The ASSL approach to specifying self-managing embedded systems

Emil Vassev* † and Mike Hinchey

Lero—Irish Software Engineering Research Centre, University of Limerick, Ireland

SUMMARY

The increasing complexity of contemporary embedded computing systems requires the use of self-management in order to handle unforeseen changes in both hardware and control software. The idea behind computer systems capable of self-management is a complex concept compound by many aspects related to both artificial intelligence and awareness. Here, the biggest challenge is still the question how to properly develop and verify such systems. In this paper, we present a formal approach to specifying embedded systems capable of self-management. In our approach, we use the ASSL (autonomic system specification language) framework as a development environment, where self-management features of embedded systems are specified and an implementation is automatically generated. ASSL exposes a rich set of specification constructs that help developers specify event-driven embedded systems. Hardware is sensed via special metrics intended to drive events and self-management policies that help the system handle critical situations in an autonomous reactive manner. We present this approach along with a simulation case study where ASSL is used to develop control software for the wide-angle camera carried on board NASA’s Voyager II Spacecraft. Copyright © 2011 John Wiley & Sons, Ltd.

Received 10 September 2010; Revised 20 December 2010; Accepted 21 March 2011

KEY WORDS: embedded systems; reactive systems; ASSL; self-management

1. INTRODUCTION

Traditionally, embedded systems combine application-specific integrated hardware with embedded control software. Embedded system engineers need to have a good understanding of both desirable and non-desirable properties of the control software, and their effect onto the controlled hardware and the quality of the system as a whole. Usually, embedded systems are designed to perform one or more dedicated functions, often with real-time constraints. In such systems, both hardware and software are embedded as parts of a complete device or system. In addition, they often have long life and 24 × 7 operational requirements. Being closely related to revolutionary innovations in computer hardware, embedded systems have become more and more powerful. As a result, today, the computational tasks we can accomplish in an embedded environment are much more complex than those of just a decade ago. As more and more software is developed for embedded systems, we must ensure that such software copes well with the high level of reliability and quality of service (QoS) requirements that we have come to expect. However, in order to build reliable embedded systems that cope well with the increased complexity, we need new, modern, development approaches. The latter must not only overcome the complexity problem, but must also address QoS in embedded critical systems, where it is often the main concern.

*Correspondence to: Emil Vassev, Lero—Irish Software Engineering Research Centre, University of Limerick, Ireland.
†E-mail: emil@vassev.com, emil.vassev@lero.ie

Copyright © 2011 John Wiley & Sons, Ltd.
We present our approach to this problem, whereby the ASSL (autonomic system specification language) \cite{1, 2} is used with its appropriate constructs to specify (or model) the event-driven behavior of an embedded system and subsequently to implement the latter via automatic code generation. ASSL is a formal method dedicated to autonomic computing (AC) \cite{3}. AC is recognized as a potential long-term solution to the problem of increasing system complexity and cost of maintenance. The idea is that software systems must manage themselves, controlling complexity through self-management based on high-level objectives. We demonstrate how ASSL can be successfully used as a formal approach to the development of embedded systems, where developers will be assisted with problem formation, system design, and system implementation.

The remainder of this paper is organized as follows. In Section 2, we review the related work, and in Section 3, we briefly present the ASSL framework, the ASSL constructs suitable for the specification of embedded systems, and the architecture of the ASSL-generated embedded systems. Section 4 presents a case study where ASSL is used to specify a simulation of the event-driven behavior of the wide-angle camera used in NASA's Voyager II mission. Finally, Section 5 provides brief concluding remarks and a summary of the future research and investigation trends.

2. PROGRAMMING EMBEDDED SYSTEMS: RELATED WORK

In general, embedded system programming is about writing software that drives hardware. In the past, embedded systems have had to run on platforms limited by memory and processor speeds, which in turn limited the programming tasks to writing simple software that drives controllers. However, for over 40 years, IT has obeyed Moore’s Law and, today, both the constant increase in processor speeds and decrease in memory costs allow for the development of new intelligent devices where real-time embedded systems become extremely complex in order to exploit the maximum advantage of the chosen platform. Nowadays, embedded system programming is targeting at applications for handheld devices, industrial control, set-top boxes, gaming devices, phones, A/V devices, and more. A subclass of embedded systems is the area of real-time systems, where timing constraints are introduced to ensure the ability to make certain calculations or decisions in a timely manner.

Often, embedded system programming is undertaken in the C/C++ programming language, combined with a variety of techniques developed to address particular problem domains. For example, many embedded systems use a real-time operating system (RTOS) to handle concurrent execution of multiple running processes, each written in a sequential language such as C \cite{4}.

Another example is the SystemC language, which is standardized by the IEEE (IEEE 1666-2005 Standard). This language originated from C++ as a language for system modeling intended to enable ‘system-to-silicon’ design flows \cite{5}. SystemC was developed to compensate for the shortcomings of the Plain old C language such as inability to represent a few key hardware-related features, notably, hierarchy, data-path widths, and concurrency. SystemC cannot be considered as a separate language, but rather as an extension (or a C++ class library) providing the missing features.

Although not considered a very efficient language due to its slow execution (sometimes less than 10% as fast as a similar program written in C), Java has also been embraced as a programming language for embedded systems. The key characteristics that helped in this regard are built-in multithreading and synchronization, automatic memory management, and lack of pointer arithmetic. Usually, in a Java-based embedded system, the software runs in the host Java VM, which executes on top of an RTOS. Java has become popular in the development of networked embedded systems \cite{6}. Note that ASSL generates executable Java code (see Section 3.5).

Formal methods have been both successful and extremely useful in the development of embedded safety-critical systems, such as modern avionics control software and control software for nuclear plants. Here, the advantages of using formal methods come from the rigorous mathematical semantics and the high levels of abstraction provided by the formal notation, and from the use of software constraints.
verification tools that help to uncover design and implementation flaws at early stages of the software life cycle. For example, to develop the control software for the C130J Hercules II, Lockheed Martin used a correctness-by-construct approach based on formal (SPARK) and semi-formal (consortium requirements engineering) methods [7].

Special formal languages called synchronous languages are dedicated to the programming of reactive systems [8]. An example of such a language is Lustre, which was successfully applied in the development of automatic control software for critical applications, e.g. the control software for nuclear plants and Airbus airplanes. Synchronous languages have also been used to develop DSP chips for mobile phones, to design and verify DVD chips, and to program the flight control software of Rafale fighters [9].

Esterel [10] is another synchronous language developed for specifying control-dominated reactive systems. This language combines the control constructs of an imperative software language with concurrency, pre-emption, and the asynchronous model of time like that used in synchronous digital circuits.

SDL [11] is a graphical specification language developed for modeling telecommunication protocols. SDL considers embedded systems consisting of concurrently running finite state machines (FSMs) connected via channels defining messages they carry. In such a system, repeatedly, each FSM receives messages and reacts to those by changing the internal state, or sends messages to other FSMs.

In our approach, we propose the use of ASSL as a development platform for embedded systems incorporating self-management features. Similar to the aforementioned approaches ranging from C, SystemC, Lustre, Esterel, and SDL, ASSL provides for a language that helps developers to develop embedded systems. Moreover, being a formal method dedicated to AC, ASSL provides for formalism that emphasizes self-management. Here, we are targeting embedded systems, whose Java implementation is automatically generated from their ASSL specifications. We believe that our approach will help in the realization of more reliable control software that maximizes the utilization of the hardware capacity through self-adaptation. The biggest shortcoming of our approach is the Java implementation of the ASSL-generated systems. There are multiple factors that make Java inherently not convenient for the development of embedded systems, e.g. performance issues, limited control over the program structures due to the integrated garbage collection, JVM dependency, etc. To overcome such issues, although Java is the primary target language of the ASSL framework, a long-term goal of our approach is the development of SystemC code generator for ASSL.

3. THE ASSL FRAMEWORK

Although intentionally dedicated to AC, ASSL can be used for the development of embedded systems with self-management capabilities. We term such systems embedded autonomic systems (EASs). In this section, we present the ASSL specification model and specific features that make the framework suitable for the development of EASs.

3.1. ASSL specification model

ASSL is based on a specification model exposed over hierarchically organized formalization tiers [1, 2]. The ASSL specification model is intended to provide both infrastructure elements and mechanisms needed by an autonomic system (AS), or in this case, by an EAS. Each tier of the ASSL specification model is intended to describe different aspects of the AS under consideration, such as: service-level objectives, policies, interaction protocols, events, actions, etc. This helps to specify an AS at different levels of abstraction imposed by the ASSL tiers (see Figure 1(a)). The ASSL specification model considers the ASs as being composed of special autonomic elements (AEs) interacting over special interaction protocols, whose specification is distributed among the ASSL tiers. However, a simple EAS can be specified with a single AE and no inter-AE interaction.
E. VASEV AND M. HINCHEY

Figure 1. ASSL specification model: (a) ASSL multi-tier specification model; (b) ASSL-specified AS; and (c) ASSL-specified AE.

protocols. Figure 1(a) presents the multi-tier specification model of ASSL. As shown, it decomposes an AS in two directions:

(i) into levels of functional abstraction,
(ii) into functionally related tiers (sub-tiers).

With the first decomposition, an AS is presented from three different perspectives, these depicted as three main tiers, each composed of sub-tiers (see Figure 1(a)):

(I) The AS Tier forms a general and global AS perspective exposing the architecture topology, general system behavior rules, and global actions, events, and metrics applied to these rules.

(II) The ASIP Tier (AS interaction protocol) forms a communication perspective exposing a means of communication for the AS under consideration. As shown, at this tier, one might specify public messages, channels, and communication functions used by the AEs to interact.

(III) The AE Tier forms a unit-level perspective, where an interacting set of the AS’s individual components is specified. These components are specified as AEs with their own behavior coping with the behavior rules from the global AS perspective. An AE specification is composed of AE rules (SLO and self-management policies), an AE interaction protocol (AEIP), special AE friends (a list of AEs forming a circle of trust), recovery protocols, special behavior models and outcomes, AE actions, AE events, and AE metrics [1, 2]. AEIP is identical to ASIP but emphasizes a private communication mechanism. Two AEs exchange messages over an AEIP only if they have an agreement on that, i.e. they are friends. In addition, AEIP provides for the specification of special interaction interfaces exposed as managed elements (MEs). Conceptually, an ME can be a software or hardware system (or sub-system) providing computational services.
THE ASSL APPROACH TO SELF-MANAGING EMBEDDED SYSTEMS

Figure 2. ASSL Self-management policy.

An ASSL-specified AS has a hierarchical composition where tier instances (specified tiers) are grouped around instances of their host tiers (nesting other sub-tiers). Figure 1(b) depicts the specification model of an AS specified with ASSL and Figure 1(c) shows the specification model for AEs composing that AS. Note that, both Figure 1(b) and (c) present generic specification models. Thus, concrete models have an arbitrary number and types of nodes derived from their corresponding ASSL specification. As shown, each node is a tier instance that possibly can be grouped around a host tier instance. For example, the AS node acts as a host tier instance for the nodes representing the AS-level sub-tiers such as SLO, policies, actions, events, and metrics. Note that the AS node organizes around itself other host tier instances, such as AE nodes and the ASIP node. Here, both the AE nodes and the ASIP node have their own surrounding nodes, these being instances of sub-tiers specified at the AE tier and at the ASIP tier, respectively.

Figure 1(c) presents the granular structure of the AE specification model. As shown, the AE node coordinates the tier instances of the sub-tiers specified for that AE; i.e. metrics (m nodes), events (e nodes), actions (a nodes), self-management policies (policy nodes), service-level objectives (slo nodes), behavior models (bm nodes), outcomes (o nodes), recovery protocols (rp nodes), and its private interaction protocol (aeip node). Here, both the policy nodes and the aeip node are host tier instances themselves.

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 model an EAS. Thus, to specify a simple EAS, we need to specify a single AE incorporating the embedded system software controlling the embedded system hardware. Moreover, self-management policies must be specified to provide self-management behavior at the level of AS (the AS tier) and at the level of AE (AE tier). Note that this rule is implied by the fact that all the ASSL specification must be AC-driven, i.e. based on self-management [3].

3.1.1. Self-management policies. The self-management behavior of an ASSL-developed EAS is specified with the self-management policies. These policies are specified with special ASSL constructs termed fluents and mappings [1, 2] (see Figure 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 self-management policies, thus making such a specification AC-driven.

Self-management policies are driven by events and actions determined deterministically. Thus, the operational evaluation of an ASSL policy is based on the evaluation of its fluents. The operational evaluation of a fluent follows the following algorithm:

If an event has occurred in the system then (see Figure 2) [1]:

1. Process the INITIATED_BY {...} clause to check if that event can initiate the policy fluent f and if so, initiate that fluent:

   • If the policy fluent f has been initiated then process only the policy MAPPING {...} clauses comprising the fluent f in their CONDITIONS {...} clause.
• Evaluate the CONDITIONS \{\ldots\} clause and if the stated conditions are held then evaluate the DO.Actions \{\ldots\} clause to perform the actions listed there.

(2) Process the TERMINATED_By \{\ldots\} clause to check whether that event can terminate the previously initiated policy fluent f and if so, terminate it.

The semantic rules 1 and 2 present the operational semantics that cope with the algorithm stated above. In these rules, each premise is a special system transition operation such as Event(ev), FluentIn(\(f, ev\)), FluentOut(\(f, ev\)), and ActionMap(\(f, a\)) [1].

\[
\begin{align*}
(1) \quad & \quad \sigma \xrightarrow{\text{Event}(ev)} \sigma' \\
& f \xrightarrow{\sigma} \text{INITIATED_BY}\{ev_1, \ldots, ev_n\} \quad \xrightarrow{\text{FluentIn}(f,ev)} \quad e \in \{ev_1, \ldots, ev_n\} \\
& \sigma \xrightarrow{\text{FluentIn}(f,ev)} \sigma' \xrightarrow{\text{Event}(ev)} \sigma'' \\
& f \xrightarrow{\sigma} \text{TERMINATED_BY}\{ev_1, \ldots, ev_n\} \quad \xrightarrow{\text{FluentOut}(f,ev)} \quad e \in \{ev_1, \ldots, ev_n\} \\
& \sigma \xrightarrow{\text{FluentIn}(f,ev)} \sigma' \\
& \text{map}\xrightarrow{\sigma} \text{CONDITIONS}\{f_1, \ldots, f_n\} \quad \xrightarrow{\text{ActionMap}(f,a)} \quad f \in \{f_1, \ldots, f_n\} \\
& \sigma \xrightarrow{\text{ActionMap}(f,a)} \sigma' \\
& \text{map}\xrightarrow{\sigma} \text{DO_ACTIONS}\{a_1, \ldots, a_n\} \quad \forall a \in A^\sigma \quad \xrightarrow{\text{Action(a)}} \quad a \in A^\sigma
\end{align*}
\]

Here, \(A^\sigma\) is the finite set of actions in the context \(\sigma\) determined by the ASSL policy under evaluation. The first premise in rule 2 evaluates whether the fluent \(f\) is initiated, i.e. only initiated fluents can be terminated [1]. Figure 2 presents a sample self-healing policy specification illustrating the semantics described above. As specified, the policy has only one fluent—inLosingSpacecraft, which is initiated by a spacecraftLost event and terminated by a earthNotified event. Further, this fluent is mapped to the notifyEarth action, which is performed as a result of the fluent’s initiation.

For the purpose of EAS development, self-management policies can be specified to control the embedded system hardware. This control goes over the ASSL-specified managed elements (see Figure 1) which describe special interface functions abstracting the communication interface exposed by the hardware. Moreover, real-time systems are bounded with deadlines, where the deadline may be a particular time or time interval, or maybe the arrival of some event. Thus, we can use ASSL to specify real-time EASs where different events can be used to trigger different policies intended to solve problems when the deadline cannot be met.

A complete description of the ASSL specification model is beyond the scope of this paper. For more information, we refer the interested reader to [1, 2].

### 3.2. ASSL features for embedded systems

In addition to the self-management policies (see Section 3.1.1), ASSL implies a number of important specification constructs and techniques, which allow for a valuable formal approach to the development of embedded systems. In this section, we present some ASSL constructs suitable for the specification of EASs.

#### 3.2.1. Events

In general, embedded systems are considered event-driven. ASSL exposes a rich set of techniques and constructs for specifying events (see Figure 3), which makes the framework suitable for the specification and code generation of event-driven embedded systems. From the EAS development perspective, events are one of the most important constructs in ASSL. By its nature, an event is a means for high-priority system messaging. ASSL uses events to specify many of the ASSL tiers and sub-tiers, such as: fluents, self-management policies, actions, etc. To specify events, one may use logical expressions over service-level objectives (SLO), metrics, other events,
THE ASSL APPROACH TO SELF-MANAGING EMBEDDED SYSTEMS

EVENTS {

EVENT lunchTime {
    ACTIVATION { ACTIV_TIME { 12:00 AM } }
    DURATION { 1 hour }
}
EVENT haveLunch {
    GUARDS { METRICS.restaurantOpen.VALUE = true }
    ACTIVATION { OCCURRED { EVENTS.lunchTime } }
    DURATION { 1 hour }
}
} // EVENTS

Figure 3. ASSL events.

messages, etc. Here, in order to specify events, ASSL introduces the following clauses:

- **DEGRADED/NORMALIZED**—to prompt an event when specified SLOs transit from normal to degraded state and from degraded to normal state, respectively,
- **RECEIVED/SENT**—to prompt an event when an ASSL message has been received or sent, respectively,
- **CHANGED**—to prompt an event when the value of a specific ASSL metric has been changed,
- **OCCURRED**—to prompt an event when another ASSL event has occurred,
- **ACTIV_TIME**—to prompt an event when a specific time has occurred,
- **PERIOD**—to prompt an event regularly on period basis,
- **DURATION**—to specify the event duration once it has been prompted. Events specified with duration have a limited active time during which they may initiate a fluent.

In addition, ASSL introduces a **GUARDS** clause to event specification to define conditions that must be stated before an event can be prompted. Figure 3 shows a specification sample specifying two events. The first (named lunchTime) is a timed event that will be prompted at 12:00 AM to notify the system that it is lunchtime. The second (named haveLunch) will be prompted by the first event, but only if the restaurantOpen metric holds true (for more on metrics see Section 3.2.2).

3.2.2. Metrics. For an embedded system, the most important success factor is the ability to sense the hardware and to react to sensed events. Together with the rich set of events, ASSL imposes metrics to gather information about external and internal points of interest, e.g. hardware in the case of an EAS. In ASSL, metrics are control parameters and observables that an embedded AS can control and/or monitor [1, 2]. Four different types of metrics are allowed:

- **resource metrics**—measure quantities of a managed element,
- **quality metrics**—measure system qualities such as performance, response time, etc,
- **scalar metrics**—monitor predefined dynamic AS variables,
- **composite metrics**—a function of other metrics.

For embedded system development the most important of these are **resource metrics**. Note that the **managed element** (see Section 3.2.3) in this case is the hardware controlled by the embedded system. In such a case, metrics are specified with a metric source that links the embedded AS with a hardware parameter that the metric in question is going to measure. As shown in Figure 4, the metric numberOfFailedNodes is updated via a special interface function called countFailedNodes and embedded in the specification of a **STAGE_ME** managed element, which represents the controlled hardware. Moreover, metrics are specified with a special range of acceptable values expressed via a special ASSL construct called **threshold class**. Note that metrics are evaluated by ASSL as valid and invalid based on their metric value and can prompt events when a new value has been detected. Thus, if a measured value does not fit into the metric threshold class, it is counted as undesirable behavior that should be carried by the EAS in question. This mechanism helps us to specify metrics prompting events when a deadline cannot be met and the EAS in question must switch to an alternative execution path.

3.2.3. Managed elements. An AE typically controls a managed resource specified in ASSL in the form of managed elements [1, 2]. A managed element is generally a functional unit, a hardware or software system that provides certain services. In an EAS, a managed element represents the controlled piece of hardware. An AE monitors and interacts with its managed elements. In ASSL, a managed element is specified with a set of special interface functions intended to provide control functionality over the managed resource (see Figure 5). ASSL provides an abstraction of a managed element through those interface functions. Thus, ASSL can specify and generate the interface controlling a managed element, but not the implementation of this interface in the controlled managed resource. Here, when developing an EAS, the generated interface must be implemented by the controlled hardware. Interface functions help to form a simple communication model for interacting with the managed elements. This model forms an extra layer at the AEIP (AE interaction protocol) (see Figure 1). The AEIP tier is normally used to specify a private communication protocol used by an AE to communicate with:

1. trusted AEs,
2. controlled managed elements.

In the case of an EAS, at this tier we should emphasize the specification of the managed element representing the controlled hardware. As shown in Figure 5, with ASSL we specify a managed element as a Java-like interface, i.e. as a named collection of functions without implementation. The parameter types and the return type of those functions are ASSL-predefined or custom-defined types. The managed element interface functions can be called by the ASSL actions to control the managed elements. In addition, these can be associated with ASSL metrics (see Section 3.2.2) to retrieve information from the hardware.
ASSL specifies managed element interface functions with four non-mandatory clauses: PARAMETERS, RETURNS, TRIGGERS, and ONERR_TRIGGERS [1, 2]. Here, the TRIGGERS and ONERR_TRIGGERS clauses are used to specify events triggered by an interface function. For example, in the sample above the runNodeReplica interface function is specified to trigger a nodeReplicaFailed event in case of erroneous execution. Recall that events drive self-management policies, which allows for handling hardware-related events, and thus, incorporating an event-driven behavior into an EAS.

3.3. Formal verification with ASSL

The ASSL framework provides a few formal verification techniques (some of those are still under development) that help in the correctness proof of ASSL-developed EASs. This is of major importance, because in some EASs failure may have severe safety or security consequences. Also, hardware and firmware designs are impossible to ‘patch’ after production, so there is an emphasis on early discovery of design flaws.

ASSL automatically generates an executable multithreaded Java application from a valid ASSL specification. Note that ASSL performs formal verification of the ASSL-specified ASs [12, 13]. A valid specification is considered that has passed through the formal verification process. The basic ASSL verification mechanism performs exhaustive traversal to check for syntax and consistency errors such as type consistency, ambiguous definitions, etc. The same mechanism checks whether a specification conforms to special correctness properties, defined as ASSL semantic definitions. Basically, the ASSL correctness properties are proof rules that make it possible to reason about the properties of the specifications created with ASSL. The correctness properties are expressed in First-order Linear Temporal Logic, which is a Temporal Logic with predicates and quantifiers [14]. An example of ASSL correctness property is Policy Initiation, which states that ‘Every policy is triggered by a finite non-empty set of fluents, and performs actions associated with these fluents.’

In addition, logical errors, such as specification and design flaws, are a subject of special ASSL model checking mechanisms [12, 13]. Here, a valid ASSL specification is considered one that has passed through the formal verification process. Finally, to allow post-implementation software verification with the ASSL framework, we are currently developing a novel test-generator tool based on change-impact analysis that helps the ASSL framework automatically generate high-quality test suites for self-management policies [15].

3.4. Code generation with ASSL

The ASSL code generator is a framework’s tool that allows for automatic Java code generation. Hence, from a valid ASSL specification, the tool generates the skeleton of an operational Java application implementing the specified EAS. The generated code consists of units and structures that inherit names and features from the ASSL specification. The architecture of an ASSL-generated EAS conforms to the ASSL multi-tier specification model. Thus, every EAS is generated with:

• a global AS autonomic manager (implements the AS tier specification) that takes care of the AS-level self-management policies and SLO,
• a set of AEs (implement the AE tier’s specification) where every AE takes care of its own self-management policies and SLO,
• a set of managed elements that help AEs communicate with the integrated hardware.

Note that both the AS autonomic manager and the AEs incorporate a distinct control loop (see Section 3.5) and orchestrate the self-management policies of the entire system. However, the AS autonomic manager is a sort of coordinator for the AEs. This coordination goes over AS-level self-management policies, SLO, events, actions, and metrics (see Figure 6). If an EAS is specified with a single AE the latter may be specified in a way that takes control over the AS manager. Moreover, instead of building a monolithic application for each ASSL-specified EAS, the ASSL framework organizes the generated AEs in a granular fashion (see Figure 1(b) and (c)). Here, at runtime, an ASSL-generated AE has a multi-granular structure composed of loosely coupled tier instances. All the tier instances form together the runtime object model of an AE. Similar to
3.5. ASSL super loop architecture for embedded systems

In addition to the suitable constructs allowing for the specification of embedded systems, ASSL also provides a suitable architecture for the automatically generated EASs. ASSL does not provide any constructs (see Sections 3.1 and 3.2) that deal with runtime conflicts due to requests for concurrent access to the same resources. Instead, this is handled by the code generator, which generates self-management policies as synchronized Java threads. Thus, a policy under execution acquires the necessary access first and then proceeds. This happens automatically every time it enters a synchronized method. For example, to evaluate self-management policies at runtime (see Section 3.1), a special synchronized `isPolicySatisfied()` method (see Figure 7) is generated for every specified policy.

ASSL generates ASs with special control loops (one per generated AE and one global for the entire AS) intended to control the system’s behavior [16]. A control loop is generated to apply control rules specified and implemented as self-management policies, SLO (service-level

```java
synchronized boolean isPolicySatisfied () {
    boolean bPolicySatisfied = true;
    Enumeration<ASSLFLUENT> eFluents = vFluents.elements();
    ASSLFLUENT currFluent = null;

    // A policy is not satisfied if there
    // is at least one initiated fluent
    // for that policy.
    while ( eFluents.hasMoreElements() ) {
        currFluent = eFluents.nextElement();
        if ( currFluent.isFluentInitiated() ) {
            bPolicySatisfied = false;
            break;
        }
    }
    return bPolicySatisfied;
}
```

Figure 6. AS architecture for ASSL.

Figure 7. isPolicySatisfied().
Figure 8. controlLoop().

Objectives) and metrics. Figure 8 presents an ASSL-generated control loop. As shown, a special controlLoop() method is generated to handle special control loop calls, and the tDelay variable is used to control the time allocated per control loop execution. The delay is intended to reduce the computational overhead introduced to the system by its AC features. The control loop calls are as follows:

(1) a perform() method is called on four distinct components: oMonitor, oAnalyzer, oSimulator, and oExecutor, to handle invalid metrics and degraded SLO.

(2) an applyPolicies() method is called to apply the self-management policies of an AE in a deterministic manner in terms of execution order.

In the first part, the control loop uses the four components to discover problems with both SLO and metrics, and uses actions to fix such problems. If there is no action set to fix a discovered problem, the control loop executes a generic action that notifies us of the discovered problem. Thus, we have a finite number of states (monitoring, analyzing, simulating, and executing), transitions between those states, and actions. The following elements describe the steps of the control loop algorithm implemented as an FSM.

(1) The FSM starts with monitoring by checking whether all the SLO are satisfied and all the metrics are valid.

(2) In case there are problematic SLO and/or metrics, the machine transits to the analyzing state. In this state, the problems are analyzed and eventually mapped to actions that can fix them.

(3) Next, the machine transits to the simulating state. In this state, for all problems still not mapped to actions, the system simulates problem-solving actions in an attempt to find needed ones. These problem-solving actions are selected from the so-called behavior models eventually specified at the AE Tier (see Section 3.1). In case, the needed problem-solving actions cannot be discovered, the system generates messages notifying about the unhandled problems.

(4) Finally, the machine transits to the executing state, where all the actions determined in both analyzing and simulating states are executed.

As shown in the second part of the control loop, an applyPolicies() method is called. Figure 9 presents the ASSL-generated implementation of that method.

Here, for each policy a doAllMappings() method is called where actions are called if a policy is activated by one or more fluents (see Section 3.1.1). Based on the control loop technique described above, ASSL generates EASs with the so-called super loop architecture [17]. The latter is a design...
pattern usually implemented as a program structure (e.g. a function) comprising an infinite loop that performs all the tasks of the embedded system in question. Figure 10 presents in pseudocode the generic implementation of the super loop architecture for embedded systems. Note that this sample is applicable to many of the implementations of the super loop architecture for embedded systems. As shown, the tasks are performed in a deterministic order with some delays between them. These delays are optional and are intended to keep the execution of tasks within a time frame allocated for each task. Here, the delays should be computed dynamically at runtime by considering the last execution time of each task for each loop pass. The computed execution time plus the allocated delay form the Worst Case Execution Time annotation of the tasks. Note that ASSL does not generate implementation of any time analysis algorithm. Instead, all the control loops are granted with delays intending to minimize the computational overhead related to the AC behavior. However, developers may manually program algorithms where the execution time of tasks is relatively measured based on previous executions and determine worst-case and best-case execution times. Task timing is important to meet the time deadlines (if such exist) of the system. Thus, this architecture targets at performing all the tasks in a correct deterministic sequential order and possibly in a reasonable amount of time.

ASSL generates a control loop that indirectly executes all the tasks that must be performed by an ASSL-generated EAS. This control loop is called on a regular basis by the run() method of the AE specified to control the embedded system in question. Note that ASSL generates AEs as Java threads, and overrides the Java Thread class’s run() method. The latter is generated as shown in Figure 11. Thus, the controlLoop() method is called on a regular basis in an endless loop and the tControlLoopDelay variable is used to control the overall time allocated for the entire AE thread.

protected void applyPolicies() {
    Enumeration<ASSLPOLICY> ePolicies = vPolicies.elements();
    ASSLPOLICY currPolicy = null;
    while (ePolicies.hasMoreElements()) {
        currPolicy = ePolicies.nextElement();
        // applies only "switched-on" policies
        if (currPolicy.isSwitchedOn()) {
            currPolicy.doAllMappings();
        }
    }
}

Figure 9. applyPolicies().

while (true) {
    Task1();
    Delay_After_Task1();
    Task2();
    Delay_After_Task2();
    ....
    TaskN();
    Delay_After_TaskN();
}

Figure 10. Super loop.
4. CASE STUDY: VOYAGER’S CAMERAS

In this section, we demonstrate how the ASSL framework can be used to specify an EAS. Our example is an ASSL specification model for the NASA Voyager Mission [18]. The NASA Voyager Mission [19] was designed for exploration of the Solar System. The original mission objectives were to explore the outer planets of the Solar System and as Voyager I and Voyager II travelled across the Solar System, they took pictures of planets and their satellites. The pictures taken by the Voyagers were transmitted to Earth via radio signals carrying image pixels. To take pictures, Voyager II, in particular, carried two television cameras on board—one for wide-angle images and one for narrow-angle images.

In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs, which follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. We emphasize the specification of the Voyager’s wide-angle camera, which can be considered as an EAS. For more information about the ASSL specification model for the NASA Voyager Mission, we refer the interested reader to [18].

4.1. ASSL specification

We specified an AE for the Voyager II spacecraft with a self-management policy to handle the image-processing behavior of the on-board wide-angle camera. Figure 12 presents the ASSL specification of the IMAGE_PROCESSING policy. As shown, we specified two fluents: inTakingPicture and inProcessingPicturePixels. The inTakingPicture fluent is initiated by a timeToTakePicture event and terminated by a pictureTaken event. This event also initiates the inProcessingPicturePixels fluent, which is terminated by the pictureProcessed event. Both fluents are mapped to the actions doTakePicture and processPicture respectively. This part of the specification is typical for any AS specified with ASSL, i.e. an ASSL specification is built around one or more self-management policies [18]. In order to specify an EAS (embedded AS) though, as described in Section 3.2.3, we must specify one or more Managed Elements intended to provide the means of control over the hardware in that embedded system.

Figure 13 presents the ASSL specification of the wideAngleCamera managed element, which is specified to control the on-board wide-angle camera via a set of interface functions. Through these interface functions, the wideAngleCamera managed element is used by the actions mapped to the fluents inTakingPicture and inProcessingPicturePixels to take pictures, apply filters, and detect interesting space objects. Owing to space limitations the ASSL specification of the narrowAngleCamera managed element is not shown here. Figure 14 presents a partial ASSL specification of the doTakePicture action mapped to the inTakingPicture fluent (see Figure 12) and calling the takePicture interface functions of both wideAngleCamera and narrowAngleCamera managed elements. As specified, the action asks the hardware to take a picture. Note that according to the

4. CASE STUDY: VOYAGER’S CAMERAS

In this section, we demonstrate how the ASSL framework can be used to specify an EAS. Our example is an ASSL specification model for the NASA Voyager Mission [18]. The NASA Voyager Mission [19] was designed for exploration of the Solar System. The original mission objectives were to explore the outer planets of the Solar System and as Voyager I and Voyager II travelled across the Solar System, they took pictures of planets and their satellites. The pictures taken by the Voyagers were transmitted to Earth via radio signals carrying image pixels. To take pictures, Voyager II, in particular, carried two television cameras on board—one for wide-angle images and one for narrow-angle images.

In this case study, we specified the Voyager II spacecraft and the antennas on Earth as AEs, which follow their encoded autonomic behavior to process space pictures, and communicate those via predefined ASSL messages. We emphasize the specification of the Voyager’s wide-angle camera, which can be considered as an EAS. For more information about the ASSL specification model for the NASA Voyager Mission, we refer the interested reader to [18].

4.1. ASSL specification

We specified an AE for the Voyager II spacecraft with a self-management policy to handle the image-processing behavior of the on-board wide-angle camera. Figure 12 presents the ASSL specification of the IMAGE_PROCESSING policy. As shown, we specified two fluents: inTakingPicture and inProcessingPicturePixels. The inTakingPicture fluent is initiated by a timeToTakePicture event and terminated by a pictureTaken event. This event also initiates the inProcessingPicturePixels fluent, which is terminated by the pictureProcessed event. Both fluents are mapped to the actions doTakePicture and processPicture respectively. This part of the specification is typical for any AS specified with ASSL, i.e. an ASSL specification is built around one or more self-management policies [18]. In order to specify an EAS (embedded AS) though, as described in Section 3.2.3, we must specify one or more Managed Elements intended to provide the means of control over the hardware in that embedded system.

Figure 13 presents the ASSL specification of the wideAngleCamera managed element, which is specified to control the on-board wide-angle camera via a set of interface functions. Through these interface functions, the wideAngleCamera managed element is used by the actions mapped to the fluents inTakingPicture and inProcessingPicturePixels to take pictures, apply filters, and detect interesting space objects. Owing to space limitations the ASSL specification of the narrowAngleCamera managed element is not shown here. Figure 14 presents a partial ASSL specification of the doTakePicture action mapped to the inTakingPicture fluent (see Figure 12) and calling the takePicture interface functions of both wideAngleCamera and narrowAngleCamera managed elements. As specified, the action asks the hardware to take a picture. Note that according to the
AESEL_MANAGEMENT {
  OTHER_POLICIES {
    POLICY IMAGE_PROCESSING {
      FLUENT inTakingPicture {
        INITIATED_BY { EVENTS.timeToTakePicture }
        TERMINATED_BY { EVENTS.pictureTaken }
      }
      FLUENT inProcessingPicturePixels {
        INITIATED_BY { EVENTS.pictureTaken }
        TERMINATED_BY { EVENTS.pictureProcessed }
      }
      MAPPING {
        CONDITIONS { inTakingPicture }
        DO_ACTIONS { ACTIONS.doTakePicture }
      }
      MAPPING {
        CONDITIONS { inProcessingPicturePixels }
        DO_ACTIONS { ACTIONS.processPicture }
      }
    } // AESEL_MANAGEMENT
  }
}

Figure 12. IMAGE_PROCESSING policy.

AEIP {
  ....
}

MANAGED_ELEMENTS {
  MANAGED_ELEMENT wideAngleCamera {
    INTERFACE_FUNCTION takePicture { }
    INTERFACE_FUNCTION applyFilterBlue { }
    INTERFACE_FUNCTION applyFilterRed { }
    INTERFACE_FUNCTION applyFilterGreen { }
    INTERFACE_FUNCTION getPixel { }
    INTERFACE_FUNCTION countInterestingObjects {
      RETURNS { integer }
    }
  } // ME wideAngleCamera
  ....
}

Figure 13. wideAngleCamera-managed element.

ASSL operational semantics [1], the pictureTaken event is prompted if the doTakePicture action is performed with no errors.

Moreover, an interestingObjects metric is specified (see Figure 15) to count all detected objects of interest, of which the Voyager AE takes pictures. The source of this metric is specified as one of the managed element interface functions (see countInterestingObjects); i.e. the metric gets updated by that interface function. Further, following the event-driven behavior specified for the EAS, we see that the timeToTakePicture event (recall that it activates the inTakingPicture fluent—see Figure 12) is prompted by a change in this metric’s value. Here, in order to simulate this condition, we also activate this event with a period of 60 s. Figure 16 presents the timeToTakePicture event specification. Thus, the doTakePicture action is performed when the inTakingPicture fluent
ACTION doTakePicture {
  DOES {
  IF AES.Voyager.isWideAngleImage THEN
  call AEIP.MANAGED_ELEMENTS.
  wideAngleCamera.takePicture;

  END ELSE
  call AEIP.MANAGED_ELEMENTS.
  narrowAngleCamera.takePicture;

  END
}
TRIGGERS {
  EVENTS.pictureTaken
}

Figure 14. Action.

METRIC interestingObjects {
  METRIC_TYPE {
    RESOURCE
  },
  METRIC_SOURCE {
    AEIP.MANAGED_ELEMENTS.
    wideAngleCamera.
    countInterestingObjects
  },
  THRESHOLD_CLASS {
    integer [ 0
  }
}

Figure 15. Metric.

EVENT timeToTakePicture {
  ACTIVATION {
    CHANGED {
      METRICS.interestingObjects
    }
    OR
    PERIOD {
      60 SEC
    }
  }
}

Figure 16. Event.

is initiated by the timeToTakePicture event (see Figure 12). In case of successful picture taking, the doTakePicture action triggers the pictureTaken event (see Figure 14), which initiates the inProcessingPicturePixels fluent (see Figure 12). The latter is mapped to (or triggers the execution of) the processPicture action.

Figure 17 presents a partial specification of the processPicture action. As shown, this action ensures first that it runs only when the inProcessingPicturePixels fluent is initiated, which is specified in the special GUARDS clause. Next, the action calls another ASSL-specified action—processFilteredPicture, to consecutively apply the ‘blue’, ‘red’, and ‘green’ filters. The processFilteredPicture action (see Figure 18) specifies a parameter (see the PARAMETERS clause) passing the filter color. Further, this action applies the appropriate filter on both wide-angle and narrow-angle
Voyager’s cameras. This is done via function calls on both wideAngleCamera and narrowAngleCamera managed elements. In case all the filters have been successfully applied, the processPicture action triggers a pictureProcessed event that terminates the inProcessingPicturePixels fluent (see Figure 12).

4.2. Test results

In this case study, we did not generate a separate EAS to test the behavior of the IMAGE_PROCESSING policy. Instead, we experimented with the prototype generated from the full ASSL specification of the Voyager II Mission [18]. Our goal was to demonstrate that the image processing behavior of the generated Voyager AE is capable of self-managing in respect of the specified with ASSL IMAGE_PROCESSING policy. It is important to mention that the generated Voyager prototype was a pure software solution, and thus we could not perform real embedded-system tests, but rather simulated ones. We specified metric-related events also as timed events, just to simulate sensing reactions from the wide-angle camera. For example, although the timeToTakePicture event was originally specified as a metric-related one, we also specified time activation to simulate changes in the metric intended to receive signals from the camera.

The test results demonstrated that, under simulated conditions (the prototype is prompted to take pictures every 60 s), the runtime behavior of the controlled wide-angle camera strictly followed the ASSL-specified IMAGE_PROCESSING self-management policy. Thus, the Voyager prototype took virtual pictures and transmitted blended images to virtual antennas on Earth, where these images were redirected to the virtual mission base for further processing [18].

Owing to specific features, common to all the Java applications generated with ASSL, at runtime, the ASSL-developed Voyager prototype produces log records, which show important state-transition
operations ongoing in the system [1, 2]. Here, we used these records to trace and evaluate the behavior of the generated prototype model. In order to perform this exercise, we compiled the generated Java code with Java 1.6.0 first, and then we ran the compiled code. First, it started all system threads as it is partially shown in the following log records. Note that starting all system threads first is a standard running procedure applied to all prototype models generated with the ASSL framework.

Log Records "Starting System Threads"

1) METRIC 'generatedbyassl.as.aes.voyager.metrics.INTERESTINGOBJECTS': started
2) EVENT 'generatedbyassl.as.aes.voyager.events.PICTUREPROCESSED': started
3) EVENT 'generatedbyassl.as.aes.voyager.events.TIMETOTAKEPICTURE': started
4) EVENT 'generatedbyassl.as.aes.voyager.events.PICTURETAKEN': started
5) FLUENT 'generatedbyassl.as.aes.voyager.aeself_management.image_processing.INPROCESSINGPIXERVICES': started
6) FLUENT 'generatedbyassl.as.aes.voyager.aeself_management.image_processing.INSTARTINGREDIMAGESESSION': started
7) POLICY 'generatedbyassl.as.aes.voyager.aeself_management.IMAGE_PROCESSING': started
8) AE 'generatedbyassl.as.aes.VOYAGER': started

Here records 1–8 show the start-up process of the VOYAGER autonomic element. Similar log records notified us that all the threads in all generated AEs started successfully. After starting up all the threads, the system ran in idle mode for 60 s, when the TIMETOTAKEPICTURE timed event occurred (record 99). This event is specified in the Voyager AE to run on regular basis every 60 s (simulated condition) and it triggers a series of system transitions following the specified autonomic behavior. The following log records demonstrate that the runtime image-processing behavior followed correctly the ASSL specification of the IMAGE_PROCESSING policy.

Records 99–103 show the initiation and termination of the voyager’s INTAKINGPICTURE fluent. This resulted in the execution of the TAKEPICTURE action (see record 100), which triggered the PICTURETAKEN event (see record 102). The latter consecutively initiated the INPROCESSINGPIXERVICES fluent. Records 104–109 and record 115 show the initiation and termination of that fluent. The INPROCESSINGPIXERVICES fluent prompted the execution of the PROCESSIPICURE action (see record 108), which executed the PROCESSFILTEREDPICTURE action thrice (records 105–107). Each time, this action was called to apply a different filter color (blue, red, or green) and the filtered image was sent to the antennas on Earth. Note that this action also uses the Voyager AE’s AEIP-specified functions [18] sendBeginSessionMsgs and sendEndSessionMsgs to send begin-session and end-session messages for each applied filter to the antennas on Earth.

Log Records "Voyager Autonomic Behavior"

99) EVENT 'as.aes.voyager.events.TIMETOTAKEPICTURE': has occurred
100) FLUENT 'as.aes.voyager.aeself_management.image_processing.INTAKINGPICTURE': has been initiated
101) ACTION 'as.aes.voyager.actions.TAKEPICTURE': has been performed
102) EVENT 'as.aes.voyager.events.PICTUREPROCESSED': started
103) FLUENT 'as.aes.voyager.aeself_management.image_processing.INTAKINGPICTURE': has been terminated
104) FLUENT 'as.aes.voyager.aeself_management.image_processing.INSTARTINGREDIMAGESESSION': has been initiated
105) FLUENT 'as.aes.voyager.aeself_management.image_processing.INSTARTINGREDIMAGESESSION': has been terminated
106) ACTION 'as.aes.voyager.aeself_management.image_processing.INSTARTINGBLUEIMAGESESSION': has been performed
107) ACTION 'as.aes.voyager.aeself_management.image_processing.INSTARTINGGREENIMAGESESSION': has been performed
108) ACTION 'as.aes.voyager.aeself_management.image_processing.INSTARTINGBLUEIMAGESESSION': has been terminated
109) EVENT 'as.aes.voyager.aeself_management.image_processing.INSTARTINGBLUEIMAGESESSION': has occurred
110) EVENT 'as.aes.antenna_spain.events.BLUEIMAGESSESSIONISABOUTTOTSTART': has occurred
111) EVENT 'as.aes.antenna_spain.events.BLUEIMAGESSESSIONISABOUTTOTSTART': has occurred
112) EVENT 'as.aes.antenna_spain.events.BLUEIMAGESSESSIONISABOUTTOTSTART': has occurred
113) EVENT 'as.aes.antenna_japan.events.BLUEIMAGESSESSIONISABOUTTOTSTART': has occurred
114) EVENT 'as.aes.antenna_japan.events.BLUEIMAGESSESSIONISABOUTTOTSTART': has occurred
115) FLUENT 'as.aes.voyager.aeself_management.image_processing.INPROCESSINGPIXERVICES': has been terminated

Subsequently, these messages prompted three [color]ImageSessionIsAboutToStart events for each antenna, one per filter color (see record 110 for the BLUEIMAGESSESSIONISABOUTTOTSTART event). Next these events initiated in the antenna AEs three instarting[color]ImageSession fluents, one per filter color (see record 113 for the INSTANTIATINGBLUEIMAGESSESSION fluent). Each of these fluents prompted the execution of the STARTIMAGECOLLECTSESSION action (see record 116). Note that this action was executed 12 times (one time for each applied filter per antenna) and it prompted the operation of receiving the begin-session messages. Subsequently, the antennas received these messages and the corresponding events were prompted to terminate instarting[color]ImageSession fluents and initiate fluents to collect the image pixels.
E. VASSEV AND M. HINCHEY

For each antenna AE, the pixel-collection fluent prompted the execution of a special pixel-collection action [18]. Thus, that action was executed for each antenna three times, one per filter color. Next, every received end-session message terminated the current active fluent for the current antenna AE. In addition, the last end-session message, for every antenna, initiated another fluent (termed inSendingImage; see [18]) that prompted the execution of a special action (termed sendImage; see [18]). The latter prepared the collected image and sent it to the Voyager Mission base on Earth. Further, this operation prompted a particular event at each antenna that terminated the inSendingImage fluent.

Further, the system continued repeating the same steps on a regular basis due to the TIMETO-TAKEPICTURE timed event (see record 99), which occurs every 60 s (simulated condition).

In the most basic of terms, experiments are said to be valid if they do what they are supposed to do. In that context, the experiments and evaluation results described here are valid and they demonstrate that the Voyager’s prototype developed with ASSL is able to perform image processing as the original mission did. Although programmed as an autonomic policy, the image-processing behavior implanted in our prototype does not extend the original event-driven behavior observed in the Voyager Mission, but rather copies the same. Here, under simulated conditions (the prototype is triggered to take pictures every 60 s), the prototype successfully transmitted blended images to virtual antennas on Earth.

5. CONCLUSIONS

We have demonstrated how ASSL—a formal tool dedicated to Autonomic Computing (AC), can be used to develop embedded systems with self-management capabilities. ASSL emphasizes self-management policies provided by special AEs intended to control special managed elements. In our approach, to develop embedded systems termed EAS (embedded autonomic systems), we use suitable ASSL specification constructs to specify AE-level self-management policies that control a piece of hardware. This control is provided via:

- ASSL events related to ASSL metrics, specified to react to changes in that hardware,
- special managed element interface functions intended to get these metrics fed with data from the controlled hardware or to trigger events related to the same.

Real-time tasks can be specified as self-management policies, where we can use timed ASSL events to bound tasks with time. Alternatively, ASSL metrics related to hardware activity can raise events to notify for the accomplishment of a particular task. Owing to the fact that ASSL provides automatic code generation, we can generate the Java implementation of successfully specified EASs. An ASSL-developed EAS is generated as a multithreaded Java application implementing a special design pattern for embedded systems termed as super loop architecture. As a proof of concept, we have successfully used ASSL to specify and generate an EAS that controls the wide-angle camera carried on board by NASA’s Voyager II spacecraft. It is important to mention though, that in its initial version, the Voyager prototype abstracts many of the spacecraft’s components without evaluating their behavior. Hence, the next prototype model will also specify the Voyager’s radio and antenna as distinct managed elements. This will allow the evaluation of their behavior (via metrics and events) and extending the IMAGE_PROCESSING policy with other self-management features. For example, fluents that react on malfunction in some of these components can trigger self-healing policies. In such a case, we are planning to implement two scenarios: remote-assistance self-healing and on-board self-healing. The former will copy the behavior of the original spacecraft, where remote assistance is provided in the form of radio contact and remote control programming. However, the on-board self-healing will add new autonomic features, which do not exist in the original spacecraft. Having the self-healing operations automated will allow us to evaluate to some extent the potential impact of autonomic computing on the maintenance required by the Voyager Mission.

The biggest flaw of our approach is the Java-based implementation of the EASs. Java does not provide direct access to hardware registers, physical memory, and handling of hardware interrupts,
which makes the Java applications inefficient at controlling hardware. To overcome this problem, we are planning to develop SystemC code generator for ASSL. The future work is also concerned with further EAS development by including real pieces of hardware attached to the control software generated by the ASSL framework. Moreover, we intend to build EAS prototypes incorporating other self-managing policies such as self-protecting and self-adapting. This will help us to investigate and develop embedded systems able to automatically detect and fix performance problems, e.g. by switching to alternative modes of execution.

ACKNOWLEDGEMENTS

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

REFERENCES