Model Checking for Autonomic Systems Specified with ASSL

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

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

Aaron Quigley
Lero, University College
Dublin, Ireland
aquigley@ucd.ie

Abstract

Autonomic computing augurs great promise for deep space exploration missions, bringing onboard intelligence and less reliance on control links. As part of our research on the ASSL (Autonomic System Specification Language) framework, we have successfully specified autonomic properties, verified their consistency, and generated prototype models for both the NASA ANTS (Autonomous Nano-Technology Swarm) concept mission and the NASA Voyager mission. The first release of ASSL provides built-in consistency checking and functional testing as the only means of software verification. We discuss our work on model checking autonomic systems specified with ASSL. In our approach, an ASSL specification is translated into a state-transition model, over which model checking is performed to verify whether the ASSL specification satisfies correctness properties. The latter are expressed as temporal logic formulae expressed over sets of ASSL constructs. We also discuss possible solutions to the state-explosion problem in terms of state graph abstraction and probability weights assigned to states. Moreover, we present an example case study involving checking liveness properties of autonomic systems with ASSL.

1 Introduction

As software increasingly exerts influence over our daily lives, the need for more reliable software systems has become critical. However, contemporary software systems are becoming extremely complex and so too is the task of ensuring their correctness. This is the reason why a great deal of research effort is devoted to developing software verification methods. A promising, and lately popular, technique for software verification is model checking [1], [2]. This approach advocates formal verification tools whereby software programs are automatically checked for specific flaws by considering correctness properties expressed in temporal logic. Numerous formal tools allowing verification by model-checking have been developed, such as Spin [3], Emc [4], Tav [5], Mec [6], XTL [7], etc. Despite best efforts and the fact that model checking has proved to be a revolutionary advance in addressing the correctness of critical systems, software assurance for large and highly-complex software is still a tedious task. The reason is that high complexity is a source of software failures, and standard model checking approaches do not scale to handle large systems very well due to the so-called state-explosion problem [1], [2].

NASA Engages in Autonomic Computing. Experience has shown that software errors can be very expensive and even catastrophic in complex systems such as spacecraft. An example is the malfunction in the control software of Ariane-5, which caused the rocket's crash 36 seconds after its launch [2].

There are a number of valid complexity-reduction approaches, including the use of formal methods and autonomic computing. The latter is an emerging field for the development of large-scale self-managing complex systems. The Autonomic Computing (AC) paradigm draws inspiration from the human (and mammalian) body's autonomic nervous system [8]. The idea is that software systems can manage themselves and deal with dynamic requirements, as well as unanticipated threats, automatically, just as the body does, by handling complexity through self-management based on high-level objectives.

NASA is currently approaching AC with great interest, recognizing in AC a bridge towards "the new age of space exploration" whereby spacecraft should be independent, autonomous, and "smart" [9]. NASA's New Millennium Project [9] addresses space-exploration missions where AC software controls spacecraft entirely. Both the Autonomous Nano-Technology Swarm (ANTS) concept mission [10] and the Deep Space One mission [9] involve the next generation of AC-based unmanned spacecraft systems. In such systems, autonomic computing software turns spacecraft into autonomic systems, i.e., capable

of planning and executing many activities onboard to meet the requirements of changing objectives and harsh external conditions. NASA is engaging in AC research to find a solution to large-scale automation for deep space exploration. However, automation implies sophisticated software, which requires new development approaches and new verification techniques.

The ASSL Approach to Autonomic Computing. The Autonomic System Specification Language (ASSL) [11] is an initiative for self-management of complex systems where we approach the problem of formal specification, validation, and code generation of autonomic systems within a 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. The current validation approach in ASSL is a form of consistency checking performed against a set of semantic definitions. The latter form a theory that aids in the construction of correct autonomic system (AS) specifications. From any valid specification, ASSL can generate an operational Java application skeleton. As a part of the framework validation and in the course of a new ongoing research project at Lero—the Irish Software Engineering Research Center, ASSL has been used to specify autonomic properties and generate prototype models of the NASA ANTS concept mission [12], [13], [14] and NASA's Voyager mission [15]. Both ASSL specifications and generated prototype models have been used to investigate hypotheses about the design and implementation of intelligent swarm-based systems and possible future Voyager-like missions incorporating the principles of AC.

Research Problem. Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. However, our experience with ASSL demonstrated that errors can be easily introduced while specifying large systems. Although the ASSL framework takes care of syntax and consistency errors, it cannot handle logical errors.

We are currently investigating a few possible approaches to ensure the correctness of the ASSL specifications and that of the generated autonomic systems:

- It is possible to improve the current ASSL consistency checker with assertion and debugging techniques. These should allow for a good deal of *static analysis* of an ASSL specification. Moreover, this approach should introduce flexibility to the ASSL consistency checker by providing various options that affect what static analysis the latter performs and what results are shown. Although currently under development, this approach will improve the verification process, but will not be sufficient to assert safety (e.g., freedom from deadlock) or liveness properties (cf. Section 2).
- ASSL generates operational Java code, which can be used to perform a sort of post-implementation model checking. For example, Java Pathfinder [16], developed at NASA Ames, can be used to verify the generated Java code. This approach is efficient when applied to smaller systems, but cannot be used to discover logical errors in the ASSL specifications because it is employed at a later stage of the software lifecycle.
- The most promising and most effective approach is model checking. In the course of this project, we consider two possible approaches:
  - Another paper presented at this event proposes mapping ASSL specifications to special Service Logic Graphs [17]. The latter support reverse model checking and games, and enable intuition of graphical models and expression of constraints in mu-calculus.
  - This paper presents a model-checking mechanism that takes as input an ASSL specification and produces as output a finite state-transition system such that a specific property in question is satisfied if and only if the original ASSL specification satisfies that property.

Benefits for Space Systems. An ASSL model-checking mechanism will be the next generation of the ASSL consistency checker based on automated reasoning. By allowing for automated system analysis and evaluation, this mechanism will complete the autonomic system development process with ASSL. The ability to verify the ASSL specifications for design flaws can lead to significant improvements in both specification models and generated autonomic systems. Subsequently, ASSL can be used to specify, validate and generate better prototype models for current and future space-exploration systems. These prototype models can be used for feature validation and simulated experimental results. The ability to simulate exploration missions with hypothesized possible autonomic features gives significant benefits.

2 Consistency Checking with ASSL

In general, ASSL considers ASs as being composed of autonomic elements (AEs) interacting over interaction protocols. To specify autonomic systems, ASSL exposes a multi-tier specification model that is designed to be scalable and to expose a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. By their virtue, the ASSL tiers are abstractions of different aspects of the AS under consideration, such as policies, communication interfaces, execution semantics, actions, etc. There are three major tiers (three major abstraction perspectives), each composed of sub-tiers [11]:

- AS tier — forms a general and global AS perspective, where we define the general system rules in terms of *service-level objectives (SLO)* and *self-management policies*, *architecture topology*, and *global actions*, *events*, and *metrics* applied in these rules.
- AS Interaction Protocol (ASIP) tier — forms a communication protocol perspective, where we define the means of communication between AEs. The ASIP tier is composed of *channels*, *communication functions*, and *messages*.
- AE tier — forms a unit-level perspective, where we define interacting sets of individual autonomic elements (AEs) with their own behaviour. This tier is composed of AE rules (*SLO* and *self-management policies*), an *AE interaction protocol (AEIP)*, *AE actions*, *AE events*, and *AE metrics*.

In general, we can group the ASSL tiers into groups of *declarative* (or imperative) and *operational* tiers. 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.

Consistency checking (cf. Figure 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 operations: *light* (checks for type consistency, ambiguous definitions, etc.) and *heavy* (checks whether the specification model conforms to special correctness properties). The correctness properties are ASSL semantic definitions [11] defined per tier. Although, they are expressed in First-Order Linear Temporal Logic (FOLTL) [1] [1], [2], 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 $\forall$ (forall) and $\exists$ (exists) work over sets of ASSL tier instances. In addition, these operators are translated by taking their first argument as a logical atom that contains a single unbound tier variable. Ideally, this atom has a relatively small number of ground tier instances, so the combinatorial explosion generally produced by these statements is controlled.

[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.

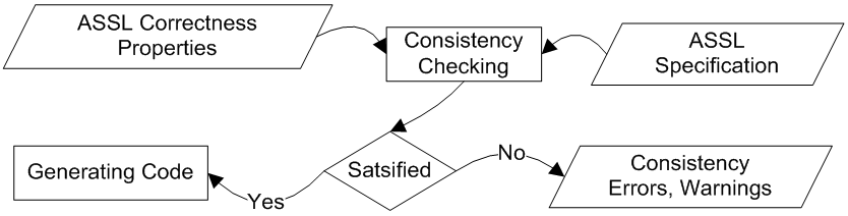

Figure 1: Consistency Checking with ASSL

It is important to mention that the consistency checking mechanism generates consistency errors or 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. Although considered efficient, the ASSL consistency checking mechanism has some major drawbacks.

- It does not consider the notion of time, and thus, temporal FOLTL operators such as $\Box$ (always), $\bigcirc$ (next), $\Diamond$ (eventually), $\mathscr{U}$ (until), $\mathscr{W}$ (waiting-for), etc., are omitted. Therefore, the ASSL consistency checking is not able to assert safety (e.g., freedom from deadlock) or liveness properties (e.g., a message sent is eventually received).
- The interpretation of the FOLTL formulas into Java statements is done in an analytical way and thus the introduction of errors is possible.
- There is no easy way to add new correctness properties to the consistency-checking mechanism.

3 Model Checking with ASSL

In general, model checking provides an automated method for verifying finite state systems by relying on efficient graph-search algorithms. The latter help to determine whether or not system behavior described with temporal correctness properties holds for the system's state graph.

In ASSL, the model-checking problem is: given autonomic system A and its ASSL specification a, determine in the system's state graph g whether or not the behavior of A, expressed with the correctness properties p, meets the specification a. Formally, this can be presented as a triple (a, p, g) where: a is the ASSL specification of the autonomic system A, p presents the correctness properties specified in FOLTL, and g is the state graph constructed from the ASSL specification in a labeled transition system (LTS) [2] format. The first step in the ASSL model-checking mechanism is to construct from an ASSL specification a the corresponding state graph g where the desired correctness properties p can be verified.

ASSL Tier States. The notion of state in ASSL is related to tiers. The ASSL operational semantics considers a state-transition model where the so-called *operational tiers* can be in different *tier states*, e.g., SLO can be evaluated as *satisfied* or *not satisfied* [11]. An ASSL autonomic system transits from one state to another when a particular tier evolves from a tier state to another tier state. Here, the LTS of an autonomic system specified with ASSL consists of high-level composite states composed of multilevel nested states; i.e., a tier state is determined by the states of the tiers (sub-tiers) nested in that tier.

In ASSL, special system operations cause ASSL tiers to evolve from one tier state to another tier state. Here, the tier state evolution caused by a system operation Op can be denoted as $\sigma \xrightarrow{Op(x_1,x_2,....,x_n)} \sigma'$ where the operation $Op(x_1,x_2,....,x_n)$ can be a parametric operation (e.g. *AS/AE action*, *ASIP/AEIP function*, or an abstract function denoting an operation performed by the framework) which takes n arguments. ASSL considers twenty system state-transition operations as shown in Figure 2.

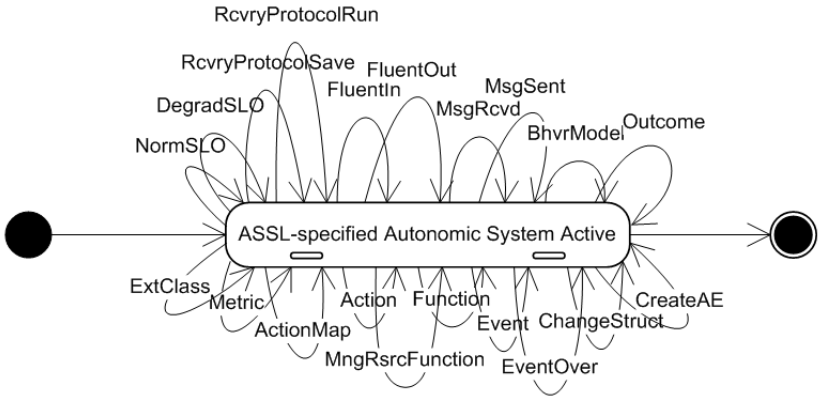

Figure 2: High-level LTS for ASs Specified with ASSL

Formally, an ASSL LTS can be presented as a tuple (S, Op, R, S_0, AP, L) [2] where: S is the set of all possible ASSL tier states; Op is the set of special ASSL state-transition operations; $R \subseteq S \times Op \times S$ are the possible transitions; $S_0 \subseteq S$ is a set of initial tier states; AP is a set of atomic propositions; $L : S \longrightarrow 2^{AP}$ is a labeling function relating a set $L(s) \in 2^{AP}$ of atomic propositions to any state s, i.e., a set of atomic propositions true in that state.

Constructing the ASSL LTS. The ASSL LTS is constructed by the ASSL framework by using the *declarative specification tree* (created by the framework when parsing an AS specification) and by applying the ASSL operational semantics [11]. The *declarative specification tree* contains the hierarchical tier structure of the actual specification. Thus, enriched with the possible tier states, it can be used to derive the composite multilevel structure of the specification's LTS by taking into consideration that all the tiers and sub-tiers run concurrently as state machines. The operational evaluation of the ASSL specification, can derive all the transition relations R and associate the tier states via the transition operations Op. Similar to the declarative specification tree, the generated ASSL LTS model is hierarchical, i.e., composed of multilevel composite tier states. Figure 3 depicts the transformation of the *declarative specification tree* into an ASSL LTS, where the latter is presented at the highest possible level of abstraction comprising a single composite state "AS Active" (cf. Figure 2).

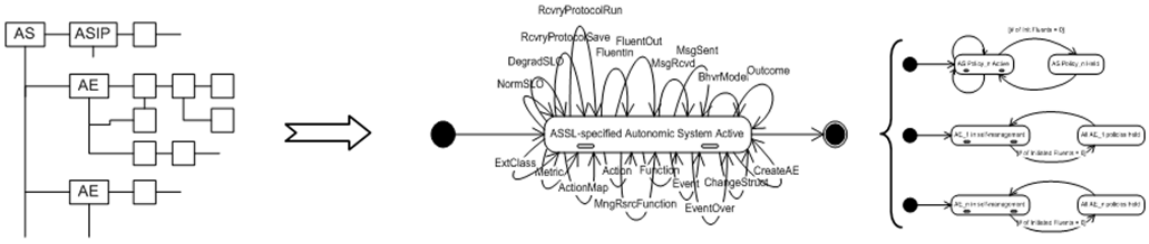

Figure 3: Transformation of the Declarative Specification Tree into an ASSL LTS

Considering implementation, the ASSL LTS can be implemented as a graph G, where the nodes will be tier states each encoded with transition relations $R \subseteq S \times Op \times S$.

ASSL Atomic Propositions and Labeling Function. In order, to make the ASSL LTS appropriate for model checking, we need to associate with each tier state a set of atomic propositions AP true in that state. In general, we should consider two types of AP — *generic* and *derivable*. Whereas the former

should be generic (independent of the specification specifics) for particular tiers, the latter should be deduced by the operational evaluation of the ASSL specification. The ASSL *AP* should be about the ASSL tiers and should be expressed in propositional logic (to include them in the correctness properties expressed as FOLTL formulas). The full set of ASSL *AP* should be predefined and passed as input to a special labeling function $L(s)$ in the model checking mechanism, which labels (relates) each tier state s with appropriate *AP*. To perform this task, $L(s)$ should employ an algorithm that considers the ASSL operational semantics, the ASSL code generation rules [11], and concurrency among the ASSL tiers.

Therefore, the implementation graph of the ASSL LTS is composed of nodes, which can be presented formally as a tuple (s, R, AP_g, AP_d) where: s is the tier state; R is a set of transition relations connecting the state s to other states via system operations; AP_g is a set of generic atomic propositions related to s; AP_d is a set of deducted atomic propositions related to s.

Model Checking Algorithm. Given that Φ is a correctness property expressed in FOLTL (i.e., a temporal logic formula), determine whether the "AS Active" tier state (cf. Figure 2) satisfies Φ, which implies that all possible compositions of nested tier states satisfy Φ. Note that $s \vdash \Phi$ iff $L(s) \vdash \Phi$.

State Explosion Problem. A typical ASSL-specified system is composed of multiple AEs (in ANTS for example they could number over 1000) each composed of large number of concurrent processes. The latter are autonomic elements implementing *self-management policies*, which spread internally into special concurrent *fluents* [2] [11], *events*, *metrics*, etc. Thus, the biggest problem stemming from this large number of concurrent processes is the so-called state explosion problem [1], [2]. In general, the size of an ASSL state graph is at least exponential in the number of tiers running as concurrent processes, because the state space of the entire system is built as the Cartesian product of the local state of the concurrent processes (concurrently running tiers).

We are currently investigating 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 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 [18] where the abstraction ensures only that correctness of G^a implies correctness of G.

The other possible solution, we have termed *prioritized tiers*, because the idea is to prioritize ASSL tiers by giving their tier states a special probability weight pw. Thus, the new formal presentation of the graph nodes will be (s, R, AP_g, AP_d, pw) where $0 \leq pw \leq 1$ is the probability weight assigned to each graph node. This can be used as a state-reduction factor to derive probability graphs G^{pw} with a specific level of probability weight, e.g., $pw > 0.3$. However, this approach is likely to introduce probability to the model-checking results, which correlates with the probability level of the graph G^{pw}.

Figure 4 depicts a global view of model checking in ASSL as it has been explained in this section. Note that in the case that a correctness property is not satisfied, the ASSL framework returns a *counterexample*. The latter is an execution path of the ASSL LTS for which the desired correctness property is not true. If model checking has been performed on the entire ASSL LTS, then the property does not hold for the original AS specification. Otherwise, in the case that a reduced ASSL LTS has been used (abstraction and/or prioritized tiers state-explosion techniques have been applied), the information provided by the counterexample is then used to refine the reduced model. For example, some composite states are presented with their low-level nested states, or a low-level probability weight is used in order to obtain a more accurate representation of the original model.

[2] An ASSL fluent is a special state with timed duration, e.g., a state like "performance is low". When the system gets into that specific condition, the fluent is considered to be initiated.

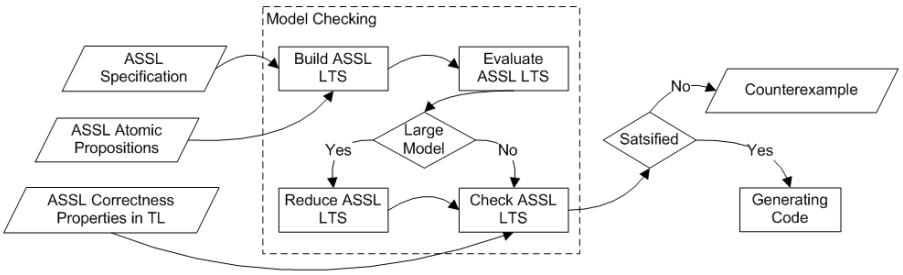

Figure 4: Model Checking in ASSL

4 Example: Checking Liveness Properties with ASSL

In this section, we demonstrate how the ASSL model checking mechanism can perform formal verification to check liveness properties of ASs specified and generated with ASSL. Our example is the ASSL specification model for the NASA Voyager Mission [15]. In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs that follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. Here 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 the specification, please refer to [15].

Figure 5 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 (cf. Figure 5), which puts a special message x on a special communication channel [15] (message x is sent over that channel). Note that the specification of both the `pictureTaken` event and the `sendBeginSessionMsgs` function is not presented here.

```
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
      ....
   }
}
```

Figure 5: ASSL IMAGE_PROCESSING policy at Voyager AE

As we have already mentioned in Section 3, the ASSL model checking mechanism should build the ASSL LTS from the ASSL specification. Here both the *declarative specification tree* and the ASSL operational semantics [11] will be 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) will label each tier state s with appropriate atomic propositions AP.

Product Machine. Figure 6, presents a partial ASSL LTS 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 LTS 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.

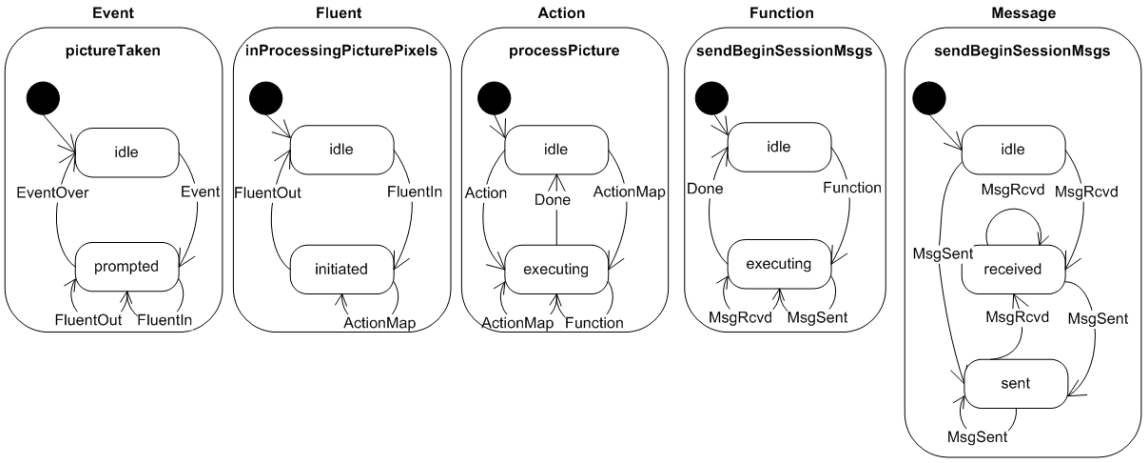

Figure 6: State machines of the Voyager AE sub-tiers

Here 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 (cf. Figure 7). Note that this is again a simplified model where not all the possible product states are shown. Figure 7 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.; cf. Figure 6).

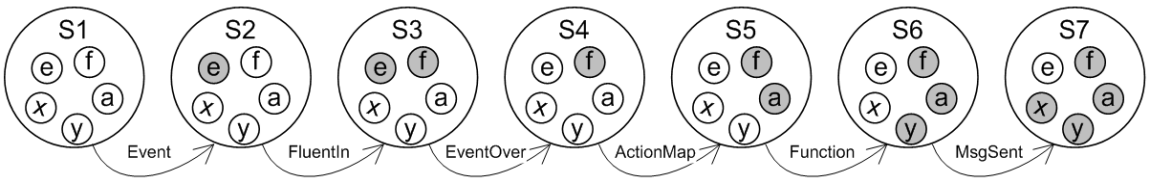

Figure 7: Partial Voyager AE product machine

Therefore, the formal presentation (S, Op, R, S_0, AP, L) (cf. Section 3) of the Voyager AE LTS is:

- $S = \{S1, S2, S3, S4, S5, S6, S7\}$
- $Op = \{\texttt{Event}, \texttt{FluentIn}, \texttt{EventOver}, \texttt{ActionMap}, \texttt{Function}, \texttt{MsgSent}\}$

- $R = \{(S1, S2, \texttt{Event}), (S2, S3, \texttt{FluentIn}), (S3, S4, \texttt{EventOver}), (S4, S5, \texttt{ActionMap}), (S5, S6, \texttt{Function}), (S6, S7, \texttt{MsgSent})\}$
- $S_0 = S1$ (initial state)
- AP ={event `pictureTaken` occurs, event `pictureTaken` terminates, action `processPicture` starts, fluent `inProcessingPicturePixels` initiates, function `sendBeginSessionMsgs` starts, sends message x}
- $L(S1)$ ={event `pictureTaken` occurs}; $L(S2)$ ={fluent `inProcessingPicturePixels` initiates}; $L(S3)$ ={event `pictureTaken` terminates}; $L(S4)$ ={action `processPicture` starts}; $L(S5)$ ={function `sendBeginSessionMsgs` starts}; $L(S6)$ ={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 shall use the correctness property formulae to check if these are held over product states considering the atomic propositions AP true for that state. Thus, the ASSL framework will be able to trace the *state path* shown in Figure 7 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 (cf. Section 3). Moreover, the Voyager AE product machine presents only product states relevant to our case study.

5 Conclusion and Future Work

We have presented our approach towards the development of a model checking mechanism in the ASSL framework. The principle technique is to transform an ASSL specification into a state graph enriched with atomic properties, which is used to verify whether special correctness properties are satisfied by that specification. Currently, ASSL provides a consistency checking mechanism to validate autonomic systems specified with ASSL against correctness properties. Although proven to be efficient with handling consistency errors, this mechanism cannot handle logical errors. Therefore, a model checking mechanism will complete the ASSL framework, allowing for automated system analysis and evaluation of ASSL specifications, and thus, it will help to validate *liveness* and *safety* properties of autonomic systems specified with ASSL.

Considering *liveness properties*, model checking will help ASSL to verify whether a self-management policy will fix a specific problem, a message sent by an AE will be eventually received by another AE, etc. In this paper, we have presented a case study example of checking with ASSL a special liveness property of the ASSL specification model for the NASA Voyager mission [15].

Considering *safety properties*, model checking will help ASSL to verify cases such as: an action will be performed when a policy is triggered; an action can be never started due to unreachable preconditions; an event will be prompted to stop a fluent (so the fluent is not endless); deadlocks among

the AEs (waiting for messages from each other) and among the policies; contradictions among policies, SLO, and among policies and SLO.

Our plans for future work are mainly concerned with further development of the model checking mechanism for ASSL. 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 a model checking mechanism for ASSL will enable broad-scale formal verification of autonomic systems. Therefore, it will make ASSL a better and more powerful framework for autonomic system specification, validation and code generation.

References

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

[2] C. Baier, J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] M. Ben-Ari. *Principles of the Spin Model Checker (Paperback)*. Springer, 2008.

[4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[5] K. G. Larsen. Efficient Local Correctness Checking. *LNCS*, 663:30–43, 1992.

[6] A. Arnold, D. Begay, and P. Crubille. Construction and Analysis of Transition Systems with MEC. *Amast Series in Computing*, 3:192, 1994.

[7] R. Mateescu and H. Garavel. XTL: A Meta-Language and Tool for Temporal Logic Model-Checking. In *Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98)*, 1998.

[8] P. Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. Technical report, IBM T. J. Watson Laboratory, October 2001.

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

[10] W. Truszkowski, M. Hinchey, J. Rash, and C. Rouff. NASA's Swarm Missions: The Challenge of Building Autonomous Software. *IT Professional*, 6(5):47–52, 2004.

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

[12] 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)*, pages 54–64. IEEE Computer Society, 2008.

[13] 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*, pages 1652–1657. ACM, 2008.

[14] E. Vassev and M. Hinchey. ASSL Specification and Code Generation of Self-Healing Behavior for NASA Swarm-Based Systems. In *Proceedings of the Sixth IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe'09)*. IEEE Computer Society, 2009.

[15] E. Vassev and M. Hinchey. ASSL specification model for the image-processing behavior in the nasa voyager mission. Technical Report Lero-2009-1, Lero—the Irish Software Engineering Research Center, 2009.

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

[17] M. Bakera, C. Wagner, T. Margaria, E. Vassev, M. Hinchey, and B. Steffen. Component-oriented behavior extraction for autonomic system design. In *The First NASA Formal Methods Symposium (NFM 2009)*. NASA, 2009.

[18] E. A. Emerson. The Beginning of Model Checking: A Personal Perspective. In *25 Years of Model Checking*, pages 27–45. Springer, 2008.