Verification, Diagnosis and Adaptation: Tool-supported enhancement of the model-driven verification process

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

*University of Potsdam, D-14482 Potsdam, Germany,
{bakera, margaria, renner}@cs.uni-potsdam.de,
http://www.cs.uni-potsdam.de/sse
**University of Dortmund, D-44227 Dortmund, Germany,
bernhard.steffen@cs.uni-dortmund.de,
http://ls5-www.cs.uni-dortmund.de

Abstract. In this paper, we use a case study from an autonomous aerospace context as running example to show how to apply a game-based model-checking approach a as a powerful technique for the verification, diagnosis and adaptation of temporal properties. This work is part of our contribution within the SHADOWS project, where we provide a number of enabling technologies for model-driven self-healing. We propose here to use GEAR, a game-based model checker for the full modal μ-calculus and derived, more user-oriented logics, as a user friendly tool that can offer automatic proofs of critical properties of such systems. 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.[1]

1 Introduction

Software self-healing is an emerging approach to address the problem of fixing large, complex software systems. Self-healing solutions presented to date commonly address a single class of problems, or they are not applicable in fielded systems. To address the need for industry-grade software self-healing, the SHADOWS EU project focuses on self-healing of complex systems, extending the state-of-art in several ways (Shehory et al., 2007). It introduces innovative technologies to enable self-healing of classes of problems not solved elsewhere. It additionally integrates several self-healing technologies into a common solution. It further adopts a model-based approach, where models of desired software behavior direct the self-healing process. These allow for lifecycle support of self-healing applicable to industrial systems.

We contribute to SHADOWS a number of enabling technologies for model-driven self-healing. In this paper, we use a case study from an autonomous aerospace context as running

[1]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.

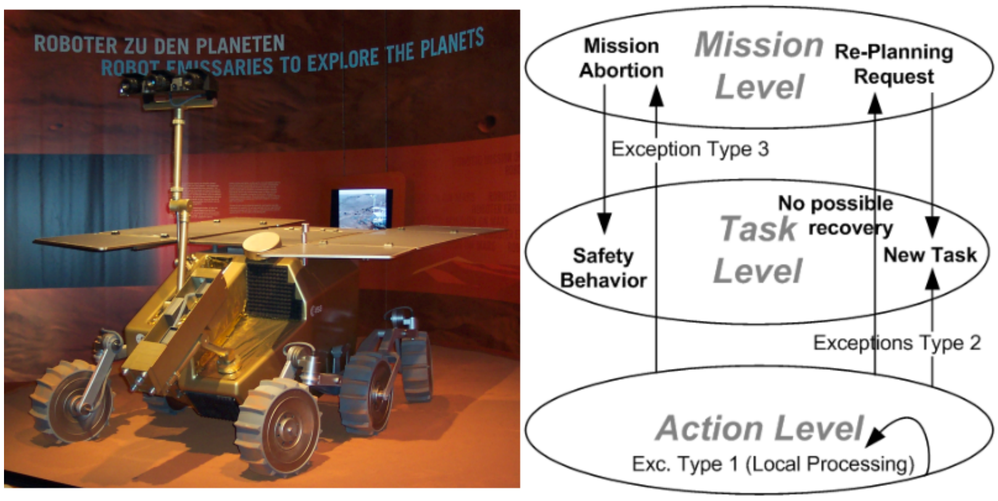


FIG. 1 – *The ExoMars Rover and the three-tier control model.*

example to show how to apply a game-based model-checking approach as a powerful technique for the verification, diagnosis and adaptation of temporal properties. In particular, we show how to model the several abstraction levels of the robot's behavior in a uniform and formal, but intuitive way in terms of processes in the jABC framework, a mature model driven service-oriented process definition platform. Subsequently, we leverage the formality of these models to prove properties by model checking, and in particular we exploit the interactive character of game based model checking to show how to discover an error, then localize, diagnose, and correct it.

In the following, we first show how we model the ExoMars Rover surface mission in the jABC, our service-oriented modelling and verification environment, according to the description provided in (Kapellos, 2005). Subsequently we discuss in depth the technique of game based model checking using the properties already considered in the ESA case study.

2 ExoMars Rover Case Study

The ExoMars Rover case study originates from ESA's FORMID Project (*FO*rmal *R*obotic *M*ission *I*nspection and *D*ebugging), that aimed at creating a development environment for the verification and analysis of robotic missions (Bormann et al., 2004). In the concrete mission example described in (Kapellos, 2005), 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. Fig. 1 shows a model of the Rover as presented on an aerospace exhibition.

As customary, the mission is organized in a hierarchical three-tier control model shown in Fig. 1, 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, which allows it to transform a task assignment into a suitable executable sequence of actions in a context dependent and error-aware way.

The case study presented in this paper relies on a system description created with FORMID (Kapellos, 2005) using Esterel as a high-level specification language (Berry and Gonthier, 1992). FORMID allows the specification of tasks and actions as well as of properties to be checked. It provides discrete event simulation with visual debugging of the scenarios and generation of code to be 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. An *observer* module for each property is generated that spies on the system model to detect violations. In that case, a violating scenario is returned to the designer. However, this kind of diagnosis is only present at run-time or simulation-time respectively.

3 Hierarchical Modelling of the ExoMars Behavior

Starting from the behavior's descriptions in (Kapellos, 2005), we model all levels of the three-tier hierarchy with the jABC (Steffen et al., 2006). We support both

1. *model engineers*, who know the desired behavior of the system. Here, we provide a modelling environment that supports modelling the mission, task and action level behavior;

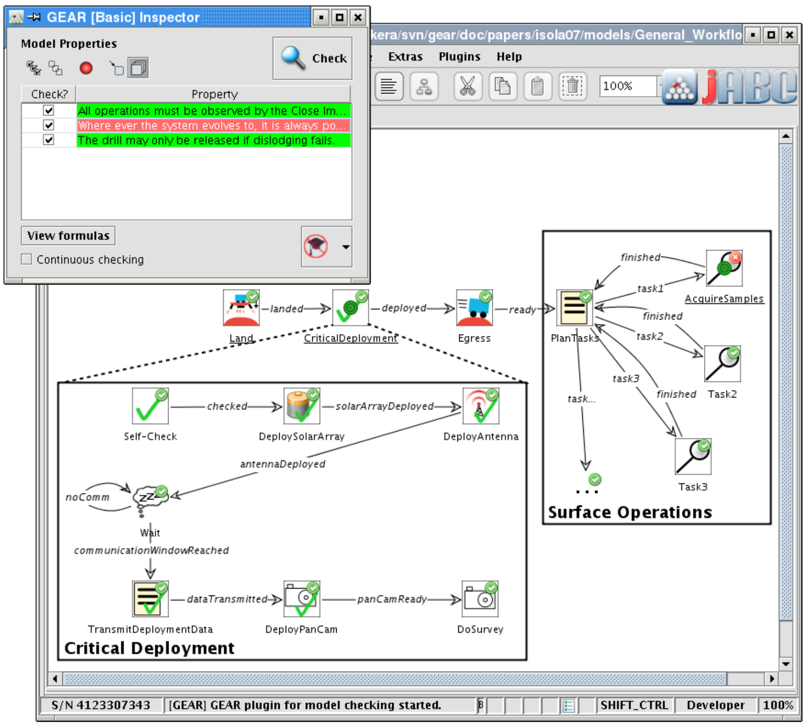
2. *dependability and validation engineers* who know the desired properties of the system. We help them here to express them in terms of logical properties, automatically checkable on the behavioral models.

Building models on the basis of a given action library is a graphical activity: The upper left corner of Fig. 4 shows references to the sub-models of the three level hierarchy. Dragging one of them into the canvas opens it for inspection and manipulation. Currently the Action level model of the sample acquisition is opened in the main canvas.

3.1 Mission level

The overall mission of the ExoMars Rover is to explore the Martian surface and to collect interesting soil samples which are acquired using a drill. Fig. 2 sketches this high-level behavioral model as it appears in the jABC (Steffen et al., 2006), our environment for model driven, service-oriented system design. There, we have modelled the whole Mission as a top-level service, consisting of the sequence of tasks Land, CriticalDeployment, and Egress followed by a choice of operational tasks, like AcquireSamples, the task described later in detail. Technically, the jABC way of modelling the behavior matches very closely the intentions of the ExoMars designers: in the original description style, typical of (autonomous) three-tier controlled systems (see Fig. 1), elementary Actions constitute re-usable, basic building blocks of behavior. They are organized in libraries and composable into Tasks, structured as flow graphs of Actions. Mission plans are then in turn composed of Tasks in a similar fashion, leading to a hierarchical model structure.

As shown in Fig. 2, the Rover's surface mission consists of the three main phases *Critical Deployment*, *Egress* and *Surface Operations*, arranged in a sequence of actions.



Check?	Property
✓	All operations must be observed by the Close Im...
✓	Where ever the system evolves to, it is always po...
✓	The drill may only be released if dislodging fails.

S/N 4123307343 | [GEAR] GEAR plugin for model checking started. | SHIFT_CTRL | Developer | 100%

FIG. 2 – *Property-controlled modelling at different abstraction levels.*

- in the *Critical Deployment* phase the operational set-up is established (solar power acquisition, communication);
- in the *Egress* phase the Rover leaves the lander dock to start surface operations;
- in the actual *Surface Operations* phase the rover travels to the next sampling location, performs required measurements and operations, and transmits the relevant data to Earth.

In the *Surface Operations* phase, the rover repeatedly chooses a task to accomplish, plans for achievement, and executes it. We focus here on the sample extraction task, (AcquireSamples), but the other alternative tasks are modelled analogously.

3.2 Task level: Critical Deployment and Egress

The task-level model of the Critical Deployment task is shown in Fig. 2. Once the ExoMars Rover has landed, it performs an initial Self-Check. If the Self-Check is passed, it proceeds to the deployment of both the solar arrays and the antenna, necessary to communicate with ground support on Earth. If planetary constellations impede communication with ground support the rover shuts down and waits until a favorable time window occurs. Once communication has been established, the rover transmits data collected during the landing phase (Entry, Descent and Landing – EDL). Subsequently, a camera is deployed to survey the surrounding landscape and to give hints while planning the pending egress task.

3.3 Task and Action level: Sample Acquisition

Fig. 3 shows a detailed model of the Acquire Samples surface operations task from Fig. 2. The green colored part (gray in the b/w version) concerns the ReferenceBehavior. This part of the model is of central interest because it represents the intended and normal behavior of the model. During sample acquisition the Rover examines one by one sampling locations of interest previously defined by ground support. It autonomously travels to the next interesting working area and performs a panoramic investigation of the site. Based on the results, specific targets for subsurface sample acquisition are identified and further investigated. This is accomplished by first cleaning up the target area and performing some measurements which – if sufficiently promising – justify the actual extraction of a sample.

The extraction process itself is shown in the SubsurfAcquireSample box at the lower right of Fig. 3. The failure recovery mechanism shown in the lower left branch will be relevant for the subsequent property verification phase. Recovery occurs upon exceptional behavior during a sample extraction if the Drill was disturbed. This is captured in the left branch of the box in Fig. 3. We will focus on this exceptional behavior in the following. However, note that exceptional behavior in other Tasks or Actions can be handled similarly.

4 How to ensure behavioral properties

As evident from Fig. 1, exception handling is of vital importance. There are three types of problems and exceptional behavior at the action level. Type 1 exceptions are locally handled by the action itself. Type 3 exceptions lead to mission abortion through safety behavior – depending on the context. Type 2 exceptions are either forwarded to the task level, where they can be handled by creating a new task to circumvent the exception, or are redirected to the mission level if no task-level recovery is known, and are dealt with by replanning the rest of the mission.

Interesting for the autonomous behavior at task level are type 2 exceptions. A central property pattern for autonomous systems depending on the action level is:

Where ever the system evolves to, it is always possible to recover.

To ensure compliance to this high-level requirement, we need to

1. Formalize the desired concretization of this *property* for the application domain (here, the ExoMars behavioral models);

2. Equip the model with these *properties* and a *checking mechanism,* so that the model expert and the validation team can pursue the modeling process respecting the required properties.

The jABC supports these two steps in two different modes: it offers

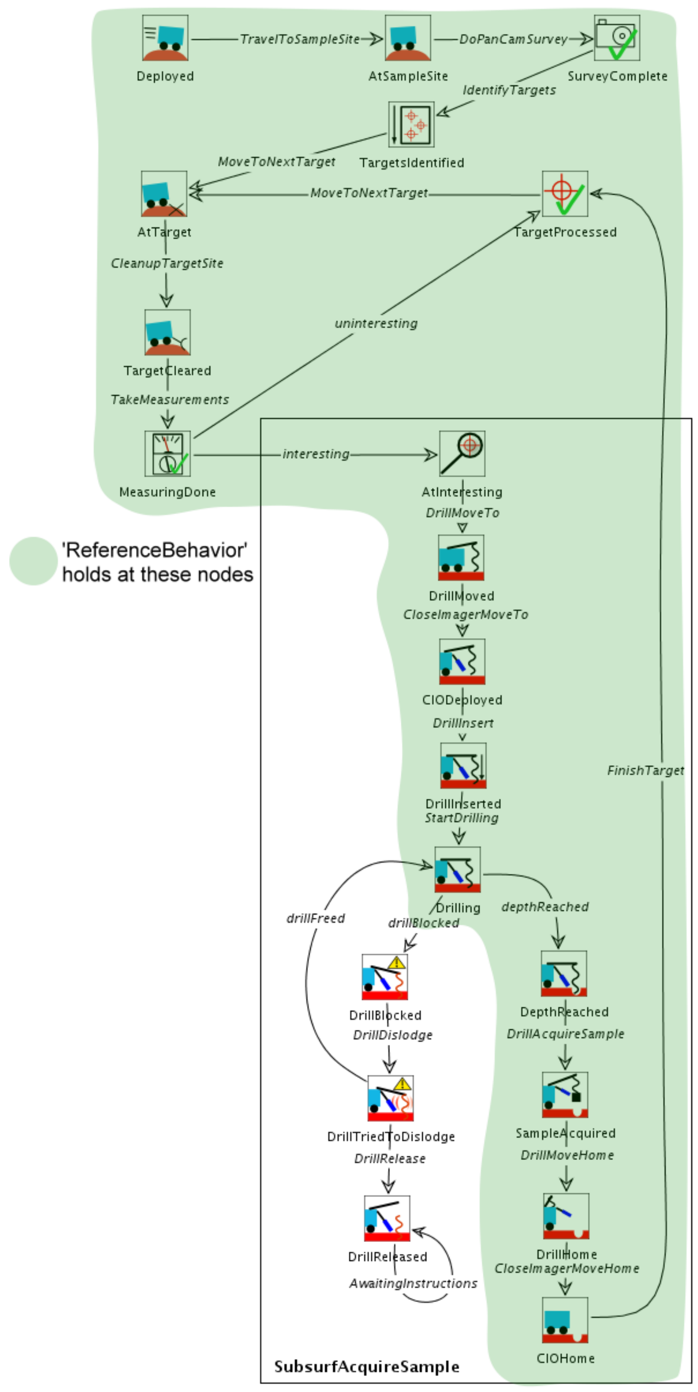


FIG. 3 – *Task and Action level model: Sample acquisition and failure recovery.*

- A *simple view* for immediate checking, adequate for an overview and for users that are not familiar with the jABC details and with the logic internally used for the property expression and checking;

- An *advanced view*, for the property experts and for the in depth model analysis, diagnosis, and repair.

To verify behavioral properties for reliable systems we use GEAR, the verification plugin of the jABC. GEAR is a game-based model checking tool capable of handling the full modal μ-calculus (Kozen, 1982) and derived, more user-friendly logics like CTL (which we use in this case study). It provides an intuitive GUI and graphical support that enables system designers and engineers to explore behavioral (temporal) properties of the designed models. GEAR is also available as a verification library, thus it can be also used on its own independently of the jABC.

5 Simple View: Easy Verification

As shown in Fig. 2, *modelling experts* can easily construct task and mission-level models on the basis of the library of elementary actions provided in the collection on the left. Already during the modelling phase the model experts can check the model's conformance with domain-specific conformance rules and properties, as shown in Fig. 2 (bottom left). In this snapshot we see that three such rules have been defined, concerning operations' visibility, correct functioning of the drill, and a general requirement guaranteeing the option of recovery. These rules are defined in GEAR's formal property language (see Sect. 6) and displayed in the Simple View mode as natural language description. By clicking on the Check button all the properties in the rule library are automatically checked for the model.

In this case we see that the first two properties are respected, since they have now a green background in the property list, and the third one is violated. If a property is violated, this view highlights the defective parts of the model. In our case, the Task *AcquireSamples* violates the property: it is the only task marked with a small red cross (at the top right in the model), while the rest of the model is compliant with this property: all the other icons are marked with a green tick symbol.

The violation detected here cannot be repaired on this model, and it also cannot be repaired with minor changes. We show therefore how to resolve the model/property inconsistency with the aid of the advanced view.

6 Advanced View: Properties and Interactive Repair

The advanced view provides access to the model and to the properties in a way adequate for the quality assurance, auditing, and validation teams. This concerns the definition of the properties subject to checking, and facilities for the game-based interactive exploration of the model's behavior wrt. a violated property, as described later in Sect. 6.2.

6.1 Defining a Property library

The properties checked by the model checker have been previously formulated in a formal way. Domain knowledge, certification and compliance conditions, safety requirements are all sources of properties that models must respect. They are captured and graphically formulated in one of the logics supported by GEAR. Since behavioral properties express conditions on system runs, temporal logics are particularly adequate here because they support primitive concepts like *always, eventually, in the successive state*, *on one run*, or *on all runs from now on*. These temporal properties, such as CTL formulas, can be specified using the Formula-Builder plugin (Jörges et al., 2006), providing graphical means of modeling properties.[2] The violated property mentioned above can be modeled using the Formula-Builder plugin, resulting in the CTL expression

$$\mathsf{AG}(\mathsf{EF}(ReferenceBehavior))$$

Moreover, properties can be graphically described without the need to master a mathematical/temporal logic or the syntax used by the tool. In particular, Specification Patterns (Dwyer et al., 1998) can be used to provide properties like e.g. *The Close Image Observer must monitor all drill operations*, corresponding to a *Universality Between* pattern. Our pattern library extends the pattern system of (Dwyer et al., 1998), for example by providing also patterns for history dependent properties, that technically require backwards temporal operators.

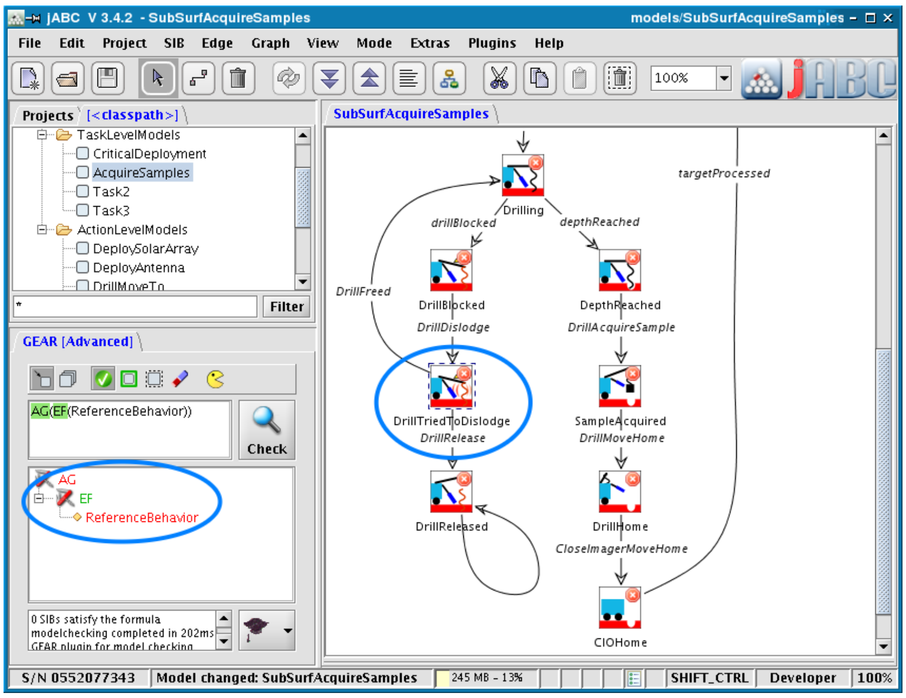
The Advanced View shown in Fig. 4 (bottom left) shows the temporal logic formula along with all its subformulas (its corresponding syntax tree), which is used for fine granular error diagnosis by means of *reverse checking*. As we see, *every* node in the model violates the property: it seems hopeless to simply adapt property or model to bring both in line without more detailed information.

With reverse checking, we can explore in detail the properties and subproperties satisfied by each single model component. In this case, selecting the model node DrillTriedToDislodge, the subproperties it violates in this context are marked red. This information is obtained from the rich output of the game-based approach without a need for further computation. Thus we see that although it does not satisfy the original property, a part of it still holds, namely $\mathsf{EF}(ReferenceBehavior)$. On this static view of the model validation engineers can conduct a very fine granular exploration of the property components and the model components, but it still does not reveal whether the inconsistency should be resolved by acting on the model or on the property. This is too a frequent case: the property may be too strong and could be relaxed. The reason for this lack of information is inherent in the separation of concerns between model and property, while the inconsistency is originated by the *interplay* of the two. To adequately pinpoint the problem and restore consistency we need to build a structure that adequately reveals this interplay.

6.2 Behavioral Diagnosis by Playing Games

The blend of model and property that reveals their interplay in a user-friendly way is the *game graph*. It represents a game that is played by two players (a demonic/angelic player that tries to refute/verify the property). The result of the game corresponds to the result of the verification. A more detailed description of game graphs and their interpretation can be

[2]The jABC also provides means to directly enter a formula close to the mathematical notation.



0 SIBs satisfy the formula
modelchecking completed in 202ms
GEAR plugin for model checking

FIG. 4 – *Verification and* reverse checking *with GEAR.*

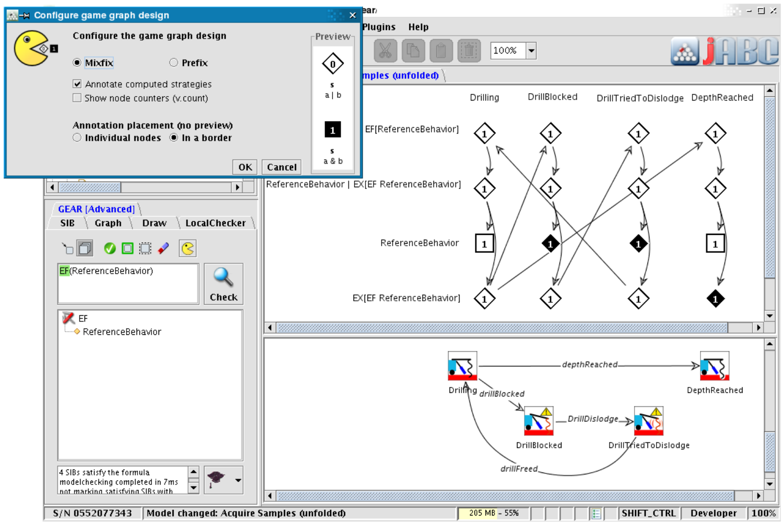
found in (Bakera et al., 2007). Here we only show how to use them to obtain a rationale for pinpointing and resolving the constraint violation.

In the advanced view, the game graph corresponding to a given model and a given property is automatically created when clicking on the doctor's hat icon at the bottom of the property inspector (Fig. 4). The presentation of the game graph is customizable to the validator's needs:

1. A *Property-oriented View* (as shown in Fig. 5) focuses on the property and the relation between its different sub-parts (in form of subformulas) and the model elements. Therefore the columns of the game graph correspond to the model's nodes, its rows correspond to the property and its parts, and a node in the game graph can be thus easily mapped to the configuration (node, subformula). This is adequate for studying the pattern of transitions which reveals systematic interactions between model elements and subformulas. It may have sparse transitions. In particular, the game graph columns correspond to the model nodes shown at the bottom of Fig. 5;

2. A *Model-oriented View* keeps the original layout of the model, and aggregates the parts of the game graphs that belong to the same action of the behavioral model, as shown in Fig. 6. This view shows only the existing transitions, and it is much more intuitive for engineers.

We now show how to use the model oriented view of the game graph in Fig. 6 obtained from the relevant part of the model of (Fig. 3) to pinpoint the violation. Here,


Configure game graph design

Configure the game graph design

[✓ Mixfix] Prefix

[✓] Annotate computed strategies

[ ] Show node counters (v.count)

Annotation placement (no preview)

Individual nodes [✓ In a border]

EF(ReferenceBehavior)

4 SIBs satisfy the formula.
modelchecking completed in 7ms
not marking satisfying SIBs with

S/N 0552077343 | Model changed: Acquire Samples (unfolded) | 205 MB - 55% | SHIFT_CTRL | Developer | 100%

FIG. 5 – *Property-oriented view on the game graph of the constraint violation.*

- A *box* indicates the part of the game graph corresponding to an action-level model node;
- *Black nodes* indicate a local mismatch between model and property;
- *White nodes* indicate a match between the annotated part of the property (a sub-formula) and the model node – i.e. this part of the model respects that part of the property;
- *Diamond-shaped nodes* indicate intended possible alternatives of the property and possible system evolutions of the model;
- *Square-shaped nodes* indicate a system choice (a system evolution) that tries to falsify the property by a faulty system run.

Thus *square black nodes* give a hint for system evolutions that do not meet the property (or parts of it). These are the important nodes in the following. The red path (dark-gray in the b/w version) in Fig. 6 reveals one system evolution that does not guarantee recovery – and thus violates the desired property. If the drill gets blocked and the dislodge attempt fails, the error recovery process is about to fail and the drill has to be released. Looking at the black square nodes we see that at the faulty node (f) the reference behavior is not necessarily reestablished once the dislodge attempt failed. This is the problem.

6.3 Adaptation

Correcting the problem requires a good understanding of both model and property, since it lies in their interplay. The system run that is about to lose the drill is (although not desired) still unavoidable. We cannot thus eliminate this run, and instead of modifying the model, we modify the property to

$$\mathsf{AG}(\mathsf{EF}(ReferenceBehavior \vee DrillReleased))$$

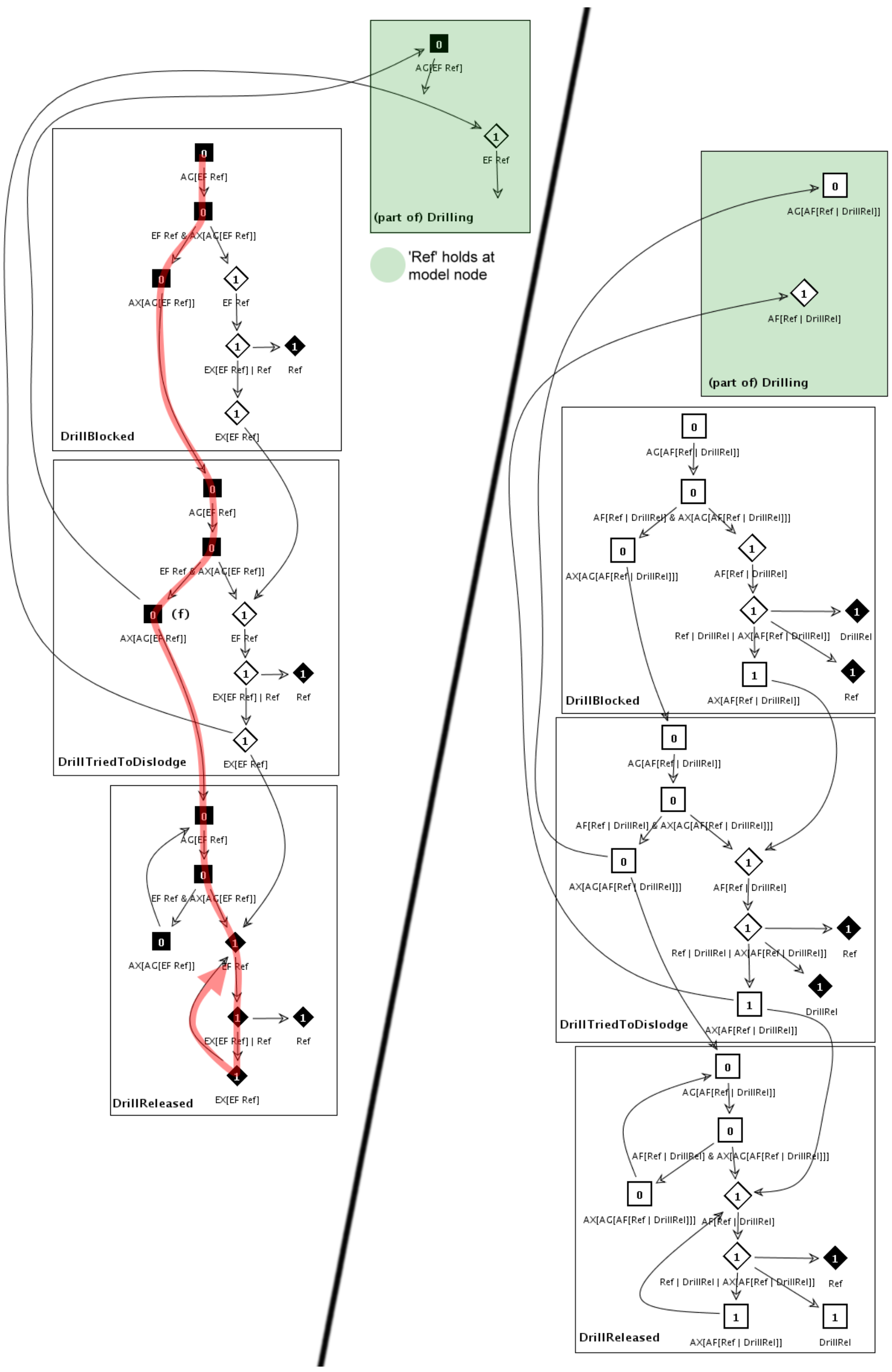


FIG. 6 – *Model-oriented views on the game graph, highlighting system evolutions not ensuring recovery (left) and the adapted property (right).*

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

This property is now satisfied by the model – i.e. it is possible to reestablish the reference behavior, or the drill is lost. In the corresponding graph (right-hand side of Fig. 6) it becomes apparent now that –due to the lack of winning opportunities for the player owning the square-shaped nodes (which have now turned white)– the following stronger property is valid:

$$\mathsf{AG}(\mathsf{AF}(ReferenceBehavior \ \vee \ DrillReleased))$$

meaning that the model *in any case* reestablishes the reference behavior or loses the drill. This property holds for the entire model and resolves the inconsistency.

7 Related Work

A prominent and successful platform for diagnosis and adaptation of autonomous systems is the Livingstone system for mode identification and reconfiguration developed at NASA Ames (Williams and Nayak, 1996). Whereas diagnosis (mode identification) and adaptation (reconfiguration) in Livingstone occurs at run-time, our approach concerns the design process. Therefore they are complementary: we provide very rich diagnosis information, but our approach is not intended for time-critical, real-time scenarios at run-time.

Various model checkers are used to verify aerospace systems. Java Pathfinder (Visser et al., 2003) developed at NASA Ames is a prominent representative for verifying smaller systems. It assists developers at the Java code level, and therefore addresses a later phase wrt. to our approach. We aim at assertions on interactions between components or of the system as a whole, with an accent on demanding properties.

The model checker SMV (Burch et al., 1990) allows for symbolic description of systems. Pecheur and Simmons (Pecheur and Simmons, 2000) automated the process of converting Livingstone models into SMV's syntax, thus enabling Livingstone users to benefit from model checking. However, SMV does not provide the detailed information about errors that we get from the games, nor it assists in diagnosis and adaptation in a graphical manner.

A different approach for coping with understanding of implementations and specifications are *temporal queries* as introduced by Chan (Chan, 2000). Such queries allow a placeholder "?" to appear exactly once in the specification. Solving the query amounts to finding a substitution for "?" that makes the whole formula true. This could be seen as an automated version of the adaptation process. However, up to now placeholder variables are limited to be substituted by propositional formulas. Therefore they do cover more demanding properties like the one we have presented here.

To our knowledge, game-based model checking as presented in this paper has not been previously used in the context of autonomous aerospace systems. We believe that it is beneficial in supporting the tighter collaboration of different expert groups (model designers, validators, etc.) and that it contributes to a better understanding of emerging problems.

8 Conclusion

We have shown on a concrete example how different user groups in a design and engineering team can be adequately and individually supported in their work to achieve a correct hierarchical model driven design in the context of autonomous aerospace systems. Central to this is the use of a design platform like jABC, that in particular offers facilities for the expression, verification, and diagnosis of behavioral properties of the system's models. In particular, the versatility of GEAR helps different user groups to benefit from the checking facilities at different competence levels, from fully automatic to interactive exploration.

In the context of the EU SHADOWS project we are currently integrating the presented approach into a platform for self-healing systems.

References

Bakera, M., T. Margaria, C. D. Renner, and B. Steffen (2007). Property-driven functional healing: Playing against undesired behavior. In *Proc. CONQUEST 2007, 10th Int. Conf. on Quality Eng. in Softw. Techn.*

Berry, G. and G. Gonthier (1992). The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19*, 87–152.

Bormann, G., L. Joudrier, and K. Kapellos (2004). FORMID: A formal specification and verification Environment for DREAMS. In *Proc. 8th ESA ASTRA Workshop*.

Burch, J. R., E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang (1990). Symbolic model checking: 10^{20} states and beyond. In *LICS*, pp. 428–439. IEEE Computer Society.

Chan, W. (2000). Temporal-logic queries. In *Proc. CAV*, Volume 1855 of *LNCS*, pp. 450–463. Springer.

Dwyer, M. B., G. S. Avrunin, and J. C. Corbett (1998). Property specification patterns for finite-state verification. In *FMSP*, pp. 7–15. ACM.

Jörges, S., T. Margaria, and B. Steffen (2006). FormulaBuilder: A tool for graph-based modelling and generation of formulae. In *Proc. ICSE*, pp. 815–818. ACM.

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

Kozen, D. (1982). Results on the propositional μ-calculus. In *ICALP*, Volume 140 of *LNCS*, Aarhus, Denmark, pp. 348–359. Springer-Verlag.

Pecheur, C. and R. G. Simmons (2000). From Livingstone to SMV. In *FAABS*, Volume 1871 of *LNCS*, pp. 103–113. Springer.

Shehory, O., S. Ur, and T. Margaria (2007). Self-healing technologies in SHADOWS: Targeting performance, concurrency and functional aspects. In *Proc. CONQUEST 2007, 10th Int. Conf. on Quality Eng. in Softw. Techn.*

Steffen, B., T. Margaria, R. Nagel, S. Jörges, and C. Kubczak (2006). Model-driven development with the jABC. In *Proc. 2nd Haifa Verification Conference*, Haifa, Israel. Springer.

Visser, W., K. Havelund, G. P. Brat, S. Park, and F. Lerda (2003). Model checking programs. *Autom. Softw. Eng 10*(2), 203–232.

Williams, B. C. and P. P. Nayak (1996). A model-based approach to reactive self-configuring systems. In *AAAI/IAAI, Vol. 2*, pp. 971–978.