Noname manuscript No.
(will be inserted by the editor)

Tool-supported enhancement of diagnosis in model-driven verification

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

the date of receipt and acceptance should be inserted later

Abstract We show on a case study from an autonomous aerospace context how to apply a game-based model-checking approach as a powerful technique for the verification, diagnosis, and adaptation of system behaviors based on 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, as a user-friendly tool that can offer automatic proofs of critical properties of such systems. Although it is a model checker for the full modal μ-calculus, it also supports derived, more user-oriented logics. With GEAR, 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.

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`.

This article is an extended version of [16] presented at ISoLA 2007, Poitiers, Dec. 2007.

T. Margaria · M. Bakera
Chair Service and Software Engineering, Universität Potsdam, D-14482 Potsdam, Germany, {margaria, bakera}@cs.uni-potsdam.de, `http://www.cs.uni-potsdam.de/sse`

C.D. Renner · B. Steffen
Chair of Programming Systems, TU Dortmund, D-44227 Dortmund, Germany, steffen@cs.tu-dortmund.de, `http://ls5-www.cs.uni-dortmund.de`

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 are highly technical and tend to be applicable with difficulty in fielded systems. To address the need for industry-grade software self-healing, the SHADOWS project focuses on self-healing of complex systems, extending the state-of-art in several ways [20]. It introduces innovative technologies to enable self-healing of new classes of problems and it additionally integrates several self-healing technologies into a common solution. Within SHADOWS, we adopt a model-based approach, where models of desired software behavior direct the self-healing process. This allows for life cycle support of self-healing applicable to industrial systems.

Fig. 1 shows an architectural overview of this system. We contribute to SHADOWS a number of enabling technologies for model-driven self-healing residing in the functional part of the architecture. Our technologies deal with self-healing issues at design-time for ensuring functional correctness – i.e. correctness with respect to the system's behavior over time. Other parts of the project address concurrency and performance issues.

In this article, we use a case study from an autonomous aerospace context as a running example. We show how to apply a game-based model-checking approach as a powerful technique for the verification, diagnosis and adaptation according to desirable temporal properties that the system's behavior must exhibit. In particular, we show how to model the several abstraction levels of the robot's behavior in a uniform and formal but intuitive way. This happens in terms

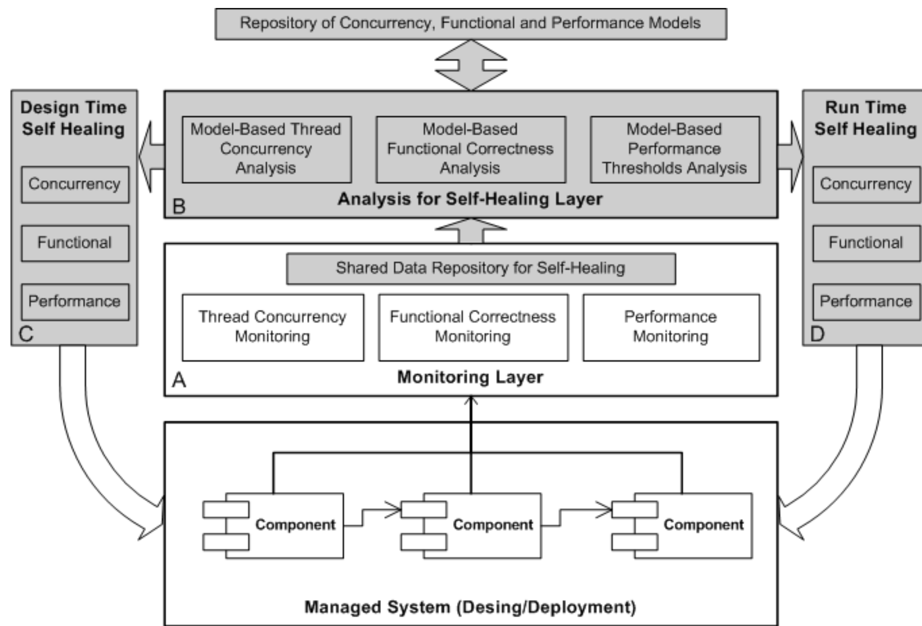

Fig. 1 The architecture of self-healing systems proposed by the SHADOWS project. [20]

of processes in the jABC framework [21], a mature, model-driven, service-oriented process definition platform. Subsequently, we leverage the formality of these models to prove properties by model checking. 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.

Design-time healing technologies that naturally emerge when dealing with self-adaptive systems, as in the context of the SHADOWS project, demand for a deeper insight of design-time faults to effectively identify and overcome them. The use of models rather than code is already a significant step towards the understandability of the actual behavior's descriptions to non programmers, like the engineers, in charge of designing a space module. This enables e.g. early discovery of misbehaviors, hazards, and ambiguities via design-time analysis.

We strive to improve the diagnostic features making them as detailed as necessary yet as intuitive as possible. Our design-time analysis technique is game-based model checking. Indeed, this approach reveals information on the system's behavioral problem that is concise enough to be used further for autonomic adaptation issues. It does not only present error traces, as conventional model checking, but it reveals complex interaction patterns between the system, the environment, and the desired systems' execution behavior over time – along behavioral properties specified in terms of temporal logics. This allows engineers to investigate in depth the non-compliant behaviors, and to pinpoint the essence of the mismatch between required property and offered behavior. In particular for self-healing or self-reconfiguring systems, this is of great value whenever deep behavioral analysis is needed. This happens first at design time, and even more often during maintenance and evolution. This is especially true for space equipment, that is inaccessible to direct inspection, and frequently happens to be re-purposed during the course of the mission. An extremely successful example are the Voyager 1 and 2 missions, where the spacecrafts were operational and alive well beyond their primary mission and are exploring the deepest universe now since Summer 1977.

In the following, we first show how we model the ExoMars Rover surface mission in the jABC, according to the description provided in [12]. 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 [4]. In the concrete mission example described in [12], 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. 2 shows a model of the Rover as presented on an aerospace exhibition.

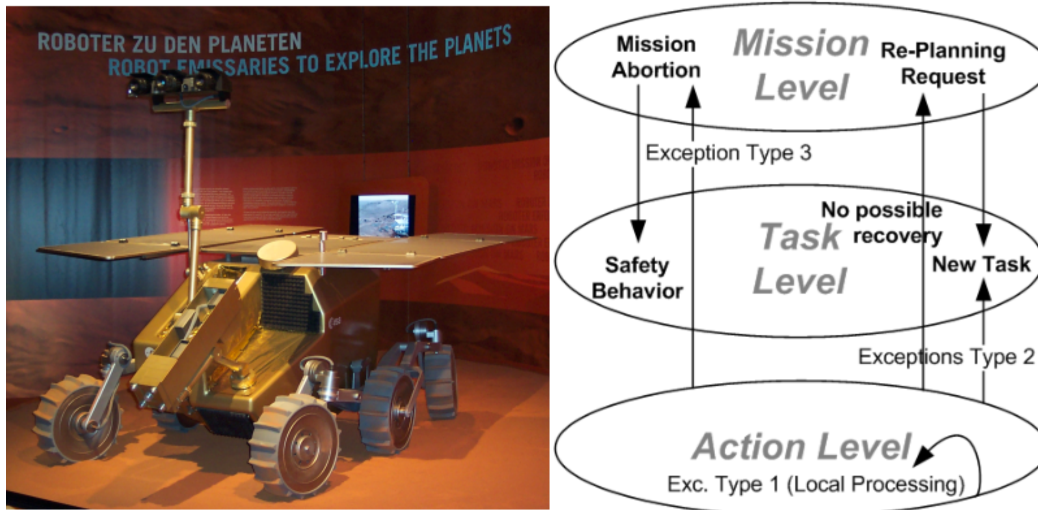

Fig. 2 The ExoMars Rover and the three-tier control model.

As customary, the mission is organized in a hierarchical three-tier control model shown in Fig. 2, which accounts for partial autonomy of the Rover. Mission plans are designed and enforced by the ground control center, 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 here relies on a system description created with FORMID [12] using Esterel as a high-level specification language [3]. 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. In the reference documents, these properties were given as informal textual descriptions. We formalized them from those descriptions in appropriate CTL [2,7] representations. The kind of formal verification considered in the ESA study concerns predefined and pre-programmed patterns of safety, liveness, and conflict-freedom properties. While the meaning of safety and liveness properties is standard, conflict-freedom means here the absence of contention of actions between sub-systems. In the ESA study, an *observer* module is generated for each property. It spies the system model to detect violations. In a violation case, a violating scenario is returned to the designer. This kind of diagnosis is of simulative character: it can be performed at run-time, or at design time as a simulation. It cannot be exhaustive, and further it cannot be used to check violation of liveness properties.

3 Hierarchical Modelling of the ExoMars Behavior

Starting from the behavioral descriptions provided in [12], we model all levels of the three-tier hierarchy with the jABC. In terms of user groups, we intend to 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, and
2. *dependability and validation engineers*, who know the desired properties of the system. We help them here to express those do's and don'ts in terms of logical properties, automatically checkable on the behavioral models.

The basis of the models is the collection of actions of which the system (here a subset of the Rover) is capable. Building models on the basis of a given action library is a graphical activity: The upper left corner of Fig. 7 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.

We examine now all three levels of behavior description and their corresponding modelling style in jABC.

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. 3 sketches this

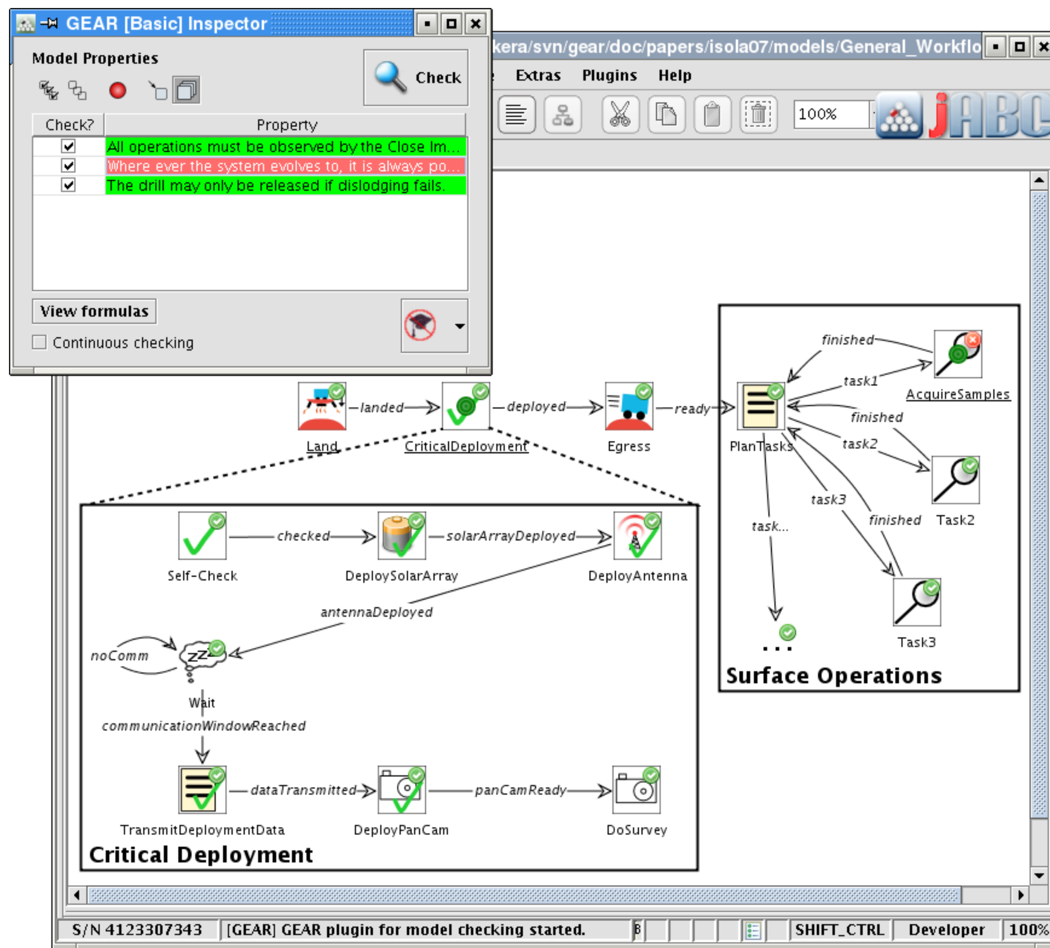

Fig. 3 Property-controlled modelling at different abstraction levels.

high-level behavioral model as it appears in the jABC. 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, such as 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. 2), elementary Actions constitute re-usable, basic building blocks of behavior. They are organized in libraries and composable into Tasks that are 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. 3, the Rover's surface mission consists of the three main phases *Critical Deployment*, *Egress* and *Surface Operations*, arranged in a sequence of actions.

- in the *Critical Deployment* phase, the Rover's operational set-up is established (solar power acquisition, communication capabilities);
- 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 the required measurements and operations, and transmits the relevant data back 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. 3. 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 unfavorable planetary constellations impede communication with ground support, the Rover shuts itself down and waits until a sufficiently favorable time window occurs. Once communication has been successfully 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. 4 shows a detailed model of the AcquireSamples surface operations task from Fig. 3. The green colored part (gray in the printed version) concerns the ReferenceBehavior. This part of the model is of central interest because it represents the intended and normal behavior of the system. During sample acquisition, the Rover examines one by one the 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 sample extraction process itself is shown in the SubsurfAcquireSample box at the lower right of Fig. 4. 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 operation was disturbed. This is captured in the left branch of the box in Fig. 4. 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 the right part of Fig. 2, exception handling is of vital importance since it directs recovery actions and it enables a successive replanning.

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.

Particularly 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), and
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 both steps in two different modes: it offers

– A *Simple View* for immediate checking, adequate for an overview. This simple view is preferable 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 [13] 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 within SHADOWS as a verification library, thus it can also be used on its own, independently of the jABC. However, here we will concentrate on the full-fledged plugin version that comes with the jABC since it provides an easy to use interface and attractive visualization capabilities.

5 The Simple View: Easy Verification

As shown in Fig. 3, *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

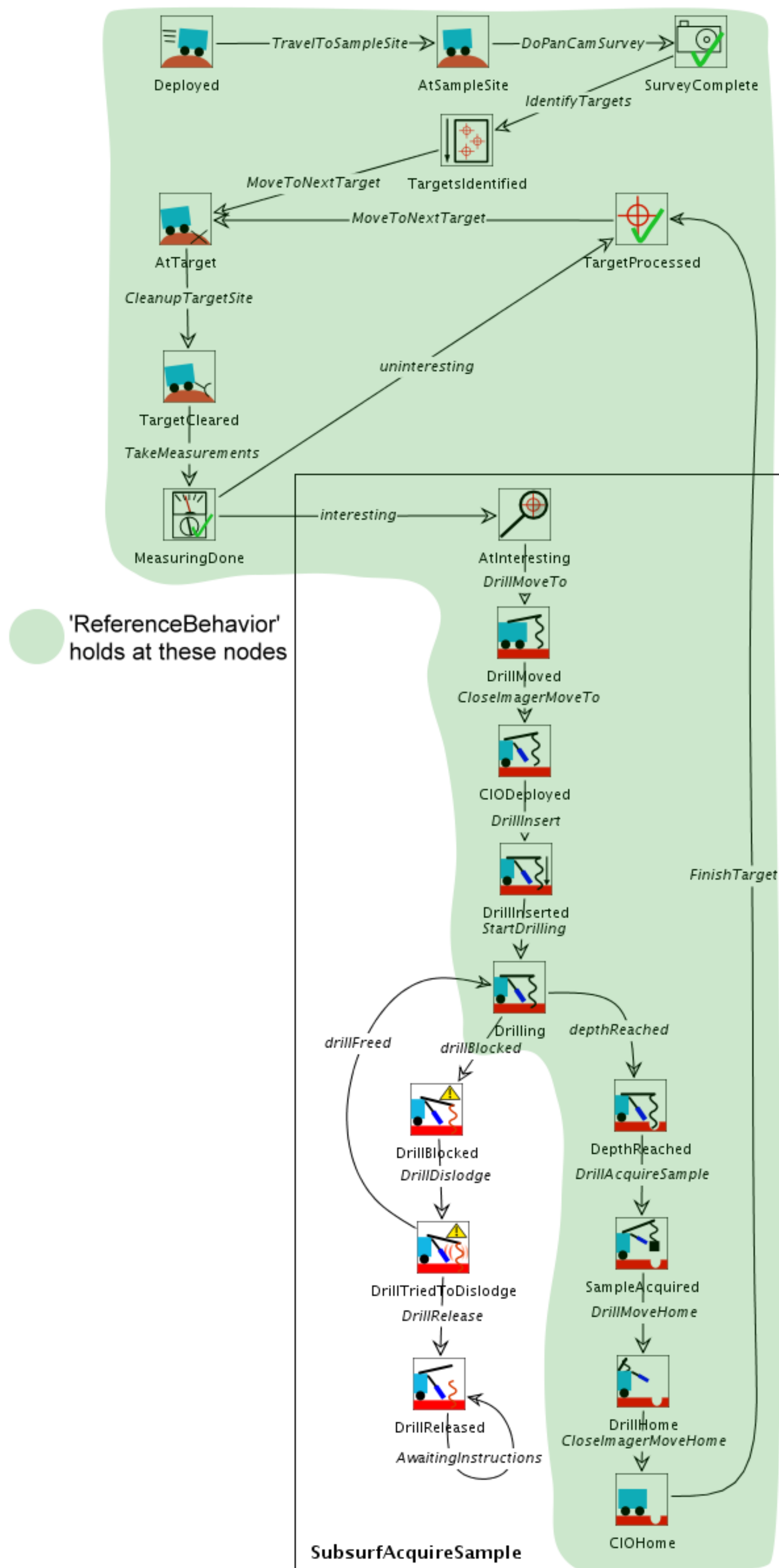

Fig. 4 Task and Action level model: Sample acquisition and failure recovery.

the model experts can check the model's conformance with domain-specific conformance rules and properties, as shown in Fig. 3 (top 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 descriptions. 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 and third property are valid, since they have now a green background in the property list, but the second one is violated. If a property is violated, this view highlights the defective parts of the model. In our case, it is the Task *Acquire-Samples* that violates the property, while the rest of the model is compliant with it. In fact, this is the only task marked with a small red cross (at the top right in the model. All the other task 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. This view is therefore useful to examine whether the described behavior is consistent with the current library of properties, and to determine which rules are not respected. To reestablish consistency, more detail is likely needed, and we show therefore how to resolve the model/ property inconsistencies with the aid of the Advanced View.

6 Advanced View: Properties and Interactive Repair

The Advanced View provides access to the model and the properties in a detailed way, adequate for the quality assurance, auditing, and validation teams. This concerns the definition of the properties subject to checking, and the 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

Collaborative specification support GEAR enables a collaborative specification approach that involves the *specification expert* who is familiar with the formal mathematical background of the system's intended temporal behavior, and the **system engineer**, that is aware of the particular problem domain and its constraints. While the specification expert narrows down the outer bound of all possible realizations for a problem, the engineer widens the inner bound of possible behaviors by realizing a set of concrete ones. To be able to remove undesired behavior from the specification, the system designer can give hints on them, derived from domain-specific knowledge, to the specification expert. Afterwards the specification expert can integrate the request into the overall specification or deny with a justification for not including it. Over time, the specification not only converges to the collaboratively desired imagination of the final system but also reveals a history of specification requests that were abandoned and therefore undesired.

To fruitfully support the communication process of the two expert groups, GEAR allows annotating the model with property specification requests via jABC's annotation editor, as shown in the left part of Fig. 5. The right part depicts a properly prepared version of all requests that can be automatically generated and displayed in a web browser. Now the specification expert can use the editor to give a justified denial of the request or integrate it into the specification and remove it from the requests.

Property specification The properties checked by the model checker have been previously formulated in a formal way. Domain knowledge, certification and compliance conditions, or 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 FormulaBuilder plugin [11], which provides graphical means of modeling properties without the need to master a mathematical/temporal logic or the syntax used by the tool.[1]

Figure 6 shows an example property for the scenario: it ensures that the sample acquisition process is guaranteed to be monitored with a camera as soon the *CloseImager* has been deployed until it will be moved back into the Rover. This ensures that the Close Image Observer must monitor all drill operations.

This rather simple property, although commonly used, unfortunately has a very large and unhandy representation when expressed in pure CTL. This CTL expression, as generated by the Formula-Builder from the graphical representation, is shown in the left panel of Fig. 6.

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

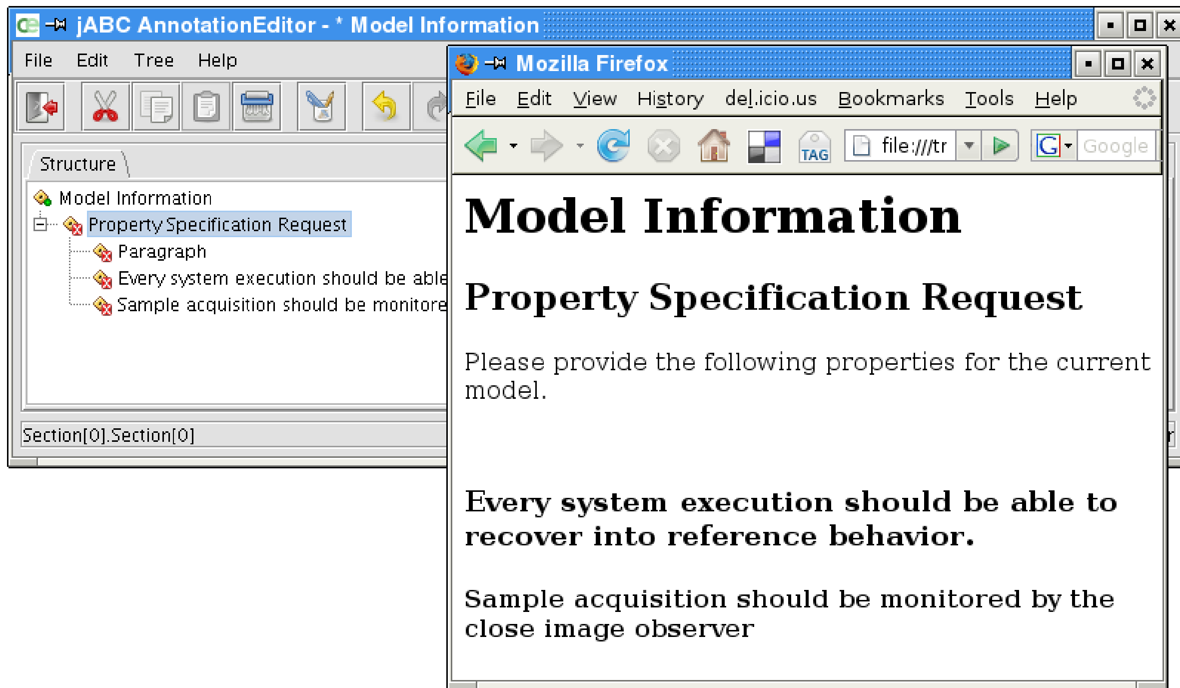

Fig. 5 Editor for specification requests as annotated by the system engineer for the specification expert (left) and auto-generated HTML-preview displayed in a web browser (right).

This property already reveals the power of domain specific specification languages grounded on *Specification Patterns* [9]. It is in fact an instance of the *Universality Between* pattern. Our own pattern library extends the cited pattern system, for example by providing also patterns for history dependent properties, that technically require backwards temporal operators.

We return now to the violated property mentioned in Sect. 4. We model it in the FormulaBuilder (top right in Fig. 6), resulting in the following CTL expression.

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

The Advanced View shown in Fig. 7 (bottom left) shows the temporal logic formula along with all its subformulas (as the corresponding syntax tree). This is useful for the fine-granular error diagnosis e.g. by means of *reverse checking*.

GEAR's reverse checking intuitively inverts the classical Model Checking question, which asks which parts of a model satisfy a given property. Now, inversely to this classical view, reverse checking selects some node of the model and asks which parts of the specification (which subformulas) it satisfies.

This capability is essential for a detailed diagnosis. As indicated by the red crosses annotating the model nodes, we see in fact that *every* node in the model violates the property. Therefore, without more detailed information, it seems hopeless to simply adapt either property or model to achieve conformance.

With reverse checking, we can explore in detail the properties and subproperties satisfied by each single model component. In this case, when selecting the model node DrillTriedToDislodge, the subproperties it violates in this context are marked red (and by the oval). Two properties are marked: the whole formula

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

and the proposition

$$ReferenceBehavior$$

. Thus we see that although the node does not satisfy the original property, a part of it still holds, namely $\mathsf{EF}(ReferenceBehavior)$.

This information is obtained from the rich output of the game-based approach without need of further computation.

On this static view of the model, validation engineers can conduct a very fine-granular exploration of the property components and of the model components, but this still does not reveal whether the inconsistency should be resolved by acting on the model or on the property. Although model checking customarily considers properties as given immutables, and thus takes them as stringent requirements that must be satisfied, this is not necessarily the case in practice. Especially in early or high-level design phases, the properties and in particular their exact formulation are subject to design as

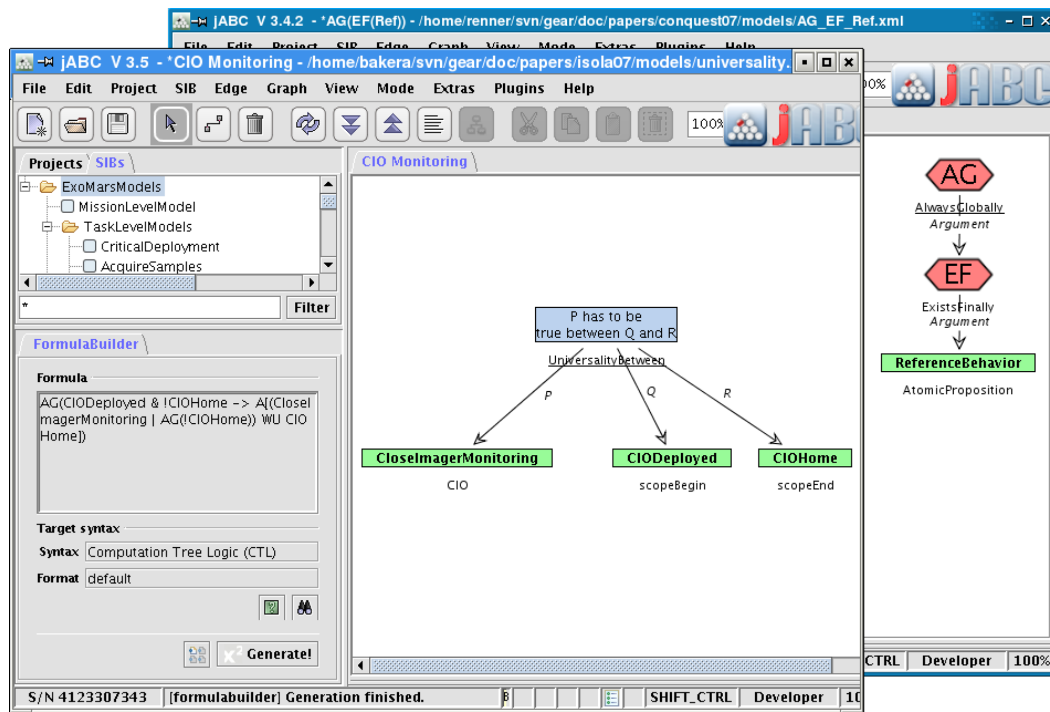

Fig. 6 Two properties graphically specified with the FormulaBuilder (left: an instance of a universality property pattern, right: a CTL recovery property for autonomous systems). The left panel shows the generated CTL version.

well, and to revisions in the course of development of the behavioral models. This need to explore the purpose and the extent of a property is therefore a frequent case too. The property may be too strong, and could be relaxed. Or have to be split in more subproperties in order to take into account exceptions or refinements. The reason for this lack of a priori complete and correct information is inherent in the separation of concerns between model and property. Inconsistencies are 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 player that tries to refute the property and an angelic player that tries to prove it. The result of the game corresponds to the result of the verification. A more detailed description of game graphs and their interpretation has been given in [1]. Here we concentrate on how to use them at an engineer's level in order to obtain a rationale for pinpointing and resolving the constraint violation.

From the Advanced View, the game graph corresponding to a given model and property is automatically created when clicking on the yellow pac-man icon in the tool bar of the property inspector (Fig. 7). The presentation of the game graph is customizable to the validator's needs in the following two ways.

1. The *Property-oriented View* (shown in the top right of Fig. 8) focuses on the property. In particular it illustrates the relation between its different subparts (in form of subformulas) and the model elements (shown in the canvas below the game graph). The columns of the game graph correspond to the model's nodes while its rows correspond to the property and its parts. Within this two-dimensional space, each node in the game graph can be easily mapped to the configuration *(node, subformula)*. This representation is adequate for studying the pattern of transitions, which reveals systematic interactions between model elements and subformulas. The game graph may have sparse transitions. In particular, the game graph's columns correspond to the model nodes shown at the bottom of Fig. 8. Transitions between these columns reflect the corresponding transitions between the model's nodes.
2. The *Model-oriented View* on the contrary 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. 9 (left

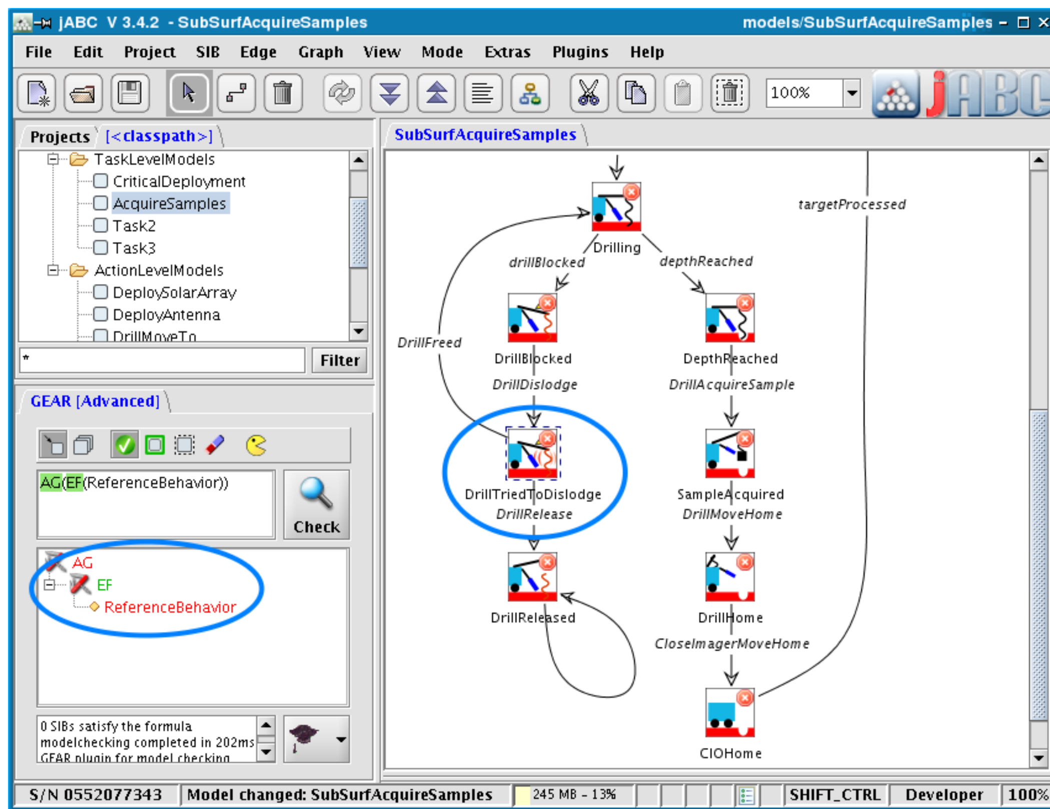

Fig. 7 Verification and *reverse checking* with GEAR.

or right). This view shows only the existing transitions, and it happens to be much more intuitive for engineers that are customarily more familiar with the behavioral model than with the internal structure of the property.

From this we see that the two views, although expressing the same content, address adequately two different groups of users:

– The property-oriented view results by systematic construction from a more concise visualization of the problem, as it arose when dealing with the problem. It is a standard 2-dimensional representation, that does not take into account the application domain or the single application, but focuses solely on the interplay between the model's substructure and the property and its substructure as a whole. Accordingly, this view is best suited for educational purposes (on game based model checking and its detailed functioning), and can be used to visualize and understand the evolution of the resulting strategies step-by-step. It is useful as a standard, application-independent representation that helps property engineers understanding the (pattern-like) structure of game graphs in dependence of (patterns of) properties. This insight does not help to understand the problem of the current model and property instances but enables to understand the complex interplay between controllable implementations and uncontrollable environments in general. It is useful for the property designers, not ideal for engineers and concrete case-by-case debugging.
– On the contrary, the model-oriented view preserves the information provided by the system developers in terms of the model's layout and arrangement of components. This view naturally focuses on the model, remains close to the model's structure, and additionally equips it with interactive, and therefore game-oriented, information of the behavioral specification over time. Thinking in analogy to a debugger tool, as commonly used in integrated development environments, the play of a game in this view provides those execution of the system that are witnesses of property violations. In contrast to ordinary step-by-step debugging – that only establishes understanding for one special case, i.e. for one algorithmic input – a property-oriented exploration of the model by playing the model checking game allows for understanding the violation in a more general sense.

In our case, the property of interest is that the system must be able to reestablish reference behavior or

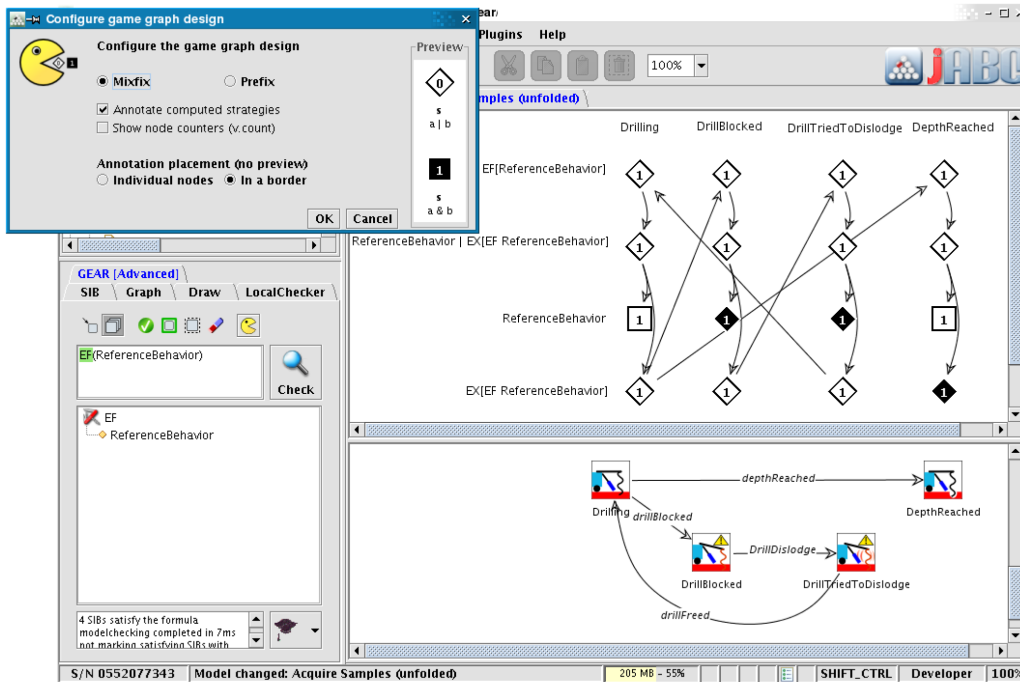

Fig. 8 Property-oriented view on the game graph of the constraint violation.

should at least wait for instructions from ground control. Depending on the setting and on the inputs, it is possible that by applying traditional testing and debugging techniques each system execution happens to avoid the execution path that waits for instructions. Therefore if the property is violated for a given run, we do not know if this part of the property is necessary or not. Removing this execution path from the property will indeed give us a hint at whether it was necessary or not, but does not reveal insight about *why*, or in which behavioral contexts this is the case. Conversely, an exploration of the model in a property-guided manner brings us right to the problematic part of the model, and reveals the essence of the mismatch. Using this insight, the system designer can now decide whether the property, the model, or both have to be adapted in order to resolve it.

We will now show how to to pinpoint the violation by using the model-oriented view of the game graph in Fig. 9 obtained from the relevant part of the model of Fig. 4. In that view,

- 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. This proves that the system playing as refuting player will win the game when playing at these nodes.
- *White nodes* indicate a match between the annotated part of the property (a subformula) and the model node – i.e. this part of the model respects that part of the property. White nodes allow the system engineer to verify the attached 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 all nodes in the game graph are partitioned according to two distinct classification criteria:

Black/white partitioning The partition into black and white nodes reveals the coverage of *how much* the desired property is satisfied by the system (satisfiability of the property by the model). The more white nodes the game graph has, the more the system designer is able to satisfy the property and parts of it. Black nodes on the other hand indicate faulty parts of the model or specification. For instance, if there are columns of game graph nodes in the model-oriented view that only contain black nodes, then for the corresponding node in the model it is not possible to prove any part of the property – i.e. the desired behavior (as expressed by the temporal property) is not respected by the corresponding component of the model implementation. See the DrillReleased block at the bottom left of Fig. 9.

Diamond/square partitioning The partition into diamond- and square-shaped nodes reveals information of

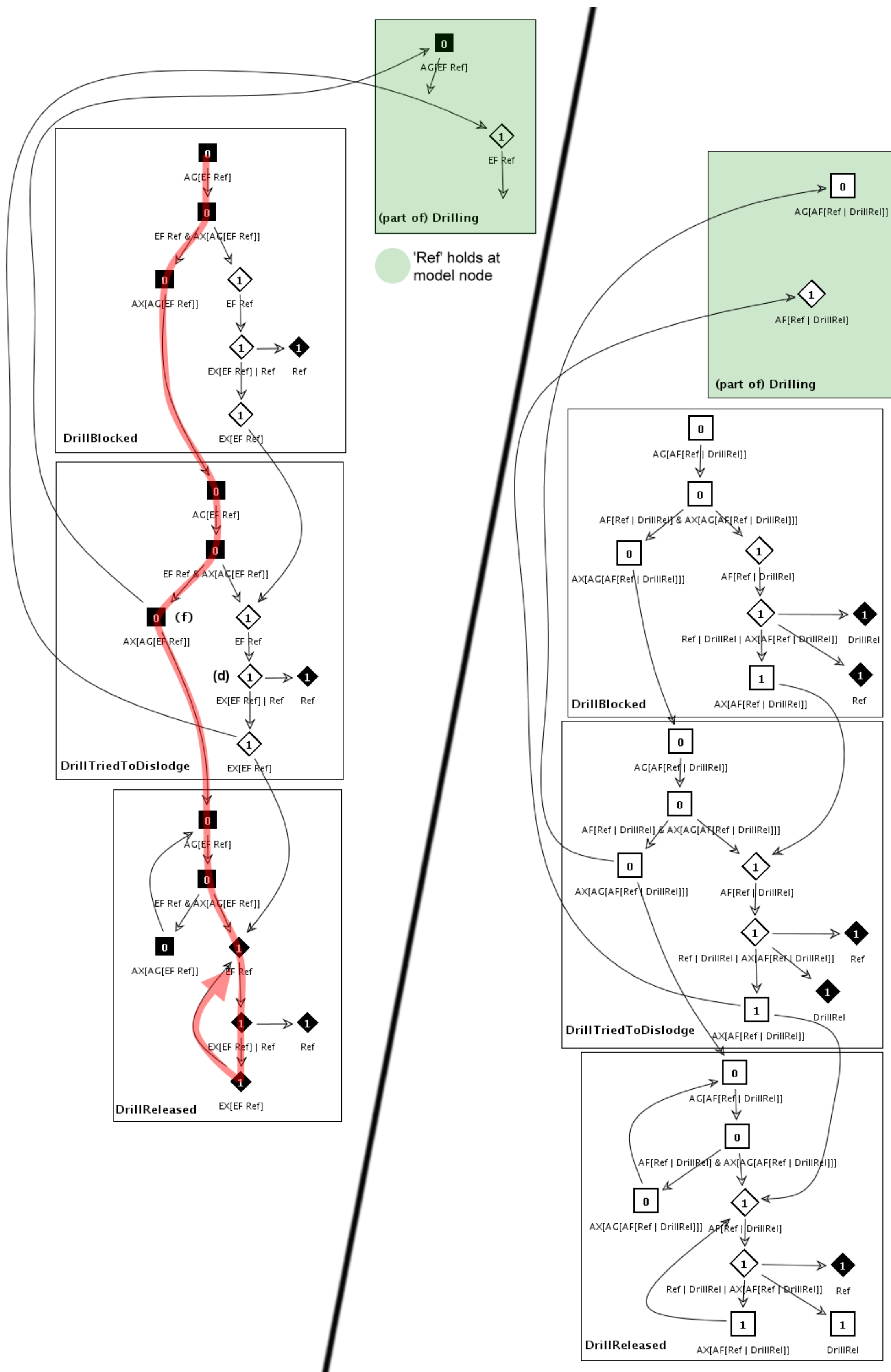

Fig. 9 Model-oriented views on the game graph, highlighting system evolutions not ensuring recovery (left) and the adapted property (right).

the structure of the property. Diamond-shaped nodes indicate a desired system evolution that eventually must occur (like the EF(*ReferenceBehavior*) part of the overall property). At such nodes, it is the system designer's turn to play and he/she must show how the model provides the desired liveness characteristic of the property. Square-shaped nodes give hints on undesired system evolutions, that the system designer must prevent from occurring. Hence the system plays at these nodes, and in its move it tries to find system evolutions that violate the property and that were not considered by the system designer.

Combining the information from the two partitions leads to the insight that *square black nodes* hint at system evolutions that do not meet the property (or parts of it). These are the important nodes, and in the following we will therefore concentrate on them. The red path (dark-gray in the b/w version) in the left part of Fig. 9 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 Comprehensive diagnosis support

For the system engineer there is no need for becoming familiar with this detailed interpretation of the game graph. GEAR in fact already enriches the game graph with such information and provides them via jABC's integrated *annotation editor*. Fig. 10 shows how to reach this editor by simply right-clicking on the node of interest and subsequently choosing the editor from the context menu. All the annotations provided for this node are presented in a tree-like structure. In this case, GEAR only provides an annotation for node `(Drill-Blocked, AG[EF[Ref]])` (where `Ref` is short for the *ReferenceBehavior* property). This information is in form of a game-based interpretation of the node and it is presented as a paragraph with a short description of the node.

Each node in the tree of annotations provides an editor to view, edit and update this information. The description of the game graph node is shown to the right in a small editor component used for paragraphs.

Therefore the diagnostic information is no longer bound to a knowledge base that resides beyond the problem scope, in a rather academic document delivered with the modelling environment, but it has become instead a living inhabitant of the current instance of the problem domain.

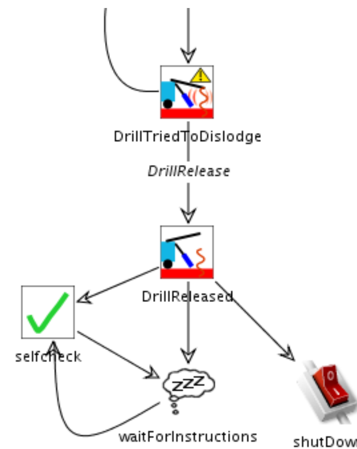

Fig. 11 Extending the model from Fig. 4 after the drill has been lost.

Interactive Counter Examples To present a more complex result that reveals more demanding types of counterexamples we now focus on a slightly refined version of the system model from Fig. 4, in which there are several options for the system to progress once the drill has been released. The Rover can now either perform a self check and wait for instructions afterwards (or vice versa), or perform a shut down of the whole system. This situation is depicted in Fig. 11. When checking the same property – i.e. whether it is possible to reestablish reference behavior – there are now two possible witnesses for violating the property:

1. The system performs a shutdown;
2. The system alternately performs a self-check and waits for instructions.

The first case grounds the property violation on an error path, while the latter uses a lasso-shaped, never-ending execution (a path leading into a circle) to falsify the property.

Figure 12 shows these two system executions and demonstrates that the system is not always able to reestablish reference behavior.

Due to the game-based interplay of model and property, the system designer is able to explore *all possible erroneous system executions* by playing against the erroneous system runs. In fact the game-based approach enables the system designer to use the model checker for debugging the system – executing it step by step – in a property-oriented manner, and focussing on the runs that are critical since they are known to violate the property.

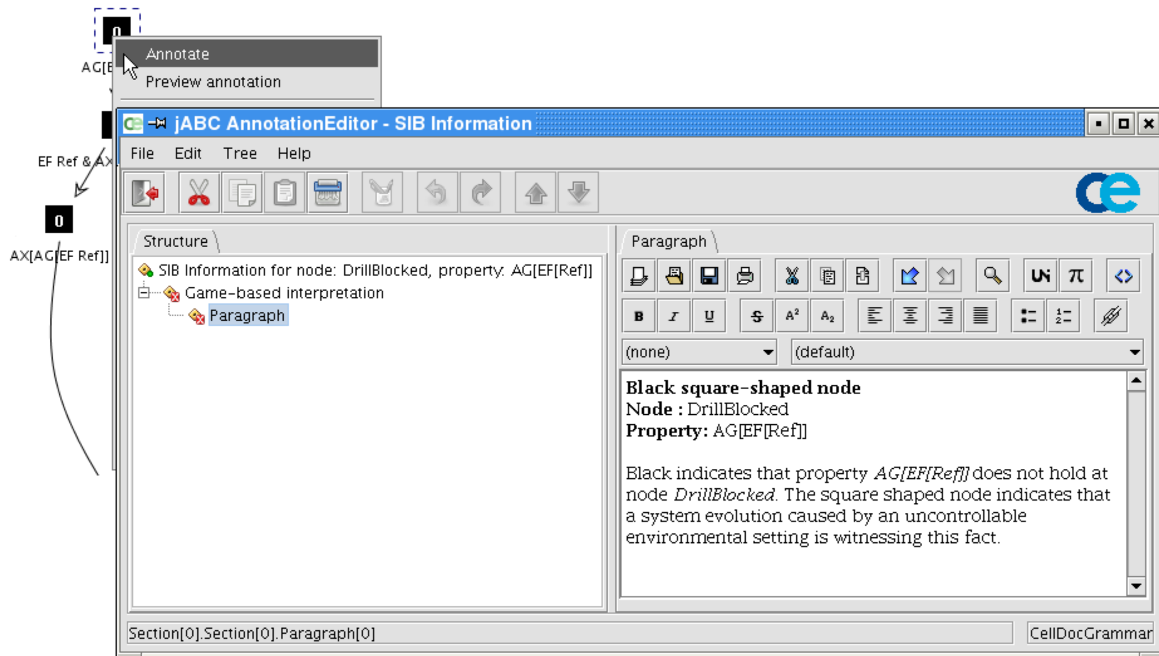

Fig. 10 Supporting the system engineer with integrated in-place information for the game graph.

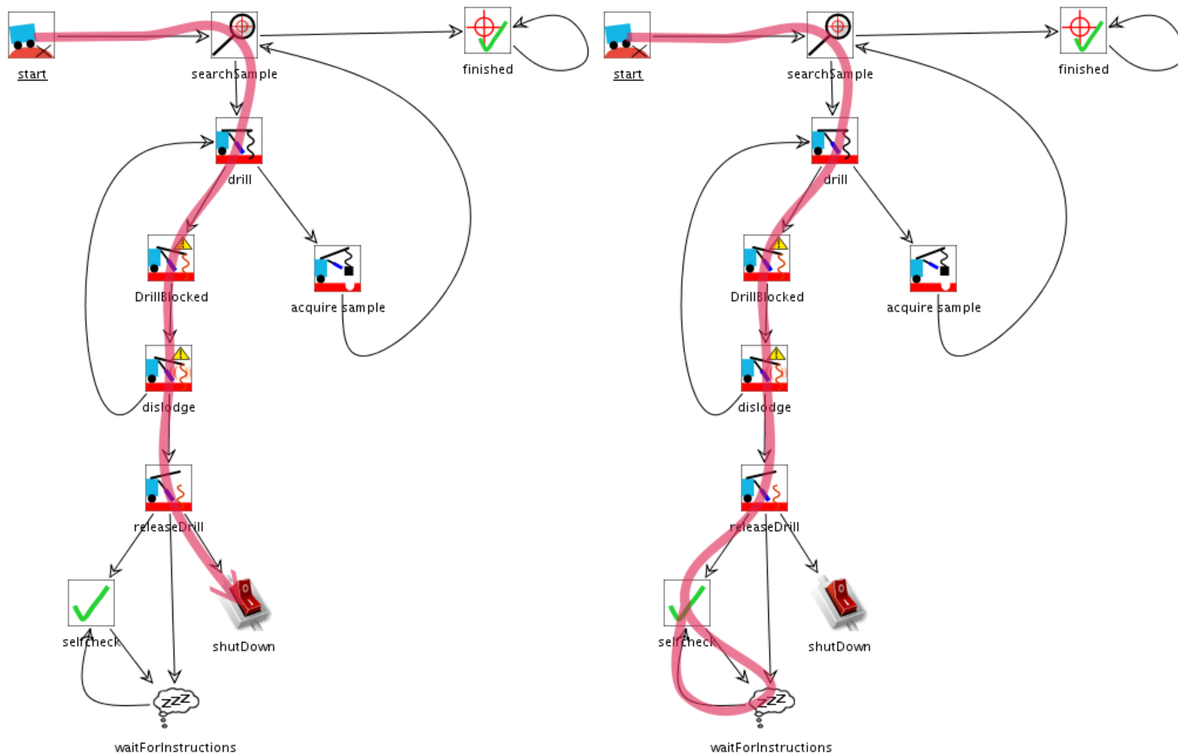

Fig. 12 Two possible system executions preventing the system from always being able to reestablish reference behavior.

We now show how to use the information obtained from playing the game graph in order to adapt the property, and to achieve again a consistence with the current model. We will focus on the original (not extended) version of the model from Fig. 4. However the same process can easily be applied for the extended version from Fig. 11.

6.4 Adaptation

Correcting the mismatch requires a good understanding of both model and property, since the problem lies in their interplay. The system run that is about to lose the drill is (although not desired) still unavoidable for the designers. Accordingly, eliminating this run is not an acceptable option, and instead of modifying the model, we need to act on the property in an adequate way. A typical way of relaxing the property is to tolerate the unavoidable situation (possibly adding some containment properties). In this light, we modify the property to

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

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

But how does this new property emerge from the information contained in the game graph? Recall that square black nodes are of special interest when analyzing the origin of the mismatch between model and property. Furthermore, groups (corresponding to columns in the property-oriented view) of solely black colored nodes indicate a "bigger" mismatch than a mix of black and white nodes. In the left part of Fig. 9 we see that the blocks corresponding to the model nodes DrillTriedToDislodge as well as to DrillBlocked both contain black and white nodes, while the game graph nodes at the bottom (those for node DrillReleased) are all black. The three white and diamond-shaped nodes indicate that it could be possible to win the game against the system run if the system designer could eventually lead the game into a white sink. The game graph node `(d)` contains the disjunctive term

$$\mathsf{EX}(\mathsf{EF}(ReferenceBehavior)) \vee ReferenceBehavior$$

which tells the system designer to be in reference behavior or at least being able to evolve the system into such.

This gives a hint that a disjunctive extension of the specification enables for turning the tide in the game and therefore allows a win. Consequently this change in the game will eliminate the model/property mismatch. Thus the disjunctively extended property

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

emerges.

This property is now satisfied by the model – i.e. it is either possible to reestablish the reference behavior, or the drill is lost. In the corresponding graph (at the right-hand side of Fig. 9) it now becomes apparent that – since there are no winning opportunities for the system owning the square-shaped nodes (which have now turned white) – even 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 Evaluation of case studies

Beside the aerospace scenario described here, the game-based approach has been analyzed in several very different contexts, such as telecommunication systems / platforms, web services, and avionics. These contexts are particularly relevant in the SHADOWS project, where validator partners work in these application domains. In all these areas, self-healing is crucial because these are highly dynamic, open systems that heavily depend on changing environments.

The exemplary case studies taken into consideration are the following.

– *ExoMars Rover* (avionics domain) is the Mars Rover scenario as presented in this article.
– *DDBJ UniProt* (web services domain) is taken from a case study in bio-informatics. We model a process that prepares, executes, combines, and evaluates a database query to several different bio-informatics databases. For a given keyword (e.g. a protein name) the process looks for a DNA sequence entry. From this it creates a table of relationship information including DDBJ Accession No., Protein ID and UniProtID.[2]
– *SWS-Discovery* (web services domain) stems from the discovery part of the Semantics Web Service (SWS) Challenge[3]. From a user request, a service repository is queried for identifying the best matching web service among an available set. The rule-based personalization framework miAmics [15] is

[2] `http://xml.nig.ac.jp/workflow/ddbj_uniprot.html`
[3] `http://www.sws-challenge.org`

System	Property	System States	Transitions	Game Graph Nodes	Edges	Runtime (ms)
ExoMars Rover	It is always possible to recover from erroneous behavior	11	15	143	162	16
DDBJ UniProt	No data that has been extracted remains unused.	16	18	272	292	38
SWS-Discovery	Every opened connection will eventually be closed.	44	52	704	816	65
EmailService	Each successfully identified spam mail will be forwarded to the admin.	27	39	459	544	36
	A requested flyer will finally be sent out.			351	402	50
	A flyer can only be requested if the mail has been checked for spam before.			189	201	22

Table 1 Evaluation of GEAR, applied to problems from avionics, web services, and telecommunication domains.

used to discover the best matching service, based on service information like pricing, shipping etc. [14]

- *EmailService* (telecommunication domain) depicts a service-based approach for processing emails taken from [8]. Each email is handled by a high-level process that abstracts from the underlying email service implementation. The example provides spam redirection to an admin for manual handling. Further messages that contain request information for a brochure are redirected to an appropriate person that is responsible for that request.

Table 1 shows the results of the survey. For each scenario we give a property, the problem size (number of states and transitions of the model), and the size of the game graph built from the model and the desired property. Finally we report GEAR's run time to solve the instance (in milliseconds).

The size of the game graph relates to both the size of the system as well as of the structural complexity of the property to be verified. As we see in the leftmost column, the measured runtimes do not reveal significant peaks. In fact, neither runtime nor memory performance have been an issue for our case studies. All properties have been checked so quickly that the model checker could be used online during the modeling process.

8 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 [23]. 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 [22] 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 a focus on demanding properties.

The model checker SMV [5] allows for symbolic description of systems. Pecheur and Simmons [19] 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 does it assist in diagnosis and adaptation in a graphical manner.

EVALUATOR, the model checking tool of the CADP toolset [10], uses a variant of alternation-free μ-calculus enriched with regular expressions over action sequences for expressing behavioral properties of systems modeled as are labeled transition systems. Both, property and system, are combined into a boolean equation system similar to the game graphs presented here. The authors point at a need for more elaborate diagnostic information for model checking results that go beyond simple yes/no answers. Although the tool does not directly produce such elaborate counter examples, in [18] the authors refer to a technique for generating a sub-graph of the graphical representation of the boolean equation system. This sub-graph contains the minimal amount of information for disproving the property with respect to the system. Unfortunately the authors neither provide a hint on how this information can be transferred back into the system, nor how the information can contribute to an adaptation process like the one presented here.

A different approach for coping with understanding of implementations and specifications are *temporal queries* as introduced by Chan [6]. 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 not cover more demanding properties like the temporal ones we have presented here.

To date, we are not aware of any symbolic approach for game-based verification capable of handling the full modal μ-calculus that allows for handling systems of size comparable to BDD-based approaches.

To our knowledge, game-based model checking as presented here 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.

9 Conclusions

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 collaborative 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 models.

The presented tool GEAR seamlessly integrates into the jABC and enriches the modelling process with elaborate verification results. The proposed approach for game-based model checking and diagnosis helps to interactively explore the system model in a property oriented manner. This new technique allows for a deeper insight of behavioral faults of the model and enables to tighter integrate user groups like system engineers into the verification and validation process. A deeper understanding consequently grounds the upcoming adaptation process and guides the user in performing the right steps. Sometimes bringing the model back in-line with the property specification can even be automated.

The number of states of the presented systems seems at first sight rather small. In our experience, however, the presented examples are realistic (within one or two orders of magnitude of the system's size, which are still well in the range of the technology[4]), and indeed they show that it is possible to debug specifications already at early description stages or at high abstraction. "Doing the right thing" is sometimes wrongly worded and formalized from the very beginning. One needs a drill down in GEAR at the game graph level only for the portions of the system that interact with the property or some parts of it. Narrowing down a problematic case to the diagnosis-level chunk of the system behavior, in fact, usually does not cover very large models.

Moreover, GEAR is not meant to be the sole analysis tool: for finding out that there is a problem in the (big, entire) system, one typically uses a non-graphical, possibly symbolic Model Checker in batch mode (like e.g. the mentioned SMV), and examines only the reported problems with diagnosis-oriented tools (like model checkers with graphical output, like CADP, or GEAR), restricting oneself to the portions that exhibit problems.

The specific charme of the method for industry is that field engineers are able and willing to adopt it, with a low entrance barrier [17]. In fact, the hints in the annotation editor are directly targeted to the engineers, who can proficiently work with such information. Using the advanced view requires understanding the meaning of the two codes (square/diamond and black/white): specification engineers used to formalizing properties have no difficulties with it, and mechanical engineers, trained in system modelling and accustomed to use debugging tools are easily familiarized with the model-oriented presentation, which is still close to the system's model they master. In particular, the debugger-like interaction while playing the game graph is here an asset, both for acceptance of the technique and for understanding and investigating the system's behavior.

Concerning the relation between user interaction and formal methods, we support analysis techniques that run completely automatically and do not require interaction, nor advanced formal methods knowledge, in order to conduct the proofs. This for the moment seems to still exclude theorem provers. In fact, the user interaction presented here is not for the proof: it is playing a game within the case study. This is a Gedankenexperiment with the system under design (here the Rover) that the designers are used to conduct anyway: it concerns making moves on the system's or on the environment's behalf.

In the future we will focus on different views on the game graph that will assist the user with identifying the faulty behavior faster and in a more focused way. Such views, e.g. in connection with dataflow analysis, slicing, or abstract interpretations, can reduce the size of the model to the essential and the number of possible faulty runs, and therefore help focussing on runs that

[4] For some problems related to data-flow analysis via model-checking, the overall problem space grew up to over 100.000 explicit states.

are for instance short or in some other way of specific interest in a particular context.

References

1. Marco Bakera, Tiziana Margaria, Clemens D. Renner, and Bernhard Steffen. Property-driven functional healing: Playing against undesired behavior. In *Proc. CONQUEST 2007, 10th Int. Conf. on Quality Eng. in Softw. Techn.*, 2007.
2. M. Ben-Ari, Amir Pnueli, and Z. Manna. The Temporal Logic of Branching Time. *Acta Informatica*, 20:207–226, 1983.
3. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
4. G. Bormann, L. Joudrier, and K. Kapellos. FORMID: A formal specification and verification Environment for DREAMS. In *Proc. 8th ESA ASTRA Workshop*, 2004.
5. Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *LICS*, pages 428–439. IEEE Computer Society, 1990.
6. William Chan. Temporal-logic queries. In *Proc. CAV*, volume 1855 of *LNCS*, pages 450–463. Springer, 2000.
7. E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs — Proceedings 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Heidelberg, Germany, 1981.
8. Dennis Saßmannshausen. Konzeption und Entwicklung prozessgestützter E-Mail Verarbeitung in serviceorientierten Architekturen. Master's thesis, Dortmund University of Technology, 2007.
9. Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Property specification patterns for finite-state verification. In *FMSP*, pages 7–15. ACM, 1998.
10. Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. CADP 2006: A toolbox for the construction and analysis of distributed processes, November 20 2007.
11. Sven Jörges, Tiziana Margaria, and Bernhard Steffen. FormulaBuilder: A tool for graph-based modelling and generation of formulae. In *Proc. ICSE*, pages 815–818. ACM, 2006.
12. Konstantinos Kapellos. MUROCO-II: FOrmal Robotic Mission Inspection and Debugging. Technical report, European Space Agency, 2005.
13. Dexter Kozen. Results on the propositional μ-calculus. In *ICALP*, volume 140 of *LNCS*, pages 348–359, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.
14. Christian Kubczak, Tiziana Margaria, Bernhard Steffen, and Stefan Naujokat. Service-Oriented Mediation with jETI/jABC: Verification and Export. In *Proc. 2007 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Int. Conf. on Intelligent Agent Technology*, pages 144–147, 2007.
15. Christian Kubczak, Tiziana Margaria, Christian Winkler, and Bernhard Steffen. An Approach to Discovery with miAamics and jABC. In *Proc. 2007 IEEE/WIC/ACM Int. Conf. on Web Intelligence and Int. Conf. on Intelligent Agent Technology*, pages 157–160, 2007.
16. C. Renner B. Steffen M. Bakera, T. Margaria. Verification, diagnosis and adaptation: Tool supported enhancement of the model-driven verification process. isola'07 workshop on formal methods in avionics, space and transport, poitiers (f), dec. 2007. *Revue des Nouvelles Techno-logies de l'Information (RNTI-SM-1)*, pages 85–98, 2007.
17. T. Margaria M. Bakera. The shadows story on implementation, verification and property-guided autonomy for self-healing systems. *ERCIM News N.75, Special theme: Safety-Critical Software*, pages 38–39, October 2008.
18. Radu Mateescu and Mihaela Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Sci. Comput. Program*, 46(3):255–281, 2003.
19. Charles Pecheur and Reid G. Simmons. From Livingstone to SMV. In *FAABS*, volume 1871 of *LNCS*, pages 103–113. Springer, 2000.
20. Onn Shehory, Shmuel Ur, and Tiziana Margaria. Self-healing technologies in SHADOWS: Targeting performance, concurrency and functional aspects. In *Proc. CONQUEST 2007, 10th Int. Conf. on Quality Eng. in Softw. Techn.*, 2007.
21. Bernhard Steffen, Tiziana Margaria, Ralf Nagel, Sven Jörges, and Christian Kubczak. Model-driven development with the jABC. In *Proc. 2nd Haifa Verification Conference*, Haifa, Israel, 2006. Springer.
22. Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng*, 10(2):203–232, 2003.
23. Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *AAAI/IAAI, Vol. 2*, pages 971–978, 1996.