MetaGame: An Animation Tool for Model-Checking Games

Markus Müller-Olm[1]* and Haiseung Yoo[2]

[1] FernUniversität in Hagen, Fachbereich Informatik, LG PI 5
Universitätsstr. 1, 58097 Hagen, Germany
mmo@ls5.informatik.uni-dortmund.de
[2] Universität Dortmund, Fachbereich Informatik, LS 5
Baroper Str. 301, 44221 Dortmund, Germany
Haiseung.Yoo@cs.uni-dortmund.de

Abstract. Failing model checking runs should be accompanied by appropriate error diagnosis information that allows the user to identify the cause of the problem. For branching time logics error diagnosis information can be given by a winning strategy in a graph game derived from the model checking instance. However, winning strategies as such are hard to grasp. In this paper we describe the MetaGame tool that computes and animates winning strategies for modal μ-calculus model checking games on finite graphs. MetaGame allows the user to play model checking games in a GUI interface thus making winning strategies more accessible.

Keywords: model checking, game, error diagnosis, branching time logic, animation

1 Introduction

Over the last two decades model checking has evolved as a useful technique that aids in the correct design of hardware and software systems. Here we are interested in checking formulas of the modal μ-calculus for small finite-state models. Such models arise, e.g., as high-level descriptions of systems as coordinated lower level components. In such scenarios, state explosion is not an issue as models typically are rather small in comparison to the models used in hardware or software model checking. Therefore, systems can be represented by explicitly given annotated graphs and global techniques can be applied.

Nowadays there is a growing awareness that model checking is most effective as an error finding technique rather than a technique for guaranteeing absolute correctness. This is partly due to the fact that specifications that can be checked automatically through model checking are necessarily partial in that they specify only certain aspects of the system behavior. Therefore, successful model checking runs while reassuring cannot guarantee full correctness. On the other hand,

* On leave from Universität Dortmund.

K. Jensen and A. Podelski (Eds.): TACAS'04, LNCS 2988, pp. 163–167, 2004.

careful investigation of the cause for failing of model checking runs may allow the user to identify errors in the system. Thus, model checkers are more and more conceived as elaborate debugging tools that complement traditional testing techniques.

For model checkers being useful as debugging tools, it is important that failing model checking attempts are accompanied by appropriate *error diagnosis information* that explains why the model check has failed. Model checkers can fail *spuriously*, i.e., although the property does not hold for the investigated abstraction it may still be valid for the real system. In order to be useful it should be easy for the user to rule out spurious failures and to locate the errors in the system from the provided error diagnosis information. Therefore, it is important that error diagnosis information is easily accessible by the user.

For linear-time logics error diagnosis information is conceptually of a simple type: it is given by an (eventually cyclic) execution path of the system that violates the given property. Thus, linear-time model checkers like SPIN [3] compute and output such an *error trace* in case model checking fails. The situation is more complex for branching-time logics like the modal μ-calculus. Such logics do not just specify properties of single program executions but properties of the execution tree. Hence, meaningful error diagnosis information for branching-time logic model checking cannot be represented by linear executions, in general.

Stirling [4, 5] developed a characterization of μ-calculus model checking as a *two player graph game* with a *Rabin chain winning condition* [6]. It is well-known that such games are *determined* (i.e., one of the players has a winning strategy) and that the winning player always has a *memory-less winning strategy*. Memoryless strategies can be presented as sub-graphs of the game graph.

In the game constructed from a model checking instance, Player II has a winning strategy if and only if model checking fails. Thus, we can use a winning strategy of Player II as error diagnosis information. Conversely, Player I has a winning strategy in the constructed game if and only if model checking succeeds. Thus, a winning strategy of Player I can be seen as justification for a successful model check. Thus, both successful and failing model checking runs give rise to the same type of justifying information, a nice symmetry.

However, it is not easy to interpret winning strategies as such. Therefore, we propose to *animate* winning strategies. The idea is that the user is put into the position of the losing player and plays games against the system. The system plays according to the computed winning strategy. Obviously, this implies that the user will lose all games. By this, the user increases his knowledge about the system behavior and hopefully understands the model checking result better. By the above mentioned symmetry, this idea is applicable for error diagnosis (user=Player I, system=Player II) as well as for understanding successful model checking results (user=Player II, system=Player I).

The MetaGame tool realizes this idea. It is integrated into the MetaFrame environment [2] and relies on its basic infrastructure and graph manipulation capabilities. MetaGame extends the MetaFrame environment with strategy synthesis and a GUI-based animation of μ-calculus model-checking games. A number

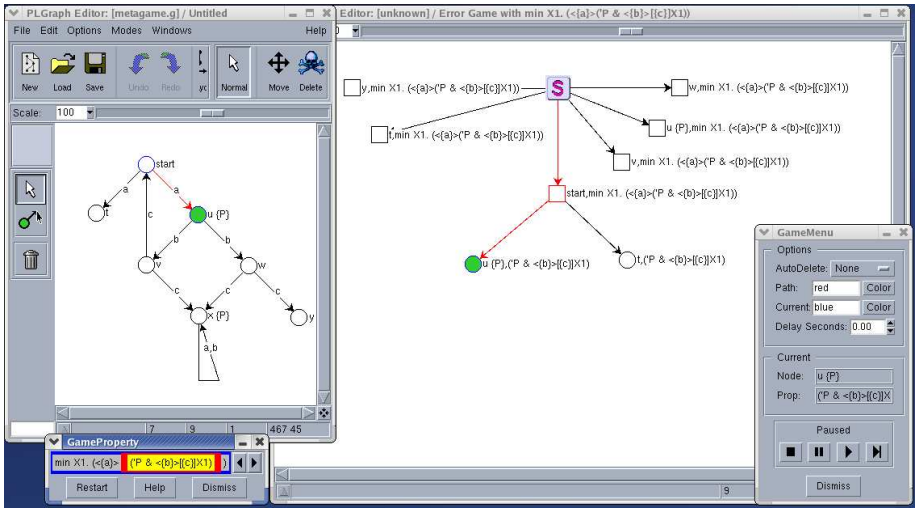

Fig. 1. A screenshot showing the main windows of MetaGame.

of features are intended to allow a more informative and more easily accessible animation. In the next section we show an example run of MetaGame on a small illustrative example and discuss the main features.

2 Playing Games with MetaGame

As usual, system models are given by finite graphs the edges of which are labeled by actions and the nodes of which are labeled by sets of atomic propositions. System models can be created and manipulated with MetaFrame's PLGraph editor or loaded from a file. The top left window in Fig. 1 shows an example system model. The logic supported by MetaGame is a variant of the modal μ-calculus. The formula to be checked onto the system model can be typed into a text field or loaded from a file. The bottom left window in Fig. 1 shows an example formula. In standard μ-calculus syntax this formula reads $\mu X_1.(\langle a\rangle(P \wedge \langle b\rangle[c]X_1))$.

After loading or creating a system and a formula, MetaGame constructs the corresponding game graph and computes the winning regions of Player I and II and their respective winning strategies. This is done simultaneously for all positions of the game graph by a strategy synthesis algorithm that achieves essentially the same asymptotic complexity as counter-based global μ-calculus model checkers [1].

After computing the winning strategies, MetaGame offers the user to play error diagnosis games; i.e., the user is put into the position of Player I. The primary view of the played game is a designated window that shows the explored game positions in a tree-like fashion together with a "Game Menu" window that offers options for proceeding with building this tree. The big window in the right

part of Fig. 1, for instance, shows a situation in an error diagnosis game for the example graph and formula in which the user has decided to play from the state `start` and proceed with the game position $(u, P \wedge \langle b \rangle [c] X_1)$.

Each game position is a pair (s, ϕ) consisting of a state s in the system model and a sub-formula ϕ of the model-checked formula. Intuitively, Player I (the user) tries to justify that ϕ holds for s while Player II (the system) tries to refute it. Accordingly, Player I plays from positions in which the outermost operator of ϕ is of a disjunctive nature (i.e., "$\vee$" or "$\langle A \rangle$") and Player II from positions in which the outermost operator is a conjunctive operator (i.e., "$\wedge$" or "$[A]$"). Positions of Player I are shown as squares and positions of Player II as circles. Fixpoint formulas $\sigma X.\phi$ (where $\sigma \in \{\mathrm{Min}, \mathrm{Max}\}$) are identified with their unfolding $\phi[\sigma X.\phi / X]$. Player II wins the game if (1) the game reaches a position of the form (s, P) (or $(s, \neg P)$) where state s does not satisfy atomic proposition P (or satisfies P, respectively); (2) the game reaches a position (s, ϕ) in which it is Player I's turn, but Player I has no move; or (3) the game becomes cyclic (i.e., a position (s, ϕ) is revisited in a game) and the outermost fixpoint operator in the cycle is a minimal fixpoint. The winning conditions for Player I are dual.

In the game window, the user can choose his next move by selecting a position in the game position tree with the mouse. After clicking the "Play" button in the Game Menu window, the system proceeds with the game according to Player II's winning strategy as far as possible. Afterwards it asks the user for a next move or, if a winning situation for Player II has been reached, it informs the user about the reason for winning. By clicking the "Fast Forward" button the user can also instruct the system to choose some next move arbitrarily.

A number of features lead to a more informative and accessible animation.

1. As shown in Fig. 1, the *projection* of the selected game position onto the state and the formula component are shown in the system model and in the formula window by coloring the corresponding state and sub-formula, respectively. This allow the user to match the game position with the model state and the sub-formula much more easily.
2. The user can *backtrack* in a play and *multiple plays* can be in progress simultaneously. This is achieved by allowing the user to select his next move at an arbitrary place in the position tree.
3. The user can *prune* the position tree, by cutting off explored pieces he is no longer interested in ("Stop"-button). In addition, various *AutoDelete* options allow the user to automatically prune the position tree according to the place of the chosen next move.
4. The user can *introduce delays* into the animation of Player I's strategy and interrupt the animation with the "Pause" button.

Fig. 2 shows a final situation for the example graph in which the user has lost all three possible plays. Besides error diagnosis games, MetaGame also allows the user to play games that explain successful model checking runs.

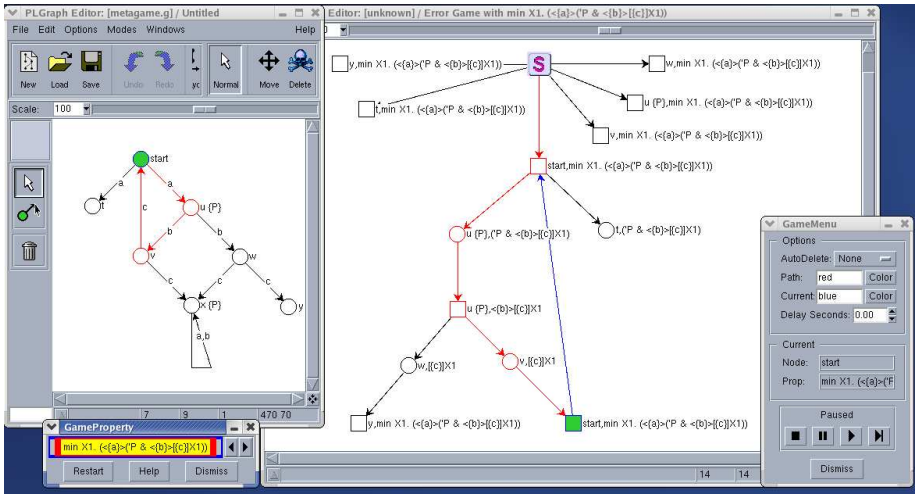

Fig. 2. A final situation.

3 Conclusion

We have described the MetaGame tool that allows the user to play model checking games in a GUI interface. As state explosion is not an issue in the intended application scenarios, we can apply global strategy synthesis and compute and store strategies for the whole game graph completely. This allows us to animate model checking games without noticeable delays and to offer the flexibility to backtrack and to have multiple plays in progress simultaneously. We consider this important factors for a wider acceptance of the idea of playing games as a means for understanding model checking results of branching time logics.

References

1. R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification (CAV'92)*, volume 663 of *Lecture Notes in Computer Science*, pages 410–422. Springer-Verlag, June/July 1992.
2. Metaframe homepage. `http://ls5-www.cs.uni-dortmund.de/projects/METAFrame/`.
3. Spin homepage. `http://spinroot.com/spin/whatispin.html`.
4. C. Stirling. Local model checking games. In S. A. Smolka, editor, *Proc. 6th Intern. Conf. on Concurrency Theory (CONCUR'95)*, volume 962 of *Lecture Notes in Computer Science*, pages 1–11. Springer-Verlag, 1995.
5. C. Stirling and P. Stevens. Practical model-checking using games. In *TACAS 1998*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101, 1998.
6. W. Thomas. On the synthesis of strategies in infinite games. In E. Mayr and C. Puech, editors, *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science, STACS '95*, volume 900 of *Lecture Notes in Computer Science*, pages 1–13. Springer-Verlag, 1995.