Nowadays, the biggest threat to continued success in information and communication technology is complexity. Contemporary software systems are becoming far too complicated, as are the tasks of building and maintaining such systems.

To remain competitive, many major software vendors, such as IBM, HP, Sun, and Microsoft, have initiated research programs to create computer systems that reduce the maintenance burden by exhibiting self-management. Autonomic computing (AC), recognized as a potential long-term solution to the problems of increasing system complexity and maintenance costs, draws inspiration from human biology. The idea is that software systems must manage themselves, as the human body does, automatically, controlling complexity through self-management based on high-level objectives.

Since its introduction in 2001 by IBM, AC has inspired many initiatives for self-management of complex systems. However, despite these efforts, it still is not pervasive across the IT industry. The only significant visible progress of AC has been the integration of self-managing autonomic features into individual products such as chips, databases, and networking components (M. Parashar and S. Hariri, eds., *Autonomic Computing: Concepts, Infrastructure and Applications*, CRC Press, 2006).

Developers cannot use traditional software approaches to create autonomic systems (ASs) because these approaches pay scant attention to many of an AS’s features. Therefore, transitioning to an autonomic culture requires new development techniques and tools that intrinsically support AC principles and provide programming concepts for implementing autonomic systems.

**ASSL Approach to AC**


**Separation of concerns**

One of the noteworthy means of complexity reduction in ASSL is the separation of the AC features from the system-service features. ASSL helps to model and generate AC wrappers in the form of ASs that embed the components of non-AC systems. These managed elements, controlled by the AS, are separate software systems performing services.

ASSL emphasizes the AC functionality and architecture, but not a managed element’s functionality and architecture. Instead, the emphasis is on the interface needed to control a managed element. As Figure 1 shows, ASSL provides an abstraction of the managed elements through this interface.

**ASSL multitier model**

ASSL exposes a hierarchical specification model defined through
formalization tiers. These tiers provide a judicious selection and configuration of AS infrastructure elements and mechanisms.

The AS comprises special autonomic elements (AEs) interacting over interaction protocols, whose specification is distributed among the ASSL tiers. Each tier describes different aspects of the AS, such as service-level objectives, policies, interaction protocols, events, and actions, which helps to specify an AS at different levels of abstraction. Table 1 presents the multitype specification model of ASSL, which decomposes an AS into levels of functional abstraction and functionally related tiers (subtiers).

The first decomposition (left column in Table 1) presents the AS from three different perspectives:

- The AS tier is a general and global AS perspective exposing the architecture topology, general system behavior rules, and global actions, events, and metrics applied to these rules.
- The AS interaction protocol (ASIP) tier exposes a means of communication for the AS.
- The AE tier is a unit-level perspective, where ASSL specifies interacting sets of the AS’s individual components as AEs with their own behavior. This behavior must synchronize with the behavior rules from the global AS perspective.

The second decomposition (right column in Table 1) divides the three major tiers into functionally related subtiers, where new AS properties emerge. This allows a flexible approach to specification:

- The top-down specification approach starts with the service-level objectives and works toward detailed actions, events, and metrics.

The bottom-up specification approach starts at the detailed levels of specification (metrics, events, actions, channels, messages, and so forth) and builds up the system.

- The merging specification approach works on both abstract and detailed levels by constantly synchronizing their specification.

Specifying with ASSL

The ASSL tiers specify different aspects of the AS in question, but developers do not need to employ all of them to model an AS. Usually, developers build an ASSL specification around self-management policies, which makes that specification AC-driven. This method aligns with AC’s main goal—self-management based on four main principles: self-configuring, self-healing, self-optimizing, and self-protecting (self-CHOP). The ASSL model addresses these self-CHOP principles as policies specified at both AS and AE tiers with special constructs called fluents and mappings:

- A fluent activates or deactivates a policy when the system fulfills a specified condition.
- Mappings connect particular fluents to ASSL-specified actions.

ASSL expresses fluents with fluent-activating and fluent-terminating events, which drive the self-management policies. To express mappings,
SOFTWARE TECHNOLOGIES

mechanism, an automated verification approach for finite state systems creating a traversal to check for both syntax and consistency errors (type consistency, ambiguous definitions, and so forth). This mechanism determines whether a specification conforms to ASSL semantic definitions.

Although efficient, the ASSL consistency-checking mechanism cannot handle logical errors including specification flaws, and thus it cannot assert safety (for example, freedom from deadlock) or liveness properties. To handle such errors, developers are creating a model-checking validation mechanism, an automated verification approach of finite state systems using efficient graph-search algorithms and correctness properties. The following approaches are currently under consideration:


- ASSL specifications are translated into special state-transition systems where a built-in model-checking mechanism determines whether a specific property is satisfied if and only if the original ASSL specification satisfies that property (E. Vassev, M. Hinchey, and A. Quigley, “Model Checking for Autonomic Systems Specified with ASSL,” Proc. First NASA Formal Methods Symp. [NFM 09], NASA, 2009, pp. 16-25).

ASSL toolset

The ASSL framework provides a toolset that developers can use to edit and validate ASSL specifications and generate Java code.

Validation. The framework toolset provides verification mechanisms for automatic reasoning of a specified AS, which helps to create reliable software that maximizes the probability of satisfying user expectations.

The base validation approach in ASSL is consistency checking, a mechanism for verifying ASSL specifications by performing exhaustive traversal to check for both syntax and consistency errors (type consistency, ambiguous definitions, and so forth). This mechanism determines whether a specification conforms to ASSL semantic definitions.

Although efficient, the ASSL consistency-checking mechanism cannot handle logical errors including specification flaws, and thus it cannot assert safety (for example, freedom from deadlock) or liveness properties. To handle such errors, developers are creating a model-checking validation mechanism, an automated verification approach of finite state systems using efficient graph-search algorithms and correctness properties. The following approaches are currently under consideration:


ASSL is currently the only complete solution for AS specification and implementation.

Code generation. An ASSL specification describes an AS solving a particular problem; it is not an implementation. However, ASSL can generate an operational Java application skeleton from any valid specification. Code generation is the most complex activity in the ASSL framework. In general, it maps validated ASSL specifications to Java classes. ASSL generates fully operational multithreaded event-driven applications with embedded messaging.

Because of automatic code generation’s synthesis approach, ASSL guarantees consistency between a specification and the corresponding implementation. Moreover, it helps software engineers transition smoothly from an AS specification to a particular implementation and saves time when changes in the specification require reimplementation.

APPLICABILITY

To validate the framework, developers have used ASSL to make existing complex systems autonomic. The results show that multiagent systems lend themselves well to ASSL’s multiter specification model.


AC does not currently provide researchers with a clear idea of what is required to develop an AS. Hence, the transition to an AC culture will be much faster in the presence of solid AC-dedicated programming techniques and technologies. To the best of our knowledge, ASSL is currently the only complete solution for AS specification and implementation. Although other solutions do exist, they emphasize policy modeling rather than multiter specification modeling and validation and code-generation tools.

The advantage of the ASSL approach is that it provides both a formal notation and tools to develop ASs. This helps to

- unambiguously identify the requirements of an AS using formal specification.
Practice has shown that the development of complex systems often requires multiple specification languages to describe different system aspects at various levels of abstraction. ASSL aims to overcome this problem by providing a specification style to address the high level of complexity in AC. This style goes beyond the initial specifications pertaining to functional and interfacing issues. Here, the tiers in ASSL are specification structures, each necessitating its own syntactical and semantic rules and providing abstractions of different aspects of the AS under consideration. They aid not only in specifying the AS at different levels of abstraction but also in reducing complexity and thus improving the overall perception of the AS features.

Emil Vassev is a senior postdoctoral researcher at Lero—the Irish Software Engineering Research Centre at University College Dublin. Contact him at emil.vassev@lero.ie.

Mike Hinchey is codirector of Lero—the Irish Software Engineering Research Centre and professor of software engineering at University of Limerick, Ireland. Contact him at mike.hinchey@lero.ie.