2010 Fourth International Symposium on Theoretical Aspects of Software Engineering

Automated Test Case Generation of Self-Managing Policies for NASA Prototype Missions Developed with ASSL

(Invited Paper)

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

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

Paddy Nixon
Lero, University College Dublin
Dublin, Ireland
Email: paddy.nixon@lero.ie

***Abstract*—Self-managing policies provide a self-management behavior for autonomic systems developed with ASSL (Autonomic System Specification Language). With ASSL we have successfully developed special autonomic prototypes of both the NASA ANTS (Autonomous Nano-Technology Swarm) concept mission and NASAs Voyager Mission. In these prototypes, we applied ASSL self-managing policies to drive the missions in critical situations in response to environmental or system changes. Therefore, the logical correctness of the ASSL specification of such policies appears to be of major importance. Experience has shown, however, that ASSL specifications may contain logical faults causing improper behavior. To handle such behavior, self-managing policies are often tested with manually injected inputs triggering events and satisfying constraints to allow for the activation, execution, and deactivation of these policies. The logical correctness of an ASSL self-managing policy currently depends solely upon the relation between inputs and conclusion. In this paper, we present our initial work on a novel tool, part of the ASSL framework, that generates test cases based on change-impact analysis. Our main goal is to reduce testing costs and effort and improve the quality of testing, thus eventually assuring the logical correctness of the self-managing policies developed with ASSL.**

***Keywords*-self-management; testing; test generation; ASSL**

I. Introduction

Safety is a major concern in safety-critical systems such as NASA exploration missions, where both reliability and maintainability are essential drivers. In that context, NASA's Reliability and Maintainability Program [1] targets engineering activities intended to asses and improve the reliability performance of systems used for both manned and unmanned space-exploration missions. The goal of this program is to help NASA system engineers develop highly reliable systems through design evaluation, automated model analysis and verification, and testing, thus establishing "confidence that those systems will function properly when needed. Moreover, the maintainability part of this program [1] targets maintenance reduction and minimization of the maintenance downtime in spacecraft systems. To achieve these goals, NASA increasingly relies on formal development approaches and on concepts from Autonomic Computing (AC) [2]. Both the Autonomous Nano-Technology Swarm (ANTS) concept mission [3] and the Deep Space One (DS1) mission [4] represent a new generation of AC-based unmanned missions. Software developed with the use of formal methods has demonstrated to be more reliable, because both a formal notation and suitable mature tool support are provided.

In order to develop reliable autonomic systems (ASs) capable of performing unmanned missions, NASA must rely on formal methods that cope well with the principles of AC. To the best of our knowledge, ASSL [5] is currently the only complete solution to the problem of AS specification and implementation. ASSL has been successfully used to develop autonomic properties and generate special prototype models of the NASA ANTS concept mission [6], [7] and NASA's Voyager mission [8]. These prototype models have helped us simulate space exploration missions.

Research Problem and Approach: Our experience with ASSL has demonstrated that errors can easily be introduced while specifying large systems such as ANTS [6] or Voyager [8]. Currently, the ASSL framework is capable of efficiently handling syntax and consistency errors [5]. As for handling logical errors, in previous work we presented a methodology for model checking ASSL specifications [9]. However, due to the so-called state-explosion problem [10], model checking cannot efficiently handle logical errors. Therefore, to detect errors introduced not only with the ASSL specifications, but also with supplementary coding, the automatic verification provided by the ASSL tools must be augmented with appropriate testing. ASSL-developed ASs are driven by special self-managing policies. To test such ASs properly, we must consider all the implemented self-managing policies and come up with test cases providing the necessary inputs and conditions to explore these policies in all possible paths of execution. Without automatic test generation, ASSL testing relies solely on the developer's intuition to identify test cases for self-managing policies. In general, the process of manually identifying appropriate test cases is laborious, and its success depends on the experience of the tester. This problem is especially serious for users of ASSL, who typically are not experienced programmers and lack background in testing. To address this problem, we have been investigating how to automate the generation of

978-0-7695-4148-8/10 $26.00 © 2010 IEEE
DOI 10.1109/TASE.2010.32

IEEE computer society

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.

test cases for self-managing policies in ways that support incremental testing and provide immediate visual feedback. We have used two techniques for test case generation: one using random selection and another using a change-impact analysis approach. This paper describes the second approach.

Benefits for Space Systems: An ASSL automatic test generator will help to complete the AS development process with ASSL. The ability to properly test the ASSL-developed ASs for implementation flaws can lead to significant improvements in both specification models and generated ASs. Subsequently, ASSL can be used to develop better prototype models for current and future space-exploration missions. Such prototypes can be used for feature validation and simulated experimental results. This eventually will help in the construction of more reliable spacecraft systems, which offers significant benefits.

II. ASSL Self-managing Policies

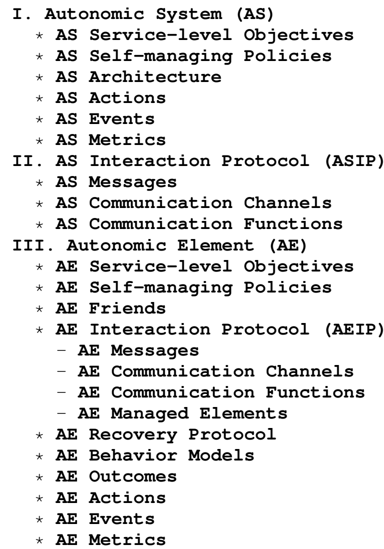
The Autonomic System Specification Language (ASSL) [11], [5] is an initiative for self-management of complex systems where we approach the problem of formal specification, validation, and code generation of autonomic systems (ASs) within a single framework. Being dedicated to AC, ASSL helps AC researchers with problem formation, system design, system analysis and evaluation, and system implementation. The framework provides tools that allow ASSL specifications to be edited and validated. From any valid specification, ASSL can generate an operational Java application.

A. ASSL Specification Model

In general, ASSL considers the ASs as being composed of autonomic elements (AEs) that communicate over interaction protocols. To specify those, ASSL is defined through formalization tiers. Over these tiers, the framework provides a multi-tier specification model that is designed to be scalable and exposes a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. The ASSL tiers and their sub-tiers (see Figure 1) are abstractions of different aspects of the AS under consideration. They aid not only to the specification of the system at different levels of abstraction, but also in reduction of complexity, and thus, improving the overall perception of the system.

There are three major tiers (three major abstraction perspectives), each composed of sub-tiers (see Figure 1):

- *AS tier* – presents a general and global AS perspective, where we define the general autonomic system rules in terms of *service-level objectives (SLO)* and *self-managing policies*, *architecture topology* and *global actions*, *events* and *metrics* applied in these rules.
- *AS Interaction Protocol (ASIP) tier* – forms a communication protocol perspective, where we define the means of communication between AEs. An ASIP is composed of *channels*, *communication functions*, and *messages*.
- *AE tier* – forms a unit-level perspective, where we define interacting sets of individual AEs with their own behavior. This tier is composed of AE rules (*SLO* and *self-managing policies*), an *AE interaction protocol* (AEIP), *AE friends* (a list of AEs forming a circle of trust), *recovery protocols*, special *behavior models* and *outcomes*, *AE actions*, *AE events*, and *AE metrics*.



```
I. Autonomic System (AS)
  * AS Service-level Objectives
  * AS Self-managing Policies
  * AS Architecture
  * AS Actions
  * AS Events
  * AS Metrics
II. AS Interaction Protocol (ASIP)
  * AS Messages
  * AS Communication Channels
  * AS Communication Functions
III. Autonomic Element (AE)
  * AE Service-level Objectives
  * AE Self-managing Policies
  * AE Friends
  * AE Interaction Protocol (AEIP)
    - AE Messages
    - AE Communication Channels
    - AE Communication Functions
    - AE Managed Elements
  * AE Recovery Protocol
  * AE Behavior Models
  * AE Outcomes
  * AE Actions
  * AE Events
  * AE Metrics
```

Figure 1. ASSL Multi-Tier Model

The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-managing policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Figure 1). The AS SLOs are a high-level form of behavioral specification that help developers establish system objectives (e.g., performance). The self-managing policies can be any of (but not restricted to) the four so-called self-CHOP policies defined by the AC IBM blueprint [12]: *self-configuring*, *self-healing*, *self-optimi-zing* and *self-protecting*. These policies are driven by events and trigger the execution of actions driving an AS in critical situations. The metrics constitute a set of parameters and observables controllable by an AS. At the ASIP Tier, the ASSL framework helps developers specify an AS-level interaction protocol as a public communication interface, expressed with special *communication channels*, *communication functions* and *communication messages*. At the AE Tier, the ASSL formal model exposes specification constructs for the specification of the system's AEs.

Conceptually, AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs. Note that ASSL targets only the AC features of a system and helps developers clearly distinguish the AC features from the system-service features.

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.

This is possible, because with ASSL we model and generate special AC wrappers in the form of ASs that embed the components of non-AC systems. The latter are considered as *managed elements*, controlled by the AS in question. Conceptually, a managed element can be any software or hardware system (or sub-system) providing services. Managed elements are specified per AE (see Figure 1) where the emphasis is on the interface needed to control a managed element. It is important also to mention that the ASSL tiers and sub-tiers are intended to specify different aspects of an AS, but it is not necessary to employ all of them in order to model such a system. For a simple AS, we need to specify:

1) the AEs providing self-managing behavior intended to control the managed elements associated with an AE;
2) the communication interface.

For more details on the ASSL multi-tier specification model and the ASSL framework toolset, the interested reader is referred to [5], [11].

B. *Specifying Self-managing Policies*

In the ASSL-developed ASs, the self-management behavior is provided by *self-managing policies* that are specified at the level of AS (see the AS Tier in Figure 1) and at the level of AEs (see the AE Tier in Figure 1). Therefore in general, an ASSL specification is built around one or more *self-managing policies*. This makes the ASSL specifications AC-driven, i.e., the ASSL-developed ASs are modeled taking into account the main goal of AC, viz. self-management. The self-managing policies are driven by *ASSL events* and *ASSL actions* and are specified with special constructs called *fluents* and *mappings* [5]:

- a fluent sets specific conditions determining when a self-managing policy is activated;
- mappings map particular fluents to particular actions to be undertaken by the specified AS.

The following ASSL code presents a sample specification of a self-healing policy.

```
ASSELF_MANAGEMENT {
 SELF_HEALING {
  FLUENT inLosingSpacecraft {
   INITIATED_BY { EVENTS.spaceCraftLost }
   TERMINATED_BY { EVENTS.earthNotified }
  }
  MAPPING {
   CONDITIONS { inLosingSpacecraft }
   DO_ACTIONS { ACTIONS.notifyEarth }
  }
 }
} // ASSELF_MANAGEMENT
```

As shown, fluents are expressed with *fluent-activating* and *fluent-terminating* events. In order to express mappings, conditions and actions are considered, where the former determine the latter in a deterministic manner.

```
SELF_PROTECTING {
 FLUENT inSecurityCheck {
  INITIATED_BY {
   EVENTS.privateMessageIsComming
  }
  TERMINATED_BY {
   EVENTS.privateMessageSecure,
   EVENTS.privateMessageInsecure
  }
 }
 MAPPING {
  CONDITIONS { inSecurityCheck}
  DO_ACTIONS { ACTIONS.checkPrivateMessage }}}
```

Figure 2. Self-protecting Policy

```
EVENT privateMessageIsComming {
 ACTIVATION {
  SENT { AEIP.MESSAGES.privateMessage }
 }
}
EVENT privateMessageInsecure {
 GUARDS { NOT METRICS.thereIsInsecureMsg }
 ACTIVATION {
  CHANGED { METRICS.thereIsInsecureMsg }
 }
}
EVENT privateMessageSecure {
 GUARDS { METRICS.thereIsInsecureMsg }
 ACTIVATION {
  CHANGED { METRICS.thereIsInsecureMsg }
 }
}
```

Figure 3. Policy Events

```
ACTION checkPrivateMessage {
 GUARDS { .... }
 ENSURES { .... }
 DOES {
  senderIdentified = call ACTIONS.checkSenderCertificate;
  ....
 }
 ONERR_DOES { .... }
 TRIGGERS { .... }
 ONERR_TRIGGERS { .... }
}
```

Figure 4. CheckPrivateMessage Action

Figure 2, Figure 3, and Figure 4 present a partial specification of a *self-protecting policy* employed by one of the prototype models [6] we built for the NASA ANTS concept mission [3]. Note that ASSL events (see Figure 3) and actions (see Figure 4) may be specified with a special *GUARDS* clause stating preconditions that must be met before an event may be raised or an action may be undertaken. In addition, events (see Figure 3) are specified with a special *ACTIVATION* clause and actions may be specified with an *ENSURES* clause to state post-conditions that must be met after the action execution. Actions may call other actions in their *DOES* or *ONERR_DOES* clauses. Finally, actions (see Figure 4) may trigger events specified in special *TRIGGERS* and *ONERR_TRIGGERS* clauses. Note

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.

that the *ONERR_DOES* and *ONERR_TRIGGERS* clauses specify the action execution path in case of an error [5]. Therefore, a policy may have multiple paths of execution depending on its fluent organization and the events driving those fluents. Moreover, each policy has an optional set of constraints defined as *GUARDS* and *ENSURES* clauses in events and actions [5]. These constraints together with the event's *ACTIVATION* clause may also determine the execution path of a policy. Therefore, to summarize, the execution of a policy may be controlled via the policy-related events (through the *ACTIVATION* clause) and policy-related *GUARDS* and *ENSURES* clauses.

III. Test Generation Methodology

Policy Execution Paths: Formally, from a policy-execution perspective, an ASSL-specified self-managing policy π may be presented as a tuple:

$$\pi\{F, A\}$$

where F presents the fluents driving the policy in question and A presents the actions that eventually will be undertaken while the policy is *active*. Here, for each fluent $f \in F$ we have:

$$f\{Ea, Af, Et\}$$

where Ea and Et are the sets of *fluent-activating* and *fluent-terminating* events respectively and $Af \subset A$ is the set of actions to be executed by f. Further, an event:

$e \in Ea \bigcup Et$ is a tuple $e\{grd, act\}$

where grd is the *GUARDS* clause and act is the *ACTIVATION* clause of the event e. Finally, an action $a \in A$ is a tuple:

$$a\{grd, ens, Etr, Eer\}$$

where grd and ens are the action's *GUARDS* and *ENSURES* clauses respectively, and Etr and Eer are sets of events triggered by the action a in case of normal and erroneous action execution.

The execution of a policy π is activation and termination of the policy's fluents. Thus, to trace the policy execution, we must consider the execution paths of all the policy's fluents F. The execution path of a fluent is a sequence of the form:

$$\{Ea, Af, Et\}$$

The execution paths of a fluent with n activation events Ea, m termination events Et, and k actions A is a product:

$$m \times n \times v(k)$$

where the function $v(k)$ gives the variations in the execution of A. This function takes into account the action's formal attributes: grd, ens, Etr, and Eer, together with their internal dependencies and ASSL formal semantics [5] as following:

- Etr and Eer are mutually exclusive, i.e., both cannot co-exist in same execution path;
- if ens is not met (denoted as $!ens$), then Eer is mandatory;
- if grd is not met (denoted as $!grd$), then the action a is not executed (denoted as $!a$).

Note that to simplify the problem, in this formal model we consider events as *activated* or *not activated*, thus helping us generalize over the event's clauses *GUARDS* and *ACTIVATION*. To illustrate the formal model, we present a simple example of a fluent

$$f\{Ea, Af, Et\}$$

where $n = 1$, $m = 1$, $k = 2$, and:

$$Ea = \{ea1\}$$
$$Et = \{et1\}$$
$$Af = \{a1, a2\}$$
$$a1 = \{grd1, ens1, Etr1, Eer1\}$$
$$a2 = \{grd2, ens2, Etr2, Eer2\}$$

Here, the possible execution paths of the fluent f are:

$Pex1 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, ens2, Etr2\}, et1\};$
$Pex2 = \{ea1, a1\{grd1, ens1, Etr1\}, !a2\{!grd2\}, et1\};$
$Pex3 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
$Pex4 = \{ea1, a1\{grd1, ens1, Etr1\}, a2\{grd2, ens2, Eer2\}, et1\};$
$Pex5 = \{ea1, !a1\{!grd1\}, a2\{grd2, ens2, Etr2\}, et1\};$
$Pex6 = \{ea1, !a1\{!grd1\}, a2\{!grd2\}, et1\};$
$Pex7 = \{ea1, !a1\{!grd1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
$Pex8 = \{ea1, !a1\{!grd1\}, a2\{grd2, ens2, Eer2\}, et1\};$
$Pex9 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, ens2, Etr2\}, et1\};$
$Pex10 = \{ea1, a1\{grd1, !ens1, Eer1\}, !a2\{!grd2\}, et1\};$
$Pex11 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
$Pex12 = \{ea1, a1\{grd1, !ens1, Eer1\}, a2\{grd2, ens2, Eer2\}, et1\};$
$Pex13 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, ens2, Etr2\}, et1\};$
$Pex14 = \{ea1, a1\{grd1, ens1, Eer1\}, !a2\{!grd2\}, et1\};$
$Pex15 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, !ens2, Eer2\}, et1\};$
$Pex16 = \{ea1, a1\{grd1, ens1, Eer1\}, a2\{grd2, ens2, Eer2\}, et1\};$

ASSL Test Generator: Our goal is to develop a novel test generator tool based on change-impact analysis that will help the ASSL framework automatically generate high-quality test suites for self-managing policies. The

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.

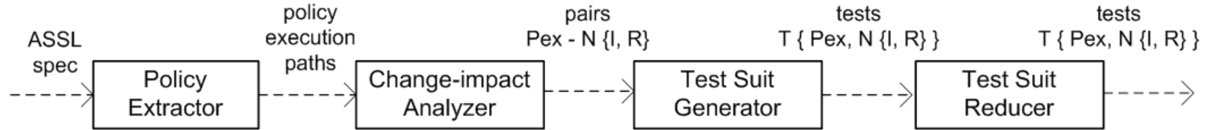


Figure 6. Operational View of the ASSL Test Generator

```
EVENT privateMessageInsecure {
 ACTIVATION { PERIOD { 1 min } }
}
```

Figure 5. Replacement Event

test generator tool accepts as input an ASSL specification comprising sets of policies Π that need to be tested and generates a set of tests T as tuples:

$$T\ \{Pex,\ N\ \{I,\ R\}\}$$

comprising an execution path Pex and test attributes N. The latter is a tuple comprising needed inputs I and optional replacement ASSL constructs R. The latter are automatically or semi-automatically determined and generated as special stubs to ensure the execution of Pex. Figure 5 presents a $privateMessageInsecure$ replacement event that shall replace the original one (see Figure 3). As shown, the new event does not have a *GUARDS* clause and its activation is time-ensured, i.e., it does not depend on external factors.

As shown in Figure 6, the tool consists of four major components: *policy extractor*, *change-impact analyzer*, *test suit generator*, and *test suit reducer*. The key notion of the tool is to synthesize two or more execution paths of the same fluent in such a way that test coverage targets (e.g., certain policies, rules, or conditions) are covered by the synthesized execution paths. The change-impact analysis component can then determine for each execution path the needed test attributes N such as inputs I and optional replacement constructs R in the form of ASSL events, ASSL actions, and *ACTIVATION*, *GUARDS*, and *ENSURES* clauses needed to be employed by a fluent execution path in order to ensure the same. Based on the determined test attributes and execution paths, the tool generates tests T. Often the number of generated tests is large (recall that the number of fluent execution paths is a product of the number of events and actions employed by a fluent) and it is not feasible for developers to manually inspect their responses. To mitigate this issue, the final step of the test generator tool reduces the number of generated tests by selecting tests based on policy structural coverage.

Change-impact Analysis: The goal of *change-impact analysis* is to determine what should be changed in the events and actions employed by a particular fluent execution path Pex in order to ensure the same. In general, ASSL facilitates change-impact analysis because ASSL specifications allow:

1) extraction of information from the model to see where a change must occur in order to force one or more execution paths;
2) calculation of the change impact on the other parts of the model for any proposed change.

Here, of major importance the evaluation of how the execution of a fluent will be affected by a change in a particular event (*GUARDS* or *ACTIVATION* clause) or action (*GUARDS* or *ENSURES* clause). Note that at the time of writing, we are working on the *change-impact analysis heuristic algorithm*. Our initial results have demonstrated that this algorithm should involve the following logical steps.

- Evaluate what the conditions that must be met to have a specific fluent execution path ensured are:
  - Evaluate the events employed by a specific fluent:
    1) For each event analyze the pre-conditions that must be met (GUARDS clause) and the activation conditions (ACTIVATION clause);
    2) Evaluate if a particular event drives (activates or terminates) multiple fluents.
  - Evaluate the actions employed by a specific fluent:
    1) For each action analyze the pre- and post-conditions that must be met (GUARDS and ENSURES clause) and the events that are triggered by the action (TRIGGERS and ON-ERR_TRIGGERS) action;
    2) Evaluate if the action itself executes other ASSL actions, or other executable constructs that may have impact on events such as ASSL interaction functions [5] and ASSL managed element functions [5].
  - Generate a test case that meets the fluent execution path's conditions. Replacement constructs must be generated when the original ones cannot ensure the path execution. For example, if an event cannot be triggered due to conditions that must be met new replacement event may be generated that simulates the old one.
- Evaluate what the impact of having two or more fluents executing simultaneously is and what the conditions that must be met for that are. Generate test cases.

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.

- Evaluate the policies involved in the tested execution path for the presence of chained fluents (the termination of a fluent activates another one, and so on). Find the conditions that must be met for that. Generate test cases.

In addition, it is important to evaluate the impact of modifying an existing construct and that of replacing the same construct with a completely new one. Another aspect that must be addressed by the change-impact analysis is the tradeoffs stemming from "disabling *GUARDS* and *ENSURES* clauses. Note that such clauses act as special "behavior constraints and are usually specified to ensure that certain conditions are met before processing (or terminating) actions or events. Therefore, by disabling (removing) those "constraints (see Figure 5 and Figure 3), we may ensure certain execution paths, but the impact of such a change needs to be also analyzed in the context of tradeoffs coming with the "unconstrained behavior.

IV. Conclusion and Future Work

We have presented our approach towards automatic test case generation for ASSL self-managing policies. An ASSL test generator tool automatically generates test suites composed of fluent execution paths and test attributes. The principle technique is to synthesize such policy execution paths paired with test attributes in the form of inputs and special replacement constructs. The test attributes are determined by change-impact analysis of the effect of a change in particular events or particular actions employed by an execution path. It is our understanding that such a testing mechanism will have a great impact on the development of prototype models for current and future space-exploration missions. Properly tested prototypes, eventually, will lead to the construction of more reliable spacecraft systems. Note that traditional methods, such as analyzing each requirement and developing test cases to verify the correctness of ASSL-implemented ASs, are not effective, because they require complete understanding of the overall complex system's self-management behavior.

Our plans for future work are mainly concerned with further development of the test generation mechanism for ASSL. Further, we plan to generate test cases for a number of self-managing policies developed for ANTS to determine the effectiveness of this approach as a test covering and generation strategy. Moreover, it is our intention to build an animation tool for ASSL, which will help to visualize counterexamples and trace erroneous execution paths.

It is our belief that automatic test generation will make ASSL a better and more powerful framework for AS specification, validation, code generation, and testing.

Acknowledgment

This work was supported in part by an IRCSET postdoctoral fellowship grant at University College Dublin, Ireland and by the Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

[1] F. Groen, "Reliability and maintainability program," 12 2007, http://www.hq.nasa.gov/office/codeq/rm/index.htm, last viewed January 2010.

[2] R. Murch, *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall, 2004.

[3] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff, "NASA's Swarm Missions: The Challenge of Building Autonomous Software," *IT Professional*, vol. 6, no. 5, pp. 47–52, 2004.

[4] M. Rayman, P. Varghese, D. Lehman, and L. Livesay, "Results from the deep space 1 technology validation mission," *Acta Astronautica*, vol. 47, p. 475, 2000.

[5] E. Vassev, *ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems*. LAP Lambert Academic Publishing, 2009.

[6] E. Vassev, M. Hinchey and J. Paquet, "Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions," in *Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track*. ACM, 2008, pp. 1652–1657.

[7] E. Vassev, M. Hinchey, and J. Paquet, "A Self-Scheduling Model for NASA Swarm-Based Exploration Missions using ASSL," in *Proceedings of the Fifth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'08)*. IEEE Computer Society, 2008, pp. 54–64.

[8] E. Vassev and M. Hinchey, "Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL," in *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09)*. IEEE Computer Society, 2009, pp. 246–253.

[9] E. Vassev, M. Hinchey and A. Quigley, "Model Checking for Autonomic Systems Specified with ASSL," in *Proceedings of the First NASA Formal Methods Symposium (NFM 2009)*. NASA, 2009, pp. 16–25.

[10] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2002.

[11] E. I. Vassev, "Towards a framework for specification and code generation of autonomic systems," Ph.D. dissertation, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2008.

[12] IBM Corporation, "An architectural blueprint for autonomic computing," IBM Corporation, Tech. Rep., 2006.

Authorized licensed use limited to: University of Canberra. Downloaded on June 17,2020 at 23:35:29 UTC from IEEE Xplore. Restrictions apply.