedge (b_B^5 \vee b_B^{11}))).
\end{aligned}$$

Table 2: Specification φ for program $\mathcal{P}$

The critical sections have been set up for the purpose of fulfilling some part of the system specification. The complete specification φ_{spec}, given in Table 2, prescribes that P satisfies *mutual exclusion* for assignments to x and y, respectively, and *absence of starvation*. There, each atom b_V^i means that the execution of process P_V is at line (or breakpoint) i of its program. Note that this system specification is, like in many cases, from the universal fragment *ACTL* of *CTL*, to which we focus of attention.

Formula φ_{spec} says, for example, that P_A must not exe-

cute instruction 5, if P_B executes instruction 5 or 11 at the same time. Absence of starvation requires that a request of a process for a resource (by setting a flag) must eventually be granted. Clearly, this makes sense only under the hypothesis that the underlying scheduler is fair; absence of starvation cannot be ensured if the scheduler always dispatches the same process. Fair schedules are specified through *fairness constraints* (Clarke, Grumberg, & Long 1994) on the Kripke structure, which constrain the paths (i.e., computations) to be considered.

Careful inspection of $\mathcal{P}$ shows that the program is not correct, even under fair schedules; instruction 2 of P_A should be $turn1B := true$. Even in this small example, however, detecting the error is not immediate for the non-expert. Model checking allows for checking the correctness of $\mathcal{P}$ (and of much larger programs) in a fully automatic way. If the program is incorrect, however, model checkers usually can not single out the error precisely, and are far from fixing a bug.

By using abductive reasoning, our method goes beyond error checking: it tries to locate a bug and proposes a repair for the program, such that the modified program meets the specification. Our approach considers possible errors both in the left and right hand side of an assignment as well as the interchange of two successive assignments.

Like abduction, program repair comes at computational cost. Even if we assume the case of a single error in the program and we plausibly restrict in Figure 1 attention to the assignments of control variables, we must consider 12 assignments ($1, 2, 6, 8, 9, 13$ in P_A and $1, 2, 6, 7, 12, 13$ in P_B) and 6 control variables. Thus, $(5 + 1) \cdot 5 + 2 = 77$ attempts of repair should be done, namely 5 corrections of the control variable on the left hand side and one correction of the right hand side of each statement, and 5 interchanges of neighbored assignments to control variables (1,2 and 8,9 in P_A and $1, 2$; 6,7; 12,13 in P_B). Each of these corrections requires a call of the model checker to see if it works.

Towards more efficient program repair, we have designed optimization techniques, based on counterexamples (Clarke, Grumberg, & Long 1994), for the case of a single error in the program, which is often considered in practice. By applying these techniques, our procedure makes only 17 (instead of 77) attempts in the worst case.

References

Aiello, L., and Nardi, D. 1991. Perspectives in Knowledge Representation. *Applied Artificial Intelligence* 5(1):29–44.

Boutilier, C., and Becher, V. 1995. Abduction as Belief Revision. *Artificial Intelligence* 77:43–94.

Brewka, G.; Dix, J.; and Konolige, K. 1997. *Nonmonotonic Reasoning – An Overview*. Number 73 in CSLI Lecture Notes. CSLI Publications, Stanford University.

Buccafurri, F.; Eiter, T.; Gottlob, G.; and Leone, N. 1999a. Enhancing Model Checking in Verification by AI Techniques. *Artificial Intelligence* 112(1-2):57–104.

Buccafurri, F.; Eiter, T.; Gottlob, G.; and Leone, N. 1999b. On ACTL Formulas Having Deterministic Counterexamples. Technical Report INFSYS RR-1843-99-01, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria. Revised version to appear in *Journal of Computer and Systems Sciences*.

Burch, J. R.; Clarke, E.; McMillan, K.; Dill, D.; and Hwang, L. 1992. Symbolic Model Checking: 10^{120} States and Beyond. *Information and Computation* 98(2):142–170.

Clarke, E.; Grumberg, O.; and Long, D. 1994. Verification Tools for Finite-State Concurrent Systems. In de Bakker, J.; de Roever, W.; and Rozenberg, G., eds., *A Decade of Concurrency - Reflections and Perspectives*, number 803 in LNCS, 124–175. Springer.

Clarke, E.; Grumberg, O.; and Peled, D. 2000. *Communication and Concurrency*. MIT Press.

Console, L.; Theseider Dupré, D.; and Torasso, P. 1991. On the Relationship Between Abduction and Deduction. *Journal of Logic and Computation* 1(5):661–690.

Eiter, T., and Gottlob, G. 1995. The Complexity of Logic-Based Abduction. *Journal of the ACM* 42(1):3–42.

Emerson, E. 1990. Temporal and Modal Logics. In van Leeuwen, J., ed., *Handbook of Theoretical Computer Science*, volume B. Elsevier Science Publishers B.V. (North-Holland). chapter 16.

Friedrich, G.; Gottlob, G.; and Nejdl, W. 1990. Hypothesis Classification, Abductive Diagnosis, and Therapy. In *Proceedings International Workshop on Expert Systems in Engineering*, number 462 in Lecture notes in AI, 69–78.

Friedrich, G.; Gottlob, G.; and Nejdl, W. 1994. Formalizing the Repair Process – Extended Report. *Annals of Mathematics and Artificial Intelligence* 11:187–201.

Hamscher, W., ed. 1992. *Readings in Model-Based Diagnosis*. Morgan Kaufman Publ.

Inoue, K., and Sakama, C. 1995. Abductive Framework for Nonmonotonic Theory Change. In *Proceedings IJCAI '95*, 204–210.

Konolige, K. 1992. Abduction versus closure in causal theories. *Artificial Intelligence* 53:255–272.

Lobo, J., and Uzcátegui, C. 1996. Abductive Change Operators. *Fundamenta Informaticae* 27:385–411.

Lobo, J., and Uzcátegui, C. 1997. Abductive Consequence Relations. *Artificial Intelligence* 89(1/2):149–171.

Peterson, G. 1981. Myths About the Mutual Exclusion Problem. *Information Processing Letters* 12:115–116.

Poole, D. 1988. A Logical Framework for Default Reasoning. *Artificial Intelligence* 36:27–47.

Provan, G., and Poole, D. 1991. The Utility of Consistency-Based Diagnostic Techniques. In *Proceedings Second International Conference on Principles of Knowledge Representation and Reasoning (KR-91)*, 461–472.

Selman, B., and Levesque, H. J. 1996. Support Set Selection for Abductive and Default Reasoning. *Artificial Intelligence* 82:259–272.