Software Verification of Autonomic Systems Developed with ASSL

Emil Vassev[1] and Mike Hinchey[2]

Lero—the Irish Software Engineering Research Centre
[1,2] University of Limerick, Limerick, Ireland
{Emil.Vassev,Mike.Hinchey}@lero.ie

Abstract. We discuss our experiences in building tools for software verification of autonomic systems developed with the Autonomic System Specification Language (ASSL). ASSL is a software framework that aims to assist developers of autonomic systems by providing a powerful combination of both notation and tools. One of the major objectives of the framework is to assure the correctness of the autonomic systems via inclusion of tools targeting consistency checking, model checking, and automatic test case generation. In this paper, we review our recent work on these tools.

Keywords: software verification, formal methods, ASSL, autonomic computing.

1 Introduction

The Autonomic System Specification Language (ASSL) [1, 2] is a formal method dedicated to the development of autonomic systems (ASs) [3]. Conceptually, ASSL assists developers with *formal specification*, *validation*, and *code generation* of such systems. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. As part of the framework validation, ASSL has been successfully used to specify autonomic features and generate AS models for a variety of computer systems including prototypes of two NASA projects—the Autonomous Nano-Technology Swarm (ANTS) concept mission [4] and the Voyager mission [5]. Our experience with ASSL has demonstrated that although the framework is very efficient, errors can be easily introduced while specifying large systems. The first release of ASSL provides built-in consistency checking and functional testing as the only means of software verification. This helps developers easily discover syntax and consistency errors, but barely handles logical errors. To increase the framework's software-verification capabilities, we have been investigating model checking [6] as the most effective approach to software verification for our purposes. In addition, in order to detect errors introduced not only in ASSL specifications, but also with supplementary coding, the automatic verification support provided by the ASSL tools is to be augmented by appropriate automatic generation of test cases. Both model checking and automatic test case generation are subjects of new research projects at Lero–the Irish Software Engineering Research Center. In this paper, we briefly present existing and new software-verification approaches for ASSL.

R. Calinescu and E. Jackson (Eds.): Monterey Workshops 2010, LNCS 6662, pp. 1–16, 2011.

The rest of this paper is organized as follows: In Section 2, we briefly present the ASSL formal specification model. In Section 3, we present the basic consistency checking mechanism. Sections 4 and 5 present our approach to model checking and automatic test case generation with ASSL. Finally, Section 6 provides brief concluding remarks and a summary of future research goals.

2 ASSL

ASSL [1, 2] is based on a specification model exposed over hierarchically organized formalization tiers (see Table 1). This specification model provides both infrastructure elements and mechanisms needed by an AS (autonomic system). Each tier of the ASSL specification model is intended to describe different aspects of the AS in question, such as *service-level objectives, policies, interaction protocols, events, actions*, *autonomic elements*, etc. This helps to specify an AS at different levels of abstraction (imposed by the ASSL tiers) where the AS in question is composed of special autonomic elements (AEs) interacting over interaction protocols (IPs).

Table 1. ASSL multi-tier specification model

AS	**AS Service-level Objectives**	
	AS Self-management Policies	
	AS Architecture	
	AS Actions	
	AS Events	
	AS Metrics	
ASIP	**AS Messages**	
	AS Channels	
	AS Functions	
AE	**AE Service-level Objectives**	
	AE Self-management Policies	
	AE Friends	
	AEIP	**AE Messages**
		AE Channels
		AE Functions
		AE Managed Elements
	AE Recovery Protocols	
	AE Behavior Models	
	AE Outcomes	
	AE Actions	
	AE Events	
	AE Metrics	

As shown in Table 1, the ASSL specification model decomposes an AS in two directions: 1) into levels of functional abstraction; and 2) into functionally related *sub-tiers*. The first decomposition presents the system at three different tiers [1, 2]:

1) *a general and global AS perspective* – we define the general system rules (providing autonomic behavior), architecture, and global actions, events, and metrics applied in these rules;

2) *an interaction protocol (IP) perspective* – we define the means of communication between AEs within an AS;

3) *a unit-level perspective* – we define interacting sets of individual computing elements (AEs) with their own autonomic behavior rules, actions, events, metrics, etc.

The second decomposition presents the major tiers AS, ASIP and as composed of functionally related sub-tiers, where new AS properties emerge at each sub-tier. The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Table 1). The AS SLOs are a high-level form of behavioral specification that helps developers establish system objectives such as performance. The *self-management 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. With 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. Note that AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs. An AE may also specify a private *AE interaction protocol* (AEIP) shared with special AE considered as "friends" (AE Friends tier).

It is important to mention that the ASSL tiers are intended to specify different aspects of the AS in question, but it is not necessary to employ all of them in order to develop an AS. Conceptually, it is sufficient to specify self-management policies only, because those provide self-management behavior at the level of AS (the AS tier) and at the level of AE (AE tier). These policies are specified within the AS/AE Self-management Policies sub-tier (the ASSL construct is **AS[AE]SELF_MANAGEMENT**) with special ASSL constructs termed *fluents* and *mappings* [1, 2]. A fluent is a state where an AS enters with fluent-activating events and exits with fluent-terminating events. A mapping connects fluents with particular actions to be undertaken. Usually, an ASSL specification is built around one or more self-management policies, which make that specification AS-driven. Self-management policies are driven by events and actions determined deterministically. 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.

Once a specification is complete, it can be validated with the ASSL built-in verification mechanisms (e.g., consistency checking) and a functional application skeleton can be generated automatically. The application skeletons generated with the ASSL framework are fully-operational multithreaded event-driven applications with embedded messaging.

3 Consistency Checking with ASSL

In general, we can group the ASSL tiers into groups of *declarative* (or imperative) and *operational* tiers [1, 2]. Whereas the former simply describe definitions in the AS under consideration, the latter not only describe definitions but also focus on the operational behavior of that AS. The ASSL framework evaluates an AS specification formally to construct a special *declarative specification tree* needed to perform both consistency checking and code generation. The declarative specification tree is created by the framework when parsing an AS specification and contains the hierarchical tier structure of that specification. Each specified tier/sub-tier is presented as a *tier instance*. Consistency checking (see Fig. 1) is a framework mechanism for verifying specifications by performing exhaustive traversing of the declarative specification tree. In general, the framework performs two kinds of consistency-checking: 1) *light* – checks for type consistency, ambiguous definitions, etc.; and 2) *heavy* – checks whether the specification model conforms to special *correctness properties*.

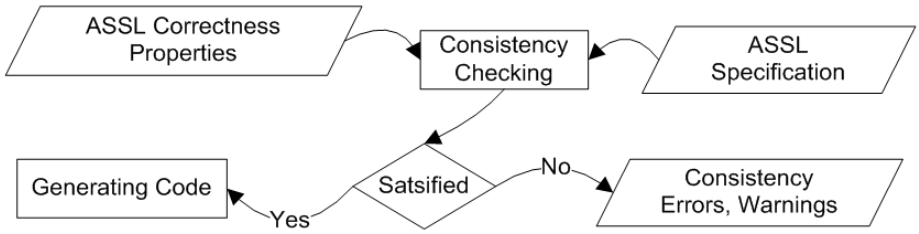

Fig. 1. Consistency Checking with ASSL

The correctness properties are *ASSL semantic definitions* [1, 2] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL)[1] [6], currently ASSL does not incorporate a FOLTL engine, and thus, the consistency checking mechanism implements the correctness properties as Java statements. Here, the FOLTL operators ∀(forall) and ∃(exists) work over sets of ASSL tier instances. It is important to mention that the consistency checking mechanism generates *consistency errors* and *consistency warnings*. Warnings are specific situations where the

[1] In general, FOLTL can be seen as a quantified version of linear temporal logic. FOLTL is obtained by taking propositional linear temporal logic and adding a first order language to it.

specification does not contradict the correctness properties, but rather introduces uncertainty as to how the code generator will handle it.

As mentioned above, a variety of predefined correctness properties are subject of consistency checking. One of those correctness properties is the so-called autonomicity rule [1, 2]. According to that rule, every autonomic system specified with ASSL must have specified at least one self-management policy. Fig. 2 shows an error reported by the ASSL's consistency checker, because the processed ASSL specification violates the autonomicity rule (the entire **ASSELF_MANAGEMENT** sub-tier comprising the self-management policies is commented).

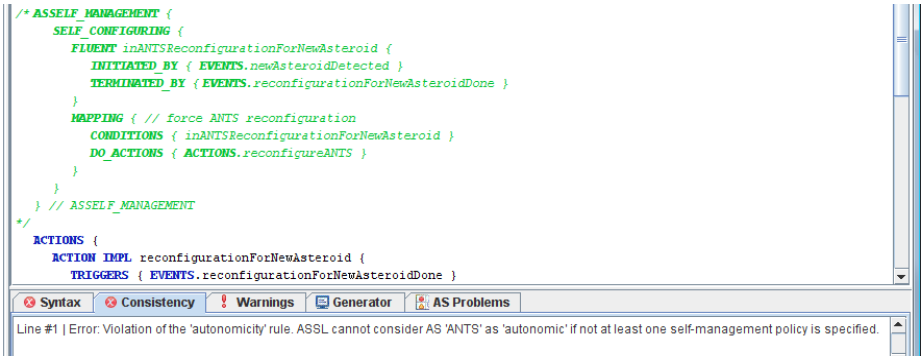

Fig. 2. Checking for "Autonomicity" with the Consistency Checker

4 Built-in Model-Checking Mechanism for ASSL

In general, model checking advocates formal verification whereby software programs are automatically checked for specific flaws by considering correctness properties expressed in *temporal logic* [6]. In this endeavor, three model-checking mechanisms for ASSL have been considered: 1) a built-in model checker [7]; 2) a mechanism for mapping ASSL specifications to formal notation with provided tool support for model checking [8]; and 3) a post-implementation model checker based on the Java PathFinder [9] tool developed at NASA Ames. Whereas the first two model-checking methods check ASSL specifications, the third one is to verify the generated Java code. Note that despite careful specification and the existence of ASSL-level model checking, it is theoretically possible to generate ASs that contain fatal errors (e.g., deadlocks). This is mainly due to the state-explosion problem, which we discuss in Section 4.2. Moreover, with the post-implementation model checker we may verify not only the newly-generated code but also all consecutively updated versions of the same. Thus, the ASSL model-checking mechanisms are intended to verify both the ASSL specifications and the corresponding AS implementations.

In this paper, we report our experience in developing the built-in model checker [7]. In this approach, an ASSL specification is translated into a *state-transition graph*, over which model checking is performed to verify whether an ASSL specification satisfies *correctness properties*. Here, the model-checking problem is: given the AS A

and its ASSL specification *a*, determine in the AS's state graph *g* (called ASG) whether the behavior of *A*, expressed with the correctness properties *p*, meets the specification *a*. An ASG formally stems from the concept of Kripke Structure [6]. The latter is basically a graph having the reachable states of an ASSL-specified system as nodes and the state transitions of the system as edges. In addition, to allow for formal verification, each system state must be labeled with properties (called atomic propositions *AP*) that hold in that state and each state transition must be associated with one or more state transition operations *Op*. The notion of state in ASSL is related to the ASSL specification constructs called *ASSL tier instances* [1, 2] (specified tiers and sub-tiers). The ASSL operational semantics [1, 2] considers a state-transition model where *tier instances* can be in different *tier states*, e.g., instances of the SLO (Service-Level Objectives) tier can be evaluated as *satisfied* or *not satisfied*. Here, an ASSL-developed AS transits from one state to another when a particular tier instance *evolves* from a tier state to another tier state. Here, transition operations *Op* cause tier instances to evolve.

4.1 Building the Autonomic System Graph

In order to build the ASSL model checker, we had to do some preliminary theoretical work to prepare the program structures holding an ASG. Here, we had to define:

1) the reference state model for ASSL-specified ASs, which appeared to be a product machine that consists of *high-level tier states* composed of multilevel *nested tier states*, and the global system state is a product of all nested states (we had to identify an initial state and all the possible tier states *S*);
2) a set of all atomic propositions *AP*, which denote the properties of individual states *S*, and present the *S-AP* relationship as tuples of the form (S_n, AP_1, …. AP_n);
3) all possible transition relations *R* as tuples of the form (S_1, *Op*, S_2).

Next, we had to implement structures holding the *S-AP* and *R* tuples. Note that those are recorded in two flat files (one per tuple type) and are loaded into the implemented program structures at the time of ASSL loading. This helps the model-checker tool cope with future extensions to ASSL. To implement the tuple structures, we used a distinct token class per tuple type (*S-AP* and *R*) and used vectors of tuple tokens. In addition, a generic algorithm is implemented to traverse those vectors and return a sub-vector of tuple tokens refined by *state*, by *operation*, or by *atomic proposition*. Thus, at runtime, the model-checking tool can obtain all the atomic propositions and related transition operations for a particular state. Here,

- tier states *S* are recorded with tier instance name and state name;
  Example: **tier** { *SLO* } **name** { *performance* } **state** { *unsatisfied* }
- transition operations *Op* are recorded with their ASSL predefined names [1];
- atomic propositions *AP* are recorded with "if" and "then" sections and optional "temporal" operators (a temporal logic operator).

Example: **if** { *event prompted* } **then** { **tempOperator** { *eventually* } *fluent initiated* }.

In the next step, we had to develop a mechanism constructing the ASG from an ASSL specification. Here, the ASG is constructed by the ASSL framework by using a

special *declarative specification tree* created by the framework when parsing an AS specification [1, 2]. The declarative specification tree contains the hierarchical tier structure of the actual specification. Thus, enriched with the tier states S, it can be used to derive the composite multilevel structure of the ASG by taking into consideration that all the tier instances run concurrently as state machines. Thus, the tier states S are derived from the declarative specification tree and enriched with the appropriate atomic propositions AP. The latter are retrieved per state.

In addition, the so-called *operational evaluation* [1, 2] performed on the ASSL specification is used to derive all the transition relations R (S_1, Op, S_2) needed to connect the states S and thus, to construct the ASG. Here, an ASG is composed of nodes that can be presented formally as a tuple (s, R, AP) where: s is the tier state; R is a set of transition relations connecting the state s to other states via system operations; AP is a set of atomic propositions held in s. Similar to the declarative specification tree, the generated ASG is hierarchical, i.e., composed of multilevel composite tier states. Note that the generated ASG is stored in a flat file, which helps us trace the graph. Fig. 3 depicts the transformation of the declarative specification tree into an ASG, where the latter is presented at the highest possible level of abstraction comprising a single composite state "AS Active", which is a product machine consisting of product states.

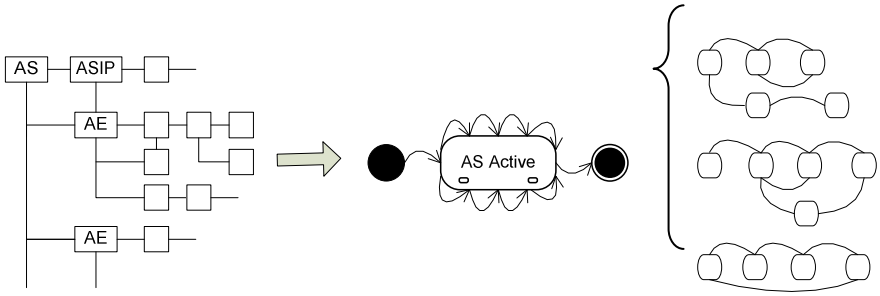

Fig. 3. Transformation of the Declarative Specification Tree into an ASG

4.2 Building the Model-Checking Engine

Next, we had to implement the model checking engine that should work over the following algorithm: *given that Φ is a correctness property expressed in a temporal logic formula, determine whether the "AS Active" tier state* (see Fig. 3) *satisfies Φ, which implies that all possible compositions of nested tier states satisfy Φ.*

Thus, the model-checking engine traverses all the possible paths in an ASG to check whether special correctness properties Φ (expressed in a temporal logic) are satisfied. In case such a property is not satisfied, the ASSL framework produces a counterexample. The latter is an execution path of the ASG for which the desired correctness property is not true.

At the time of writing, the model-checking engine is still under development. We are currently examining two possible solutions: 1) developing our own engine; or 2) integrating an already existing engine that can process the generated ASG file. Engines of current interest are SPIN [10] and GEAR [11]. In all approaches though, we

need to consider the so-called *state-explosion problem*. In general, the size of an ASG is at least exponential in the number of ASSL tier instances running concurrently in the system (recall that an ASG is a product machine). We are currently working on two possible solutions to that problem—*abstraction* and *prioritized tiers*. The first solution is to use composite tier states to abstract their nested tier states. Thus, given an original state graph G (derived from an ASSL specification) an abstraction is obtained by suppressing low-level tier states yielding a simpler and likely smaller state graph G_a. This reduces the total amount of states to be considered but is likely to introduce a sort of conservative view of the system where the abstraction ensures only that correctness of G_a implies correctness of G. The other possible solution is to prioritize ASSL tiers by giving their tier states a special probability weight pw. This can be used as a state-reduction factor to derive probability graphs G_{pw} with a specific level of probability weight, e.g., $pw > 0{,}5$. However, this approach is likely to introduce probability to the model-checking results, which correlates with the probability level of the graph G_{pw}.

4.3 Checking Liveness Properties

This section demonstrates how the ASSL built-in model-checking mechanism can perform formal verification to check *liveness* properties of an AS specified and generated with ASSL. Our example is the ASSL specification model for the NASA Voyager Mission [5]. In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs (autonomic elements) that follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. In this section, we use a sample from this specification to demonstrate how a liveness property such as "*a picture taken by the Voyager spacecraft will eventually result in sending a message to antennas on Earth*" can be checked with the ASSL model-checking mechanism. Note that the ASSL specification model for the NASA Voyager Mission is relatively large (over 1000 lines of specification code). Thus, we do not present the entire specification but a specification sample. For more details on that specification, please refer to [5].

```
POLICY IMAGE_PROCESSING {
   ....
   FLUENT inProcessingPicturePixels {
      INITIATED_BY { EVENTS.pictureTaken }
      TERMINATED_BY { EVENTS.pictureProcessed }
   }
   ....
   MAPPING {
      CONDITIONS { inProcessingPicturePixels }
      DO_ACTIONS { ACTIONS.processPicture }
   }
}

ACTION processPicture {
   ....
   DOES {
      ....
      call AEIP.FUNCTIONS.sendBeginSessionMsgs
      ....
   }
}
```

Fig. 4. The IMAGE_PROCESSING policy

Fig. 4 presents a partial ASSL specification of the **IMAGE_PROCESSING** self-management policy of the Voyager AE. Here the **pictureTaken** event will be prompted when a picture has been taken. This event initiates the **inProcessingPicturePixels** fluent. The same fluent is mapped to a **processPicture** action, which will be executed once the fluent

gets initiated. As it is specified, the **processPicture** action prompts the execution of the **sendBeginSessionMsgs** communication function (see Fig. 4), which puts a special message **x** on a special communication channel [5] (message **x** is sent over that channel). Note that the specification of both the **pictureTaken** event and the **sendBeginSessionMsgs** function is not presented here. As we have already mentioned in Section 4.1, the ASSL model-checking mechanism builds the ASG (autonomic system graph) from the ASSL specification. Here both the *declarative specification tree* and the *ASSL operational semantics* [1, 2] are used to derive tier states S and transition relations R, and to associate those tier states via the ASSL transition operations Op. Next the labeling function $L(s)$ (integrated in the model-checking mechanism) labels each tier state s with appropriate atomic propositions AP.

Fig. 5 presents a partial ASSL ASG of the sub-tiers of the Voyager AE. These sub-tiers are derived from the declarative specification tree constructed for the Voyager AE. Note that this ASG is a result of our analytical approach and for reasons of clarity it is simplified, i.e., not all the possible tier states are presented here.

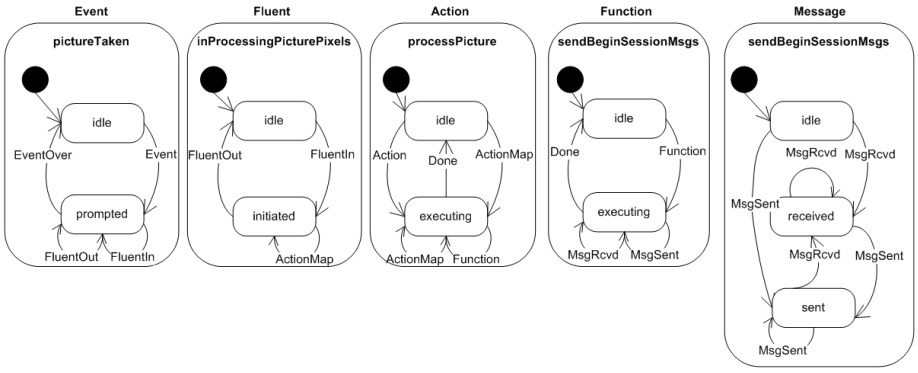

Fig. 5. State machines of the Voyager AE sub-tiers

As shown, each sub-tier instance forms a distinct *state machine* (basic machine) within the AE state machine and the AE state machine is a *Cartesian product* of the state machines of its sub-tiers. It is important to mention that by taking the Cartesian product of a set of basic sub-tier machines, we form a product machine consisting of product states. The latter are tuples of concurrent basic sub-tier states. Moreover, in the AE product machine, the ASSL state-transition operations Op are considered product transitions that move from one product state to another. Note that the states in the state machine of the whole AS product machine can be obtained by the Cartesian product of all the AE product machines. Thus, by considering the sub-tier state machines we construct the Voyager AE product machine (see Fig. 6). Note that this is again a simplified model where not all the possible product states are shown.

Fig. 6 presents the AE product states as large circles embedding the sub-tier states (depicted as smaller circles). Here we use the following aliases: e states for *Event state machine*; f states for *Fluent state machine*; a states for *Action state machine*; y states for *Communication Function state machine*; x states for *Message state machine*.

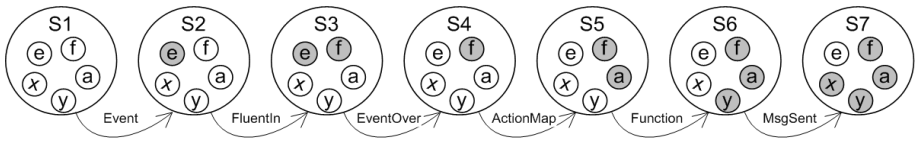

Fig. 6. Voyager AE product machine

Moreover, white circles present ”idle” state and gray circles present the corresponding ”active” state of the sub-tier state machine under consideration (such as: prompted for events, initiated for fluents, etc.; see Fig. 5).

Therefore, the formal presentation *(S; Op; R; S_0; AP; L)* (see Section 4.1) of the Voyager AE ASG is:

- *S = {S_1; S_2; S_3; S_4; S_5; S_6; S_7}*
- *Op = {Event; FluentIn; EventOver; ActionMap; Function; MsgSent}*
- *R = {(S_1;S_2;Event); (S_2;S_3;FluentIn); (S_3;S_4;EventOver); (S_4;S_5;ActionMap); (S_5;S_6;Function); (S_6;S_7;MsgSent)}*
- *S_0 = S_1 (initial state)*
- *AP = {* event *pictureTaken* occurs, event *pictureTaken* terminates, action *processPicture* starts, fluent *inProcessingPicturePixels* initiates, function *sendBeginSessionMsgs* starts, sends message *x }*
- *L(S):*
  - *L(S_1) = {* event *pictureTaken* occurs *};*
  - *L(S_2) = {* fluent *inProcessingPicturePixels* initiates *};*
  - *L(S_3) = {* event *pictureTaken* terminates *};*
  - *L(S_4) = {* action *processPicture* starts *};*
  - *L(S_5) = {* function *sendBeginSessionMsgs* starts *};*
  - *L(S_6) = {* sends message *x };*

Moreover, we consider the following correctness properties applicable to our case:

- *If an event occurs eventually a fluent initiates.*
- *If an event occurs next eventually it terminates.*
- *If a fluent initiates next actions start.*
- *If an action starts eventually a function starts.*
- *If a function starts eventually it sends a message.*

The ASSL model-checking mechanism uses the correctness property formulae to check if these are held over product states considering the atomic propositions *AP* true for every state. Thus, the ASSL framework is able to trace the state path shown in Fig. 6 and to validate the *liveness property* stated above. Note that in this example, we intentionally presented a limited set of atomic propositions *AP* and correctness properties. The former are derivable, that is, deduced from the operational evaluation of the ASSL specification. Moreover, the Voyager AE product machine presents only product states relevant to our case study.

5 Automatic Test Case Generation with ASSL

To allow post-implementation software verification with the ASSL framework, we are currently developing a novel *test-generator tool* based on change-impact analysis that will help the ASSL framework automatically generate test suites for self-managing policies. Conceptually, the test generator tool accepts as input an ASSL specification (see Section 2) comprising sets of policies Π that need to be tested and generates a set of test cases T as tuples $T\ \{P_{ex}, A\ \{I, R\}\}$ comprising an execution path P_{ex} and test attributes A. The latter is a tuple comprising needed inputs I and optional replacement ASSL constructs R. The replacement ASSL constructs are automatically or semi-automatically specified and generated as supplementary software stubs to ensure the execution of P_{ex}.

Table 2 presents a **privateMessageInsecure** replacement event that is intended to replace the original **privateMessageInsecure** event. As shown, the replacement event guarantees that this event will occur in the system because: 1) it does not have a **GUARDS** clause that prevents the event from firing if special conditions are not met; and 2) its activation (see the **ACTIVATION** clause in Table 2) is time-ensured; i.e., it does not depend on external factors.

Table 2. Original and replacement ASSL events

Original Event	Replacement Event
EVENT privateMessageInsecure { **GUARDS** { **NOT** **METRICS**.thereIsInsecureMsg } **ACTIVATION** { **CHANGED** { **METRICS**.thereIsInsecureMsg } } }	**EVENT** privateMessageInsecure { **ACTIVATION** { **PERIOD** { 1 **min** } } }

5.1 Test Generation Methodology

5.1.1 Policy Execution Paths

Formally, from a policy execution perspective, an ASSL-specified self-management 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:

$e \in Ea \cup Et$ 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 (state post-conditions that must be met after the action execution [1, 2]) 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 number of 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 [1, 2] 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:

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

5.1.2 ASSL Test Generator

With the ASSL Test Generator we are aiming at a novel tool based on change-impact analysis that helps the ASSL framework automatically generate high-quality test suites for self-management policies.

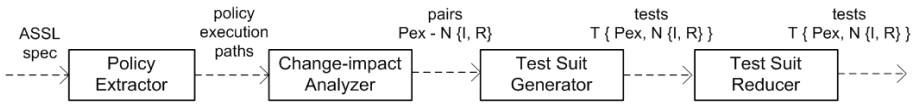

Fig. 7. Operational view of the ASSL Test Generator

As shown in Fig. 7, the test generator 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 policy 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 an 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.

5.1.3 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.

A. Evaluate what the conditions that must be met to have a specific fluent execution path ensured are:
   a. 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.
   b. 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 **ONERR_TRIGGERS** clauses);
      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* and ASSL *managed element functions* (both are sub-tiers in the ASSL specification model [1, 2]).
   c. 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.
B. Evaluate what the impact of having two or more fluent executing simultaneously is and what the conditions that must be met for that are. Generate test cases.
C. 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 Table 2), 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*.

6 Conclusion and Future Work

We have presented software verification mechanisms for ASs (autonomic systems) developed with the ASSL framework. The family of software-verification framework tools includes: *consistency checker*, *model checker*, and *test case generator*. Currently, the ASSL consistency checker is the only fully implemented tool. It automatically checks ASSL specifications for consistency errors and some design flaws. The latter are verified against special consistency rules implemented as semantic definitions.

We have also presented our experiences to-date in developing the model checker and test case generator tools for ASSL. To implement the model checker, we developed program structures and algorithms that help an ASSL specification be transformed into a state-transition graph composed of special tier states with associated atomic propositions and transition relations connecting those states. We are currently developing a model-checking engine that works on the state transition graph. In addition, possible solutions to the so-called state-explosion problem are considered.

The test case generator tool aims at automatic generation of test suites for self-management policies. A test case is generated with a policy-execution path and test attributes that come in the form of inputs and special replacement ASSL constructs ensuring the execution of a tested policy. 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 is going to 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 model checker and test-case generator tools 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 test-generation strategy. Moreover, it is our intention to build an animation tool for ASSL that will help to visualize counterexamples and trace erroneous execution paths.

Acknowledgment

This work was supported in part by Science Foundation Ireland grant 03/CE2/I303_1 to Lero—the Irish Software Engineering Research Centre.

References

1. Vassev, E.: Towards a Framework for Specification and Code Generation of Autonomic Systems. PhD Thesis, Computer Science and Software Engineering Department, Concordia University, Quebec, Canada (2008)
2. Vassev, E.: ASSL: Autonomic System Specification Language - A Framework for Specification and Code Generation of Autonomic Systems. LAP Lambert Academic Publishing (2009)
3. Murch, R.: Autonomic Computing: On Demand Series. IBM Press (2004)
4. Vassev, E., Hinchey, M., Paquet, J.: Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. In: Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008), pp. 1652–1657. ACM, New York (2008)
5. Vassev, E., Hinchey, M.: Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. In: Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), pp. 246–253. IEEE Computer Society, Los Alamitos (2009)
6. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
7. Vassev, E., Hinchey, M., Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 16–25 (2009)
8. Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., Steffen, B.: Component-oriented Behavior Extraction for Autonomic System Design. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA, pp. 66–75 (2009)
9. Visser, W., Havelund, K., Brat, G., Park, S.-J.: Model Checking Programs. In: Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE 2000). IEEE Computer Society, Los Alamitos (2000)
10. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston (2003)
11. Bakera, M., Renner, C.: GEAR - Game-based, Easy and Reverse Model Checking (2008), `http://jabc.cs.tu-dortmund.de/modelchecking/`