Property-driven functional healing: Playing against undesired behavior[1]

Marco Bakera[1]

Tiziana Margaria[1]

Clemens D. Renner[1]

Bernhard Steffen[2]

[1] Chair Service and Software Engineering,
Institute of Informatics, University of Potsdam,
August-Bebel-Straße 89, 14482 Potsdam, Germany.
{bakera,margaria,renner}@cs.uni-potsdam.de

[2] Chair Programming Systems,
Department of Computer Science, University of Dortmund,
Otto-Hahn-Straße 14, 44227 Dortmund, Germany.
steffen@cs.uni-dortmund.de

Abstract

In this paper, we show how to use GEAR, a game-based model checker, for property-driven functional healing of high-assurance systems. Designers and engineers can interactively investigate the winning strategies resulting from the games. These reveal in-depth information about the connection between the property, the system, and the proof, both as explanation in case of a successful proof, and as detailed, fine-granular error diagnostics in the case of failure. This results in an interactive use of the tool where debugging and redesign are carried out by playing against undesired behavior.

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 [Kap05], which was designed as part of a European Space Agency (ESA) project.

[1] This work has been partially supported by the European Union Specific Targeted Research Project SHADOWS (IST-2006-35157), exploring a Self-Healing Approach to Designing cOmplex softWare Systems. The project's web page is at https://sysrun.haifa.ibm.com/shadows.

1 Motivation

Game-based model checking is an established field of logics and theoretical computer science. Lange and Stirling [Sti95, LS00] have focused on the application of games to branching-time temporal logics, such as CTL. Stirling also introduced the notion of a verifier and a refuter, the game players we call prover and disprover.

The strategies that provide the background for the rationale used in our work stem from the work of Müller-Olm and Yoo [MOY04] while first steps in the direction of game-based μ-calculus model checking were already taken by Emerson et al. [EJS93].

There have also been approaches to the synthesis of game strategies by Vöge [Voe00]. Here, we are mainly interested in the application of these techniques to provide fine granular diagnostic capabilities that enhance modeling and development environments for high-assurance systems.

In this paper, we show how to use GEAR, a game-based model checker for the full modal μ-calculus, and more user-oriented derived logics, as a user-friendly tool that can offer automatic proofs of critical properties of such systems. Designers and engineers can interactively investigate the winning strategies resulting from the games. These reveal in-depth information about the connection between the property, the system, and the proof, both as explanation in case of a successful proof, and as detailed, fine-granular error diagnostics in the case of failure.

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 [Kap05], which was designed as part of a European Space Agency (ESA) project.

Before explaining our modeling and verification approach, we briefly present the concrete case study.

2 Case Study: The ExoMars Rover

ESA's FORMID Project (FOrmal Robotic Mission Inspection and Debugging) aimed at creating a development environment for the verification and analysis of robotic missions [GJK04]. In the concrete mission example [Kap05], a robot (the ExoMars Rover) is sent on a surface mission on Mars where it has to accomplish several tasks, including the acquisition of subsurface soil samples using a drill. In this case study, the mission is organized in a hierarchical way, which accounts for partial autonomy of the rover.

Mission plans are designed and enforced by the ground control, while finer grained operational decisions, at the *Task* level, are completely autonomous: the rover has its own planning capabilities, enabling it to transform a task assignment into a suitable, executable sequence of *Actions* in a context-dependent and error-aware way.

The functional reference model of the control architecture was realized in FORMID, internally using Esterel [BG92] as a high-level specification language. FORMID allows the specification of tasks and actions and of properties to be checked, the discrete event

simulation with visual debugging of the scenarios, and to generate the code which is then uploaded to the robot controller.

The kind of formal verification considered in the ESA study concerns predefined patterns of safety, liveness, and conflict-freedom properties. Then, an *observer* module for each property is generated, that spies the system model to detect violations. In that case, a violating scenario is returned to the designer.

In this paper we illustrate the power of our game-based model checker GEAR on a central property pattern for remote/autonomous (space) systems, with the intuitive meaning that it is not always avoidable that such systems can run into situations from which they cannot recover, even with the ground support from Earth.

In the following, we first show how we model the ExoMars Rover surface mission in the jABC [SMN+06], our service-oriented modeling and verification environment, according to the description provided in [Kap05]. We then show how GEAR helps in identifying the culprit of a property violation.

3 Behavioral service-oriented models of the ExoMars Rover

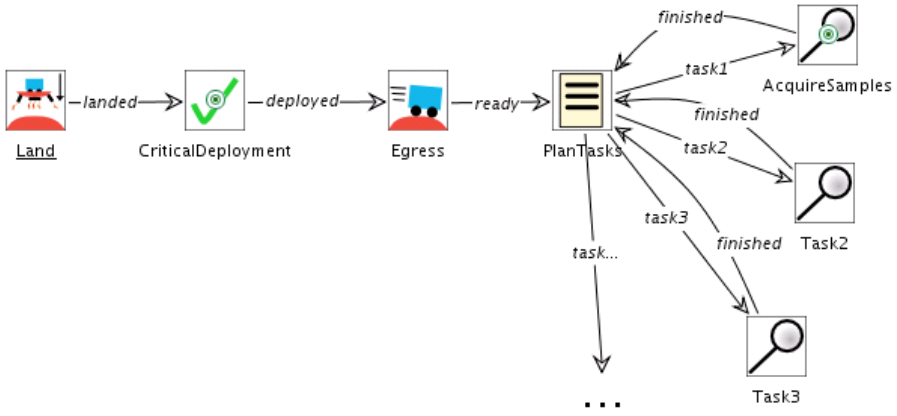

Fig. 1: Service Logic Graph of the global ExoMars Rover mission workflow

The ExoMars Rover's mission is to explore the Martian surface and in doing so to collect interesting soil samples which are acquired using a drill. Problems may occur, for example when the drill gets blocked and the Rover is no longer able to act according to its agenda.

3.1 Mission workflow

Fig. 1 shows the high-level behavior of the Mars Rover (on *Mission* level), consisting of the actual landing on the planetary surface, followed by critical deployment procedures

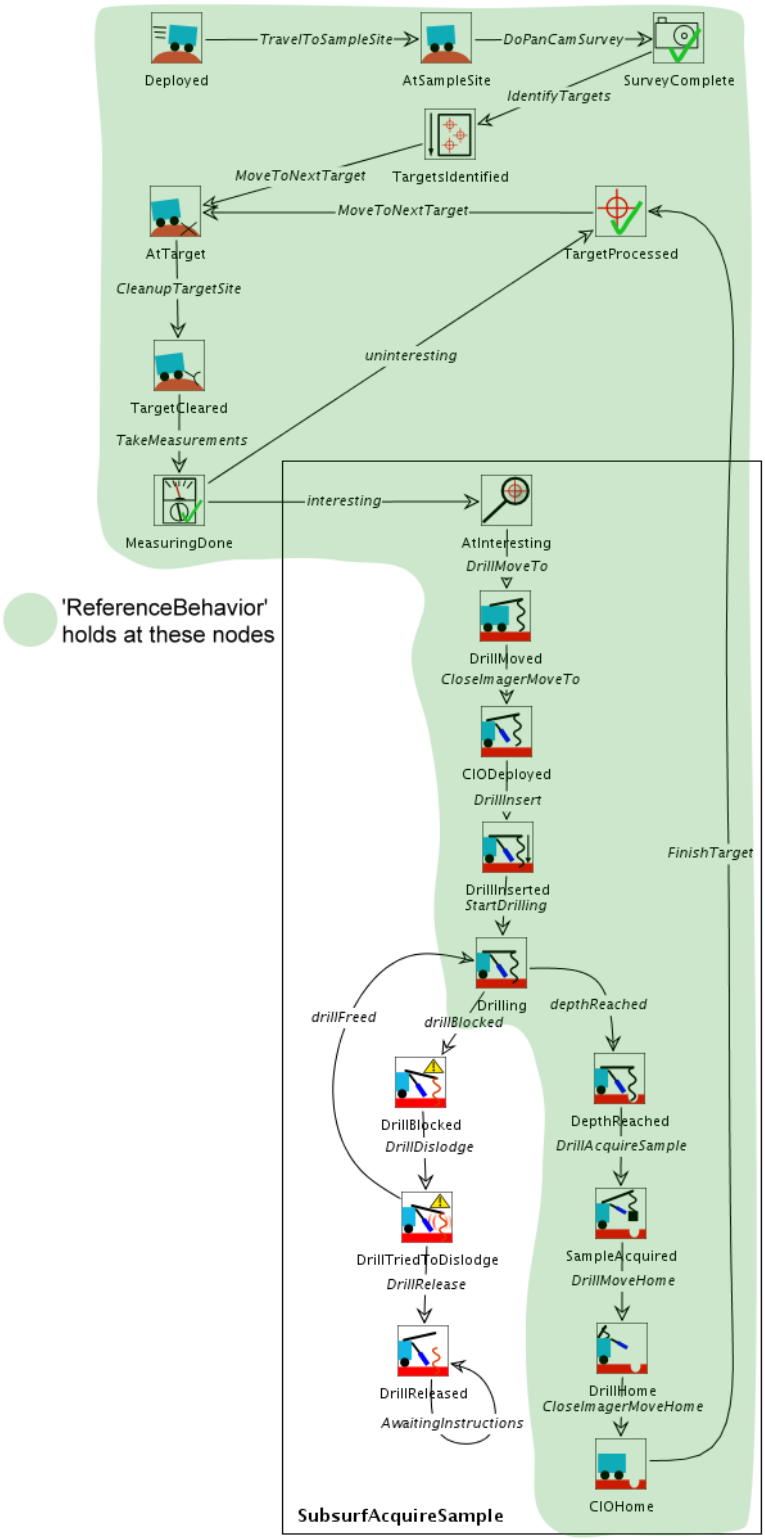

Fig. 2: The acquisition of subsurface samples (AcquireSamples)

(such as the set-up of solar power acquisition and radio communications with ground support). Subsequently, the Rover egresses from its landing pad and plans its future operations. The two activities *CriticalDeployment* and *AcquireSamples* represent submodels containing the actual workflow for deployment and sample acquisition, respectively.

The submodel for *CriticalDeployment* shall be omitted here for brevity.

The acquisition of subsurface soil samples is only one of the possible tasks the Rover may pursue. While the tasks itself are pre-defined for the mission, the actual execution order is determined autonomously by the Rover itself.

3.2 Sub-surface sample acquisition (AcquireSamples)

The submodel that is used to refine the *AcquireSamples* task from Fig. 1 above is shown in Fig. 2. The Rover first travels to the site where potential sample extraction targets are located. Using its panorama-camera, the Rover now compiles a list of specific extraction point candidates which are then evaluated sequentially. Preliminary measurements are evaluated to assess the chance of interesting findings for each target. If the target is considered interesting, the Rover then prepares its drill and close image observer camera (which monitors the entire drilling process) and attempts to drill until the desired depth has been reached and the sample can be extracted (right branch in the box in Fig. 2). Before the next target location is examined, both drill and close image observer are retracted into safe positions.

However, the drill might get stuck during the acquisition of a sample (left branch in the box in Fig. 2). In this case, the Rover attempts to free the drill using various dislodge maneuvers and resumes drilling if the dislodging succeeded. If all recovery fails, the drill has to be released to keep the Rover functional and allow it to pursue other predefined tasks based on new instructions from ground control on Earth.

The states in the Acquire Samples model are tagged with so-called atomic propositions (used e.g. in Kripke structures cf. [MSS99]). These are basic, local properties, such as “ReferenceBehavior” for the states in the green (darker) area in Fig. 2. Moreover, every state has at least its name attached as an atomic proposition.

4 Playing against undesired behavior

In the following, we analyze behavioral aspects of the system introduced in the previous sections. These behavioral aspects are best described and formalized by the use of temporal logics [MSS99] like CTL or LTL which are both theoretically well-understood and backed by strong tool support. In our case we use CTL.

Space systems have a basis functioning mode, called in this case *reference behavior*, which constitutes the normal activity mode of the spacecraft or of the device. Additional modes can be activated when opportune, for special operations or for exceptional behavior, e.g. in case of emergency, but it is of central importance to be able to return to the reference behavior once the operations or emergency terminate.

A central property is therefore the following:

"However the system evolves in the future, it is always possible to re-establish the reference behavior."

This property is concisely expressed by the CTL formula

$$\phi = \mathrm{AG}[\mathrm{EF}\ \mathrm{ReferenceBehavior}]$$

For the model in Fig. 2, ϕ is not satisfied due to the fact that the Rover may lose its Drill under certain circumstances (when the dislodging fails).

Proving or disproving behavioral properties can be understood as a game that is played by two players – one that tries to prove the property correct for some particular process model (Prover), and one that tries to refute it (Refuter, Disprover). The interaction between the model and the property is revealed by playing the game.

Whenever a property has to be verified for a particular state of the system model it is crucial whether the property requires to *achieve a desired* or to *avoid undesired* system behavior. The former case is called a *liveness property* while the latter is called a *safety property*.

Properties that emerge from real-world applications rarely appear as pure liveness or safety properties but rather they combine these two aspects. The considered property is mixed: it requires system evolutions to eventually reach reference behavior – the liveness aspect. On the other hand, the system shall not evolve in ways such that it is not able to re-establish the reference behavior – the safety aspect.

Both this mixture of a) safety and liveness aspects in specifications and b) the system evolution competing against or supporting the specification constitute the game-based approach. We can imagine the system as being the player that tries to refute the property – by intuition the system holds the property if no system evolution is able to disprove it. The system developer can be imagined as the player that tries to satisfy the property – by intuition, he tries to develop the system to the specification needs.

The consistency match between a temporal specification and a behavioral model is called the model checking problem. By using the game-based model checking approach one can think of the model checking problem as being a game played by the system developer against all possible system behaviors.

This change of the point of view in the context of verifying process models allows for detailed in-depth analyses of faults in the system under consideration simply by exploring the system model with respect to a behavioral limitation specified as a temporal formula.

The actual game is visualized on a game graph such as the one shown in Fig. 3. It consists of two types of nodes:

- *Diamond-shaped nodes* belong to the developer who is convinced that the property holds for the system. At these nodes, the developer decides how to proceed in the game.

- *Box-shaped nodes* belong to the system which demonstrates the problematic behavior that is responsible for the violation of the property. This is where the system decides how to proceed.

In addition, *black nodes* indicate that the system will succeed in showing a violation of a property whereas *white nodes* indicate that the developer's assumption is validated.

The game graph in Fig. 3 is laid out in a way similar to the model in Fig. 2. Those nodes which belong together because they describe the same state in the system model are grouped in boxes named after the corresponding system state. The game graph's nodes inside these boxes reflect the liveness and safety aspects of the investigated property.

This game view not only provides diagnostic information but additionally gives hints about how to re-establish consistency between model and specification.

5 Verification and games

In this section, we illustrate Game-based Model Checking on the model of the ExoMars Rover and a property whose violation needs diagnostic means beyond the usual error paths. We now investigate the problematic behavior as it emerges from the game played "against the system".

Recall that the desired system execution is left once the Rover's drill gets blocked. Therefore, the nodes *DrillBlocked*, *DrillTriedToDislodge* and *DrillReleased* do not constitute reference behavior. In the game graph, this is represented by the fact that the nodes named Ref (which is short for the desired reference behavior) are black.

However, we are interested in the more complex property ϕ. By following the red path in Fig. 3, we can observe that the system succeeds in demonstrating the problematic behavior by moving down to the nodes in the *DrillReleased* box. Having arrived there, the system chooses a circle from which the developer is unable to break out. Moreover, the developer is persistently kept from moving into the white nodes because of the system's choices at nodes (a) and (b).

Due to the developer being forced into the black nodes in the *DrillReleased* box, it becomes apparent that a change in the property ϕ is needed. As the release of the drill cannot be prevented, there is essentially one natural way for adapting the property specification:

$$\mathsf{AG[EF[Ref} \vee \mathsf{DrillReleased]]}$$

which tolerates this unavoidable situation, but maintains the original intent otherwise.

It turns out that the revised property holds for the system. Even better, investigating the corresponding game graph it becomes apparent that the following stronger property is valid:

$$\mathsf{AG[AF[Ref} \vee \mathsf{DrillReleased]]}$$

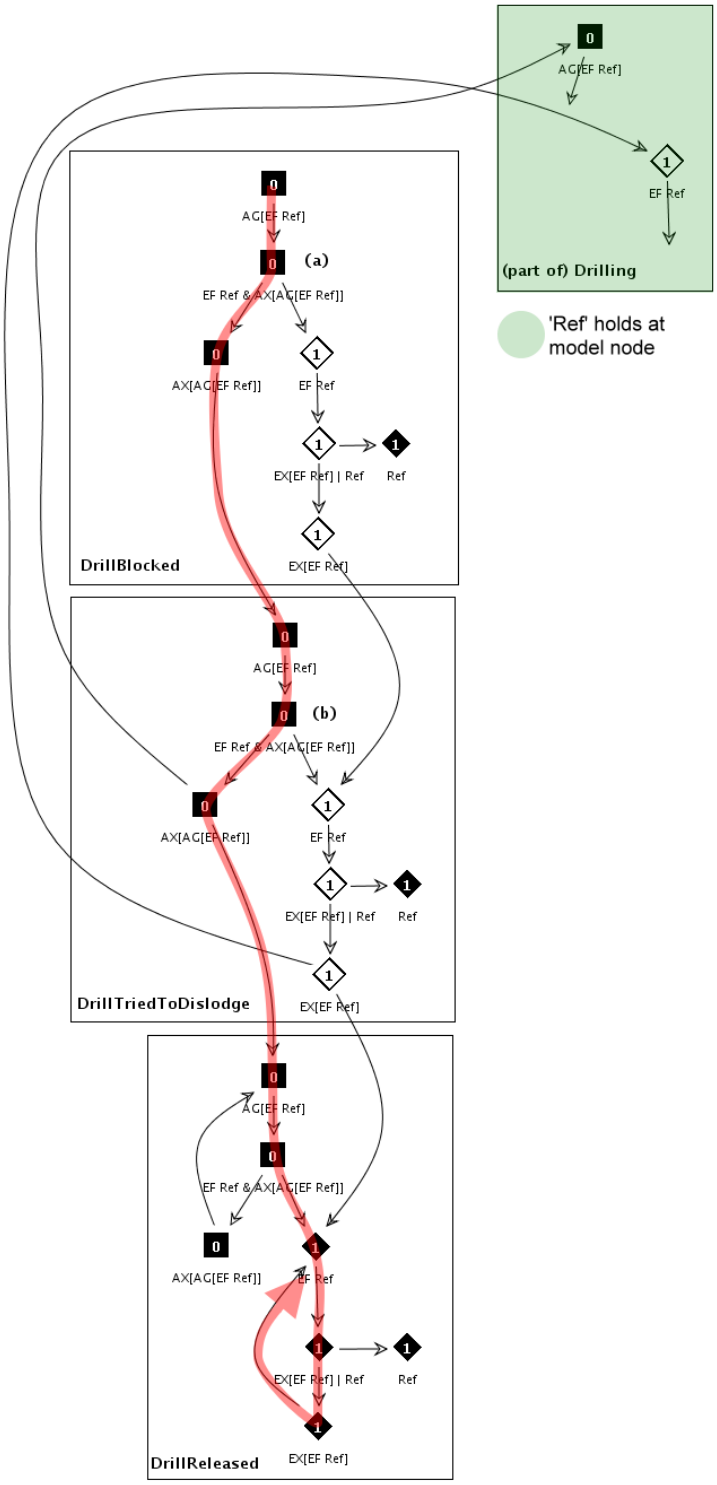

Fig. 3: A part of the game graph that reveals the problematic behavior

This proves that the reference behavior will inevitably be reached, unless the drill is released.

6 Conclusion

We have illustrated the power of GEAR on a central property pattern for remote/autonomous (space) systems which is branching-time in nature, and cannot be expressed in linear-time logics. Thus, it requires diagnostic means beyond classical error paths. The power of the diagnostic information on the basis of winning strategies has been exploited in order to 'repair' the considered property.

Our case study concerned a task-level model of a Mars robot. Models and questions become much more interesting and realistic, as soon as the robot's task-level autonomy controller is included in the modelling.

References

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. Science of Computer Programming, 19:87-152, 1992.

[Cur03] S.A. Curtis and colleagues. Ants for the human exploration and development of space. In Proc. IEEE Aerospace Conf., 2003.

[EJS93] E. A. Emerson, C. S. Jutla, and A. P. Sistla. On model-checking for fragments of μ-calculus. In Proc. CAV. 1993.

[GEAR] GEAR – a game-based model checking tool. http://jabc.de/gear.

[GJK04] G. Bormann, L. Joudrier, and K. Kapellos. FORMID: A formal specification and verification environment for DREAMS. In Proc. 8th ESA Workshop on Adv. Space Techn. for Robotics and Automation ASTRA, 2004.

[HS07] M. G. Hinchey and R. Sterritt. 99% (biological) inspiration..., In Proc. 4th IEEE Int. Workshop on Engineering of Autonomic and Autonomous Systems, 2007.

[IBM01] IBM. Autonomic computing: IBM's perspective on the state of information technology. Technical report, IBM, 2001.

[Kap05] K. Kapellos. MUROCO-II: FOrmal Robotic Mission Inspection and Debugging. Technical report, European Space Agency, 2005.

[KC03] J. O. Kephart and D. M. Chess. The vision of autonomic computing. Technical report, IBM, 2003.

[Koz82] D. Kozen. Results on the propositional μ-calculus. In 9th Colloquium ICALP, Springer ,1982.

[LS00] M. Lange and C. Stirling. Model checking games for CTL. In Proc. Int. Conf. on Temporal Logic, ICTL 2000.

[MOY04] M. Müller-Olm and H. Yoo. Metagame: An animation tool for model-checking games. In Proc. TACAS, Springer, 2004.

[MS04] T. Margaria and B. Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. STTT, 5(2-3), 2004.

[MSS99] M. Müller-Olm, D. A. Schmidt and B. Steffen. Model-Checking: A Tutorial Introduction. In Proc. SAS, 1999.

[SMN+06] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak. Model-driven development with the jABC. In Proc. 2nd Haifa Verification Conference, Springer, 2006.

[Sti95] C. Stirling. Local model checking games. Lecture Notes in Computer Science, 962, 1995.

[Voe00] J. Voege. Strategiesynthese für Paritätsspiele auf endlichen Graphen. PhD thesis, RWTH Aachen, 2000.