Component-Oriented Behavior Extraction for Autonomic System Design [*]

Marco Bakera, Christian Wagner, Tiziana Margaria
University of Potsdam, Germany
bakera,wagner,margaria@cs.uni-potsdam.de

Emil Vassev
Lero, University College Dublin, Ireland
emil.vassev@lero.ie

Mike Hinchey
Lero, University of Limerick, Ireland
mike.hinchey@lero.ie

Bernhard Steffen
TU Dortmund, Germany
steffen@cs.tu-dortmund.de

Abstract

Rich and multifaceted domain specific specification languages like the Autonomic System Specification Language (ASSL) help to design reliable systems with self-healing capabilities. The GEAR game-based Model Checker has been used successfully to investigate properties of the ESA ExoMars Rover in depth. We show here how to enable GEAR's game-based verification techniques for ASSL via systematic model extraction from a behavioral subset of the language, and illustrate it on a description of the Voyager II space mission.

1 Introduction

The SHADOWS project (**S**elf-**h**ealing **A**pproach to **D**esigning **C**o**m**plex Soft**w**are **S**ystems) [14, 15] aims at developing technologies that augment large software systems with a sort of immune response against various issues and contingencies that can occur at design-time or runtime. Focusing on functional healing at design time, we developed a number of enabling techniques for functional self-healing. In particular, we introduced game based model checking of behavioral models in the GEAR tool [1, 2] as a deep diagnosis tool for early realignment between behavioral models and requirements expressed as temporal properties that we applied to the analysis of the recovery behavior of the ESA ExoMars Rover.

In this paper we show 1) how we are able to link the behavioral modelling style of our techniques with ASSL [17], a rich domain-specific language for the specification of autonomous systems, equipped with a formal semantics [17], and 2) how we can easily and systematically translate (parts of) the specification of the Voyager's behavior into Service Logic Graphs (SLGs, introduced formally in Section 3.1), thus enabling the application of the SHADOWS technologies to the large class of autonomous systems describable in ASSL. The advantage of SLGs over other models is that they are closer to the field engineer's understanding, thus making advanced game-based diagnosis features accessible to non-experts in formal methods and models.

1.1 The NASA Voyager Mission Case Study

The NASA Voyager Mission started in 1977 and was designed for exploration of the outer planets of the Solar System. As the twin spacecraft Voyager I and Voyager II flew, they took pictures of planets and their satellites in 800x800 pixel resolution, then radiotransmitting them to Earth. Voyager II has two on-board television cameras - one for wide-angle images and one for narrow-angle images - that record images in black and white. Each camera is equipped with a set of colour filters, which help images to be reconstructed as fully-colored ones. Voyager II uses radar-like microwave frequencies to send the stream of pixels toward Earth. The signal suffers on this distance a 20 billion times attenuation [4]. In Vassev and Hinchey [18], the mission is specified as an autonomic system composed of the Voyager II spacecraft and four antennas on Earth, all specified as distinct autonomic elements. This paper is based on this specification and on results regarding the behavior of the system.

E. Denney, D. Giannakopoulou, C.S. Păsăreanu (eds.); The First NASA Formal Methods Symposium, pp. 66-75

[*]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`.

In the rest of this paper, we briefly sketch ASSL (Sect. 2) and then how to map the ASSL specification with our models (Sect. 3), and illustrate it on the model for the NASA Voyager mission. We then discuss verification issues (Sect. 4), related work (Sect. 5), and finally conclude (Sect. 6).

2 ASSL

The Autonomic System Specification Language (ASSL) is a framework that provides a multi-tier structure for specifying and validating autonomic systems and targets the generation of an operational prototyping model for any valid ASSL specification [17]. ASSL provides a multi-tier specification model that tackles autonomic systems (ASs) as composed of autonomic elements (AEs) interacting over interaction protocols (ASIP and AEIP). We concentrate here on the behavioral aspects of the AS and AE description, since they are the part of ASSL that finds direct counterpart in the GEAR behavioral models.

- The AS tier - provides a general and global AS perspective. It defines the general system rules in terms of *service-level objectives (SLO)* and *self-management policies*, *architecture topology*, and global *actions*, *events*, and *metrics* applied in these rules. It is similar to the mission and task level of the ExoMars description.
- the AE tier - provides a unit-level perspective, It defines interacting sets of individual autonomic elements (AEs) with their own behavior. This tier is composed of AE rules (*SLO* and *self-management policies*), an *AE interaction protocol (AEIP)*, *AE actions*, *AE events*, and *AE metrics*. It is similar to the Action level of the ExoMars description.

2.1 How the Voyager Takes Pictures

When a space picture must be taken and sent to Earth, the Voyager exhibits autonomous-specific behavior. The spacecraft must detect interesting objects on-the-fly and take their pictures. This reveals a sort of autonomic event-driven behavior that can be easily specified with ASSL at the three main tiers - AS (autonomic system) tier, ASIP (autonomic system specification protocol) tier, and AE (autonomic element) tier [17]. The Voyager II spacecraft and the antennas on Earth are specified at both AS and AE tiers as autonomic elements that follow their autonomic behavior encoded as a self-management policy called `IMAGE_PROCESSING`. ASSL specifies self-management policies with fluents[1], denoting specific system states, and mappings, that map fluents to ASSL actions, i.e. actions to be performed when the system gets into a fluent [17].

2.2 AS Tier Specification.

The `IMAGE_PROCESSING` self-management policy is specified at the AS tier to process images from four antennas on Earth located in Australia, Japan, California, and Spain. In fact, we consider this specification as forming the autonomic image-processing behavior of the Voyager Mission base on Earth.

As shown in Figure 1, the policy is specified with four policy *fluents*—one per antenna. Fluents denote specific system states. They are *initiated* by events prompted when an image has been received and *terminated* by events prompted when the received image has been processed. Further, all the four fluents are *mapped* to an ASSL *action*: that is to be performed when the system enters in one of the fluents. Figure 2 shows the specification of the events that initiate and terminate the fluent presented by Figure 1. Note that the first event is prompted to occur in the system when a special message has been received. In addition, a `processImage` action (see [18] for this action's specification) is specified to process images from all four antennas.

At the autonomic system interaction protocol (ASIP) tier, the image messages (one per antenna), a communication channel that is used to communicate these messages, and communication functions to send and receive these messages over that communication channel to the Earth are specified [18].

[1]ASSL adopts some AI-planning terminology: a fluent is comparable to a state variable in our transition system view.

```
FLUENT inProcessingImage_AntSpain {
  INITIATED_BY { EVENTS.imageAntSpainReceived }
  TERMINATED_BY { EVENTS.imageAntSpainProcessed }}

MAPPING {
  CONDITIONS { inProcessingImage_AntSpain}
  DO_ACTIONS { ACTIONS.processImage("Antenna_Spain") }}
```

Figure 1: An IMAGE_PROCESSING Fluent

```
EVENT imageAntSpainReceived {
  ACTIVATION { RECEIVED {
    ASIP.MESSAGES.msgImageAntSpain } }}

EVENT imageAntSpainProcessed { }
```

Figure 2: AS-tier Events

```
AESELF_MANAGEMENT {
  OTHER_POLICIES {
    POLICY IMAGE_PROCESSING {
      FLUENT inTakingPicture {
        INITIATED_BY { EVENTS.timeToTakePicture }
        TERMINATED_BY { EVENTS.pictureTaken }
      }
      FLUENT inProcessingPicturePixels {
        INITIATED_BY { EVENTS.pictureTaken }
        TERMINATED_BY { EVENTS.pictureProcessed }
      }
      MAPPING {
        CONDITIONS { inTakingPicture }
        DO_ACTIONS { ACTIONS.takePicture }
      }
      MAPPING {
        CONDITIONS { inProcessingPicturePixels }
        DO_ACTIONS { ACTIONS.processPicture }
      }
    }
  }
} // AESELF_MANAGEMENT
```

```
FLUENT inStartingGreenImageSession {
  INITIATED_BY { EVENTS.
          greenImageSessionIsAboutToStart }
  TERMINATED_BY { EVENTS.
          imageSessionStartedGreen }
}
FLUENT inCollectingImagePixelsBlue {
  INITIATED_BY { EVENTS.imageSessionStartedBlue }
  TERMINATED_BY { EVENTS.imageSessionEndedBlue }
}


EVENT greenImageSessionIsAboutToStart {
  ACTIVATION { SENT { AES.Voyager.
    AEIP.MESSAGES.msgGreenSessionBeginAus } }
}
EVENT imageSessionStartedBlue {
  ACTIVATION { RECEIVED { AES.Voyager.
    AEIP.MESSAGES.msgBlueSessionBeginAus } }
}
```

Figure 3: AE antenna self-management policies, fluents, events

2.3 AE Tier Specification.

At this tier, we have five autonomic elements: the Voyager II spacecraft and the four antennas on Earth. For each, an own part of the `IMAGE_PROCESSING` self-management policy is specified.

AE Voyager. The spacecraft's `IMAGE_PROCESSING` self-management policy (see Figure 3) uses two fluents. The `inTakingPicture` fluent is initiated by a `timeToTakePicture` event and terminated by a `pictureTaken` event. This event also initiates the `inProcessingPicturePixels` fluent, which is terminated by the `pictureProcessed` event. The fluents are mapped to the actions `takePicture` and `processPicture` respectively. Metrics are used, for example, to count all the detected interesting objects which the Voyager AE takes pictures of.

AE Antenna. The four antennas receiving signals from the Voyager II spacecraft are specified as autonomic elements. Their `IMAGE_PROCESSING` self-management policy uses pairs of fluents `inStarting-ImageSession` - `inCollectingImagePixels`, one for each color filter. These sets of fluents determine the states of the antenna AEs when an image-receiving session is starting and when an antenna AE is collecting the image pixels.

Since the `Voyager` AE processes the images by applying different filters and sends each filtered image separately, we have distinct fluents for each color and antenna. This allows an antenna AE to process a collection of multiple filtered images simultaneously.[2] It is the `Voyager` AE that notifies an

[2]Note that according to the ASSL formal semantics, a fluent cannot be re-initiated while it is initiated, thus preventing the same fluent from being initiated simultaneously twice or more times [17].

antenna that an image-sending session begins and ends. Figure 3 shows two of the `IMAGE_PROCESSING` fluents. They are further mapped to AE actions that collect the image pixels per filtered image (see [18]).

In Figure 3 we see how two of the events initiate the AE Antenna fluents. The `greenImageSession-IsAboutToStart` event is prompted (triggered) when the Voyager's `msgGreenSessionBeginSpn` message has been sent and the `imageSessionStartedBlue` event is prompted when the Voyager's `msg-BlueSessionBeginSpn` message has been received by the antenna.

3 Mapping ASSL to GEAR Models

ASSL specifications describe all the different aspects of an autonomic system in one comprehensive document. This is practical, but by nature in realistic cases it becomes very complex, the complexity to a good extent due to the many cross references between the specification elements. A trace through the specified autonomic system may request jumping between different aspects (e.g. from messages $\rightarrow$ events $\rightarrow$ fluents $\rightarrow$ mappings $\rightarrow$ actions) and "pages". Another paper in this proceedings proposes mapping ASSL specifications to LTS, in order to verify LTL properties [19]. Here, we address a different mapping, that enables intuition of the graphical models, expression of constraints in any mu-calculus derivative, and a deep support to diagnosis by means of reverse model checking and games. Our models are Service Logic Graphs (SLG).

3.1 Behavioral models: Service Logic Graphs

To complement the original textual view, and in perspective to visualize and reify certain aspects of the SOS semantics of ASSL, we map selected behavioral elements of the specification to GEAR's behavioral models. These can be visualized as Service Logic Graphs (SLG) in the jABC framework [13, 16] (of which GEAR is the model checking plugin) and analyzed, guiding the user through the processes and workflows of the specified autonomic system. These models are directly amenable to model checking.

SLGs themselves are composed of reusable building blocks that are called *Service Independent Building Blocks* (SIBs) [9], and may represent both a single atomic service or a whole subgraph (i.e., another SLG). Thus SLGs can be hierarchical, which grants a high reusability not only of the building blocks, but also of the models themselves, within larger systems. SLGs formally stem from the concept of Kripke Transition Systems [11].

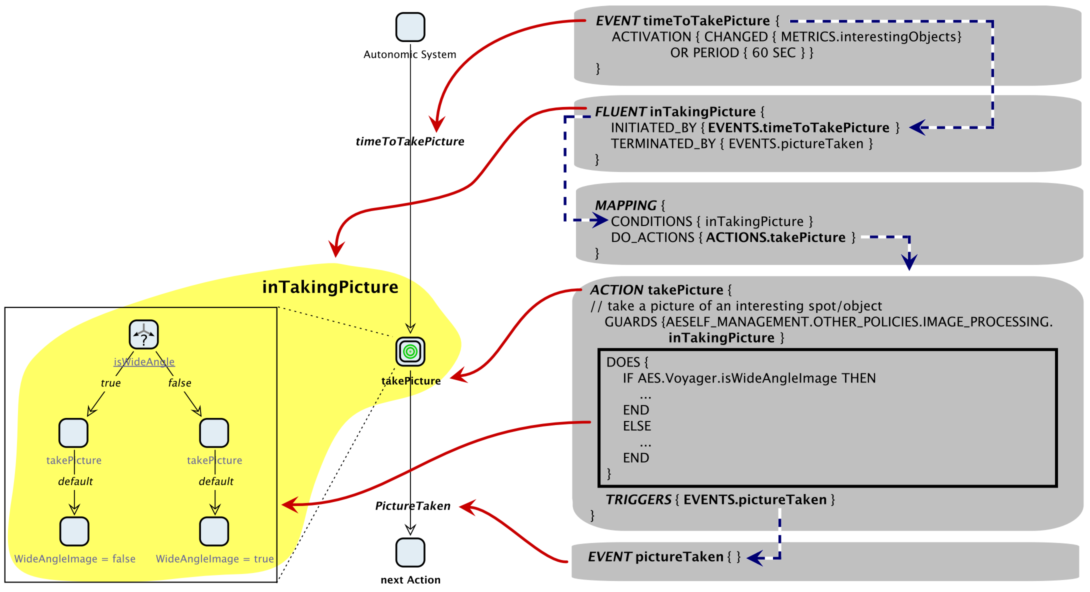
Kripke Transition System A Kripke Transition System K is defined as a tuple $(S, \mathsf{Act}, \rightarrow, I)$ over a set of atomic propositions AP, disjoint from Act, where

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

A KTS is best-suited for verification tasks that focus on transitions of the system as being the edges. On the contrary, one can think of an SLG as being the engineer's view of the system that focuses on the *actions* of the system as being the nodes.

Service Logic Graph A *Service Logic Graph (SLG)* is defined as a tuple $(S, \mathsf{Act}, \rightarrow, I)$ over a set of atomic propositions AP, disjoint from Act, where

- S represents the occurrences of the Service Independent building blocks (SIBs), which are the actions or functions in the graph
- *Act* is the set of possible branching conditions, to be determined upon execution of the preceding SIB,
- $Trans = (s, a, s)$ is a set of transitions where $s, s \in S$ and $a \in Act$, and
- a labeling interpretation function $I : S \rightarrow 2^{\mathsf{AP}}$ equips SIB occurrences with atomic propositions.



```
EVENT timeToTakePicture {
  ACTIVATION { CHANGED { METRICS.interestingObjects}
           OR PERIOD { 60 SEC } }
}
```

```
FLUENT inTakingPicture {
  INITIATED_BY { EVENTS.timeToTakePicture }
  TERMINATED_BY { EVENTS.pictureTaken }
}
```

```
MAPPING {
  CONDITIONS { inTakingPicture }
  DO_ACTIONS { ACTIONS.takePicture }
}
```

```
ACTION takePicture {
// take a picture of an interesting spot/object
   GUARDS {AESELF_MANAGEMENT.OTHER_POLICIES.IMAGE_PROCESSING.
            inTakingPicture }
   DOES {
      IF AES.Voyager.isWideAngleImage THEN
         ...
      END
      ELSE
         ...
      END
   }
   TRIGGERS { EVENTS.pictureTaken }
}
```

```
EVENT pictureTaken { }
```

Figure 4: Action, Event, Fluent, and *Mapping* in KTS behavioral model representation

The structural match, KTS and SLGs are both graph structures with labeled branches and nodes that are enriched with atomic propositional properties, suffices to adopt the established model checking technologies for the SLGs.[3]

In mapping the elements of the ASSL specification to a graphical representation in the behavioral model we focus on those constructs that describe behavioral and self-* aspects: these are the central elements which will be most frequently used to specify autonomic systems. We currently cover

- the AS tier: Service-Level Objectives, Self-Management Policies, Actions and Events.
- the AE tier: Service-Level Objectives, Self-Management Policies, Actions and Events, and additionally behavioral models and outcomes.

Architectural, communication, and quantitative aspects will be dealt in future work.

3.2 Mapping ASSL Elements

From the point of view of model generation, AS and AE specifications are structurally similar w.r.t. events, self management policies, and actions, but differ in the scoping—while the AS specification has a global scope, the AE specification is only valid for the local element. Due to the similarities, we focus on the description of the autonomic element (AE) tier. The AS tier is captured similarly, by means of hierarchy (where single nodes of the AS-level KTS are expandable to AE-level models).

We refer to Figure 4, showing a specification fragment for the Voyager (right) and the corresponding section of the behavioral model (left). In the textual specification (right), we have two events, one fluent with a mapping, and one action. Dashed arrows illustrate a trace of an event within the specs. Arrows indicate the correspondence between elements of the ASSL-specification and of the behavioral. The InTakingPicture cloud defines the current state of the system (an atomic proposition).

AE Event. Event is the central language element in ASSL. It specifies fluents, actions, and policies globally in the AS tier and locally in the AE tier. Events could be activated by messages, other events,

[3]In fact both system representation styles can easily be translated into each other by adequately mapping edges to nodes and vice versa.

actions or metrics. In our behavioral model, events are mapped to homonymous Branches. In Figure 4, the behavioral model starts with the event *timeToTakePicture*, activated for interesting objects or after a time period of 60 s. It initiates the self management policy (fluent) *inTakingPicture*.

AE Self Management Policy. It defines the behavior of the autonomic system by connecting specific system states with the intended (re)action. A policy consists of two elements:

- A *fluent*, similar to a state. It is initiated (ie., that state is reached) when the system satisfies specific *conditions*. It will be terminated (left) if specific events occur. Fluent activation and termination is driven by events.
- A *mapping* of certain conditions to *actions*. The conditions test fluents: in a certain state, certain actions (in the AS or AE tier) are performed. Actions activate specified actions.

They are central to the model extraction: the information contained in a self management policy is used and useful both for model construction and for verification.

Together, fluent and mappings define the control flow, i.e. create branches with the name of the initiating event. They define all possible incoming branches of an action. The specific condition that activates the fluent is stored in the *context* of the system's model. The context represents the current global state of the system, like a global Blackboard or shared memory-mechanism. For model checking purposes, the fluent is additionally associated as atomic proposition to the corresponding node(s) of the behavioral model. This enables global model checking. The fluent can be used as preconditions of actions. They hold on all states in the region between initiation and termination.

The fluent in our example is activated by the *timeToTakePicture*-event and the overall status of the autonomous system is changed to *intakingPicture*. This change activates an action: *takePicture* which is specified in the Mapping section of the self management object.

The self management policy which connects the event to actions is additionally used to annotate the nodes in the behavioral model with atomic propositions (AP). The name of the AP is equal to the name of the fluent. They can later be used for model checking.

AE Action. Actions are routines performed by AE or AS (global and local). In our behavioral model, they are the second essential element: the nodes of our behavioral model, named as the action. The different elements of an action are used to describe the nodes and for verification purposes. Action *parameters* become parameters of a node; the *does* part represents the body of a node. It can be a single action (then the node is an atomic node), but for complex *does* it is an entire behavioral model. We then model them as a SLG hierarchy, as in Figure 4: the node *takePicture* has a corresponding sub-model, presented on the left. The *guards*, *returns* and *outcomes* are used for verification. We offer two possibilities for verification:

- The Localchecker uses the Guard to verify if an Action could be executed within the current system state (defined by the fluents and stored in a global context).
- We can use a model checker to verify relations of nodes and actions expressed as temporal logic constraints. Internally, GEAR uses the modal mu-calculus [10] enriched with forward and backward modalities, so it is best equipped, for example, to express dataflow properties, or other behavioral constraints such as CTL formulas.

The specified action in Figure 4 contains a guard which must conform to the AP annotated at the node.

How the outgoing branches of a node are defined depends on the information found in the action's specification. Actions can use events; triggers are communication functions to communicate with the autonomic system and its elements. We thus have several possibilities to detect outgoing branches.

- a trigger statement in the specification of an action will create an event which introduces the next fluent and/or action, and is comparable to an outgoing branch,

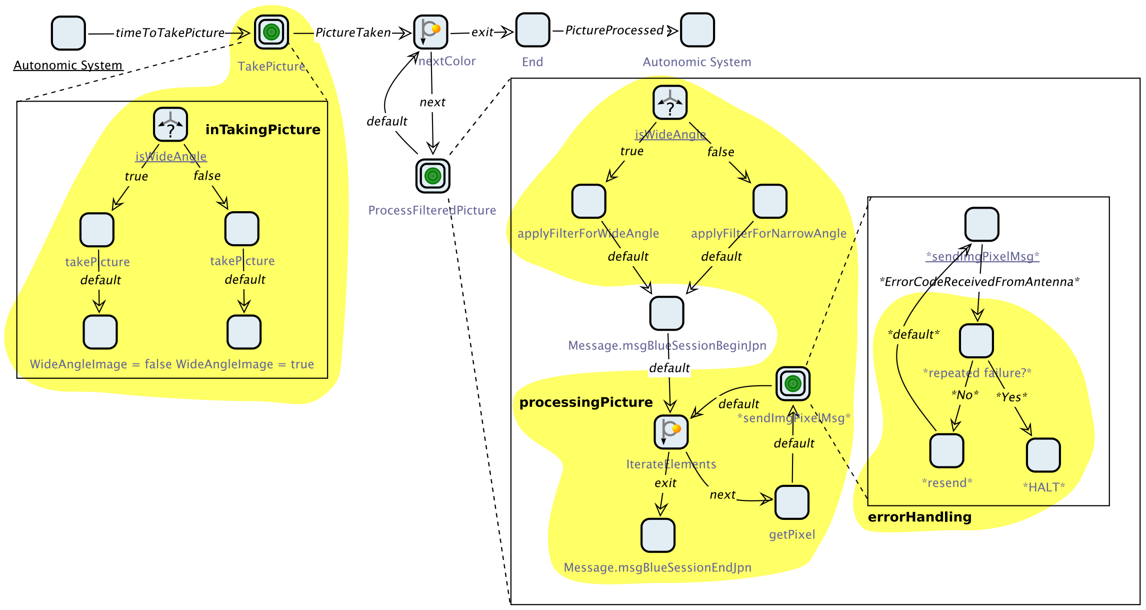


Figure 5: Behavioral model of the picture transmission process. Bottom right: a new error handling recovery mechanism.

- event statements in the Does part are added as possible outgoing branches,
- if communication functions are used, we follow the chain from the function to the communication channel to the events which will be activated by a specific message in the channel. It is not unusual that more than one event will be created from one message.

The *takePicture* action of Figure 4 is closed by a new event *pictureTaken*. This is specified in the triggers section and represents the outgoing branch of this node. The new event will again initiate a fluent, and it terminates the *inTakingPicture* fluent. Therefore, the next action has another AP.

AE Outcomes, AE Behavioral Models, and AE Recovery Protocol. These elements are not yet treated in depth. They will become relevant when applying the SHADOWS methodology. In short, **AE Outcomes** are post-conditions of actions or behavioral models — they are useful for verification purposes. An **AE Behavioral Model** is comparable, from the model generation point of view, to a further mapping in the self-management policy. It consists of conditions, a do element where an action is activated, and outcomes. We can model the behavioral model similarly to an action (atomic or hierarchical). Condition and outcomes become the pre- and post-condition and the action is the implemented behavior. An **AE Recovery Protocol** should guarantee fault-tolerant operation of the autonomous system (e.g. create snapshots, log messages, consistency checking). A recovery protocol specification is rather complex, and it is specified in a separate submodel.

4 Verifying the Voyager's Behavioral Model

Figure 5 contains the behavioral model of the Voyager II spacecraft. Note that the error handling graph at the right was not part of the original ASSL specification.

A simple verification issue that immediately emerges is whether the system takes care of an error-handling process whenever picture pixels are transmitted. This can be easily expressed in CTL [6] as

$$\mathsf{AG}(\mathsf{inProcessingPicturePixels} \Rightarrow \mathsf{EF}(\mathsf{errorHandling}))$$

This formula can be interpreted as follows:

> Wherever the system evolves to (the AG-part), whenever picture pixels are about to be processed (the atomic proposition inProcessingPicturePixels) it follows that the system has an option to evolve into an error-handling process (the EF(errorHandling)-part).

Since the original model of Figure 5 does not support any kind of self-healing capabilities, this property does not hold.

Therefore, in a first attempt to reconcile model and property, we added an error-handling routine directly in the model. We slightly changed the design manually, by refining the sendImgPixelMsg action, originally atomic, to an entire routine. Now, if problems during the transmission process occur, the system tries to resend those picture pixels that were not transmitted correctly. If the problem still exists afterwards, the system is halted and needs manual interaction from ground control.

A game-based approach as presented in Bakera et al. [1] would do much more than just allowing the identification of the missing recovery mechanism in the original specification. Enabling this investigation for self-healing and self-healing enactment is our aim. A domain-dependent guidance also enables to pinpoint that part of the model which is best-suited for integration of recovery mechanisms. Due to space limitation, we cannot discuss this process in detail in this contribution.

Enabling Model based Self-healing 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. In particular, we show how to model the several abstraction levels of the system's behavior in a uniform and formal but intuitive way. This happens in term of processes in the jABC framework [16], 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, for example, 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. GEAR [1], our model checker capable for the full modal μ-calculus, has a rich user interface that allows pinpointing problems in system design. This is achieved by interactively exploring the problem space in a game-based way. The game-based nature of GEAR's verification algorithm supports the system designer at design-time to interactively explore the problem space upon property mismatches.

In the case of the Voyager mission case study, such properties can be used to check for complete picture transmission to the four antennas in case of transmission interrupts. Further, the verification process is able to assure the application of all four color filters before picture transmission. In addition it is essential for the picture transmission to send closing notification signals of transmission endings to the antennas. This as well can be assured by the aforementioned Model Checking techniques.

If problems occur in the verification task, one immediate result of the game-based algorithm of the Model Checker is an interactive counter-example. This counter-example both pinpoints the problem of the property mismatch and provides a strategy encoded into the counter-example to adapt and self-heal the system. GEAR [1] elaborates on the application of this technique on an ESA mission example.

5 Related Work

Nowadays, there is a growing consensus that model checking is most effective as an intelligent and early error-finding technique rather than a technique for guaranteeing correctness. This is partly due to the fact that specifications that can be checked automatically through model checking are necessarily partial in that they specify only certain aspects of the system behavior. Therefore, successful model checking runs, while reassuring, cannot guarantee full correctness. Rather, model checkers are increasingly conceived as elaborate debugging tools that complement traditional testing techniques.

Various model checkers are used to verify aerospace systems. Java Pathfinder [7], developed at NASA Ames, is a prominent example for verifying smaller systems. It assists developers at the Java code level, and therefore addresses a later phase than our approach. We aim at assertions on interactions between components or of the system as a whole, with a focus on demanding properties.

For model checkers to be useful as debugging tools it is important that failing model checking attempts are accompanied by appropriate *error diagnosis* information that explains *why* the model check has failed. Model checkers may in fact also fail *spuriously*, i.e., although the property does not hold for the investigated abstraction it may still be valid for the real system. In order for model checking to be useful, it should therefore be easy for the user to rule out spurious failures and to locate the errors in the system based on the provided error diagnosis information. Therefore, it is important that error diagnosis information is easily accessible by the user.

Currently, ASSL provides a consistency checking mechanism to validate specifications of autonomic systems against correctness properties. Although proven to be efficient with handling consistency errors, this mechanism cannot handle logical errors. Another paper in this proceedings [19] proposes a different model checking approach for ASSL, based on Labelled Transition systems and LTL properties.

For linear-time logics (LTL), error diagnosis information is conceptually of a simple type: It is given by a (possibly cyclic) execution path of the system that violates the given property. Thus, in case model-checking fails, linear-time model checkers like SPIN [8] compute an output in the form of an *error trace* that represents a violating run, and is therefore valuable for the subsequent diagnosis and repair.

The produced error traces (also called counter-examples) can be used for simulation to reproduce the execution that leads to an error, e.g., a counter-example could be translated into a UML sequence diagram. However, SPIN supports multiple counter-examples; i.e., if a correctness property is not satisfied, there will be multiple error traces generated leading to the same error. Although, the shortest path can be determined manually, automatic analysis and generation of relevant executions requires a formal approach to the possible variations of a counter-example.

In general, the LTL model checking can be done though generalization to a broader class of linear formalism than LTL. This is possible through transformation into the so-called Büchi automaton and is often implemented on top of CTL model-checking algorithms by transforming the LTL model checking over one structure into checking fairness over another structure [5].

The situation is more complex for properties that embody recovery issues. These claim for more demanding properties expressible in branching-time logics like CTL or the modal μ-calculus. Such logics do not just specify properties of single program executions but properties of the entire execution tree, comprising the locale of decision points. Hence, meaningful error diagnosis information for branching-time logic model checking cannot be represented by linear executions in general. Here games help.

6 Summary and Conclusion

We have shown how to translate parts of an ASSL specification for autonomic systems into a behavioral model. This task implied mapping the ASSL specific *self-management policy, action*, and *event* parts that made up the system to corresponding counterparts in a behavioral system model that is based on a Service Logic Graph. We applied this translation step to the NASA Voyager II mission case study, opening up several options for verifying issues related to e.g. recovery issues. Having detected the absence of a recovery mechanism upon transmission error within the system specification, we may leverage GEAR to fix this problem. A game-based exploration of the problem space as already suggested a tool supported enhancement of the model-driven verification process [1] can help in identifying those parts of the model that need adaptation to overcome this specific problem. However, we did not elaborate on this exploration here since the translation of the specification is still incomplete.

We have previous experience of automatic generation of control flow graphs from a language's Structured Operational Semantics [12] (SOS). In [3] we showed how to do it for a process algebra, later extended for object oriented languages. Accordingly, we plan to examine the SOS for ASSL provided in [17] and possibly take it as a starting point for an SOS-driven generation of the SLGs. This way, the palette of model analyses developed in the jABC and the self-healing specific techniques developed in SHADOWS would become immediately applicable to all ASSL descriptions.

References

[1] M. Bakera, T. Margaria, C. Renner, and B. Steffen. Verification, diagnosis and adaptation: Tool supported enhancement of the model-driven verification process. *ISSE, Innovations in System and Software Engineering - a NASA Journal , Springer Verlag*, in print.

[2] M. Bakera, T. Margaria, C. Renner, and B. Steffen. Game-Based Model Checking for Reliable Autonomy in Space. *AIAA Journal of Aerospace Computing, Information and Communication(JACIC)*, to appear.

[3] V. Braun, J. Knoop, and D. Koschützki. *cool: A control-flow generator for system analysis*. Technical Report MIP-Bericht Nr. 9801, Faculty of Mathematics and Informatics, University of Passau, Germany, 1998.

[4] M. W. Browne. Technical magic converts a puny signal into pictures. *NY Times*, 1989.

[5] J. Burch, E. Clarke, K. Mcmillan, D. Dill, and L. Hwang. Symbolic model checking: 10 pow 20 states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33. IEEE Computer Society Press, 1990.

[6] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Logics of Programs — Proc. 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Heidelberg, Germany, 1981.

[7] K. Havelund and T. Pressburger. Model Checking JAVA programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000.

[8] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, Massachusetts, USA, 2003.

[9] ITU. General recommendations on telephone switching and signaling - intelligent network: Introduction to intelligent network capability set 1, Recommendation Q.1211. Technical report, Standardization Sector of ITU, Geneva, March 1993.

[10] D. Kozen. Results on the propositional μ-calculus. In *ICALP*, volume 140 of *LNCS*, pages 348–359, Aarhus, Denmark, 12–16 July 1982. Springer-Verlag.

[11] M. Müller-Olm, D. A. Schmidt, and B. Steffen. Model-Checking: A Tutorial Introduction. In *SAS*, pages 330–354, 1999.

[12] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[13] Ralf Nagel. jABC. `http://www.jabc.de`.

[14] SHADOWS. A self-healing approach to designing complex software systems. `https://sysrun.haifa.ibm.com/shadows/`.

[15] O. Shehory, S. Ur, and T. Margaria. Self-healing technologies in SHADOWS: Targeting performance, concurrency and functional aspects. In *10th (CONQUEST)*, 2007.

[16] B. Steffen, T. Margaria, R. Nagel, S. Jörges, and C. Kubczak. Model-Driven development with the jABC. In *Hardware and Software, Verification and Testing*, pages 92–108, 2007.

[17] E. Vassev. *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[18] E. Vassev and M. Hinchey. ASSL Specification Model for the Image-processing Behavior in the NASA Voyager Mission. Technical report.

[19] E. Vassev, M. Hinchey, and A. Quigley. Model checking for autonomic systems specified with ASSL. In *Proc. First NASA Formal Methods Symposium (NFM 2009)*. NASA, 2009.