

ULRR

Developing self-managing embedded systems with ASSL

Item Type	Meetings and Proceedings
Authors	Vassev, Emil;Hinchey, Mike
Citation	Proceedings of the IMCSIT;2009
Publisher	IEEE Computer Society
Download date	2026-06-13 09:35:39
Item License	https://creativecommons.org/licenses/by-nc-sa/1.0/
Link to Item	https://hdl.handle.net/10344/1797

Developing Self-Managing Embedded Systems with ASSL

Emil Vassev*, Mike Hinchey**

*Lero—the Irish Software Engineering Research Centre, University College Dublin
Ireland (e-mail: emil.vassev@lero.ie)

** Lero—the Irish Software Engineering Research Centre, University of Limerick
Ireland (e-mail: mike.hinchey@lero.ie)

Abstract: We present a new formal approach to the implementation of embedded systems, arrived at by introducing self-management capabilities to the same. We use the ASSL (Autonomic System Specification Language) framework to approach the problem of formal specification and automatic code generation of embedded systems. Some features of ASSL help to specify event-driven embedded systems where hardware is sensed via special metrics intended to drive events and self-management policies. The latter can be specified to handle critical situations in an autonomous reactive manner. Moreover, we present a case study where we use ASSL to specify control software for the wide-angle camera carried on board by NASA's Voyager II spacecraft.

Keywords: embedded systems, real-time systems, reactive systems, self-management, ASSL

1. INTRODUCTION

Embedded systems have taken their fair share of the tremendous expansion of IT in our daily lives. A most important feature of these systems is that they are able to provide a secure and highly available environment in conjunction with the ability to deliver deterministic real-time services. In addition, they often have long-life and 24x7 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 just ten years ago. 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 also must address the quality of service (QoS) in embedded critical systems where it is often the main concern.

We present our approach to this problem, whereby the ASSL (Autonomic System Specification Language) (Vassev, 2008; Vassev and Hinchey, 2009a) 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) (Murch, 2004). AC is recognized as a potential long-term solution to the problem of increasing system complexity and costs of maintenance. The idea is that software systems must manage themselves automatically by controlling complexity through self-management based on high-level objectives. In this paper, 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 rest of this paper is organized as follows. In Section 2, we review 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 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 future research and investigation trends.

2. RELATED WORK: PROGRAMMING EMBEDDED SYSTEMS

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 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 real-time systems which have timing constraints introduced to assure 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 (Labrosse, 1998).

Another example is the SystemC language, which is standardized by the IEEE (cf. IEEE 1666-2005 Standard). This language originated from C++ as a language for system modeling intended to enable “system-to-silicon” design flows (OSCI, 2009).

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. Key characteristics that helped in this 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 a RTOS. Java has become popular in the development of networked embedded systems (Fleischmann, Buchenrieder, and Kress, 1999). Note that ASSL generates executable Java code (cf. Section 3.3).

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 level of abstraction provided by the formal notation, and from the use of software verification tools that help to discover design and implementation flaws at early stages of the software lifecycle. 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 (Amey, 2002).

Special formal languages called *synchronous languages* are dedicated to the programming of reactive systems (Halbwachs, 1993). 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 were also used to develop DSP chips for mobile phones, to design and verify DVD chips, and to program the flight control software of Rafale fighters (Benveniste et al., 2003).

Esterel (Berry and Gonthier, 1992) 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 asynchronous model of time like that used in synchronous digital circuits.

SDL (Ellsberger, Hogrefe, and Sarma, 1997) 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 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-managing features. Here, we are targeting embedded systems, whose Java implementation is automatically generated from their ASSL specification. 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.

3. ASSL

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 special 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. 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 (cf. Table 1).

Table 1. ASSL Multi-Tier Specification Model

AS	AS Service-Level Objectives	
	AS Self-Management Policies	
	AS Architecture	
	AS Actions	
	AS Events	
ASIP	AS Metrics	
	AS Messages	
	AS Channels	
AE	AS Functions	
	AE Service-Level Objectives	
	AE Self-Management Policies	
	AE Friends	
	AEIP	AE Messages
		AE Channels
		AE Functions
		AE Managed Elements
	AE Recovery Protocols	
	AE Behavior Models	
	AE Outcomes	
AE Actions		
AE Events		
AE Metrics		

The ASSL specification model considers the ASs as being composed of special autonomic elements (AEs) interacting over 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 protocols.

Table 1 presents the multi-tier specification model of ASSL. As shown, it decomposes an AS in two directions:

- 1) into levels of functional abstraction;
- 2) into functionally related tiers (sub-tiers).

With the first decomposition (cf. first column in Table 1), an AS is presented from three different perspectives, these depicted as three main tiers:

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.

The ASIP Tier (AS interaction protocol) forms a communication perspective exposing a means of communication for the AS under consideration.

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, which must be synchronized with the behavior rules from the global AS perspective.

Here, 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 (Murch, 2004).

In ASSL, self-management policies are specified with special constructs termed *fluents* and *mappings*.

- 1) A fluent is a state where an AS enters with *fluent-activating events* and exits with *fluent-terminating events*.
- 2) A mapping connects fluents with particular *actions* to be undertaken.

Here, self-management policies are driven by events and actions determined in a deterministic manner, similar to finite state machines. For the purpose of EAS development, self-management policies can be specified to control the EAS hardware. Moreover, real-time systems are bounded with deadline, where the deadline may be a particular time or time interval, or may be 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.

In the following section, we emphasize the ASSL constructs suitable for the specification of EASs. A complete description of the ASSL specification model is beyond the scope of this paper. For more information, we refer the interested reader to (Vassev, 2008).

3.2 ASSL Features for Embedded Systems

ASSL implies a number of important specification constructs and techniques, which allow for a valuable formal approach to the development of embedded systems.

3.2.1 Events

In general, embedded systems are considered event-driven. ASSL exposes a rich set of techniques and constructs for specifying events, which makes the framework suitable for the specification and code generation of event-driven embedded systems. From the EAS development perspective, ASSL events are one of the most important aspects in ASSL. By its nature, an ASSL 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 ASSL events, one may use logical expressions over *service-level objectives (SLO), metrics, other 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.

In addition, ASSL introduces a GUARDS clause to event specification to define conditions that must be stated before an event can be prompted.

```
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
```

The ASSL code above shows a specification sample specifying two events. The first one (named lunchTime) is a timed event that will be prompted at 12:00 AM to notify the system that it is lunchtime. The second one (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, perhaps 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 (Vassev, 2008). Four different types of metrics are allowed:

- *resource metrics* – measure managed resource quantities;
- *quality metrics* – measure system qualities like *performance*, *response time* etc;
- *scalar metrics* – monitor predefined dynamic AS variables;
- *composite metrics* – a function of other metrics.

For the needs of embedded system development the most important are resource metrics. Note that the *managed resource* (cf. 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.

```

METRICS {
  // increments when a failed node has been discovered
  METRIC numberOfFailedNodes {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE {
      AEIP.MANAGED_ELEMENTS.STAGE_ME.countFailedNodes }
    DESCRIPTION { "counts failed nodes" }
    VALUE { 0 }
    THRESHOLD_CLASS { integer [0] } // valid only when holds 0
  }
}

```

In the sample above, the metric numberOfFailedNodes gets updated via a special interface function called countFailedNodes and embedded in the specification of a STAGE_ME managed element. The latter represent the controlled managed resource, which in an EAS is the hardware.

Moreover, metrics are specified with special range of acceptable values expressed with a special ASSL construct called *threshold class*. In general, a threshold class determines rules for *valid* and *invalid* metric values. 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 is very useful, because we can specify metrics prompting events when a real-time system's deadline cannot be met and the EAS in question must switch to an alternative execution path. For example, the sample above specifies a metric, which is valid only when zero (0) holds.

3.2.3 Managed Resource

An AE typically controls a *managed resource* specified in ASSL in the form of *managed elements* (Vassev, 2008). 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 same. ASSL provides an abstraction of a managed element through specified interface functions. Here, ASSL can specify and generate the interface controlling a managed element, but not the implementation of this interface in that controlled managed element. Here, when developing an EAS, the generated interface must be implemented by the piece of 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) (cf. Table 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.

For the EAS case, at this tier we should emphasize the specification of the managed element representing the controlled hardware.

```

AEIP {
  MANAGED_ELEMENTS {
    MANAGED_ELEMENT STAGE_ME {
      INTERFACE_FUNCTION countFailedNodes {
        RETURNS { integer }
      }
      // runs the replica of a failed node
      INTERFACE_FUNCTION runNodeReplica {
        PARAMETERS { NetNode node }
        ONERR_TRIGGERS { EVENTS.nodeReplicaFailed }
      }
    }
  }
} // AEIP

```

As shown by the sample above, 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 (cf. 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 (Vassev, 2008). 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 ASSL Super Loop Architecture for Embedded Systems

ASSL automatically generates an executable multithreaded Java application from a valid ASSL specification. Note that ASSL performs formal verification of the ASSL-specified ASs before and after generating the Java code (Vassev, Hinchey, and Quigley, 2009a, 2009b). Here, a valid specification is considered one that has passed through the formal verification process.

ASSL automatically generates an executable multithreaded Java application from a valid ASSL specification. Note that ASSL performs formal verification of the ASSL-specified ASs before and after generating the Java code (Vassev, Hinchey, and Quigley, 2009a, 2009b). 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. In addition, logical errors, such specification and implementation flaws, are a subject of special ASSL model checking mechanisms, which are still under development. Here, a *valid specification* is considered one that has passed through the formal verification process.

Although considered efficient, the ASSL consistency checking mechanism cannot handle *logical errors* (specification flaws) and thus, it is not able to assert safety (e.g., freedom from deadlock) or liveness properties. Thus, a *model checking*¹ validation mechanism able to handle such errors is under development.

In addition to the suitable constructs allowing for the specification of embedded systems, ASSL also provides a suitable architecture for the automatically generated EASs. In this section, we present the architecture of the ASSL-generated EASs.

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 (IBM, 2006; Vassev, 2008). A control loop is generated to apply control rules specified and implemented as self-management policies, SLO (service-

level objectives) and metrics. The following Java code fragment presents an ASSL-generated control loop.

```
protected void controlLoop() {
    try {
        //monitor-analyzer-simulator-executor
        oMonitor.perform();
        oAnalyzer.perform();
        oSimulator.perform();
        oExecutor.perform();

        //applies self-management policies
        applyPolicies();

        Thread.sleep(tDelay);
    }
    catch ( InterruptedException ex )
    {....}
}
```

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 control loop calls are as following:

- 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 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 notifying about the discovered problem. The algorithm implemented here follows the behavior exposed by a finite state machine. 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 a finite state machine.

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

In the second part of the control loop, as shown, an `applyPolicies()` method is called. The following code

¹ Model checking is a formal verification approach to automated verification of finite state systems by employing efficient graph-search algorithms and correctness properties.

fragment presents the generated implementation of that method.

```
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();
        }
    }
}
```

Here, for each policy a `doAllMappings()` method is called where actions are called if a policy is activated by one or more fluents (cf. Section 3.1).

Based on the control loop technique described above, ASSL generates EASs with the so-called *super loop architecture* (Kurian and Pont, 2007). 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.

The following pseudocode presents 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.

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

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. Note that 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 executes indirectly 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 following.

```
public void run() {
    ....
    /**** runs the control loop
    while ( !bStopAE ) {
        controlLoop();
        try {
            Thread.sleep( tControlLoopDelay );
        }
        catch ( InterruptedException ex )
        { .... }
    }
}
```

Here, 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.

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 (Vassev and Hinchey, 2009b). The NASA Voyager Mission (The Planetary Society, 2009) was designed for exploration of the Solar System. The original mission objectives were to explore the outer planets of the Solar System and as the Voyager I and Voyager II flew 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. In this paper, we emphasize the specification of the Voyager's wide-angle camera, which could be considered as an EAS. For more information on the ASSL specification model for the NASA Voyager Mission, we advise the interested reader to refer to (Vassev and Hinchey, 2009b).

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. The following ASSL code presents the specification of the `IMAGE_PROCESSING` policy.

```
AESELF_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.takePicture }
}
MAPPING {
  CONDITIONS { inProcessingPicturePixels }
  DO_ACTIONS { ACTIONS.processPicture }
}
}
} // AESELF_MANAGEMENT

```

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 `takePicture` 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 (Vassev and Hinchey, 2009a). Therefore, in order to specify an EAS (embedded AS) we must specify one or more managed elements intended to provide the means of control over the hardware in that embedded system (cf. Section 3.2.3).

```

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
  }
} // AEIP

```

Here, the `wideAngleCamera` managed element 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. The following partial specification presents the `doTakePicture` action mapped to the `inTakingPicture` fluent and calling the `takePicture` interface function to ask the hardware to take a picture. Note that the `pictureTaken` event is prompted if the `doTakePicture` action is performed with no errors.

```

ACTION doTakePicture { // takes a picture of an interesting spot/object
  ....
  DOES {
    IF AES.Voyager.isWideAngleImage THEN
      call AEIP.MANAGED_ELEMENTS.wideAngleCamera.takePicture;
    }
  }
  END
}
TRIGGERS { EVENTS.pictureTaken }
}

```

Moreover, an `interestingObjects` metric is specified to count all detected objects of interest, which the Voyager AE takes

pictures of. The source of this metric is specified as one of the managed element interface functions (cf. `countInterestingObjects`); i.e., the metric gets updated by that interface function. The following ASSL code presents the specification of this metric.

```

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

```

Further, following the event-driven behavior specified for the EAS, we see that the `timeToTakePicture` event (recall that it activates the `inTakingPicture` fluent) is prompted by a change in this metric's value. Here, in order to simulate this condition, we also activate this event every 60 seconds on a periodic basis. The following ASSL code sample presents the `timeToTakePicture` event specification.

```

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

```

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 entire ASSL specification of the Voyager II Mission (Vassev and Hinchey, 2009b). 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 simulated ones. Here, 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 triggered to take pictures every 60 sec), the run-time 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 (Vassev and Hinchey, 2009b).

5. CONCLUSIONS

In this paper, we have demonstrated how ASSL – a formal tool dedicated to 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 structures and techniques 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. Due 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*. Here, as a proof of concept, we have successfully used ASSL to specify and generate an EAS that controls the wild-angle camera carried on board by NASA's Voyager II spacecraft.

Future work is 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 self-managing policies such as self-healing, 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.

REFERENCES

- Amey, P. (2002). Correctness By Construction: Better Can Also Be Cheaper. *CrossTalk Magazine*. The Journal of Defense Software Engineering.
- Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., and De Simone, R. (2003). The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, Vol. 91(1), pp. 64–83. IEEE Computer Society Press.
- Berry, G. and Gonthier, G. (1992). The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, Vol. 19(2), pp. 87–152.
- Kurian, S. and Pont, M.J. (2007). Maintenance and Evolution of Resource-Constrained Embedded Systems Created Using Design Patterns. *Journal of Systems and Software*. Vol. 80(1), pp. 32–41.
- Ellsberger, J., Hogrefe, D., and Sarma, A. (1997). *SDL: Formal Object-Oriented Language for Communicating Systems*. 2 ed. Prentice Hall. New Jersey, USA.
- Fleischmann, J., Buchenrieder, K., and Kress, R. (1999). Java Driven Codesign and Prototyping of Networked Embedded Systems. Proceedings of the 36th ACM/IEEE conference on Design automation, p.794–797. New Orleans, Louisiana, USA.
- Halbwachs, N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer Academic Publishers, Boston.
- IBM Corporation (2006). *An Architectural Blueprint for Autonomic Computing*. White Paper, 4th ed. IBM Corporation.
- Labrosse, J. (1998). *MicroC/OS-II*. CMP Books. Kansas, USA.
- Murch, R. (2004). *Autonomic Computing: On Demand Series*. IBM Press, Prentice Hall.
- OSCI: Open SystemC Initiative (2009). *SystemC*. <http://www.systemc.org/> (last visited on July 7, 2009).
- The Planetary Society (2009). *Space topics: Voyager – the story of the mission*. http://planetary.org/explore/topics/space_missions/voyager/objectives.html (last visited on July 10, 2009).
- Vashev, E. (2008). *Towards a Framework for Specification and Code Generation of Autonomic Systems*. PhD Thesis. Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.
- Vashev, E. and Hinchey, M. (2009a). ASSL: A Software Engineering Approach to Autonomic Computing. *IEEE Computer*, Vol. 42(6), pp. 90–93. IEEE Computer Society.
- Vashev, E. and Hinchey, M. (2009b). Modeling the Image-Processing Behavior of the NASA Voyager Mission with ASSL. *Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09)*. IEEE Computer Society (to appear).
- Vashev, E., Hinchey, M., and Quigley, A. (2009a). Towards Model Checking with Java PathFinder for Autonomic Systems Specified and Generated with ASSL. *Proceedings of the 4th International Conference on Software and Data Technologies (ICSOF 2009)*. INSTICC, Sofia, Bulgaria (to appear).
- Vashev, E., Hinchey, M., and Quigley, A. (2009b). Model Checking for Autonomic Systems Specified with ASSL. *Proceedings of the First NASA Formal Methods Symposium (NFM 2009)*. pp.16–25. NASA.