TOWARDS MODEL CHECKING WITH JAVA PATHFINDER FOR AUTONOMIC SYSTEMS SPECIFIED AND GENERATED WITH ASSL

Emil Vassev[1], Mike Hinchey[2], Aaron Quigley[1]
[1]*Lero – The Irish Software Engineering Research Centre, University College Dublin, Ireland*
[2]*Lero – The Irish Software Engineering Research Centre, University of Limerick, Ireland*
emil.vassev@lero.ie, mike.hinchey@lero.ie, aquigley@ucd.ie

Keywords: autonomic computing, model checking, ASSL, Java PathFinder

Abstract: Autonomic computing has been recognized as a valid approach to the development of large-scale self-managing complex systems. The Autonomic System Specification Language (ASSL) is an initiative for the development of autonomic systems where we approach the problem of formal specification, validation, and code generation of such systems within a framework. As part of our research on ASSL, we have developed and investigated different approaches to software verification. Currently, the latter is possible via built-in consistency checking and functional testing where handling logical errors is a daunting task. In this paper, we discuss our work on model checking with NASA's Java PathFinder tool, which is an explicit-state model checker that works directly on the generated Java code. We propose optional automatic generation of test drivers in the form of PathFinder API calls seeded in the ASSL-generated code.

1 INTRODUCTION

Nowadays software permeates everywhere where information technology can be useful to some extent. However, contemporary software faces the expanding burden of complexity in software development and of ensuring both its correctness and reliability. Hence, initiatives such as *autonomic computing* (Parashar and Hariri, 2006) have risen to introduce theories and techniques intended to reduce the complexity of managing systems through automation. Moreover 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* (Clarke, Grumberg and Peled, 2002; Baier and Katoen, 2008).

The vision and metaphor of autonomic computing (AC) (Murch, 2004) is to apply the principles of self-regulation and complexity hiding to the design of complex computer-based systems. However, the very complexity of many systems that lend themselves well to AC can often imply difficulty in designing the autonomic system itself.

The ASSL Approach to AC. The Autonomic System Specification Language (ASSL) (Vassev, 2008) is a framework dedicated to AC. By providing a powerful formal notation and computational tools, ASSL helps AC researchers with problem formation, system design, system analysis and evaluation, and system implementation. The framework's tools 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 (Vassev, 2008). The latter form a theory that aids in the construction of correct autonomic system (AS) specifications. In addition, from any valid specification, ASSL can generate an operational Java application skeleton.

As part of the framework validation and in the course of a new ongoing research project at Lero — the Irish Software Engineering Research Centre, ASSL has been successfully used to specify autonomic properties and generate prototype models of the NASA ANTS concept mission (Vassev, Hinchey and Paquet, 2008) and the NASA Voyager mission (Vassev and Hinchey, 2009).

1.1 Research Problem and Proposed Solution

ASSL provides automatic code generation, which ensures consistency between a specification and its implementation. However, our experience with ASSL has demonstrated that errors can be easily introduced while specifying large systems. Although the ASSL framework takes care of syntax and consistency errors, it still cannot handle logical errors.

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

- We are working on improving the current ASSL consistency checker with assertion and debugging techniques, which should allow for a good deal of static analysis of an ASSL specification. This approach will improve the verification process, but will not be sufficient to assert *safety* (e.g., freedom from deadlock) or *liveness* properties.
- We are investigating model checking as the most effective approach to software verification for our purposes.

Model checking is a *formal methods* technique that allows formal specifications to be tested exhaustively (Clarke, Grumberg and Peled, 2002; Baier and Katoen, 2008). The approach advocates formal verification tools whereby the formal specifications are automatically checked for specific flaws by considering special correctness properties expressed in *temporal logic* (Baier and Katoen, 2008).

In the course of this long-term project, we consider three different possible approaches to model checking.

- A part of this research is on a model-checking mechanism that takes an ASSL specification as input and produces as output a finite state-transition system (also called a state graph) such that a specific property in question is satisfied if and only if the original ASSL specification satisfies that property (Vassev, Hinchey and Quigley, 2009).
- Another research direction is towards mapping ASSL specifications to special *service logic graphs*, which support the so-called r*everse model checking* (Bakera et al., 2009).
- In this paper we present our third approach to model checking with ASSL. The latter generates operational Java code, which we use to perform a sort of *post-implementation* model checking with the Java PathFinder tool developed at NASA Ames (Visser et al., 2000). In this approach, we use Java PathFinder to verify the generated Java code. We are at the beginning of our research and the results presented here are preliminary.

1.2 Why Post-Implementation Model Checking?

Although it is widely accepted that model checking should be applied to the design phase rather than to the implementation phase of the software lifecycle, we believe that post-implementation model checking is worthy of investigation and probably well integrated with ASSL.

In (Vassev, Hinchey, Quigley, 2009) we reveal the so-called *state-explosion problem* we are currently facing with *specification-phase model checking*. Due to the highly concurrent nature of the ASSL-specified ASs, the size of an ASSL state graph is at least exponential in the number of running ASs internal concurrent processes. This is because the state space of the entire AS is built as the Cartesian product of the local state of these concurrent processes. Here, our possible solutions to the state-explosion problem are *abstraction techniques* that reduce the number of states to be tested; i.e., the model checking mechanism does not explore all the possible paths of execution, but only those considered important. However, this approach makes it possible to generate ASs with ASSL that contain fatal errors (e.g., deadlocks), which cannot be detected, despite careful specification and the existence of model checking.

Another good reason for having post-implementation model checking is the possibility to verify not only the newly-generated code but also all consecutively updated versions of the same. Thus, we can check the code even if it has evolved following its automatic generation with ASSL.

Paper Organization. The rest of this paper is organized as follows. In Section 2, we review related work on post-implementation model checking. As a background to the remaining sections, Section 3 provides a brief description of the ASSL framework and the built-in consistency checking mechanism. Section 4 introduces Java PathFinder and the proposed post-implementation approach to model checking by integrating that tool in ASSL. Finally, Section 5 presents concluding remarks and future work.

2 RELATED WORK

In the course of this research, we found two interesting projects targeting post-implementation model checking, where the implementation language is C.

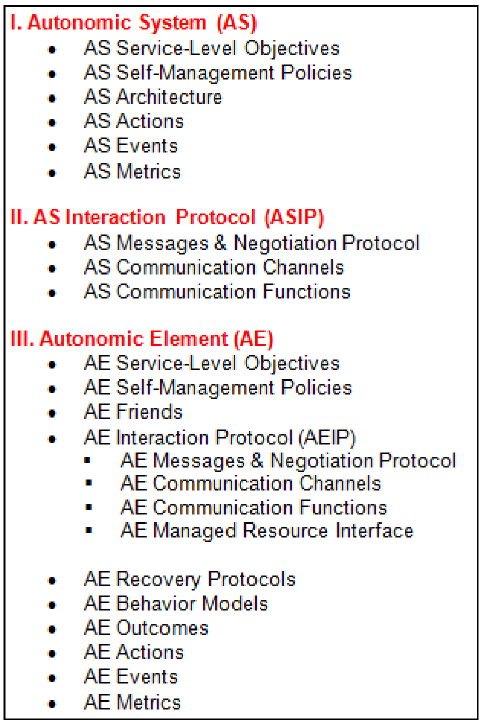
In (Ball, T., Podelski. A., Rajamani, S., 2001) the SLAM project at Microsoft is described. This project is similar to Java PathFinder in the sense that it also relies on different techniques to accomplish model checking. SLAM uses techniques such as *static analysis*, *abstraction*, and *symbolic model checking* (Clarke, Grumberg and Peled, 2002; Baier and Katoen, 2008). A special model checker for Boolean programs (Ball and Rajamani, 2000) is used where all the program variables are of the Boolean type. The idea is to apply abstraction on the original C program by extracting a state graph and then to check whether program statements are reachable. For any reachable statement, the path of instructions to that statement is symbolically executed on the original program. If the executed path does not match the expected path, a Boolean variable is created to catch the point where the difference begins. Further, the same process is repeated with a new Boolean condition (involving the newly created Boolean variable) that removes the path difference.

Another post-implementation model checker is the FeaVer tool (Holzmann and Smith, 2000). The latter is based on the SPIN (Ben-Ari, 2008) model checker mechanism. FeaVer extracts an abstract verification model in PROMELA (Iosif, 1998) (a special verification modeling language) from the C program, and verifies it against special logic properties. The process of abstraction is semi-automated because a special lookup table is used to translate C code to PROMELA code. Once the verification model is created, in a series of steps a complete verification of the model is performed with the construction of increasingly detailed PROMELA models. The SPIN model checker is used to perform model checking on the PROMELA models.

3 ASSL

In general, ASSL considers ASs as composed of autonomic elements (AEs) interacting over interaction protocols. To specify ASs, ASSL uses a *multi-tier* specification model (Vassev, 2008). By their nature, the *ASSL tiers* are abstractions of different aspects of the AS under consideration, such as self-management policies, communication interfaces, execution semantics, actions, etc. There are three major tiers (abstraction perspectives), each composed of sub-tiers (cf. Figure 1):

- 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. Note that ASSL express policies with special states called *fluents* (Vassev, 2008).
- AS Interaction Protocol (ASIP) tier — forms a perspective that defines the means of communication between AEs.
- AE tier — forms a unit-level perspective, where we define interacting sets of individual AEs with their own behavior.


I. Autonomic System (AS)
- AS Service-Level Objectives
- AS Self-Management Policies
- AS Architecture
- AS Actions
- AS Events
- AS Metrics

II. AS Interaction Protocol (ASIP)
- AS Messages & Negotiation Protocol
- AS Communication Channels
- AS Communication Functions

III. Autonomic Element (AE)
- AE Service-Level Objectives
- AE Self-Management Policies
- AE Friends
- AE Interaction Protocol (AEIP)
  - AE Messages & Negotiation Protocol
  - AE Communication Channels
  - AE Communication Functions
  - AE Managed Resource Interface
- AE Recovery Protocols
- AE Behavior Models
- AE Outcomes
- AE Actions
- AE Events
- AE Metrics

Figure 1: ASSL multi-tier specification model.

AEs are intelligent agents and the ASSL-developed ASs are considered multi-agent systems. Similar to any multi-agent system (Sycara, 1998), AEs are autonomous entities that interact either cooperatively or non-cooperatively (on a selfish base). The interaction though in the ASSL approach is going over predefined messages. In addition, the ASSL-developed AEs provide the needed decision-making capability that underlies autonomy and self-management. Moreover, ASSL allows for defining the architecture topology of an AS where AEs can be grouped into groups forming bigger intelligent entities (mini ASs). Here, the group formation can be centralized or GRID-alike (Vassev, 2008).

3.1 Consistency Checking in ASSL

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 2) is a framework mechanism for verifying specifications by performing exhaustive traversal of the declarative specification tree. In general, the framework performs two kinds of consistency-checking operations: 1) *light* – checks for type consistency, ambiguous definitions, etc.; and 2) *heavy* – checks whether the specification model conforms to special *correctness properties*.

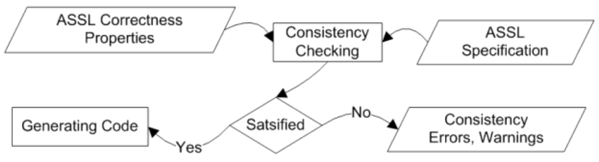


Figure 2: Consistency checking with ASSL.

The correctness properties are ASSL semantic definitions defined per tier (Vassev, 2008). Although, these are expressed in First-Order Linear Temporal Logic (FOLTL) (Baier and Katoen, 2008), 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. 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.

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 *always*, *next*, *eventually*, *until*, *waiting-for*, etc., are omitted. Therefore, 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.

4 POST-IMPLEMENTATION MODEL CHECKING

In this section, we present our preliminary work on model checking with ASSL and Java PathFinder. As we have already stated, we are at the beginning of our research on this model checking approach. Thus, the results presented here are rather theoretical.

4.1 Java PathFinder

Java PathFinder is a post-implementation model checker tool written in Java and targeting at Java programs (Visser et al., 2000; Java PathFinder, 2008). It can check Java programs for deadlocks, invariants and user-defined assertions in the code. Moreover, properties expressed in Linear Temporal Logic (Baier and Katoen, 2008) 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 (Java PathFinder, 2008).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.

Figure 3 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.

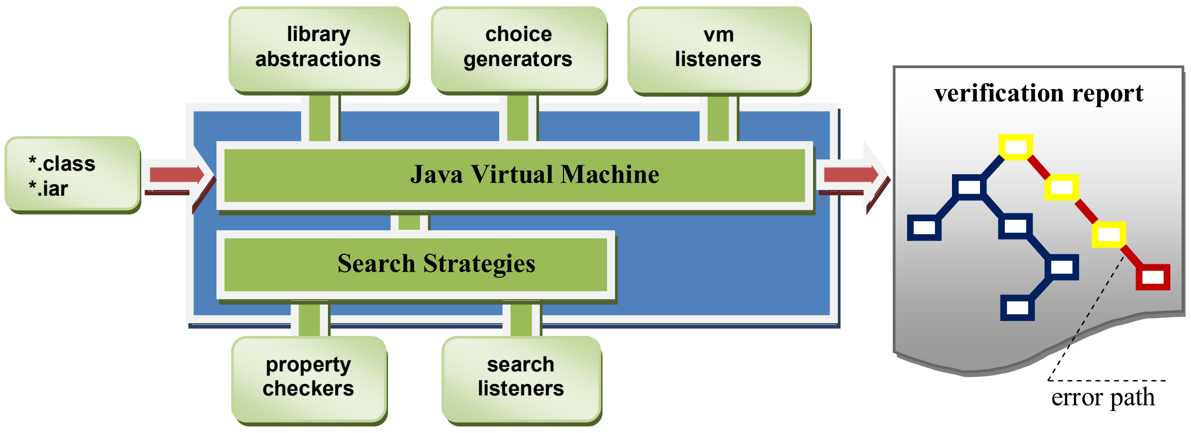


Figure 3: Java PathFinder operational model (elaborated from Java PathFinder, 2008)

As shown by Figure 3, 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.

4.2 Embedding Java PathFinder in ASSL

In general, Java PathFinder provides capabilities for non-deterministic testing via random input data generators (Java PathFinder, 2008) 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 (Vassev, 2008). Here, a non-deterministic PathFinder choice point will be generated (cf. `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`.

5 CONCLUSION

A model checking mechanism will complete the ASSL framework by allowing for automated system analysis and evaluation of any ASSL-generated autonomic system, and thus, it will help to validate liveness and safety properties of the same.

As a part of our long-term research on model checking with ASSL, we are currently investigating post-implementation model checking with NASA's Java PathFinder tool. In this paper, we have justified and presented our approach to applying Java PathFinder on ASSL-generated autonomic systems. We propose automatic generation of special PathFinder choice points in the generated Java applications. These choice points, together with the provided backtracking and state matching PathFinder mechanisms, will allow for possibly efficient post-implementation model checking.

Future research is concerned with further development of this approach and experimental results. 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 ASs. Therefore, it will make ASSL a better and more powerful framework for AS specification, validation and code generation.

REFERENCES

Clarke, E., Grumberg, O., and Peled, D., 2002. *Model Checking*. MIT Press.

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

Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., Steffen, B., 2009. Component-Oriented Behavior Extraction for Autonomic System Design. In *Proceedings of the First NASA Formal Methods Symposium (NFM 2009)*. NASA.

Ball, T., Podelski. A., Rajamani, S., 2001. Boolean and Cartesian Abstractions for Model Checking C Programs. In *Proceedings of TACAS01: Tools and Algorithms for the Construction and Analysis of Systems*. Genova, Italy.

Ball, T., Rajamani, S., 2000. Bebop: A symbolic Model Checker for Boolean Programs. In *Proceedings of the 7th International SPIN Workshop*. Vol. 1885 of LNCS, Springer-Verlag.

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

Holzmann, G., Smith, M. H., 2000. Automating Software Feature Verification. *Bell Labs Technical Journal*, Vol. 5(2), Issue on Software Complexity.

Iosif, R., 1998. *The PROMELA Language*, http://www.dai-arc.polito.it/dai-arc/manual/tools/jcat/main/node168.html, last visited on April 25, 2009.

Java PathFinder, 2008. http://javapathfinder.sourceforge.net/, last visited on April 25, 2009.

Parashar, M. and Hariri, S. (editors), 2006. *Autonomic Computing: Concepts, Infrastructure and Applications*. CRC Press.

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

Sycara, K. P., 1998. Multiagent Systems. In *AI Magazine*, vol. 19(2). Association for the Advancement of Artificial Intelligence.

Vassev, E., 2008. *Towards a Framework for Specification and Code Generation of Autonomic Systems.* A PhD Thesis in the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

Vassev, E., Hinchey, M., Paquet, J., 2008. Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions. In *Proceedings of 23rd Annual ACM Symposium on Applied Computing (SAC2008) - AC Track*. ACM.

Vassev, E., Hinchey, M., 2009. 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'09)*. IEEE Computer Society.

Vassev, E., Hinchey, M., Quigley, A., 2009. Model Checking for Autonomic Systems Specified with ASSL. In *Proceedings of the First NASA Formal Methods Symposium (NFM 2009),* NASA.

Visser, W., Havelund, K., Brat, G., Park, S.-J., 2000. Model Checking Programs, In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE'00)*. IEEE Computer Society.