JOURNAL OF AEROSPACE COMPUTING, INFORMATION, AND COMMUNICATION
Vol. 8, April 2011

Game-Based Model Checking for Reliable Autonomy in Space

Marco Bakera[*, †],
University of Potsdam, Potsdam, Germany and TU Dortmund, Dortmund, Germany

Tiziana Margaria[‡]
University of Potsdam, Potsdam, Germany

Clemens D. Renner[*, ‡]
University of Potsdam, Potsdam, Germany and TU Dortmund, Dortmund, Germany

and

Bernhard Steffen[†]
TU Dortmund, Dortmund, Germany

DOI: 10.2514/1.32013

Autonomy is an emerging paradigm for the design and implementation of managed services and systems. Self-managed aspects frequently concern the communication of systems with their environment. Self-management subsystems are critical, they should thus be designed and implemented as high-assurance components. Here, we propose 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 resulting from the games, this way exploring the connection between the property, the system, and the proof. The benefits of the approach are illustrated on a case study that concerns the ExoMars Rover.

I. Motivation

SINCE its inception, a central challenge to the IT industry is the ability to deliver increasingly complex and increasingly reliable systems. Systems that both are composed of complex parts and that provide rich communication capabilities become more and more heterogeneous. Thus the management of such systems is an increasingly critical and daunting problem. The adequate control of the interaction between the subsystems and the environment is one of the main issues in the development of upcoming systems [1–3]. However, the space of such interactions is so huge, that developers of these systems cannot oversee it adequately anymore. Tools can provide enormous help by narrowing down the universe of possible alternatives and decisions to the critical amount a single developer is able to handle. In the past this has proven to be extremely useful in critical areas like hardware design and large telecommunication systems. The usage of tools and techniques that support the development process and guide the

Received 7 May 2007; accepted for publication 15 March 2009.

[*] Chair Service and Software Engineering, University of Potsdam, August-Bebel-Str. 89, Haus 4, 14482 Potsdam, Germany.
[†] Chair Programming Systems, TU Dortmund, Dortmund, Germany.
[‡] Chair Service and Software Engineering, University of Potsdam, August-Bebel-Str. 89, Haus 4, 14482 Potsdam, Germany, margaria@cs.uni-potsdam.de

developer while deciding how to integrate highly reliable and interacting components constitutes an essential help for the outcome of the overall development process.

Dealing with interactions between loosely coupled system components is of critical importance in the context of space systems [4]. These systems are not controlled by humans directly, but rather communicate with a ground control center on Earth that steers the mission. In this situation, the natural demand for autonomy and autonomy-supporting development techniques is evident: ground-controlled systems cannot assist in situations where immediate decisions are required. As an example, the self-protecting local navigation of a spacecraft that examines an extra-terrestrial surface has to avoid particles flying around and endangering the spacecraft. Such kind of reaction has to be carried out by the spacecraft itself and cannot be controlled from the ground control.

In long-distance missions, communication between ground-control and the mission entity is additionally burdened by transmission latencies of several seconds up to tens of minutes. Thus direct communication between ground-control and the spacecraft for reacting to sudden unexpected events becomes unfeasible.

A third issue to consider are disturbances and total breakdowns of communication due to solar storms that affect radio. In these situations the spacecraft has to operate in isolation, until the connection can be reestablished.

All three scenarios have in common that reliable systems—especially systems in space—must be able to rely on themselves. Hence there is a strong need for autonomic decisions and behavior for these systems, especially for long-term and long-distance missions.

As a consequence

1) The interaction of components or whole systems must be managed autonomously, either by *foreseeing* the possible events and designing the system in a way that it is capable of coping with the foreseen events, or by equipping the system with means to tackle the actual challenges independent of specific scenarios. In this second case we speak of *emergent* autonomy.
2) System developers should be guided by tools and techniques to be constantly aware of the critical (autonomy-oriented) properties that the system has to satisfy. Since it can be very hard to anticipate all possible system evolutions, the tools must be able to explain to the developer reasons for their analyzes, in order to ensure a deep understanding of the system and to provide sufficient means for a targeted countermeasure.

In this paper, we show how to use GEAR [5–8], a game-based model checker for the full modal μ-calculus, 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 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 [9],which was designed as part of an European Space Agency (ESA) project. Here we focus on a central property pattern for remote/autonomous (space) systems, with the intuitive meaning that these systems cannot run into situations where they cannot recover, even with ground support from Earth.

Game-based model checking is an established field of logics and theoretical computer science. Lange and Stirling [10,11] 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 [12,13] while first steps in the direction of game-based μ-calculus model checking were already taken by Emerson et al. [14]. There have also been approaches to the synthesis of game strategies by Vöge [15].

However most of the work concentrates on the theoretical questions, like expressivity and complexity, and on the algorithmic or tool-related aspects. In contrast, 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.

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

II. 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 [16]. In the concrete mission example [9], a robot (the *ExoMars Rover*) is sent on a surface mission on Mars where it has to accomplish several tasks, including the

Fig. 1 Three-dimensional model of the ExoMars Rover at the International Aerospace Exhibition ILA 2006.

acquisition of subsurface soil samples using a drill. A three-dimensional model (shown in Fig. 1) [9] was created for simulation purposes. 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 own planning capabilities, which allow 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 as a high-level specification language [17]. 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 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. 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 central property pattern for remote/autonomous (space) systems, with the intuitive meaning that such systems cannot run into situations from which they cannot recover, even with the ground support from Earth. We formalize an alternative formulation of this pattern, which expresses the following:

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

This property pattern is *branching-time* in nature, and cannot be expressed in linear-time logic, as it comprises both

1) a universally quantified part: *Where ever the system evolves to*, and
2) an existentially quantified part: *it is always possible to recover*.

In the following, we first show how we model the ExoMars Rover surface mission in the jABC [18,19], our service-oriented modeling and verification environment, according to the description provided in [9]. Subsequently we discuss in depth the technique of game-based model checking using the properties already considered in the ESA case study.

III. Service-oriented Models of the ExoMars Behavior

The three-tier control model presented in [9] concerns a Mission, a Task, and an Action level.

A. The Mission

The ExoMars Rover 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.

The Rover's surface mission consists of three main phases:

1) in the *Critical Deployment* phase the operational set up is established (solar power acquisition, communication),
2) in the *Egress* phase the Rover leaves the lander dock to start surface operations, and
3) 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 to the Earth.

Figure 2 sketches this high-level behavioral model as it appears in the jABC, our environment for model-driven, service-oriented system design. There, we have modeled 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 modeling matches very closely the intentions of the ExoMars designers: in the original description style, typical of (autonomous) three-tier controlled systems (see Fig. 3), elementary *Actions* constitute re-usable, basic building blocks of behavior. They are organized in libraries and are 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.

We also see in Fig. 3 that central attention is devoted to exception handling: Type 1 events are locally processed in the Action. Type 2 exceptions are treated in the Task, resulting in a switch to a different Action under nominal conditions and a new Task in case of failure. Type 3 exceptions lead to a mission abortion through safety behavior which is context dependent.

Both, exception handling and reaction in the course of autonomous behavior (this is located at Task level) concern behavioral rules and properties that must be ensured reliably. Our work addresses exactly these issues: how to formulate such properties in a declarative way, and how to provide detailed analysis, verification, and diagnostic capabilities in order to check their fulfillment and help repair them at design time.

Recoverability is a mission critical central issue. Tasks are defined as '*a standalone entity that propagates only unrecoverable errors outside its scope*', thus we show here how to formulate and check a recoverability requirement on a task-level behavioral model within the jABC.

In the jABC, we see the executable actions as basic services providing atomic units of behavior, called Service Independent Building blocks (SIBs), organized in sharable collections called SIB palettes. Behavioral models that coordinate actions or tasks express the logic of composite services in terms of flow graphs with fork/join parallelism,

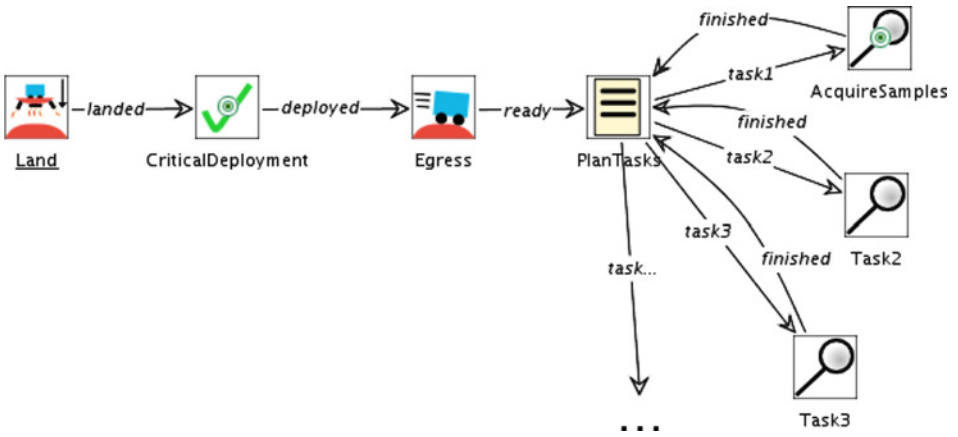

Fig. 2 Service logic graph of the Rover mission's process.

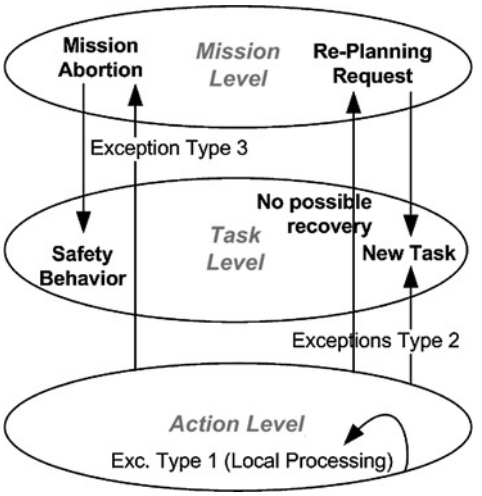

Fig. 3 Three layer abstraction: model layers and exceptions (from [9]).

called Service Logic Graphs (SLGs). Figure 2 shows the Mission SLG, expanded to its top Task level. The interpretation is intuitive: once the rover is successfully *landed* it proceeds to the Egress phase. Once this is completed, it proceeds with one of the operations, for example AcquireSamples. Subsequently, when a Task has been completed, a different Task may be initiated.

In the following we briefly examine the three main tasks.

B. Critical Deployment and Egress

The model of the Critical Deployment task shown in Fig. 4 has been built according to the verbal description of [9]. Once the ExoMars Rover has landed, it performs an initial *Self-Check*. If this is successful, it proceeds to the deployment of solar arrays (subtask *DeploySolarArray*) in order to ensure proper power provisioning, and it deploys an antenna (*DeployAntenna*) used to communicate with the support on Earth. Given that the planetary constellation may impede Rover-ground communications, the Rover may shut down (*Wait*) until a favorable time window occurs. Once communication has been established, the Rover transmits the necessary data collected during Entry, Descent, and Landing along with some information about the preceding deployment (*TransmitDeploymentData*).

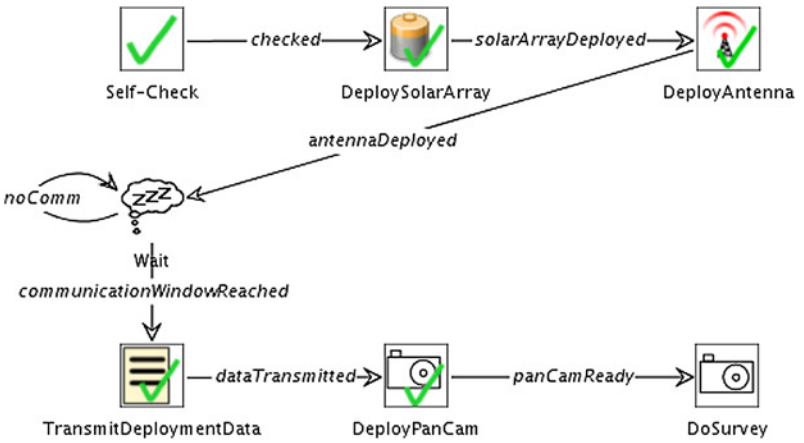

Fig. 4 The Critical Deployment task.

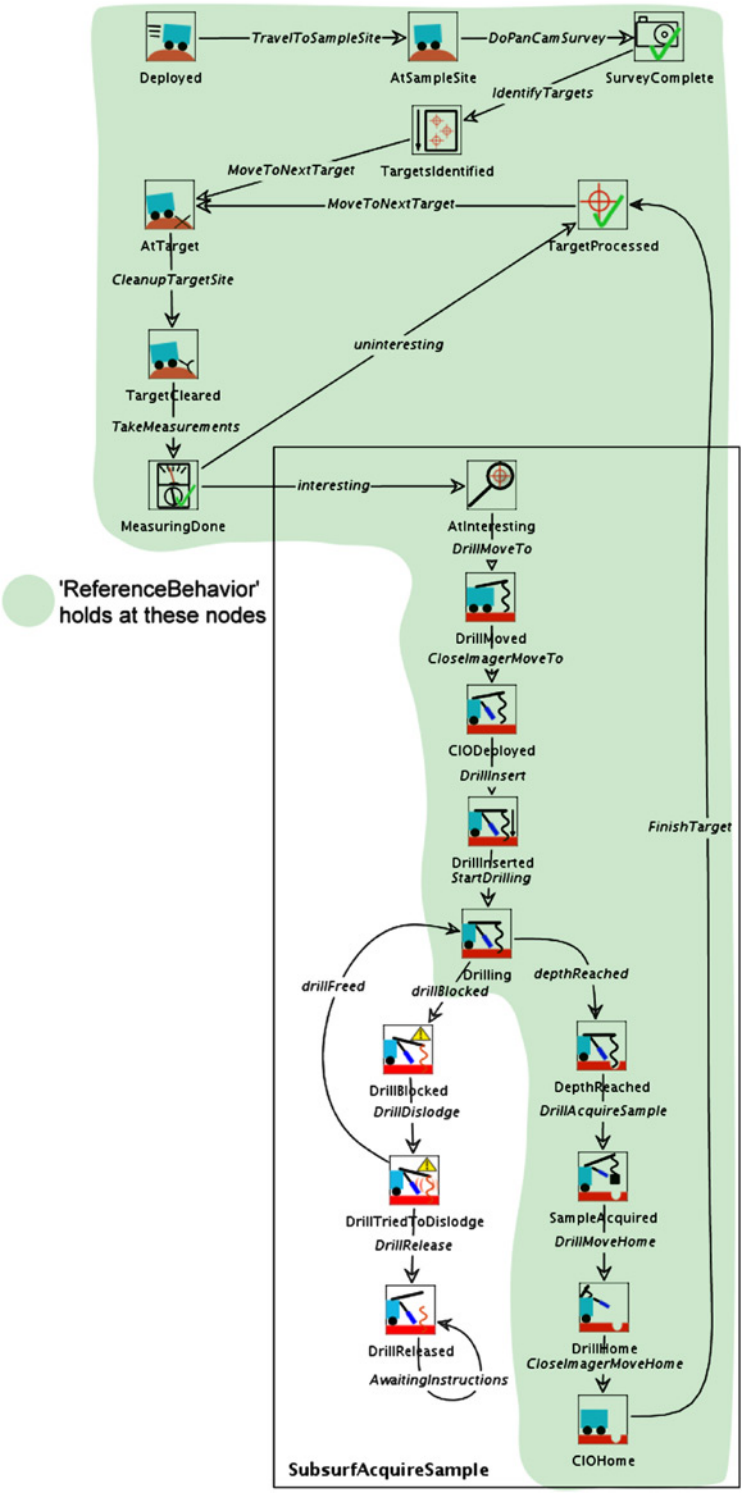

Fig. 5 The acquire samples service logic graph.

The next step is the planning of the actual *Egress Task*. To this aim, a panorama-camera is deployed (*Deploy-PanCam*) and the resulting survey of the surrounding landscape (*DoSurvey*) helps ground personnel to devise a plan for the actual egression. Supporting technology is available to allow the Rover to leave the lander dock safely and unharmed. We have abstained from explicitly modeling the accompanying communication with ground support for the sake of simplicity.

The actual egression from the lander dock (*Egress Task*) is initiated by minor adjusting maneuvers, to bring the Rover in the best possible position for the egress path planned by the ground support on Earth. The Rover leaves the lander dock using this predefined path. Final preparation of Rover equipment, such as the drill subsystem, concludes the Egress Task.

We have decided not to refine the Egress Task and focus instead on the actual core process: the collection of soil samples.

C. Acquire Samples

Figure 5 shows our model of the Acquire Samples surface operations task, intentionally closely resembling the original description from [9].

In this phase, the Rover examines one by one a number of 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 of Fig. 5. The task-level model uses the following actions (elementary SIBs):

1) *DrillMoveTo, CloseImagerMoveTo*—Drill and Close Image Observer are moved into the correct positions for the upcoming sample extraction.
2) *DrillInsert, StartDrilling*—The drill is inserted into the ground and brought into operation.
3) *depthReached*—This guard expresses an event relevant for the drill control.
4) *DrillAcquireSample*—Acquires a rock sample for later analysis.
5) *DrillMoveHome, CloseImagerMoveHome*—Drill and Close Image Observer are moved back into their 'home' positions.
6) *FinishTarget*—The Rover prepares to move on to the next target.

During a sample extraction the regular behavior of the Drill might be disturbed. This is captured in the left branch of the box in Fig. 5:

1) *drillBlocked*—The Drill gets stuck.
2) *DrillDislodge, drillFreed*—If the drill is blocked, attempts are made to dislodge it, freeing it again so that the drilling can continue.
3) *DrillRelease, AwaitingInstructions*—If the drill is blocked despite all dislodging attempts, the drill is released and the Rover awaits new instructions from ground support.

IV. Verification with Games

The model used for the case study has been expressed as a Kripke Transition System [20], formalized in the following definition.

Definition 1 (Kripke Transition System):
A Kripke Transition System K is defined as a tuple $(S, \mathsf{Act}, \rightarrow, I)$ over a set of atomic propositions AP, disjoint from Act, where

1) S are the states of the model;
2) Act is a set of actions;
3) $\rightarrow \subseteq S \times \mathsf{Act} \times S$ are the possible transitions in the model; and
4) a labeling interpretation function $I : S \rightarrow 2^{\mathsf{AP}}$ equips states with atomic propositions.

In the following we will use Computation Tree Logic (CTL) [14] to express desired system properties.

Definition 2 (Syntax of CTL):
For $p \in \mathsf{AP}$, the set of CTL formulas is defined by:

$$\phi ::= p | \neg\phi | \phi \vee \phi | \mathsf{EX}[\phi] | \mathsf{E}[\phi \,\mathsf{U}\, \phi] | \mathsf{A}[\phi \,\mathsf{U}\, \phi]$$

The semantics of CTL are defined using paths in a Kripke Transition System. These paths represent the possible execution traces of the system.

Definition 3 (Path):
A path is a sequence of states $s_0, s_1, s_2, \ldots$, such that $(s_i, a, s_{i+1}) \in \rightarrow$ for some $a \in \mathsf{Act}$. S^ω denotes the set of all paths. We refer to the $(i+1)$th state of a path $\pi \in S^\omega$ as $\pi[i]$.

$P_K(s) = \{\pi \in S^\omega | \pi[0] = s\}$ denotes the set of paths starting from state s in the Kripke Transition System K.

This allows us to specify the semantics of basic CTL operators as follows.

Definition 4 (Semantics of CTL):
Let $K = (S, \mathsf{Act}, \rightarrow, I)$ be a Kripke Transition System over atomic propositions AP, $p \in \mathsf{AP}$ be an atomic proposition, $s \in S$ be a state, and ϕ, ψ be CTL formulas. The satisfaction relation $\models$ is defined by

$$\begin{array}{ll}
s \models p & \text{iff } p \in I(s) \\
s \models \neg\phi & \text{iff } s \models \phi \text{ does not hold} \\
s \models \phi \vee \psi & \text{iff } s \models \phi \text{ or } s \models \psi \\
s \models \mathsf{EX}[\phi] & \text{iff } \exists \pi \in P_K(s).\pi[1] \models \phi \\
s \models \mathsf{E}[\phi U \psi] & \text{iff } \exists \pi \in P_K(s).\exists j \geqslant 0.\pi[j] \models \psi \wedge \forall 0 \leqslant k < j.\pi[k] \models \phi \\
s \models \mathsf{A}[\phi U \psi] & \text{iff } \forall \pi \in P_K(s).\exists j \geqslant 0.\pi[j] \models \psi \wedge \forall 0 \leqslant k < j.\pi[k] \models \phi
\end{array}$$

As usual, we can derive the following (dual) operators.

1) $\phi \Rightarrow \psi \equiv \neg\phi \vee \psi$ and $\phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$
2) $\mathsf{AX}[\phi] \equiv \neg\mathsf{EX}[\neg\phi]$
3) $\mathsf{EF}[\phi] \equiv \mathsf{E}[\text{true}\,\mathsf{U}\,\phi]$ and $\mathsf{AF}[\phi] \equiv \mathsf{A}[\text{true}\,\mathsf{U}\,\phi]$
4) $\mathsf{EG}[\phi] \equiv \neg\mathsf{AF}[\neg\phi]$ and $\mathsf{AG}[\phi] \equiv \neg\mathsf{EF}[\neg\phi]$

Structurally, CTL operators consist of a *path quantifier* that specifies whether we are interested in at least one possible execution trace (E) or every possible execution trace (A), and a *path formula* describing the behavior along paths.

1) X (next-time) talks about the next state
2) F (finally) talks about a state that is reached eventually
3) G (generally) talks about the entire path
4) $\phi \,\mathsf{U}\, \psi$ (until) requires ϕ to hold until ψ holds at some future state, or ψ may also hold immediately

Example: If we intend to expose a satellite in a geostationary (or geosynchronous) orbit, the corresponding model concerns navigation and stabilization at a Lagrangian point. For this purpose the corresponding model concerns should have an atomic proposition *lagrangePointReached* characterizing the states where the satellite is at a Lagrange point. (Lagrange points are points of zero gravity in two-body systems where small objects like satellites are theoretically stationary (if only gravity is considered).) A required property for a mission could be "*The satellite is guaranteed to reach the Lagrange Point*", expressed as the temporal property

$$\mathsf{AF}[\text{lagrangePointReached}]$$

A. Games

In this section, we sketch the basic principles of game-based model checking. An application of this technique to the Mars rover example is shown in the next section.

In general, model checking is used to decide whether an abstraction of a reactive system, modeled, e.g., using a transition system or a Kripke structure, satisfies a requirement, specified, e.g., using temporal logics. In the case of failure, typically error paths are provided as diagnostic information. This is unfortunately not possible, as soon as branching-time properties are considered, as their violation cannot be explained in terms of paths. Rather, the diagnostic information has to be generalized to winning strategies of parity games.

Parity games are played by two players, both having complete information—Go or Chess are examples of such games. They can be used for game-based model checking as introduced in [14], which is available for the full modal μ-calculus [21] and thus also applicable to, e.g., CTL and CTL*, which are expressible in μ-calculus.

In a parity game, the *game graph* derived from a model has *game-graph nodes* partitioned in two sets, one per player. Whenever the game reaches a game-graph node, the player who "owns" that game-graph node has to move to another game-graph node—otherwise he loses the game. Formally, we have the following definitions.

Definition 5 (Game graph):
A game graph G is a tuple $(V_\Box, V_\Diamond, E, p)$ where

1) V is a set of nodes
2) V is partitioned in two disjoint sets $V_\Box$ and $V_\Diamond$
3) $E \subseteq V \times V$ is a set of edges and
4) $p : V \to \{0, \ldots, d-1\}$ is a function (for some natural d).

The priority function p is needed to determine the winner of the game if the game becomes cyclic; i.e., whenever the same state will be reached a second time (cf. Definition 7). (Technically, p is used to keep track of fixed-point alternation. By intuition, the priority of a game-graph node increases if the fixed-point type changes (from least to greatest or vice versa) when traversing the formula, starting with 0 for least and with 1 for greatest fixed-points, respectively.)

Definition 6 (Game):
A game consists of a game graph $G = (V_\Box, V_\Diamond, E, p)$ with some start node $n \in V$ played by two players $\Diamond$ and $\Box$ according to the following rule:

Player $i \in \{\Box, \Diamond\}$ chooses some successor w of the current node v such that $(v, w) \in E$ and $v \in V_i$.

Definition 7 (Winning condition):
Player $\Diamond$ (the Prover) wins when one of the following conditions is met:

(S) A $\Box$-sink is reached.
(C) The game becomes cyclic—i.e., a game-graph node is reached twice—and the least priority appearing in the cycle is even.

Otherwise player $\Box$ (the Disprover) wins.

Definition 8 (Winning set):
A set of game-graph nodes W_i is a winning set for player i with $i \in \{\Box, \Diamond\}$ if and only if i can win any game starting in a node $v \in W_i$, regardless of the other player's moves.

We will now show how the model-checking problem relates to parity games. In the following, the $\Diamond$-player acts as the prover while the $\Box$-player acts as the disprover of the considered property. (Other authors sometimes call the players $\vee$-player/0-player/*E*loise (for the prover), suggesting an existential nature of its behavior and $\wedge$-player/1-player/*A*belard (for the disprover), suggesting an all-quantified behavior.)

B. Relation Between Games and Model Checking

Game graphs originate from a cross-product of the property and the model. Each game-graph node represents a pair $(\mathbf{s}, \phi)$, where $\mathbf{s}$ is a node from the model and ϕ is a subformula of the property considered. If the $\Diamond$-player

can win from a given game-graph node, the node belongs to his winning set and because he acts as the prover, this implies $\mathbf{s} \models \phi$.

$V_\Diamond$ contains all the game-graph nodes $(\mathbf{s}, \phi)$, where ϕ has disjunctive characteristic, i.e., its outermost operator is $\vee$, EX, or μ. Dually, $V_\Box$ contains those game-graph nodes where ϕ has conjunctive characteristic, i.e., $\wedge$, AX, or ν.

Game-graph nodes where ϕ is an atomic proposition are sinks. They belong to $V_\Diamond$ if $s \models \phi$. Otherwise the game-graph node belongs to $V_\Box$.

Edges in a game graph reveal *dependencies* between (sub)formulas and model nodes in the model-checking problem. We distinguish three types of subformulas ϕ:

1) *Model-independent* (*propositional*) *formulas* are of the form $\phi' \wedge \phi''$ or $\phi' \vee \phi''$. Their evaluation does not depend on the model but solely on the formula: to evaluate it, it suffices to evaluate its subformulas ϕ' and ϕ'' at the unchanged model node. Outgoing edges from this type of game-graph nodes target game-graph nodes $(\mathbf{s}, \phi')$ and $(\mathbf{s}, \phi'')$: the edge's source and target game-graph node share the same model node $\mathbf{s}$.
2) *Model-dependent subformulas* are of the form $\mathsf{EX}\phi'$ or $\mathsf{AX}\phi'$. They depend on the valuation of the subformula ϕ' at the succeeding nodes in the model. Edges leaving such game-graph nodes lead to game-graph nodes of the form $(\mathbf{s'}, \phi')$ if and only if there is in the model an edge from s to s'.
3) *Fixpoint subformulas* $\sigma X.\phi'(X)$—with $\sigma \in \{\mu, \nu\}$—are recursive, thus in the game graph they are unfolded into $\phi'[\sigma X.\phi'(X)/X]$ due to their self-referencing nature ($\phi[x/t]$ denotes syntactic substitution, i.e., t is replaced by x in ϕ). For this reason, two edges are inserted: one from nodes of the form $(\mathbf{s}, \sigma X.\phi'(X))$ to $(\mathbf{s}, \phi'(X))$, and one from $(\mathbf{s}.X)$ back to $(\mathbf{s}, \sigma X.\phi'(X))$.

Winning Conditions: By intuition, the Prover (Player $\Diamond$) is played by the user who attempts to verify the property. Thus, he decides where to go in the game graph at disjunctive game-graph nodes ($\vee$, EX, and μ) while the Disprover (Player $\Box$), played by the model checker, moves at conjunctive game-graph nodes ($\wedge$, AX, and ν).

1) If the game ends in a sink in the game graph and the player that should move is unable to do so, he loses the game (according to winning condition (S) in Definition 7).
2) If the game becomes cyclic, game-graph nodes (a game situation) are reached twice or more. Cycles in the game graph arise due to fixed-points in the formula. For the evaluation the type (least or greatest) of the outermost fixed-point of the formula is crucial. Cycles in greatest fixed-points, which very much resemble an infinite conjunction, belong to the $\Box$-player and are won by the $\Diamond$-player, who, on the other hand, loses in case of minimal fixed-points (winning condition (C) in Definition 7).

The following section illustrates the use of strategies of parity games for diagnostics, explaining both validity and property violation.

V. Game-based Model Checking the ExoMars Rover

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.

In the model of Fig. 5, the green (darker) background indicates the portion describing the desired *Reference Behavior*. Formally this is expressed by equipping all the green node with the atomic proposition Ref, which allows us to formalize the following important requirement:

$$\mathsf{AG}[\mathsf{EF}\ \mathrm{Ref}]$$

meaning that, however, the system evolves (the AG part), it is always possible to reestablish the reference behavior (the EF part).

In order to understand the following analysis, we need the following fixed-point characterizations:

$$\mathsf{AG}[\mathsf{EF}\ \mathrm{Ref}] =_\nu \mathsf{EF}[\mathrm{Ref}] \wedge \mathsf{AX}[\mathsf{AG}[\mathsf{EF}\ \mathrm{Ref}]] \tag{1}$$

and

$$\mathsf{EF}\ \mathrm{Ref} =_\mu \mathrm{Ref}\ \vee \mathsf{EX}[\mathsf{EF}\ \mathrm{Ref}]$$

where $=_\mu$ / $=_\nu$ denotes the least/greatest fixed-point of the equation, respectively.

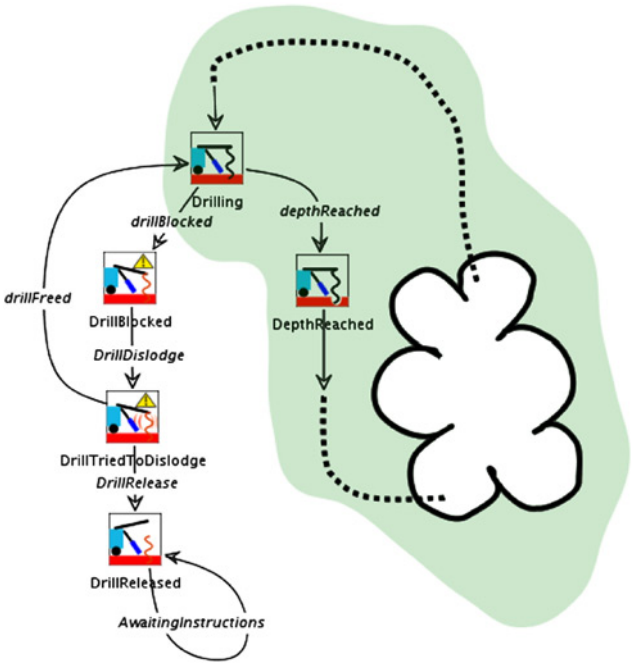

Fig. 6 Relevant part of the analyzed model.

Remark: In general, the first characterization would be AG[EF Ref] $=_\nu$ EF[Ref] $\wedge$ AX[AG[EF Ref]] $\wedge$ EX[true]. However, the subformula EX[true] can be safely omitted because there are no sinks in the analyzed model.

Figure 6 shows the relevant part of the analyzed Acquire Samples Task model. Nodes that are not relevant for the following discussion are abstractly represented by the cloud.

Let us now investigate the relevant part of the game graph shown in Fig. 7, which uses the following convention:

- ◇ nodes belong to the ◇-player (since diamond-shaped) and are in its winning set (since white).
- ◆ nodes belong to the ◇-player (diamond-shaped) but are in the winning set of the □-player (black).
- □ nodes belong to the □-player (box-shaped) but are in the winning set of the ◇-player (white).
- ■ nodes belong to the □-player (box-shaped) and are in its winning set (black).

The game we consider starts at node **DrillBlocked** since this is the node where the system has left the reference behavior. It is played by the following two players:

1) The Prover (◇ player) is convinced that AG[EF Ref] holds,
2) the Disprover (□ player) states that AG[EF Ref] does not hold.

They start playing at the game-graph node (**DrillBlocked**, AG[EF Ref]), the top node (a) in Fig. 7. The game develops as follows:

1) From (a) □ can only move to the next node, EF Ref $\wedge$ AX[AG[EF Ref]] (b). At (b) there are two alternatives (c) and (c′). As (c′) belongs to the winning set of ◇, □ chooses (c) as his next move.
   *At **DrillBlocked*** AG[EF Ref] *does not hold because the fixed-point characterization of the property is not satisfied. In fact, referring to Equation (1), at least one subformula* EF Ref *or* AX[AG[EF Ref]] *is not satisfied—in this case the latter.*
2) Now □ moves to the only successor (d).
   To disprove AX[AG[EF Ref]] *at node **DrillBlocked**, it is necessary to prove that* AG[EF Ref] *does not hold for at least one successor. The only successor to state **DrillBlocked** is state **DrillTriedToDislodge**, as shown in Fig.* 6.
3) Now the only possible successor is (e). There, the [ ] player has an important decision. If he picks the wrong successor (f′), he would lose. Therefore he picks (f).
   *As seen for game-graph node (a), we unfold the fixed-point, in this case for **DrillTriedToDislodge**.* EF Ref $\wedge$

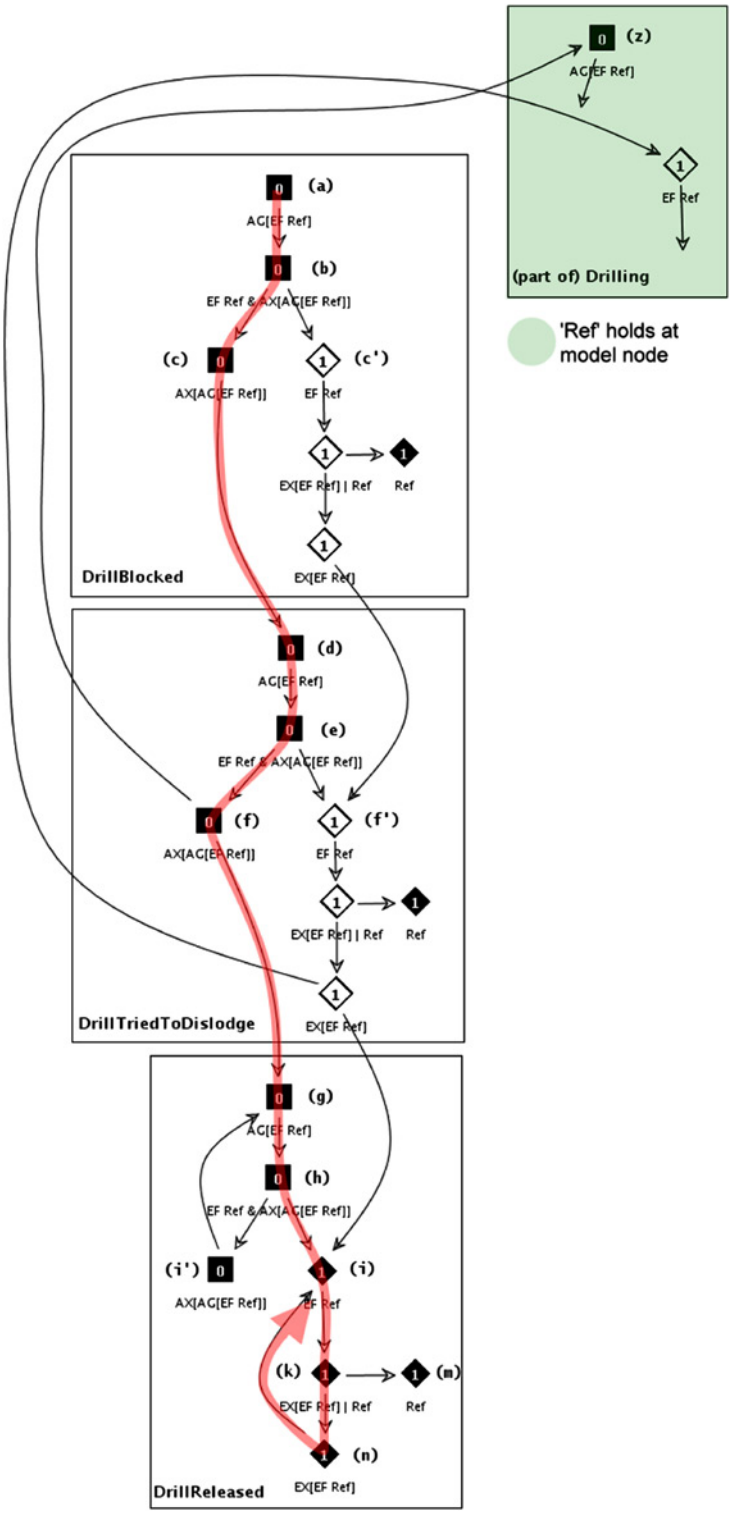

Fig. 7 The relevant part of the game graph.

AX[AG[EF Ref]] *can only be falsified via subformula* AX[EG[AF Ref]] *at this model node. The other subformula* EF Ref *holds at* ***Drilling****, as it is itself part of the reference behavior. Thus the reference behavior can be re-established from* ***DrillTriedToDislodge***

4) Now □ has to decide whether to go to (g) or (z). Since he has no reason for preference, he chooses (g).
To falsify AX[AG[EF Ref]] *it is sufficient to find one successor for which* AG[EF Ref] *does not hold. This is true for both* ***Drilling*** *and* ***DrillReleased****.*

5) As before □ steps down to (h). Now, choosing (i′) would force him to close a cycle with even priority (g, h, i′), and therefore to lose according to winning condition (C) (Definition 7). Therefore he picks (i) and turns over the game to ◇ for the first time.
To falsify EF Ref ∧ AX[AG[EF Ref]] *it is sufficient to disprove one subformula as mentioned earlier. However, since there is no finite counter example for* AX[AG[EF Ref]] *starting at node* ***DrillReleased*** *it is not possible to disprove this subformula.*

6) The only possible choice for ◇ at (i) is to move on to (k). From there ◇ can only move to (m) or (n). He will lose if he picks (m) according to winning condition (S). However, once he has moved down to (n) his only choice is going back to (i) where he loses due to winning condition (C).
To prove that EF[Ref] *holds at* ***DrillReleased*** *it is necessary to establish a finite path back to the reference behavior. However, since the state does not belong to the reference behavior and only has a reflexive edge this is impossible.*

This shows that

$$\mathsf{AG}[\mathsf{EF}\ \mathrm{Ref}]$$

does not hold: there are system evolutions where no way leads back to the reference behavior. More precisely, the execution where the drill is eventually lost when all the dislodge attempts fail does not allow any way back to the reference behavior.

As this particular execution cannot be excluded, there is essentially one natural way for adapting the property specification:

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

which tolerates this unavoidable situation, but maintains the original intent otherwise. It turns out that the revised property holds of the system. Even better, investigating the corresponding game graph it becomes apparent that the following stronger property is valid:

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

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

VI. Conclusion

In this paper, we have shown 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 strategies 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 error diagnostics in the case of failure.

We have illustrated the power of our game-based model checker GEAR on a central property pattern for remote/autonomous (space) systems, with the intuitive meaning that systems cannot run into situations from which they cannot recover, even with ground support from Earth. This property pattern 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 autonomicity controller is included in the modeling. We are planning to extend our model checking scenario along these lines.

Acknowledgments

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.

References

[1] IBM. "Autonomic computing: IBM's Perspective on the State of Information Technology," TR, IBM, 2001.

[2] Kephart, J. O., and Chess, D. M., "The vision of Autonomic Computing," TR, IBM, 2003.

[3] Hinchey, M. G., and Sterritt, R., "99% (Biological) Inspiration," *19th World Computer Congress, 1st International Conference on Biologically Inspired Computing*, Santiago, Chile, edited by Pan, Y., Rammig, F. J., Schmeck, H., and Solar, M., Vol. 216, Biologically Inspired Cooperative Computing, Springer, 21–24 August 2006, pp. 7–20, http://dblp.uni-trier.de/db/conf/ifip10/bicc2006.html#HincheyS06

[4] Curtis, S. A., "Ants for the Human Exploration and Development of Space," *Proceedings of the IEEE Aerospace Conference*, 2003, Link zu den Proceedings, http://ieeexplore.ieee.org/xpl/RecentCon.jsp?punumber=8735

[5] Bakera, M., Margaria, T., Renner, C. D., and Steffen, B., "Property-driven Functional Healing: Playing Against Undesired Behavior," *Proceedings of the CONQUEST 2007, 10th International Conference on Quality Engineering in Software Technology*, Potsdam, 26–28 September 2007, Dpunkt Verlag GmbH.

[6] Bakera, M., Margaria, T., Renner, C. D., and Steffen, B., "Model Checking with the jABC Framework—From METAGame to GEAR," *QEES'08, International Symposium on Quality Engineering for Embedded Systems*—In Conjunction with the ECMDA 2008, Berlin (D), Fraunhofer IRB Verlag, June 2008, pp. 45–62.

[7] Bakera, M., Margaria, T., Renner, C. D., and Steffen, B., "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)*, December 2007. Extended version in ISSE, Innovations in Systems and Software Engineering—A NASA Journal, in print.
doi: 10.1007/s11334-009-0091-6

[8] GEAR, "A Game-based Model Checking Tool," http://jabc.de/gear [retrieved 19 June 2008].

[9] Kapellos, K., "MUROCO-II: FOrmal Robotic Mission Inspection and Debugging," TR, European Space Agency, 2005.

[10] Stirling, C., "Local Model Checking Games," *Lecture Notes in Computer Science*, Vol. 962, 1995, pp. 1–11.

[11] Lange, M., and Stirling, C., "Model Checking Games for CTL," *Proceedings of the International Conference on Temporal Logic, ICTL 2000*, Leipzig, Germany, October 2000.

[12] Müller-Olm, M., and Yoo, H., "Metagame: An Animation Tool for Model-Checking Games," *Proceedings of the TACAS 2004, 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Barcelona, Spain, edited by K. Jensen and A. Podelski, Vol. 2988, Lecture Notes in Computer Science, Springer-Verlag, 29 March–2 April 2004, pp. 163–167.

[13] Yoo, H., "Fehlerdiagnose beim Model-Checking durch animierte Strategie-Synthese," Ph.D. Dissertation, TU Dortmund, 2007.

[14] Emerson, E. A., Jutla, C. S., and Sistla, A. P., "On Model-checking for Fragments of μ-Calculus," *Proceedings of the 5th International Conference, CAV'93*, Elounda, Greece, edited by Courcoubetis, C., Vol. 697, Lecture Notes in Computer Science, Springer, 28 June–1 July 1993, pp. 385–396, http://dblp.uni-trier.de/db/conf/cav/cav93.html#EmersonJS93

[15] Vöge, J., "Strategiesynthese für Paritätsspiele auf endlichen Graphen," PhD Thesis, RWTH Aachen, 2000.

[16] Bormann, G., Joudrier, L., and Kapellos, K., "FORMID: A Formal Specification and Verification Environment for DREAMS," *Proceedings of the 8th ESA Workshop on Advanced Space Technologies for Robotics and Automation ASTRA*, Noordwijk, Netherlands, 2004.

[17] Berry, G., and Gonthier, G., The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, 1992, pp. 87–152.
doi: 10.1016/0167-6423(92)90005-V

[18] Steffen, B., Margaria, T., Nagel, R., Jörges, S., and Kubczak, C., "Model-driven Development with the jABC," *Proceedings of the HVC'06, IBM Haifa Verification Conference*, Haifa, Israel, Vol. 4383, Lecture Notes in Computer Science, Springer-Verlag, October 2006, pp. 92–108.

[19] Margaria, T., and Steffen, B., "Lightweight Coarse-grained Coordination: A Scalable System-level Approach," *International Journal on Software Tools for Technology Transfer*, Vol. 5, No. 2–3, pp. 107–123.

[20] Müller-Olm, M., and Schmidt, D. A., and Steffen, B., "Model-Checking: A Tutorial Introduction," *Proceedings of the Static Analysis, 6th International Symposium, SAS '99*, Venice, Italy, Vol. 1694, Lecture Notes in Computer Science, Springer-Verlag, 22–24 September 1999, pp. 330–354.

[21] Kozen, D., "Results on the Propositional μ-Calculus," *Automata, Languages and Programming, 9th Colloquium*, edited by M. Nielsen and E. M. Schmidt, Vol. 140, Lecture Notes in Computer Science, Aarhus, Denmark, 12Ž01316, July 1982. Springer-Verlag, pp. 348–359.
doi: 10.1007/BFb0012782

Roy Sterritt
Associate Editor