Developing Model-Checking Mechanisms for ASSL: An Experience Report

Emil Vassev and Mike Hinchey

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

Abstract. The Autonomic System Specification Language (ASSL) is a formal method dedicated to autonomic computing, and as such, assists developers with *formal specification*, *validation* and *code generation* of autonomic systems. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. Moreover, one of the major objectives of the framework is to assure the correctness of autonomic systems via the inclusion of tools targeting model checking. In this paper, we report our experience in developing model-checking mechanisms for ASSL.

Keywords: model checking, formal methods, ASSL, autonomic computing

1 Introduction

The Autonomic System Specification Language (ASSL) [1, 2] 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 framework. Being dedicated to autonomic computing (AC) [3], 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 skeleton. The ASSL formal validation is addressed by multiple model checking mechanisms, some fully implemented and some still under development.

In general, model checking advocates for formal verification whereby software programs are automatically checked for specific flaws by considering correctness properties. In ASSL, some of those properties are defined as semantic definitions forming a theory that aids in the construction of correct AS specifications. For the purpose of developing flawless ASs, we are considering four distinct model checking mechanisms for ASSL: 1) a consistency checker; 2) a built-in model checker; 3) a mechanism for mapping ASSL specifications to a formal notation with provided tool support for model checking; and 4)a post-implementation model checker. Whereas the first three model-checking methods check ASSL specifications, the fourth one

verifies the generated Java code. Note that despite careful specification and the existence of ASSL-level model checking, it is possible to generate ASs containing fatal errors (e.g., deadlocks). This is mainly due to the so-called state-explosion problem. 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. In this paper, we report our experience in developing the ASSL model checking mechanisms in the course of research projects at Lero–the Irish Software Engineering Research Centre.

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 our experience in the development of the four ASSL model checking mechanisms. Section 4 briefly outlines a case study where the built-in model checking approach is applied. Finally, Section 5 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).

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	

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 allows us 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). 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 and ASIP, 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 Model Checking with ASSL

The ASSL framework helps developers edit and validate ASSL specifications and generate Java code, i.e., the ASSL toolset provides powerful tools needed to formally process an ASSL specification and automatically generate the corresponding implementation. The following subsections present the ASSL model checking mechanisms, used to validate the ASSL specifications.

3.1 Consistency Checking

The ASSL tiers can be classified as *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 eventually model checking) and code generation. The declarative specification tree is created by the framework when parsing an AS specification and copes with 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*. The "*heavy*" consistency checking can be considered as a form of model checking, where the model is verified against predefined correctness properties.

The correctness properties are *ASSL semantic definitions* [1, 2] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL)[1] [5], currently ASSL does not incorporate a FOLTL engine, and thus, the consistency

[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.

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

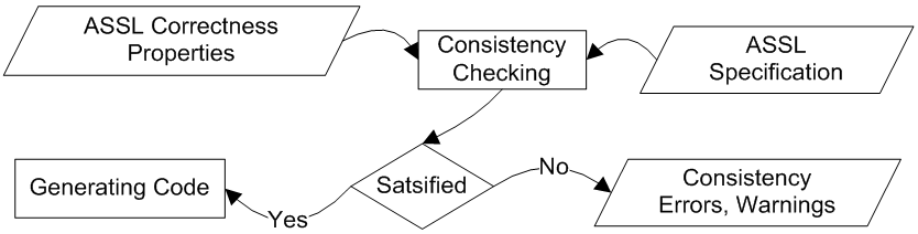

Fig. 1. Consistency Checking with ASSL.

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 consistency checker, because the processed ASSL specification violates the autonomicity rule (the entire ASSELF_MANAGEMENT sub-tier comprising the self-management policies is commented).

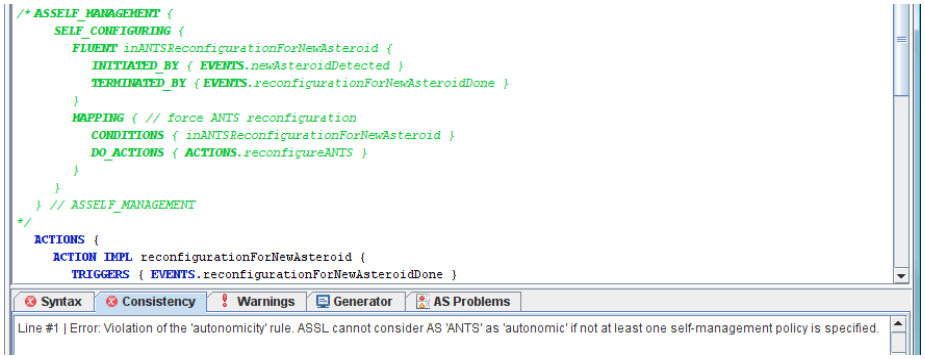

Fig. 2. Checking for "Autonomicity" with the ASSL Consistency Checker.

3.2 Built-in Model Checker

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 [4]. An ASG formally stems from the concept of Kripke Structure [5].

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.

3.2.1 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.

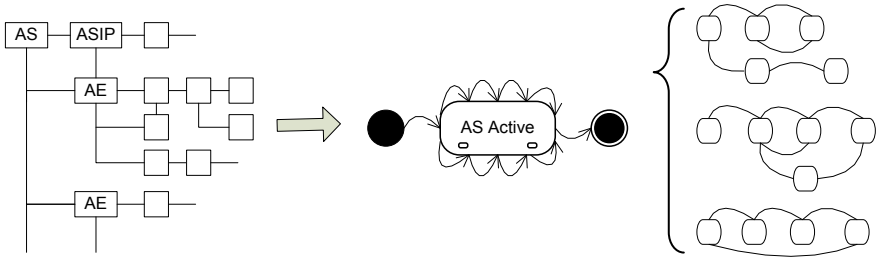

Fig. 3. Transformation of the Declarative Specification Tree into an ASG.

3.2.2 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 [6] and GEAR [7]. 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}.

3.3 External Model Checker

Another research direction of ours is towards mapping ASSL specifications to special *service logic graphs* supporting the so-called r*everse model checking* [8]. In this approach, to complement the original textual view of an ASSL specification, and in perspective to visualize and reify certain aspects of the operational semantics of ASSL, we map selected ASSL-specified behavioral elements to GEAR's behavioral models. These can be visualized as special Service Logic Graphs (SLGs) in the jABC framework [9] (of which GEAR is the model checking plugin) and analyzed, guiding the user through the processes and workflows of the specified autonomic system. Note that these models are directly amenable to model checking. SLGs themselves are composed of reusable building blocks that are called Service Independent Building Blocks, and may represent 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 ofa Kripke Transition System (KTS) [5]. Similar to any KTS, SLGs are graph structures with labeled branches and nodes that are enriched with atomic propositional properties, thus sufficing to adopt the established model checking technologies for SLGs.

From the point of view of model generation, AS and AE specifications are structurally identical with reference to events, self-management policies and actions, but differ in terms of scoping - while the AS specification has a global scope, the AE specification is only valid for the element in question (see Section 2). Due to the similarities, we focus on the description of the 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).

Fig. 4 shows a specification fragment of an ASSL specification of the Voyager spacecraft [10] (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).

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 can be activated by messages, other events, actions or metrics [1, 2]. In our behavioral model, events are mapped to homonymous Branches. In Fig. 4, the behavioral model starts with the

event timeToTakePicture. It initiates the self-management policy (fluent) inTakingPicture.

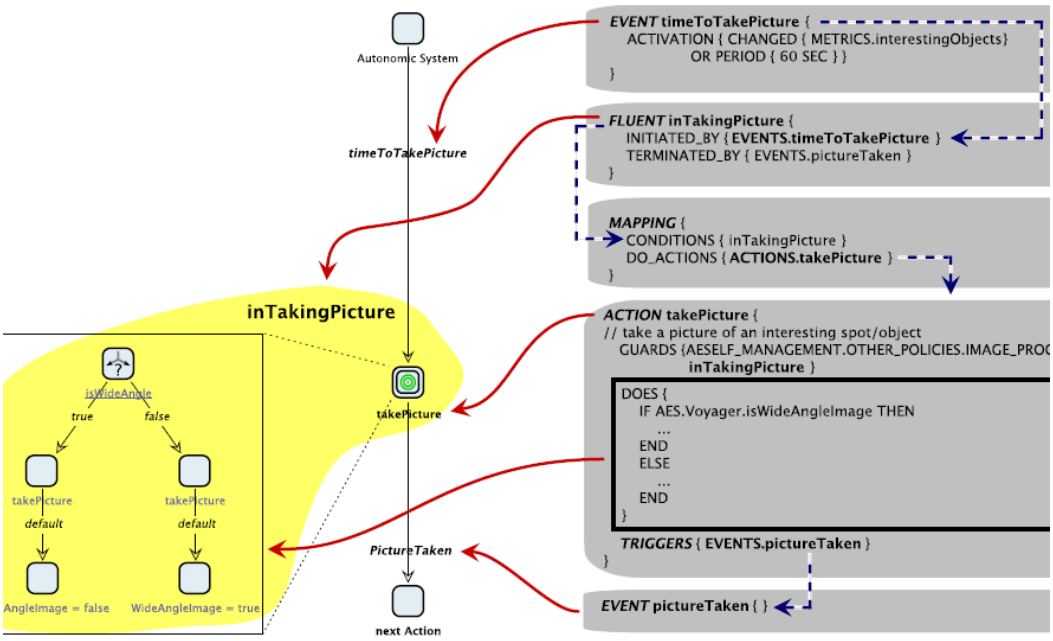

Fig. 4. Action, Event, Fluent, and Mapping in KTS behavioral model representation.

An AE self-management policy defines the behavior of the AE by connecting specific system states (expressed with fluents) with the intended (re)action (expressed with mappings) (see Section 2). Fluents and mappings 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 AS is changed to *intakingPicture*. This change activates an action: *takePicture* which is specified in the *mapping* section of the self-management policy. The self-management policy which connects the event to actions is additionally used to annotate the nodes in the behavioral model with atomic propositions (APs). The name of the AP is equal to the name of the fluent. They can later be used for model checking.

Actions are routines performed by an AE or AS (global and local) [1,2]. In our behavioral model, they are the second essential element. 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 [1, 2] (see the ASSL specification in Fig.4) represents the body of a node. It can be a single action (then the node is an atomic node), or a more complex structure where the latter is represented as an entire behavioral model. We then model them as a SLG hierarchy, as shown in Figure 4: the node *takePicture* has a corresponding submodel presented on the left.

The action's *guards*, *returns* and *outcomes* [1, 2] are used for verification. We offer two possibilities for verification:

- The GEAR's Localchecker mechanism 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* [5] constraints. Internally, GEAR uses the modal μ - calculus [11] enriched with forward and backward modalities, so it is best equipped, for example, to express dataflow properties, or other behavioral constraints such as temporal logic formulas.

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

3.4 Post-implementation Model Checker

In this approach, we rely on the Java PathFinder [12] tool to perform model checking on the ASSL-generated Java code.

3.4.1 Java PathFinder

Java PathFinder is a post-implementation model checker tool written in Java and targeted at Java programs [12, 13]. It can check Java programs for deadlocks, invariants and user-defined assertions in the code. Moreover, properties expressed in Linear Temporal Logic [14] can be checked. In general, it is claimed that Java PathFinder is capable of checking any Java program that does not rely on native methods. However, it is important to mention that the state-explosion problem limits the size of the applications that can be checked effectively up to 10,000 lines of code. Similar to any regular model checking tool, Java PathFinder performs exhaustive testing. The difference is that it works on the real Java code instead of on a state graph. Here, the basic technique is multiple execution of the program under consideration to check all the possible executions for paths that can lead to property violations, such as deadlocks or unhandled exceptions. If an error is found, Java PathFinder reports the execution path that leads to it. Note that every execution step is recorded, so we can trace the execution path that gets to property violation.

Fig. 5 depicts the operational model of Java PathFinder. As depicted, different components (tools) work by accompanying the execution of the compiled Java program (in Java bytecode), e.g., an ASSL-generated AS compiled to Java bytecode

with a regular Java Compiler. As shown in Fig. 5, special *configurable search strategies* are provided to solve the problem of state explosion. Because for large (more than 10,000 lines of code) applications the whole state space cannot be searched effectively, these search strategies are used to direct the search.

In addition, different *state-reduction techniques* can help to reduce the number of states that have to be stored:

- Special *heuristic choice generators* are provided to set possible choices where a certain state does not have to be complete. These generators have the form of Java PathFinder APIs that can be embedded in the tested applications.
- A special *library abstraction* per state reduces the overhead coming from tracking the run-time data changes taking place in the checked Java application. Note that all the heap, stack, and thread changes are stored by default. This can cause a big overhead if abstraction is not provided.

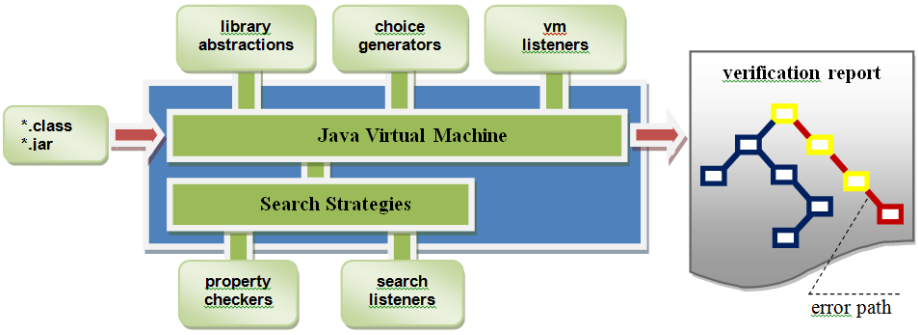

Fig. 5. Java PathFinder operational model (elaborated from [12])

3.4.2 Embedding Java PathFinder in ASSL

In general, Java PathFinder provides capabilities for non-deterministic testing via random input data generators [12] that can be embedded in the tested Java application. Special APIs are provided, which can significantly ease the creation of test drivers. Hence, the ASSL framework can automatically generate such test drivers based on the Java PathFinder API. ASSL could generate these special test drives as *non-deterministic choices* implemented in the generated code. Here, to simulate non-deterministic testing we rely on two Java PathFinder capabilities – *backtracking* and *state matching*.

With *backtracking*, we use the Java PathFinder tool to restore previous execution states, which helps to determine whether there are unexplored choices left. Therefore, if an end state is reached, backward steps can be performed to find execution paths that are still not executed, and thus, the program does not have to be re-executed from the very beginning.

With *state matching*, the Java PathFinder checks whether a specific execution path has already been explored any time when an ASSL-generated non-deterministic

choice is reached. In such a case, model checking does not continue along the current execution path, but does backtracking to reach the *nearest non-explored path* that starts from the nearest non-deterministic choice. For example, the following *run()* method could be generated by the ASSL framework for an autonomic element.

```
public class AE_WORKER {
   ...
   public void run () {
      boolean cond = Verify.getBoolean();
      if (cond)   {  ... }
      else  {  ... }
   }
   ...
}
```

Note that autonomic elements are generated by ASSL as Java Threads [1, 2]. Here, a non-deterministic PathFinder choice point will be generated (see `cond = Verify.getBoolean`) to test two different paths of execution of the autonomic element.Both *backtracking* and *state matching* techniques will be used to trace the two possible execution path – when `cond = true` and when `cond = false`.

4 Case Study: Checking Liveness Properties with ASSL

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 [10]. 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 [10].

```
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. 6. The IMAGE_PROCESSING policy.

Fig. 6 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. 6), which puts a special message **x** on a special communication channel [10] (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 3.2, 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. 7 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.

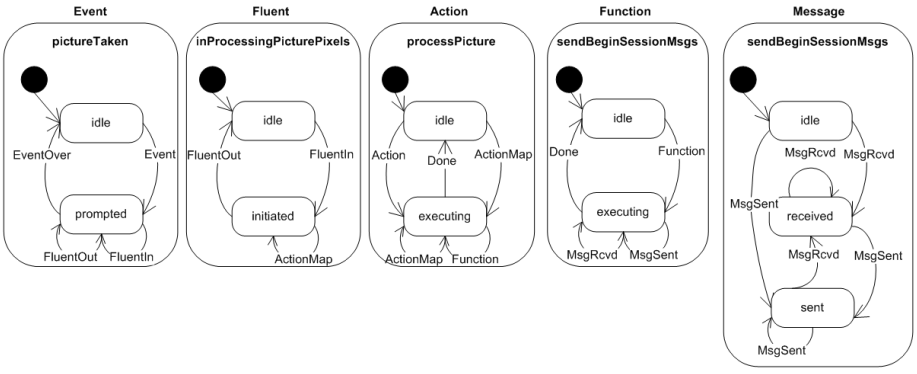

Fig. 7. State machines of the Voyager AE sub-tiers.

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. 8). Note that this is again a simplified model where not all the possible product states are shown.

Fig. 8 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*. 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. 7).

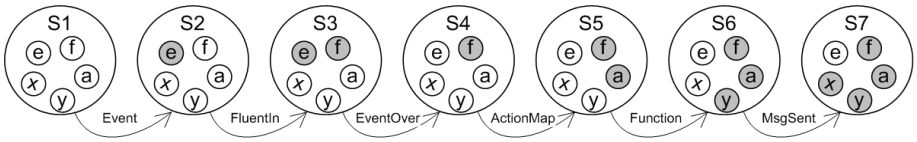

Fig. 8. Voyager AE product machine.

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 Conclusion and Future Work

We have presented our experience to-date in developing model-checking software verification mechanisms for the ASSL framework. Currently ASSL supports a family of software-verification framework tools (implemented or still under implementation) including a *consistency checker*, a *built-in model checker*, an *ASSL-to-SLG specification mapper* to support external model checking with the GEAR model checker and an *integration of the Java PathFinder* model checker to support post-implementation model checking. 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.

The other model mechanisms for ASSL require different implementation approaches. For example, to implement the built-in 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.

Our plans for future work are mainly concerned with further development of the model checker and test-case generator tools for ASSL. Moreover, in addition to the model-checking mechanisms, we are currently working on a special *test-case generator*, which 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 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.

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., and Quigley, A.: Model Checking for Autonomic Systems Specified with ASSL. Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA (2009) 16–25
5. Clarke, E. M., Grumberg, O., and Peled, D.: Model Checking. MIT Press (1999)
6. Holzmann, G. J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Boston, Massachusetts, USA (2003)
7. Bakera, M., Renner, C.: GEAR - Game-based, Easy and Reverse Model Checking. url: http://jabc.cs.tu-dortmund.de/modelchecking/ (2008)
8. Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., and Steffen, B.: Component-oriented Behavior Extraction for Autonomic System Design. Proceedings of the First NASA Formal Methods Symposium (NFM 2009), NASA (2009) 66–75
9. Nagel, R.: jABC. http://www.jabc.de
10. Vassev, E. and Hinchey, M.: Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09), IEEE Computer Society (2009) 246–253
11. Kozen, D.: Results on the propositional μ-calculus. ICALP. Vol. 140 of LNCS. Springer-Verlag (1982) 348–359
12. Java PathFinder. http://javapathfinder.sourceforge .net/
13. Visser, W., Havelund, K., Brat, G., Park, S.-J.: Model Checking Programs. Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00). IEEE Computer Society (2000)
14. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press (2008)