TLQSolver: A Temporal Logic Query Checker

Marsha Chechik and Arie Gurfinkel

Department of Computer Science, University of Toronto,
Toronto, ON M5S 3G4, Canada.
Email: {chechik,arie}@cs.toronto.edu

1 Introduction

Query checking was proposed by Chan [Cha00] to speed up design understanding by discovering properties not known *a priori*. A *temporal logic query* is an expression containing a symbol $?_x$, referred to as the *placeholder*, which may be replaced by any propositional formula[1] to yield a CTL formula, e.g. $AG?_x$, $AG(?_x \wedge p)$. A propositional formula ψ is a *solution* to a query φ in state s if substituting ψ for the placeholder in φ is a formula that holds in state s. A query φ is *positive* [Cha00] if when ψ_1 is a solution to φ and $\psi_1 \Rightarrow \psi_2$, then ψ_2 is also a solution. For example, if $p \wedge q$ is a solution to φ, then so is p. For positive queries, we seek to compute a set of strongest propositional formulas that make them true. For example, consider evaluating the query $AG?_x$, i.e., "what are the invariants of the system", on a model in Figure 1(a), ignoring the variable m. $(p \vee q) \wedge r$ is the strongest invariant: all others, e.g., $p \vee q$ or r, are implied by it. Thus, it is the solution to this query. In turn, if we are interested in finding the strongest property that holds in all states following those in which $\neg q$ holds, we form the query $AG(\neg q \Rightarrow AX?_x)$ which, for the model in Figure 1(a), evaluates to $q \wedge r$. Chan also showed [Cha00] that a query is positive iff the placeholder appears under an even number of negations.

Alternatively, a query is *negative* if a placeholder appears under an odd number of negations. For negative queries, we seek to compute a set of *weakest* propositional formulas that make them true.

In solving queries, we usually want to restrict the atomic propositions that are present in the answer. For example, we may not care about the value of r and m in the invariant computed for the model in Figure 1(a). We phrase our question as $AG(?_x\{p, q\})$, thus explicitly restricting the propositions of interest to p and q. The answer we get is $p \vee q$. Given a fixed set of n atomic propositions of interest, the query checking problem defined above can be solved by taking all 2^{2^n} propositional formulas over this set, substituting them for the placeholder, verifying the resulting temporal logic formulas, tabulating the results and then returning the strongest solution(s) [BG01]. The number n of propositions of interest provides a way to control the complexity of query checking in practice, both in terms of computation, and in terms of understanding the resulting answer.

In his paper [Cha00], Chan proposed a number of applications for query checking, mostly aimed at giving more feedback to the user during model checking, by providing a partial explanation when the property holds and diagnostic information when it does

[1] A propositional formula is a formula built only from atomic propositions and boolean operators.

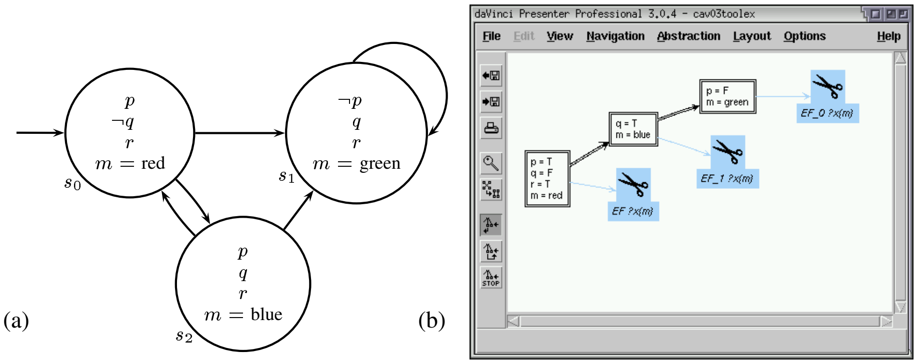

Fig. 1. (a) A simple state machine; (b) A witness to query $EF?_x\{m\}$.

not. For example, instead of checking the invariant $AG(a \vee b)$, we can evaluate the query $AG?_x\{a,b\}$. Suppose the answer is $a \wedge b$, that is, $AG(a \wedge b)$ holds in the model. We can therefore inform the user of a stronger property and explain that $a \vee b$ is invariant because $a \wedge b$ is. We can also use query checking to gather diagnostic information when a CTL formula does not hold. For example, if $AG(\text{req} \Rightarrow AF\,\text{ack})$ is false, that is, a request is not always followed by an acknowledgment, we can ask *what* can guarantee an acknowledgment: $AG(?_x \Rightarrow AF\,\text{ack})$.

In his work, Chan concentrated on *valid* queries, that is, queries that always have a single strongest solution. All of the queries we mentioned so far are valid. Chan showed that in general it is expensive to determine whether a CTL query is valid. Instead, he identified a syntactic class of CTL queries such that every formula in the class is valid. He also implemented a query-checker for this class of queries on top of the symbolic CTL model-checker SMV.

Queries may also have multiple strongest solutions. Suppose we are interested in exploring successors of the initial state of the model in Figure 1(a), again ignoring variable m. Forming a query $EX?_x$, i.e., "what holds in any of the next states, starting from the initial state s_0?", we get two incomparable solutions: $p \wedge q \wedge r$ and $\neg p \wedge q \wedge r$. Thus, we know that state s_0 has at least two successors, with different values of p in them. Furthermore, in all of the successors, $q \wedge r$ holds. Clearly, such queries might be useful for model exploration. Checking queries with multiple solutions can be done using the theoretical framework of Bruns and Godefroid [BG01]. They extend Chan's work by showing that the query checking problem with a single placeholder can be solved using an extension of alternating automata.

This paper introduces TLQSolver – a query checker that can decide positive and negative queries with a single or multiple strongest solutions. In addition, it can decide queries with multiple placeholders. TLQSolver is built on top of our existing multi-valued symbolic model-checker χChek [CDG02]. Our implementation not only allows one to compute solutions to the placeholders, but also gives *witnesses* – paths through the model that explain why solutions are as computed. We also give a few examples of use of TLQSolver, both in domains not requiring witness computation and in those that depend on it. Further uses are described in [GDC02].

2 Implementation and Running Time

The architecture of TLQSolver is given in Figure 2. TLQSolver is implemented on top of our symbolic multi-valued model-checker χChek [CDG02]. χChek is written in Java, and provides support for both model-checking with fairness and the genera-

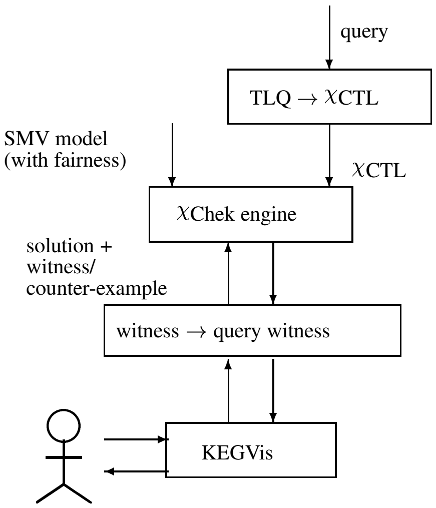

Fig. 2. Architecture of TLQSolver.

tion of counter-examples (or witnesses). It uses the *daVinci Presenter* for layout and exploration of counter-examples. TLQSolver takes a query and translates it into χCTL – a temporal logic used by χChek. χCTL is a multi-valued extension of CTL that includes multi-valued constants and operations on them. Details of this encoding are given in [GC03b]. Output of χChek is translated into a form expected for query-checking and passed to KEGVis [GC03a] – an interactive counter-example visualization and exploration tool. In addition to visualization, the user can also define a number of strategies for exploration: forward and backward, setting step granularity, choosing witnesses based on size, branching factor, etc.

The running time of solving a query φ on a Kripke structure K with the state space S is in $O(|S| \times |\varphi| \times dd)$, where dd is the complexity of computing existential quantification using decision diagrams [GC03b]. The dominating term in dd is $|SS(\varphi)|$ – the number of strongest solutions in the computation of φ. For queries about states, e.g., $AG(\neg q \Rightarrow AX?_x)$, $|SS(\varphi)|$ is in $O(|S|)$, so query checking is in the same time complexity class as model-checking. All queries we used in our applications (see Section 3 and [GDC02]) are in this category. For queries about paths, e.g., $EG?_x$, $|SS(\varphi)|$ is in $O(2^{|S|})$ [HS02], so the overall running time is in $O(|S| \times |\varphi| \times 2^{|S|})$, which is infeasible even for small problems.

To obtain a copy of TLQSolver, please send e-mail to `xchek@cs.toronto.edu`.

3 Applications

TLQSolver can be effectively used for a variety of model exploration activities. In particular, it can replace several questions to a CTL model-checker to help express reachability properties and discover system invariants and transition guards [GDC02]. For example, it can determine that the query $AG(\neg q \Rightarrow AX?_x\{q, r\})$ on the model in Figure 1(a) evaluates to $q \wedge r$. We now consider a novel application of query-checking – to guided simulation.

The easiest way to explore a model is to simulate its behavior by providing inputs and observing the system behavior through outputs. The user is presented with a list of available next states to choose from. In addition, some model-checkers, such as NuSMV, allow the user to set additional constraints, e.g., to see only those states where p holds. However, it is almost impossible to use simulation to guide the exploration towards a given objective. Any wrong choice in the inputs in the beginning of the simulation can result in the system evolving into an "uninteresting" behavior. For example, let our objective be the exploration of how the system shown in Figure 1(a) evolves into its different modes (i.e., different values of variable m). We have to *guess* which set of inputs results in the system evolving into mode *red* and then which set of inputs yields transition into mode *green*, etc. Thus, the process of exploring the system using a simulation is usually slow and error prone.

In *guided simulation* [GDC02], the user provides a set of higher-level objectives, e.g. $EF(m\text{=green})$, and then only needs to choose between the different paths through the system in cases where the objective cannot be met by a single path. Moreover, each choice is given together with the set of objectives it satisfies. Note that this process corresponds closely to that of *planning*, and its version has been realized on top of NuSMV.

Query-checking is a natural framework for implementing guided simulations. The objective is given by a query, and the witness serves as the basis for the simulation. Suppose that the goal is to illustrate how the system given in Figure 1(a) evolves into all of its modes. The following would be the interaction of the user with TLQSolver. The user then expresses the query as $EF?_x\{m\}$ and uses KEGVis [GC03a] to set his/her preference of the witness with the largest common prefix. The output produced by TLQSolver is shown in Figure 1(b). In this figure, the witness is presented in a proof-like [GC03a] style where state nodes (double lines) are labeled with proof steps that depend on them. The "scissors" symbol on a proof node means that more information is available, e.g. expanding `EF ?x{m}` tells the user that two more states, indicated by `EF_2 ?x{m}`, are required to witness the solution to the query. Since our objective was achieved by a single trace, no further user interaction is required.

References

[BG01] G. Bruns and P. Godefroid. "Temporal Logic Query-Checking". In *Proceedings of LICS'01*, pages 409–417, Boston, MA, USA, June 2001.

[CDG02] M. Chechik, B. Devereux, and A. Gurfinkel. "χChek: A Multi-Valued Model-Checker". In *Proceedings of CAV'02*, volume 2404 of *LNCS*, July 2002.

[Cha00] William Chan. "Temporal-Logic Queries". In *Proceedings of CAV'00*, volume 1855 of *LNCS*, pages 450–463, Chicago, IL, USA, July 2000. Springer.

[GC03a] A. Gurfinkel and M. Chechik. "Proof-like Counter-Examples". In *Proceedings of TACAS'03*, April 2003. To appear.

[GC03b] A. Gurfinkel and M. Chechik. "Temporal Logic Query Checking through Multi-Valued Model Checking". CSRG Technical Report 457, University of Toronto, Department of Computer Science, January 2003.

[GDC02] A. Gurfinkel, B. Devereux, and M. Chechik. "Model Exploration with Temporal Logic Query Checking". In *Proceedings of FSE'02*, November 2002.

[HS02] S. Hornus and Ph. Schnoebelen. On solving temporal logic queries. In *Proceedings of AMAST'2002*, volume 2422 of *LNCS*, pages 163–177. Springer, 2002.