Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Model Checking with the jABC Framework*

From METAGame to GEAR

Marco Bakera, Tiziana Margaria, Clemens D. Renner, Bernhard Steffen

[1] Chair of Service and Software Engineering, University of Potsdam, D-14482 Potsdam, Germany,
e-mail: `margaria@cs.uni-potsdam.de`

[2] Chair of Programming Systems, TU Dortmund, D-44227 Dortmund, Germany,
e-mail: {`marco.bakera,clemens.renner,steffen`}`@cs.tu-dortmund.de`

The date of receipt and acceptance will be inserted by the editor

Abstract. This paper describes significant improvements of GEAR, our tool for game-based model checking, that concern its extensibility, the ability to inspect the strategies in parity games and clearly laid out game graphs, as well as a new approach we call reverse checking. The tool is well documented (featuring extensive help and animated online tutorials) and supports both novice and advanced users. This work is part of our contribution within the SHADOWS project, where we provide a number of enabling technologies for model-driven self-healing at design time.

1 Introduction

GEAR, our game-based model checker for the full modal μ-calculus and derived, more user-oriented logics, is a user-friendly tool that can offer automatic proofs of critical properties of complex system models [3]. In the application area of embedded systems/autonomous systems, the technique and the tool have been successfully applied to a case study from an autonomous aerospace context. The benefits of the approach are illustrated on a case study that concerns the design of the task-level control part of the processes of the ExoMars Rover [12], which was designed as part of a European Space Agency (ESA) project. We called our approach *property-driven functional healing*, and we showed that game-based model-checking can be a powerful technique for design-time healing via verification, diagnosis, and adaptation of temporal properties. As shown in [2] designers and engineers can interactively investigate automatically generated winning strategies for the games, this way exploring the connection between the property, the system, and the proof. Playing against undesired behavior is a very effective way of discovering insufficiencies or imprecisions in otherwise seemingly correct behavioral models. This work is part of our contribution within the SHADOWS project, where we provide a number of enabling technologies for model-driven self-healing.

In this paper, we discuss from a practical point of view the tool features that make it adequate for adoption by field engineers. Background and detailed exposition of the game based model checking technique are available online at [1] and in [2].

In the following, in Sect. 2 we recall the purpose of game based model checking and summarize the development history that led to GEAR, in Sect. 3 we illustrate how a user can profit of GEAR in the two main modes of use: for unexperienced users or users just interested in a pass/fail answer when checking properties of the system under consideration, as well as in the advanced mode. In Sect. 4 we recall the principles of parity games, and how they are implemented in GEAR. Games disclose detailed information on how to pinpoint very precisely what goes wrong and how to repair it. In Sect. 5 we compare GEAR with UPPAAL TIGA [4], another successful game based tool. Finally, in Sect. 6 we draw our conclusions.

2 Related Work

Techniques for formal verification of software have grown in importance over the last few years. 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 considered as coordinated systems of lower-level components. In such scenarios, state-explosion is not an issue, as models typi-

* This work has been partially supported by the European Union Specific Targeted Research Project *SHADOWS* (IST-2006-35157), exploring a *S*elf-*H*ealing *A*pproach to *D*esigning c*O*mplex soft*W*are *S*ystems. The project's web page is at `https://sysrun.haifa.ibm.com/shadows`.

cally 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 behaviour. Therefore, successful model checking runs, while reassuring, cannot guarantee full correctness.

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

For model checkers to be 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 may in fact also 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 for model checking to be useful, it should therefore be easy for the user to rule out spurious failures and to locate the errors in the system based on 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 a (possibly cyclic) execution path of the system that violates the given property. Thus, in case model-checking fails, linear-time model checkers like SPIN [8] compute an output in form of an *error trace*. The situation is more complex for branching-time logics like CTL or the modal μ-calculus. Such logics do not just specify properties of single program executions but properties of the entire execution tree, comprising the local of decision points. Hence, meaningful error diagnosis information for branching-time logic model checking cannot be represented by linear executions in general. This is where games help.

Stirling [23, 24] developed a characterization of μ-calculus model checking as a *two-player graph game* with a *Rabin chain winning condition* [25]. It is well-known that such games are *determined*: one of the players has a winning strategy and this strategy is *memory-less*, i.e. does not depend on the history of the play.

In the game constructed from a model checking instance, the disproving player (sometimes called *player II* or the *AND player*) has a winning strategy if and only if model checking fails. Thus, we can use the disprover's winning strategy as error diagnostics. Conversely, the proving player (also *player I* or *OR player*) has a winning strategy in the constructed game if and only if model checking succeeds. Therefore, a winning strategy for the prover can be seen as a justification for a successful model check. Hence, 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. As Müller-Olm and Yoo [19] suggested and presented, an interactive approach that involves the user's participation in the parity game can be used to animate the error diagnosis information. In the METAGame tool, the user and the system (i.e. the model checker) are adversaries. If a state in the model does not satisfy the property in question, the user plays the role of the prover. Since the property is not satisfied by that state, the user will inevitably lose the game, but while (hopefully) gaining knowledge derived from that particular game run.

We have taken the METAGame tool as a basis to further improve its usability, visualization, and extensibility. METAGame's successor GEAR is built on top of the jABC Framework [22]. The jABC is our environment for model-driven development of high assurance systems. It can be tailored to domain-specific needs using plugins, one of them being the GEAR model checker itself. The jABC Framework enables users to graphically develop structural and behavioral (process-oriented) models of systems. Automatic code generation for various platforms, among them the LEGO Mindstorms robotic controller [10], is available as well. As a result, systems modelled or entirely developed within the jABC are amenable to being animated, validated, verified, and executed in an integrated fashion, within the same platform. This way, the verification and debugging using the GEAR model checker moves very close to the actual system being developed, yielding more precise or more timely results than traditional model checking on manually derived abstract models of the system under development. The combination of these technologies (graphical modeling, verification on the actual system model, and code generation for embedded systems) makes the jABC Framework a suitable platform for the GEAR model checker.

Of course, GEAR can also be used independently of jABC. In fact, in the SHADOWS project we are going to offer GEAR as a remote service library, that can be accessed by other tools without any need of resorting to any specific framework.

In the following we are going to illustrate how to use GEAR in the most typical settings.

3 Using the Tool

The models to be checked in GEAR are graphs in which both vertices and edges are annotated: While nodes hold atomic propositions (i.e. basic properties), the edges are labeled to support selectivity for the modal operators of the μ-calculus (box and diamond). Such models can be easily created and edited using the jABC framework. In the terminology of this framework, the nodes are

SIBs (Service Independent Building Blocks) and represent modular units of behaviour (be they components, services, classes, procedures, or uninterpreted symbols) while the edges are called branches and describe the possible (normal or exceptional) continuations after a SIB's execution.

Given a jABC model, the SIBs need to be equipped with atomic propositions for the model checker. This is done using the AP-Manager inspector (shown in Fig. 1, upper right corner), that manages the atomic propositions.

Of course it is also possible to import models produced independently from the jABC (this is e.g. the case in SHADOWS), and it is also possible to import annotations.

3.1 Simple View

Once the plugin is started, one can notice the inspector tab shown in the bottom left corner of Fig. 1. This is a Simple View of the formally specified properties (or, to be more exact: *descriptions* of these properties) for the currently active graph. This view is intended for users that are not familiar with temporal logics or those who want to benefit from model checking without the required logical background. The different background colors indicate the status of the property: red means that the model does not satisfy the property, green indicates that the property is satisfied and yellow points to a syntax error. This way it is very simple to keep the verification aligned with the development.

3.2 Advanced View

Clicking the scholar's hat at the bottom of this inspector brings up the Advanced View (upper left corner of Fig. 1). The Advanced View allows users to develop formulas (both CTL and modal μ-calculus are supported) which are then associated with the model[1].

Once a formula is parsed, its detailed structure is shown as a syntax tree below the entry field. GEAR is not restricted to checking the validity of the whole formula: users can inspect the sets of satisfying nodes for each subformula in the tree by simply selecting the desired subformula. On the other hand, selecting a subset of nodes in the model will highlight in green those subformulas that are satisfied by these nodes. Thus, the user may get a first hint at why certain nodes do not satisfy certain formulas. We call this mode of use *reverse checking*.

Every formula that has been checked is added to the list of associated formulas for the respective graph. To make use of the Simple View we mentioned earlier, descriptions should be provided for all formal properties; this is done in the window shown in the bottom right corner of Fig. 1. It is also possible to transfer properties (along with their descriptions) between graphs.

In the Advanced View, after checking (universal) safety properties, *error paths* (i.e. counter-examples) might help users to identify the problem's source for a non-satisfying node.

Although the tool's primary usage may be with the framework, it can be used in a stand-alone version, as a command-line tool where graph (and formula) are simply provided as files. Additionally, the tool is not restricted to the so-called SIB graphs but can build graphs from e.g. a very simple CSV-based model description[2]. It is also open to new model specifications which can be provided via a Java implementation of a corresponding interface.

Implementation and usage aspects aside, documentation has been a primary focus during the development of the tool. Extensive online documentation along with Flash-based tutorials is available from the tool's website.[3] The tutorials show animated, step-by-step example runs introducing all key features.

4 Parity Games

Game graphs can be constructed as a cross-product of formula and model, where each game graph node is a tuple of the form (s, ϕ) where s is a node of the model and ϕ is a subformula. A game graph is created in GEAR by simply clicking the joystick button in the advanced view (top left corner of Fig. 1). The generated graph contains annotations regarding the computed strategies (for both players) and is laid out in a matrix with the rows describing the subformulas and the columns representing nodes from the model (Fig. 2).

For understanding a property violation, the user plays the game in the role of the prover. Thus, he decides where to go in the game graph at disjunctive nodes ($\vee$ and the diamond operator) while the system (acting as the disprover) moves at conjunctive nodes ($\wedge$ and the box operator). In our example (as shown in Fig. 2), conjunctive nodes are identified by circular icons while disjunctive nodes have rectangular icons.

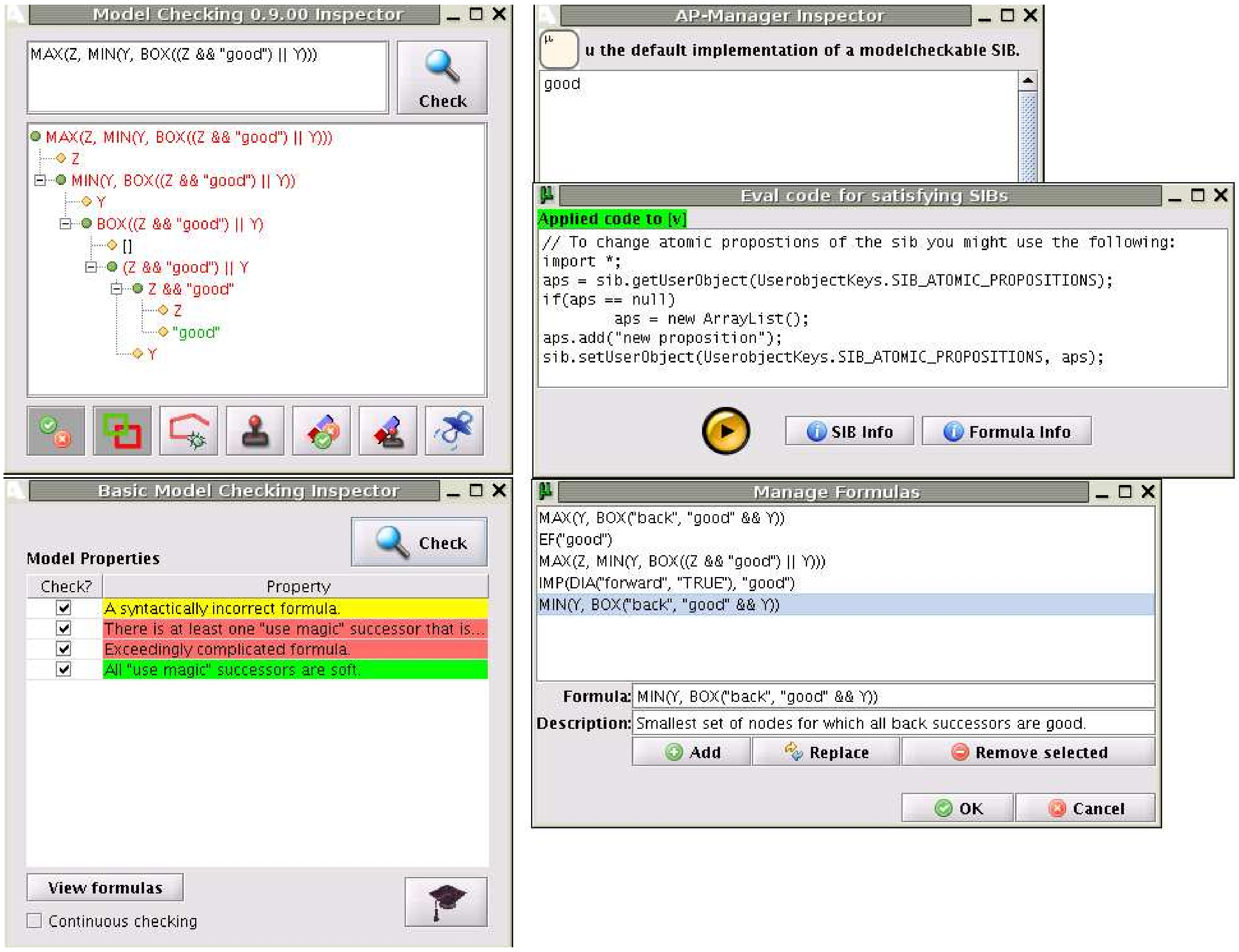
The prover wins when one of the following conditions is met (dual conditions hold for the disprover):

1. A $\wedge$-sink is reached in the game graph (especially the *true* sink). In this situation, the $\wedge$-player cannot choose any next node, thus the other player wins.
2. The game becomes cyclic – i.e. a node is reached twice and the outermost operator of this cycle is a maximal fixed point.

[1] Using the FormulaBuilder plugin, formulas from other formal languages can be created (e.g. quantifiers, regular expressions, simple arithmetic, Specification Patterns) [11, 7].

[2] CSV Specification: `http://rfc.net/rfc4180.html`

[3] GEAR Homepage: `http://jabc.cs.uni-dortmund.de/gear`



```
MAX(Z, MIN(Y, BOX((Z && "good") || Y)))
```

```
MAX(Z, MIN(Y, BOX((Z && "good") || Y)))
  Z
  MIN(Y, BOX((Z && "good") || Y))
    Y
    BOX((Z && "good") || Y)
      []
      (Z && "good") || Y
        Z && "good"
          Z
          "good"
        Y
```

u the default implementation of a modelcheckable SIB.

good

Applied code to [v]

```
// To change atomic propostions of the sib you might use the following:
import *;
aps = sib.getUserObject(UserobjectKeys.SIB_ATOMIC_PROPOSITIONS);
if(aps == null)
        aps = new ArrayList();
aps.add("new proposition");
sib.setUserObject(UserobjectKeys.SIB_ATOMIC_PROPOSITIONS, aps);
```

Model Properties

Check?	Property
✓	A syntactically incorrect formula.
✓	There is at least one "use magic" successor that is...
✓	Exceedingly complicated formula.
✓	All "use magic" successors are soft.

☐ Continuous checking

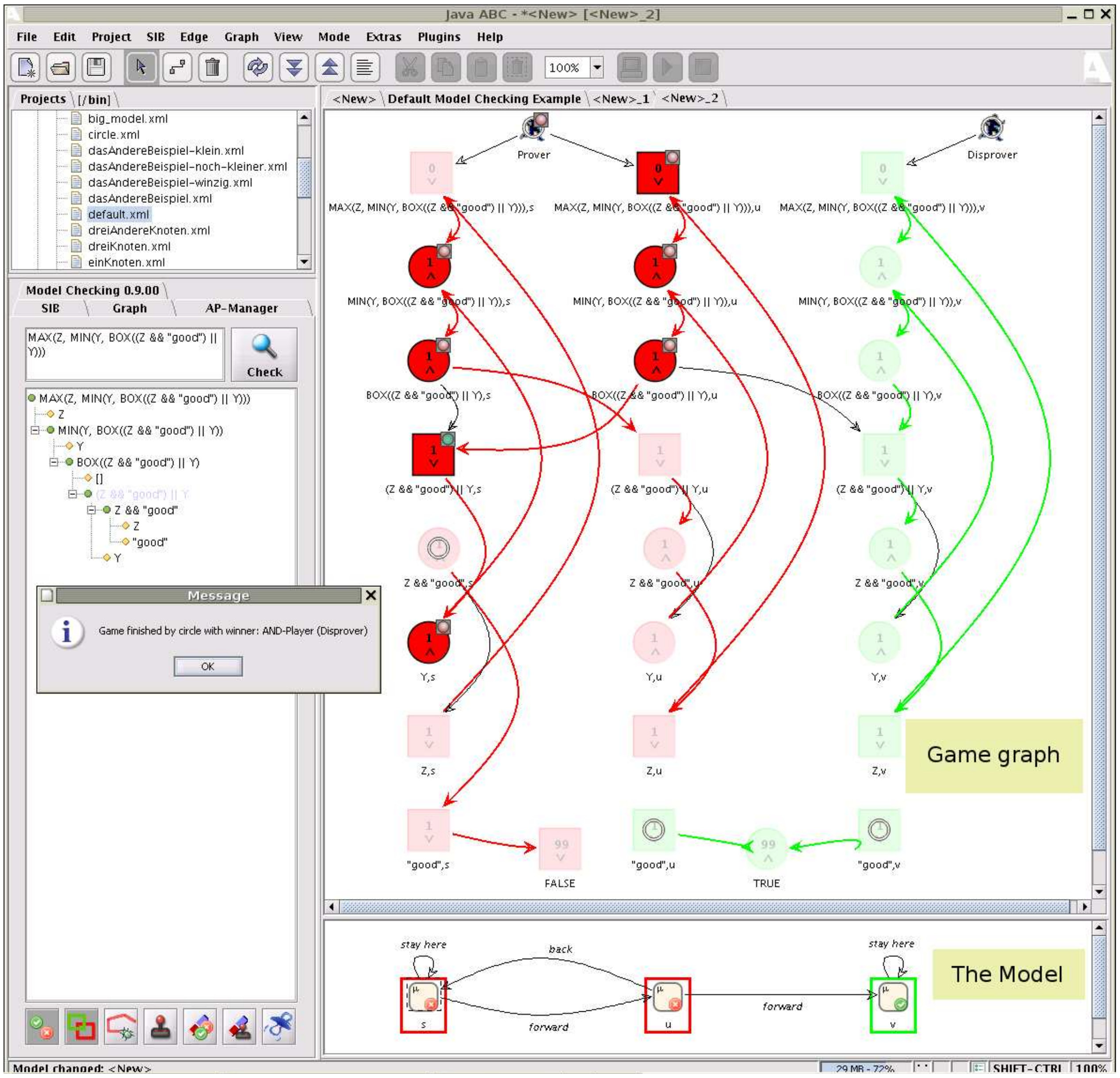
```
MAX(Y, BOX("back", "good" && Y))
EF("good")
MAX(Z, MIN(Y, BOX((Z && "good") || Y)))
IMP(DIA("forward", "TRUE"), "good")
MIN(Y, BOX("back", "good" && Y))
```

Formula: MIN(Y, BOX("back", "good" && Y))

Description: Smallest set of nodes for which all back successors are good.

Fig. 1. The Simple and Advanced user interfaces and facilities to manage propositions and formulas, as well as execute arbitrary code.

[t]


Java ABC - *<New> [<New>_2]
File Edit Project SIB Edge Graph View Mode Extras Plugins Help
100%
Projects [/bin]
big_model.xml
circle.xml
dasAndereBeispiel-klein.xml
dasAndereBeispiel-noch-kleiner.xml
dasAndereBeispiel-winzig.xml
dasAndereBeispiel.xml
default.xml
dreiAndereKnoten.xml
dreiKnoten.xml
einKnoten.xml
Model Checking 0.9.00
SIB Graph AP-Manager
MAX(Z, MIN(Y, BOX((Z && "good") || Y)))
Check
MAX(Z, MIN(Y, BOX((Z && "good") || Y)))
Z
MIN(Y, BOX((Z && "good") || Y))
Y
BOX((Z && "good") || Y)
[]
(Z && "good") || Y
Z && "good"
Z
"good"
Y
<New> Default Model Checking Example <New>_1 <New>_2
Prover
Disprover
MAX(Z, MIN(Y, BOX((Z && "good") || Y))),s
MAX(Z, MIN(Y, BOX((Z && "good") || Y))),u
MAX(Z, MIN(Y, BOX((Z && "good") || Y))),v
MIN(Y, BOX((Z && "good") || Y)),s
MIN(Y, BOX((Z && "good") || Y)),u
MIN(Y, BOX((Z && "good") || Y)),v
BOX((Z && "good") || Y),s
BOX((Z && "good") || Y),u
BOX((Z && "good") || Y),v
(Z && "good") || Y,s
(Z && "good") || Y,u
(Z && "good") || Y,v
Z && "good",s
Z && "good",u
Z && "good",v
Y,s
Y,u
Y,v
Z,s
Z,u
Z,v
"good",s
"good",u
"good",v
FALSE
TRUE
Game graph
stay here
back
stay here
forward
forward
s
u
v
The Model
Model changed: <New>
29 MB - 72%
SHIFT-CTRL
100%

Message

Game finished by circle with winner: AND-Player (Disprover)

OK

Fig. 2. A sample game in the game graph.

In the annotated game graph, nodes contain also a number, which is the node's priority, i.e. the fixed point alternating depth. In the visualization, nodes are colored according to the player: green nodes are in the domain of the prover's winning strategy (which is simply a partial function $f : (s, \phi) \mapsto (s', \phi')$) while the red nodes belong to the disprover.

As long as there is only one edge leaving a node, this step is taken automatically. The same applies when it is not the user's turn to pick a successor. If no game has been played yet, the nodes are dimly colored. Once the game starts and players make their moves, the nodes reached in this run are vividly highlighted.

During the game, if a game graph node (s, ϕ) is highlighted, the corresponding model node s is highlighted too, as well as the subformula ϕ in the formula tree. This way a visual clue relates the elements of the three representations as a means of orientation for the user.

While not visible to the user, the winning strategies derived from parity games are used to compute all model-checking results as described in [19].

5 Comparison with TIGA

TIGA (TImed Game Automata, introduced by [5]) is an extension of UPPAAL [4] that allows for solving games based on timed game automata. Edges of such automata are equipped with clocks and guards as usual timed automata, and they are further partitioned into controllable and uncontrollable edges. Uncontrollable edges represent the system's environment. The winning condition of such a game is specified by a property defined in a CTL-like extension of the UPPAAL query language over a set of so-called *goal* and *lose* states given by the modeler of the automaton. For instance the property

```
control: A[ not(lose) U win]
```

asks for a controllable execution of the automaton that must reach a win state while avoiding lose states.

Unfortunately TIGA does not support convenient exploration and play on the game graph. The strategies are given as textual output whose value for a user decreases with growing size. The specification language used by TIGA (a specialized version of CTL) does not allow users to express more complex (e.g. nested) properties, that however naturally arise when dealing with the interplay of control and environment in self-adaptive systems. This lack of expressiveness in the properties arises from the complex nature of the examined models of timed automata compared to finite state systems. Furthermore, TIGA is mainly controlled by a command-line tool and needs (beside syntactic information of the specification language) deep insight into its many command-line arguments. Finally, there is no support for collaboration between specifying and implementing groups of the development team.

6 Conclusions

We have presented a tool for game-based model checking that supports a number of views in order to give unique feedback when checking branching time formulas: winning strategies, highlighted sets of satisfied formulas or satisfying states, and interactive game playing support summarize the available knowledge in a user-oriented fashion, in order to provide a better intuition of the models and the formulas, and to help locating errors. All this comes in two modes, specifically addressing either novice or advanced users.

The tool accepts a number of model specification languages and logics, and supports the easy graphical definition of properties using on-the fly defined graphical macros by means of the FormulaBuilder, which can also be used to integrate new logics. As part of the jABC framework, it can easily be combined with other tools, but it is also available as a command line-based stand alone version.

We believe that the different views and the ability to thoroughly inspect the parity game on an adequately highlighted game graph make this tool a valuable addition to the investigation techniques currently in use in the design of high-assurance software and systems.

References

1.
2. M. Bakera, T. Margaria, C. D. Renner, and B. Steffen. Verification, Diagnosis and Adaptation: Tool-supported Enhancement of the Model-driven Verification Process. In *ISoLA'07 Workshop on Formal Methods in Avionics, Space and Transport, Poitiers (F)*, pages 85–98, ISBN 2854288148. In Revue des Nouvelles Techno-logies de l'Information (RNTI-SM-1), Dec. 2007.
3. M. Bakera, T. Margaria, C. D. Renner, and B. Steffen. Property-Driven Functional Healing: Playing Against Undesired Behavior. In *CONQUEST 2007, 10th Int. Conf. on Quality Engineering in Software Technology*, Sept. 2007.
4. G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *SFM*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–236. Springer, 2004.
5. F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient On-the-Fly Algorithms for the Analysis of Timed Games. In Martin Abadi and Luca de Alfaro, editors, *CONCUR 2005 – Concurrency Theory (16th CONCUR'05)*, volume 3653 of *Lecture Notes in Computer Science (LNCS)*, pages 66–80. Springer-Verlag (New York), San Francisco, CA, USA, August 2005.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
7. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Property Specification Patterns for Finite-state Verification. In *Proceedings of the Second Workshop on Formal Methods in Software Practice*, pages 7–15, March 1998.
8. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.
9. H. Hungar, T. Margaria, and B. Steffen. Test-Based Model Generation For Legacy Systems. In *Proc. 2003 Int. Test Conf. ITC 2003, Breaking Test Interface Bottlenecks*, pages 971–980, 2003.
10. S. Jörges, C. Kubczak, F. Pageau, and T. Margaria. Model Driven Design of Reliable Robot Control Programs Using the jABC. In *Proc. 4th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe) '07*, pages 137–148, Baltimore, Maryland, USA, 2006. IEEE Computer Society.
11. S. Jörges, T. Margaria, and B. Steffen. FormulaBuilder: A tool for graph-based modelling and generation of formulae. In *Proc. ICSE*, pages 815–818. ACM, 2006.
12. K. Kapellos. MUROCO-II: Formal Robotic Mission Inspection and Debugging. In *Technical report, European Space Agency*, 2005.
13. M. Karusseit and T. Margaria. Feature-based Modelling of a Complex, Online-Reconfigurable Decision Support Service. In *First Int. Workshop on Automated Specification and Verification of Web Site WWV 2005*, pages 9–25, 2005.
14. D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
15. A. Lamprecht, T. Margaria, and B. Steffen. Data-Flow Analysis as Model Checking within the jABC. In *15th Int. Conf. on Compile Construction (CC)*, pages 101–104, 2006.

16. T. Margaria and B. Steffen. Lightweight Coarse-Grained Coordination: A Scalable System-Level Approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):107–123, March 2004.
17. T. Margaria, B. Steffen, and M. Reitenspieß. Service-Oriented Design: The Roots. In *Third Int. Conf. on Service-Oriented Computing*, pages 450–464, 2005.
18. Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. Remote integration and coordination of verification tools in jETI. In *12th IEEE Int. Conf. on the Engineering of Computer-Based Systems (ECBS)*, pages 431–436. IEEE Computer Society, 2005.
19. M. Müller-Olm and H. Yoo. MetaGame: An Animation Tool for Model-Checking Games. In *TACAS 04, LNCS 2988*, pages 163–167. Springer-Verlag, 2004.
20. J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings of the Fifth International Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, New York, 1982. Springer.
21. B. Steffen and T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In *Correct System Design, Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 390–415, 1999.
22. B. Steffen, T. Margaria, R. Nagel, S. Jörges, and Christian Kubczak. Model-driven development with the jABC. In *Proc. 2nd Haifa Verification Conference*, Haifa, Israel, 2006. Springer.
23. C. Stirling. Practical model-checking using games. In *CONCUR 95, LNCS 962*, pages 1–11. Springer-Verlag, 1995.
24. C. Stirling and P. Stevens. Practical model-checking using games. In *TACAS 98, LNCS 1384*, pages 85–101. Springer-Verlag, 1998.
25. W. Thomas. On the synthesis of strategies in infinite games. In *STACS 95, LNCS 900*, pages 1–13. Springer-Verlag, 1995.