An ASSL Approach to Handling Uncertainty in Self-adaptive Systems

Emil Vassev, Mike Hinchey
Lero - the Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
emil.vassev@lero.ie, mike.hinchey@lero.ie

Dharini Balasubramaniam, Simon Dobson
School of Computer Science
University of St Andrews
St Andrews, United Kingdom
dharini@cs.st-andrews.ac.uk, sd@cs.st-andrews.ac.uk

Abstract—**Both modularity and loose-coupling properties inherent to the self-adaptive systems offer the opportunity for ad-hoc service compositions, dynamic change and adaptation. To provide such a dynamic and self-adapting behavior, developers emphasize special self-management policies. ASSL (Autonomic System Specification Language) is a formal tool where such policies might be formally specified, validated and implemented. Intrinsically, the ASSL-developed policies are very strict and may impose quite restrictive behavior, which sometimes is undesirable. To solve the problem, we are currently developing special mechanisms for ASSL that help to specify policies that might evolve in order to satisfy system goals changing in the course of system adaptation. This paper presents our work on a mechanism imposing special loose self-management policies introducing flexibility into the self-adapting behavior.**

Keywords-self-adaptive systems; formal methods; ASSL

I. INTRODUCTION

A self-adaptive system changes its behavior in response to stimuli from its execution and operational environment [1]. As software is used for more pervasive and critical applications, support for self-adaptation is increasingly seen as vital in avoiding costly disruptions for repair, maintenance and evolution of systems. However, the wider use of self-adaptive systems in a variety of domains also leads to more challenges in designing and developing them. Salehie and Tahvildari [1] identify some of these challenges including building multi-property self-adaptive systems and deciding on adaptive processes in a dynamic and uncertain environment. Any long-running system is subject to uncertainty in its execution environment due to potential changes in requirements, business conditions, available technology, etc. Thus, it is important to capture and cater for uncertainty as part of the development process. Failure to do so may result in systems that are too rigid to be fit for purpose, which is of particular concern for the domains that typically make use of self-adaptive technology. We hypothesize that modeling uncertainty and developing mechanisms for managing it as part of software design and implementation will lead to systems that are:

- more expressive of the real world;
- fault tolerant due to fluctuations in requirements and conditions being anticipated;
- flexible and able to manage dynamic changes.

Achieving this goal requires languages and notations that are able to model uncertainty at different stages of the software lifecycle and tools that are able to work over these models and produce system code that reflects them. The ability to specify flexible policies is an important factor in dealing with uncertainty. For example, rather than mandating a repetition of `x` seconds for an operation, we may only require that the operation is carried out as often as possible. However it is also essential to identify properties that must remain invariant to ensure correctness of system execution.

Whittle et al. [2] have carried out some work in explicitly capturing uncertainty as part of specifying system requirements. Our approach builds on the ideas presented in [2] to deal with uncertainty throughout the development lifecycle. We extend ASSL (Autonomic System Specification Language) [3], a dedicated to Autonomic Computing (AC) [4, 5] specification language for self-adaptive systems, with constructs to capture and manage uncertainty in specifying system behavior, configuration and properties. ASSL provides a strong foundation for this work as it has already been used to model non-trivial self-adaptive systems [6, 7, 8] and has an established toolset including a code generator and a model checker. The code generation tool for the extended language will be expanded to produce an implementation skeleton, which codifies the so-called *relaxed properties* of the system.

This paper is structured as follows. We outline some related work in Section II. A brief overview of the ASSL language is provided in Section III while the next section describes the extensions to the language to capture uncertainty. A case study illustrating the extended ASSL is given in Section V. We discuss our conclusions from this work and provide some thoughts on future work in the final section.

II. RELATED WORK

Self-adaptive systems have inspired growing interest in the formalization of such systems. In general, the formal approaches help developers precisely describe with the logical underpinning of mathematics features of a self-adaptive system and validate those features at a higher level of abstraction than the one provided by implementation. Policy models, goal models and feature models are used to specify possible behaviors of autonomic systems (ASs), emphasizing the self-configuration, self-healing, and self-optimization aspects present in self-adaptive systems.

IBM Research has developed a framework called Policy Management for Autonomic Computing (PMAC) [9]. This framework provides a standard model for the definition of policies and an environment for the development of software objects that hold and evaluate policies. PMAC is used for development and management of intelligent autonomic software agents. With PMAC, these agents incorporate the ability to change dynamically their behavior. This is provided by a formal specification of policies by encompassing the scope under which these policies are applicable.

A NASA-developed formal approach, named R2D2C (Requirements to Design to Code) is described in [10]. In this approach, system designers may write specifications as scenarios in constrained (domain-specific) natural language, or in a range of other notations (including UML use cases). These scenarios are then used to derive a formal model that fulfills the requirements stated at the outset, and which is subsequently used as a basis for code generation. R2D2C relies on a variety of formal methods to express the formal model under consideration. The latter can be used for various types of analysis and investigation, and as the basis for fully formal implementations as well as for use in automated test case generation.

Banatre et al. [11] use the chemical reaction metaphor to express the coordination of computations. In this approach the Gama Formalism is used to describe computation in terms of chemical reactions (described as rules) in solutions (described as multi-sets of elements). When applied to AS specification, the Gama Formalism captures the intuition of a collection of cooperative components that evolve freely according to some predefined constraints (rules). System self-management arises as a result of interactions between components, in the same way as "intelligence" emerges from cooperation in colonies of biological agents.

Whittle et al. [2] define a special requirements language called RELAX that allows developers to specify requirements that may be relaxed at run-time. RELAX helps to address uncertainty in requirements to support self-adaptive systems development, in a way such that the uncertainty can be specified declaratively rather than by simply enumerating all alternative goals.

As mentioned above, this research builds on the ideas of *relaxing properties* presented in [2]. In this paper we present an extension of the ASSL language intended to tackle uncertainty in self-management policies forming the self-adapting behavior of a system.

III. Overview of ASSL

By its virtue, the Autonomic System Specification Language (ASSL) [3] provides both formal notation and tools for building software mechanisms for self-management in complex systems where the problem of *formal specification*, *validation*, and *code generation* of autonomic systems (ASs) is approached within a framework. Here, being a formal method dedicated to AC, ASSL helps AC researchers with *problem formation*, *system design*, *system analysis* and *evaluation*, and *system implementation*. A powerful and domain-specific formal notation is provided to specify required features and to model high-level models of ASs incorporating those features. Moreover, suitable mature tool support is provided to allow ASSL specifications to be edited and validated and Java code to be generated from any valid specification.

A. *The ASSL Specification Model*

The ASSL formal notation [3] is based on a specification model exposed over hierarchically organized *formalization tiers* (see Table 1). The specification model provides both infrastructure elements and mechanisms needed by an AS. Thus, each tier of the ASSL specification model is intended to describe different aspects of the AS in question, such as *service-level objectives, self-management policies, interaction protocols, events, actions*, *autonomic elements*, etc. This helps to specify an AS at different levels of abstraction (imposed by the ASSL tiers) where the AS in question is composed of special *autonomic elements* (AEs) interacting over special *interaction protocols*.

TABLE I. ASSL Multi-tier Specification Model

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

As shown in Table 1, the AS Tier specifies an AS in terms of *service-level objectives* (SLO), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics*. The AS SLO is a high-level form of behavioral specification that establishes system objectives such as performance. The metrics constitute a set of parameters and observables controllable by the AEs. At the AS Interaction Protocol tier, the ASSL framework specifies an AS-level interaction protocol (ASIP), a public communication interface, expressed with channels, communication functions and messages. Finally, at the AE Tier, the ASSL formal model considers AEs to be analogous to software agents able to manage their own behavior and their relationships with other AEs. In this tier, ASSL describes the individual AEs of the AS.

In general, an ASSL specification is built around one or more self-management policies. This makes the ASSL specifications AC-driven, where ASs are modelled taking into account the main goal of AC - self-management based on four main principles: *self-configuring*, *self-healing*, *self-optimizing*, and *self-protecting* (self-CHOP). ASSL addresses these self-CHOP principles as *self-management policies* specified at both AS and AE tiers. ASSL specifies such policies with special constructs termed as *fluents* and *mappings*. Whereas the former are considered as specific policy conditions, the latter map these conditions to appropriate actions. Fluents are expressed with *fluent-activating* and *fluent-terminating* events, i.e., the self-management policies are driven by events. In order to express mappings, conditions and actions are considered, where the former determine the latter in a deterministic manner. The following ASSL code presents a sample specification of a self-healing policy.

```
ASSELF_MANAGEMENT {
    SELF_HEALING {
        FLUENT inLosingSpacecraft {
            INITIATED_BY { EVENTS.spaceCraftLost }
            TERMINATED_BY { EVENTS.earthNotified }
        }
        MAPPING {
            CONDITIONS { inLosingSpacecraft  }
            DO_ACTIONS { ACTIONS.notifyEarth }
        }
    }
} // ASSELF_MANAGEMENT
```

B. *Operational Evaluation*

The formal evaluation of the operational behavior of ASSL specification models is a stepwise evaluation of the specified ASSL tiers, where the latter are evaluated as state transition models in which operations cause a current state to evolve to a new state [3]. Thus, if we use the convention for semantic function in which **σ** states for a current state and **σ′** states for a new state then the state evolution caused by an operation Op is denoted as $\sigma \xrightarrow{Op(x_1,x_2,\dots,x_n)} \sigma'$, where the operation $Op(x_1, x_2, \dots, x_n)$ is an abstraction of a transition operation performed by the framework that potentially takes n arguments. All the arguments are evaluated to their expression value first, and then the operation is performed. Here, in the *standard* ASSL Op is a transition operation of type O^{trans} (see the set definition below).

$$O^{trans}\ \{\ DegradSLO, NormSLO, FluentIn, FluentOut, ActionMap, Action, Function, MsgRcvd, MsgSent, Event, EventOver, Metric, ChangeStruct, CreateAE, ExtClass, BhvrModel, RcvryProtocol, MngRsrcFunction, Outcome\ \}$$

In addition, the operational semantics of the ASSL tiers introduces the notion of *tier environment* ρ presenting the *host tier* of the sub-tiers or clauses under evaluation. For example, the AS Tier is a host tier of the AS Actions sub-tier (see Table 1). Thus, we write $\boldsymbol{\rho} \vdash_{\sigma}$ to mean that ρ is evaluated in context σ and $\boldsymbol{\rho} \vdash_{\sigma} \boldsymbol{e} \longrightarrow \boldsymbol{e}'$ to mean that, in a given *tier environment* ρ (host tier for the expression e) one step of the evaluation of expression e in the context σ results in the expression e'. Here, the context σ is defined by the tier content, i.e., sub-tiers, tier clauses, etc. Note that the ASSL tiers may participate in expressions. For example, AS/AE SLO, AS/AE policies, fluents, AS/AE events, and AS/AE metrics can participate in Boolean expressions, where they are evaluated as **true** or **false** in the context of their host tier based on their performance.

1) ASSL Self-management Policy Evaluation: Originally, an ASSL policy is evaluated over the evaluation of its fluents and mappings and it is closely related to the events occurring in the system. The original operational evaluation of a fluent f follows the following algorithm:

If an event has occurred in the system then:

1. Process the *INITIATED_BY {...}* clause to check if that event initiates the fluent f and if so, initiate that fluent with the *FluentIn()* system transition operation:
    - Process the policy's *MAPPING {....}* clauses comprising the fluent f in their *CONDITIONS {....}* clause.
    - Evaluate the *CONDITIONS {....}* clause and if the stated conditions are held then evaluate the *DO_ACTIONS {....}* clause to perform the actions listed there.
2. Process the *TERMINATED_BY {...}* clause to check if that event terminates the fluent f and if so, terminate it. Fluent termination is possible iff that fluent has been initiated.

The semantic rules 1 through to 4 present the operational semantics that cope with the algorithm stated above. In these rules, each premise is a system transition operation such as $Event\ (ev)$, $FluentIn\ (f,\ ev)$, $FluentOut\ (f,\ ev)$, and $ActionMap\ (f, a)$.

1) $$\frac{\sigma \xrightarrow{Event(ev)} \sigma'}{f \vdash_{\sigma'} \boldsymbol{INITIATED\_BY}\{ev_1,\dots,ev_n\} \xrightarrow{FluentIn(f,ev)} \sigma''}\ ev \in \{ev_1,.,ev_n\}$$

2) $$\frac{\sigma \xrightarrow{FluentIn(f,ev)} \sigma' \quad \sigma' \xrightarrow{Event(ev')} \sigma''}{f \vdash_{\sigma''} \boldsymbol{TERMINATED\_BY}\ \{ev_1,\dots,ev_n\} \xrightarrow{FluentOut(f,ev')} \sigma'''}\ ev' \in \{ev_1, \dots, ev_n\}$$

3) $$\frac{\sigma \xrightarrow{FluentIn(f,ev)} \sigma'}{map \vdash_{\sigma'}\ \boldsymbol{CONDITIONS}\{\ f_1,\dots,f_n\} \xrightarrow{ActionMap(f,a)} \sigma''}\ f \in \{f_1, \dots, f_n\}$$

4) $$\frac{\sigma \xrightarrow{ActionMap(f,a)} \sigma'}{map \vdash_{\sigma'}\ \boldsymbol{DO\_ACTIONS}\{\ a_1,\dots,a_n\} \xrightarrow{\forall a \in \{\ a_1,\dots,a_n\} \bullet Action(a)} \sigma''}\ a \in A^{\sigma}$$

Here, A^{σ} is the finite set of actions in the context σ and the first premise in rule 2 evaluates whether the fluent f is initiated, i.e., we can terminate initiated fluents only.

2) ASSL Action Evaluation: ASSL actions comprise the following tier clauses: *PARAMETERS {...}, RETURNS {...}, GUARDS {...}, ENSURES {...}, DOES {...}, ONERR_DOES {...}, TRIGGERS {...},* and *ONERR_TRIGGERS {...}* [3]. The following is an example of ASSL action specified with some of the clauses listed above. Note that only the *DOES {...}* clause is mandatory.

```
ACTION doPlanning {
    PARAMETERS  { State initialState; State goalState; TIME deadline }
    GUARDS { EVENTS.newAsteroidFoundReceived }
    ENSURES { EVENTS.planningDone }
    DOES {
        IF deadline > 00:00:00 THEN
            set METRICS.teamTaskDeadline.VALUE = deadline;
            set METRICS.teamTaskTime.VALUE = 00:00:00;
            apply AES.ae1.BEHAVIOR_MODELS.modelPlanning
        END;
        instrumentTasks = call IMPL planTask (initialState, goalState, deadline)
    }
    TRIGGERS { EVENTS.planningDone }
    ONERR_TRIGGERS { EVENTS.planningImpossible }
}
```

The operational evaluation of an ASSL action follows the following algorithm:

1. Map the arguments, if any, from the action call to the parameters (*PARAMETERS {...}* clause).
2. Process the action guards, if any (*GUARDS {...}* clause):
    - If the guards are held then perform the action.
    - Otherwise, deny the action.
3. Evaluate the variable declarations, if any.
4. Process the DOES {…} clause:
    - If a return statement is hit, then stop the action and return a result.
    - Else, process all the statements until the end of the *DOES {...}* clause.
5. If the DOES {…} clause is evaluated correctly, then evaluate the ENSURES {…} clause (in respect to the TRIGGERS {…} clause):
    - If the *ENSURES {...}* clause is held then trigger notification events via the *TRIGGERS {...}* clause and exit the action normally.
    - Else, process the *ONERR_DOES {...}* clause and trigger error events via the *ONERR_TRIGGERS {...}* clause.
6. If an error occurs while evaluating the action clauses, then stop the evaluation process and:
    - Process the *ONERR_DOES {...}* clause (similar to the evaluation of the *DOES {...}* clause), if any.
    - Trigger error events via the *ONERR_TRIGGERS {...}* clause, if any.

Note that an ASSL action is evaluated operationally when it is mapped to a fluent (see Section III.B.1) or called internally from another action via a special *CALL* statement [3]. For example:

```
CALL ACTIONS.checkInstrument;
```

IV. ASSL MAY MECHANISM

The standard ASSL does not provide specification constructs that help developers specify policies evolving in the course of system adaptation. Moreover, the ASSL self-management policies are very strict and may impose quite *restrictive* behavior, which sometimes leads to undesirable persistency in the form of constant system's attempts to follow the predefined self-management policies. In the course of this project, we have developed a special ASSL mechanism that introduces *relaxed properties* in the ASs developed with ASSL. This mechanism is termed "The MAY Mechanism" and helps developers specify special *loose self-management policies* capable of *flexible self-managing behavior* allowing ASs to be more agile. Such *loose policies* introduce points of flexibility and nondeterministic choice in their behavior. The MAY Mechanism is an extension of ASSL introducing a special *MAY* specification modifier for ASSL sub-tiers. This modifier is intended to provide flexibility and fault-tolerance in the autonomic behavior via the specification of *self-management policies*, *actions* and *SLO*.

A. MAY Self-management Policies

With the ASSL MAY Mechanism, the self-management policies might be specified at an agile level of autonomic behavior, where an AS is more flexible in terms of decision making involving uncertainty issues. A policy specified with the MAY Mechanism allows an AS to decide on-the-fly if a certain behavior "may" be followed rather than "must".

A MAY policy is specified by using the new *MAY* specification modifier when specifying the policy's fluents. Note that a self-management policy might have one or more MAY fluents coexisting with other non-MAY fluents. The following ASSL specification demonstrates the specification of the *inLosingSpacecraft* fluent with the *MAY* modifier.

```
SELF_HEALING {
    MAY FLUENT inLosingSpacecraft {
        INITIATED_BY { EVENTS.spaceCraftLost }
        TERMINATED_BY { EVENTS.earthNotified }
    }
    MAPPING {
        CONDITIONS { inLosingSpacecraft  }
        DO_ACTIONS { MAY ACTIONS.notifyEarth }
    }
}
```

A MAY fluent requires MAY actions and is always terminated after the execution of its mapped actions even no fluent-terminating events have occurred in the system. Thus, the operational evaluation of a *MAY* fluent f adds a 3-rd step to the fluent evaluation (see Section III.B.1):

3. Silently terminates the fluent f if it is still active.

The operational evaluation of a MAY fluent introduces a new state transition operation (see Section 3.B) termed $MayFluentOut$. The following inference rule demonstrates the new behavior introduced to MAY fluents (also described in step 3 above).

5) $$\frac{\sigma \xrightarrow{FluentIn(f,ev)} \sigma' \quad \sigma' \xrightarrow{ActionMap(f,a)} \sigma'' \quad \sigma'' \xrightarrow{\forall a \in \{a_1,\ldots,a_n\} \bullet Action(a)} \sigma'''}{f \vdash_{\sigma'''} \xrightarrow{MayFluentOut(f)} \sigma^{IV}}$$

Therefore, a MAY fluent may be terminated either in normal way when a fluent-terminating event is triggered in the system or silently after the mapped actions have been executed. This gives the AS the right to try only one execution of the fluent's actions even the goals after their execution are not achieved.

B. *MAY Actions*

An ASSL MAY action is an ASSL action called with the *MAY* modifier (see the example below). The ASSL MAY Mechanism requires that a MAY action performs as a normal ASSL action (see the operational evaluation of ASSL action in Section III.B.2) if no exception is raised during its execution. However, actions called with the *MAY* modifier do not raise exceptions if cannot succeed and thus, they do not trigger erroneous events even the latter are specified in the *ONERR_TRIGGER {....}* clause. Note though, that a MAY action always triggers events specified in the *ON_TRIGGER {....}* clause. Although not defined as a semantic rule, to comply with the MAY Mechanism the *MAY fluents* shall be mapped to *MAY actions*. For example:

```
MAPPING {
     CONDITIONS { inLosingSpacecraft  }
     DO_ACTIONS { MAY ACTIONS.notifyEarth }
}
```

Note that according to the ASSL Operational Semantics [3] the act of mapping an action to a fluent requires that action be evaluated operationally (see Section III.B.2). However, when the mapping is done with the *MAY* modifier the action's evaluation follows the following algorithm:

1. Map the arguments, if any, from the action call to the parameters (*PARAMETERS {…}* clause).
2. Evaluate the variable declarations, if any.
3. Process the DOES {…} clause:
   - If a return statement is hit, then stop the action and return a result.
   - Else, process all the statements until the end of the *DOES {…}* clause.
   - If the DOES {…} clause is evaluated correctly, then trigger notification events via the *TRIGGERS {…}* clause and exit the action.
   - If an error occurs while evaluating the action clauses, then stop the evaluation process and process the *ONERR_DOES {…}* clause (similar to the evaluation of the *DOES {…}* clause), if any.

Therefore, the MAY evaluation of an ASSL action excludes the evaluation of specified *GUARDS {…}*, *ENSURES {…}* and *ONERR_TRIGGERS {…}* clauses. Thus, a MAY action does not raise erroneous events, which helps the system handle uncertainty without propagating erroneous events when an action cannot be performed due to problems in the *GUARDS {…}*, *ENSURES {…}* or *DOES {…}* clauses.

C. *MAY Service-level Objectives*

According the ASSL Operational Semantics [3], an ASSL event might be prompted by specified SLO (Service-Level Objectives). This is possible when an ASSL event is specified with one of the two clauses: *DEGRADED {…}* or *NORMALIZED {…}*. Whereas the former specifies that the event will be prompted when specific SLO have *degraded* their performance, the latter specifies that the event will be prompted when the SLO have *normalized* their performance. The following rules evaluate these event clauses in a given event tier environment ev defined in the *tier context* σ (see Section III.B):

6) $$\frac{\sigma \xrightarrow{DegradSLO(sloID)} \sigma'}{ev \vdash_{\sigma'} \textbf{DEGRADED}\{sloID\} \xrightarrow{Event(ev)} \sigma''}$$

7) $$\frac{\sigma \xrightarrow{NormSLO(sloID)} \sigma'}{ev \vdash_{\sigma'} \textbf{NORMALIZED}\{sloID\} \xrightarrow{Event(ev)} \sigma''}$$

The MAY Mechanism introduces SLO specified with the *MAY modifier*. For example:

```
MAY SLO Safety_RiskGroup4 {
  IF  ASSLO.Safety_RiskGroup1 and  ASSLO.Safety_RiskGroup2 THEN
    FOREACH  member in AES {  not member.EVENTS.highRadiationLevel  }
  END
}
```

SLO defined as MAY do not trigger events associated with *SLO degradation*, but do trigger events associated with *SLO normalization*. Note that when encountered in expressions, SLO are evaluated as Booleans based on their performance - *false* if degraded and *true* if not [3]. The evaluation of *MAY* SLO is like the evaluation of the regular SLO. However, degraded *MAY* SLO are not considered by the special system's *control loop*, which strives to get the degraded regular SLO back to normal.

V. CASE STUDY

To demonstrate the MAY Mechanism, we used one of the previously developed and published ASSL specification models for the NASA ANTS (Autonomous Nano-Technology Swarm) prospective mission [12]. Here, we applied the MAY Mechanism to the ASSL model for Emergent Self-Adapting Behavior in NASA ANTS Missions [13].

A. *NASA ANTS*

The Autonomous Nano Technology Swarm (ANTS) concept sub-mission PAM (Prospecting Asteroids Mission) is a novel approach to asteroid belt resource exploration. ANTS provides extremely high autonomy, minimal communication requirements to Earth, and a set of very small explorers with a few consumables [12]. These explorers forming the swarm are pico-class, low-power, and low-weight spacecraft units, yet capable of operating as fully autonomous and adaptable agents.

Figure 1 depicts the PAM (Prospecting Asteroid Mission) sub-mission scenario of the ANTS concept mission. As shown, there are three classes of spacecraft: *rulers*, *messengers* and *workers*. By grouping them in certain ways, ANTS forms teams that explore particular asteroids. Hence, ANTS exhibits self-organization since there is no external force directing its behavior and no single spacecraft unit has a global view of the intended macroscopic behavior. The internal organization of a swarm depends on the global task to be performed and on the current environmental conditions. In general, a swarm consists of several sub-swarms, which are temporal groups organized to perform a particular task. Each swarm group has a group leader (*ruler)*, one or more *messengers*, and a number of *workers* carrying a specialized instrument. The *messengers* are needed to connect the team members when they cannot connect directly.

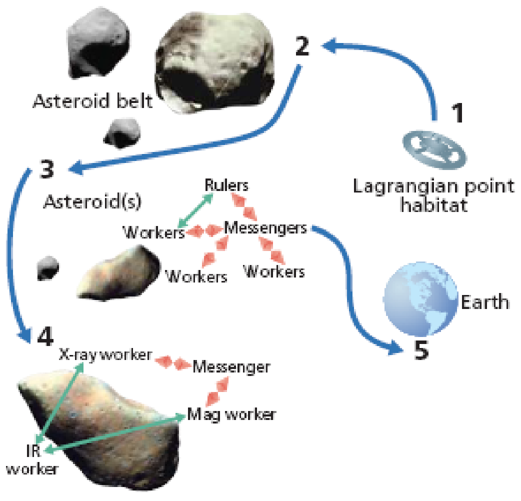

FIGURE 1. ANTS MISSION CONCEPT [12]

B. ASSL Self-transformation Model for ANTS

The ASSL specification model described in [13] involves policies and actions leading to operational transformation of workers. This happens when a worker cannot perform its duties anymore, due to a damage or instrument loss. If so, it

1. asks the ruler to assign a new replacement worker;
2. strives to transform to another category (messenger or ruler) useful to the swarm unit.

A worker may transform to a ruler or a messenger. Moreover, in the case that these transformations are not possible, it may transform to a stand-by "shield". A shield unit sails nearby and strives to protect the replacement worker from different hazards. For example, a shield unit could take the impact of an incoming small asteroid which is about to hit the replacement worker.

The original specification model [13] specifies this self-transformation behavior as a self-management policy as following (note that this is a partial specification):

```
AESELF_MANAGEMENT {
  OTHER_POLICIES {
    SELF_TRANSFORMATION {
      FLUENT unableToExplore {
        INITIATED_BY  { EVENTS.instrIsNonfunctional }
        TERMINATED_BY  {
          EVENTS.canBeRuler , EVENTS.canBeMessenger,
          EVENTS.canBeShield , EVENTS.mustBeDestroyed  }
      }
      FLUENT inTransformToRuler  {
        INITIATED_BY  { EVENTS.canBeRuler }
        TERMINATED_BY  { EVENTS.transformedToRuler ,
          EVENTS.canBeMessenger,  EVENTS.canBeShield }
      }
      FLUENT inTransformToMessenger {
        INITIATED_BY  { EVENTS.canBeMessenger }
        TERMINATED_BY  { EVENTS.transformedToMessenger ,
          EVENTS.canBeRuler ,  EVENTS.canBeShield }
      }
      FLUENT inTransformToShield {
        INITIATED_BY  {
          EVENTS.canBeShield, EVENTS.transformedToShield }
        TERMINATED_BY  { EVENTS.mustBeDestroyed  }
      }
      FLUENT inSelfDestruction {
        INITIATED_BY  { EVENTS.mustBeDestroyed }
      }
      MAPPING  {
        CONDITIONS  { unableToExplore }
        DO_ACTIONS  { ACTIONS.checkTransformation }  }
      MAPPING  {
        CONDITIONS { inTransformToRuler }
        DO_ACTIONS  { ACTIONS.transformToRuler }  }
      MAPPING  {
        CONDITIONS  { inTransformToMessenger }
        DO_ACTIONS  { ACTIONS.transformToMessenger }  }
      MAPPING  {
        CONDITIONS  {  inTransformToShield }
        DO_ACTIONS  { ACTIONS.transformToShield }  }
      MAPPING  {
        CONDITIONS  {  inSelfDestruction }
        DO_ACTIONS  { ACTIONS.selfDestroy }  }
    }
  }
}
```

As shown, we specify the self-transformation behavior as a self-management policy specified at the individual spacecraft level (AE Tier – see the ASSL multi-tier specification model in Section III.A). As specified, the worker can make a few possible choices for transformation when is no longer operational. To specify the self-sacrifice policy we used:

- SELF-TRANSFORMATION – a self-management policy structure. We use a set of fluents and mappings to specify this policy. With fluents, we expressed specific situations, in which the policy is interested, and with mappings, we mapped those situations to actions;
- actions – a set of actions (not shown here) that can be undertaken by the worker in response to certain conditions, and according to that policy;
- events – a set of events (not shown here) that initiate fluents and possibly are prompted by actions according to that policy;
- metrics – a set of metrics (not shown here) needed by that policy.

The unableToExplore fluent takes place when the worker is no longer operational, due to heavy damage or instrument loss. The fluent is initiated by an instrIsNonfunctional event and terminates if one of the events canBeRuler, canBeMessenger, canBeShield, or mustBeDestroyed occurs.

Further, the unableToExplore fluent is mapped to a checkTransformation action (not shown here), which checks for a possible worker transformation and triggers one of the events that terminate the current fluent. Moreover, each of the terminating events initiates a new fluent respectively. The "transform" fluents are mapped to "transformTo" actions in an attempt to transform the worker into a *ruler*, a *messenger*, or a *shield* respectively. As specified, the transformation attempts are hierarchically related. Thus, when possible, the transformation process starts with a transformation into ruler or messenger, and then, in case of failure the algorithm attempts to perform a transformation into shield. At the end of the hierarchically ordered transformations, we have self-destruction of the worker, in case none of the transformations is successful.

The following ASSL code presents a partial specification of one of the "transformTo" actions - transformToRuler. As shown, in order to make the transformation from worker to ruler, this action changes the unit's service-level objectives (SLO) – removes the old ones and adds new ones. In addition, this action re-specifies the unit in accordance with the new SLO and the new goals appropriate for a ruler. Note that some of the statements like the add statements are not complete due to space limitations.

```
ACTION transformToRuler { ….
   DOES {
      call  IMPL saveAESPEC;
      call  ASIP.FUNCTIONS.sendRulerSpecRequest;
      call  ASIP.FUNCTIONS.receiveRulerSpecification;
//remove the old spec structures
      remove  AESLO { };
      remove  AESELF_MANAGEMENT { };
            ….
//produce the new spec structures based on the received spec
      add  AESLO  {….};
      add  AESELF_MANAGEMENT { SELF_HEALING {….}  };
         ….
      call  IMPL doRulerTransformation
   }
   ONERR_DOES  {  call  IMPL restoreAESPEC  }
   TRIGGERS  {  EVENTS.transformedToRuler  }
   ONERR_TRIGGERS   {
      IF  METRICS.antennaAvailability.VALUE > 80  THEN
         EVENTS.canBeMessenger
      END  ELSE
         EVENTS.canBeShield
      END
   }
}
```

As specified, the self-transformation policy is very restrictive and cannot handle cases when the worker is *uncertain* about the ongoing transformation. To handle this uncertainty we apply the MAY Mechanism and specify the fluents triggering transformation with the MAY modifier. The new SELF_TRANSFORMATION policy specification is the following. Note that this is a partial specification emphasizing the applied MAY Mechanism.

```
 AESELF_MANAGEMENT {
   OTHER_POLICIES {
      SELF_TRANSFORMATION  {
         FLUENT unableToExplore  { …. }
         MAY FLUENT inTransformToRuler  { …. }
         MAY FLUENT inTransformToMessenger { …. }
         MAY FLUENT inTransformToShield { … }
         MAY FLUENT inSelfDestruction { ….}
         MAPPING   {
            CONDITIONS   { unableToExplore }
            DO_ACTIONS   { ACTIONS.checkTransformation }  }
         MAPPING   {
            CONDITIONS  { inTransformToRuler }
            DO_ACTIONS   { MAY ACTIONS.transformToRuler }  }
         MAPPING   {
            CONDITIONS   { inTransformToMessenger }
            DO_ACTIONS   { MAY ACTIONS.transformToMessenger }  }
         MAPPING   {
            CONDITIONS   {  inTransformToShield }
            DO_ACTIONS   { MAY ACTIONS.transformToShield }  }
         MAPPING   {
            CONDITIONS   {  inSelfDestruction }
            DO_ACTIONS   { MAY ACTIONS.selfDestroy }  }
      }
   }
}
```

With the new specification, the worker will attempt a transformation when the conditions require so, but will not continue trying to transform if the conditions have changed meanwhile and the first transformation operation is not successful. Let us assume that the worker's instrument has stopped functioning for a while due to a power disruption and the worker starts a "transformation to a ruler" operation. Thus, the MAY inTransformToRuler fluent gets initiated and the MAY transformToRuler action is started. Because the fluent is specified as a *MAY fluent* it "dies" silently once the transformToRuler action is performed (recall the new operational evaluation of the MAY fluents – see Section IV.A). Therefore, the fluent will not start the transformToRuler action again even the latter has not succeeded with the transformation process.

Moreover, because the transformToRuler action is called as a *MAY action* its operational evaluation excludes the evaluation of the otherwise specified *ONERR_TRIGGERS {….}* clause (see Section IV.B). Thus, no erroneous events such as canBeMessenger or canBeShield will be fired and the transformation process will not be propagated to other fluents (e.g., inTransformToMessenger or inTransformToShield). Also, because the operational evaluation of a *MAY action* does not exclude the evaluation of the *ONERR_DOES {….}* clause, the latter will be evaluated and the restoreAESPEC action will be called to "wipe out" the leftovers from the unsuccessful transformation. Therefore, once the worker is restored from the unsuccessful transformation, in order to start a new transformation again, it shall check its instrument first. If the instrument is properly functioning (the power is on and the instrument is working) the worker will not initiate a transformation. Instead, it will continue operating as a worker.

VI. Conclusion and Future Work

In this paper, we have presented an approach to handling uncertainty in self-adaptive systems developed with ASSL. We have introduced a new ASSL mechanism (termed *MAY Mechanism*) that helps developers specify special *loose self-management policies* capable of *flexible self-managing behavior* allowing autonomic systems to be more agile and not so persistent in following the specified behavior. Such *loose policies* introduce points of flexibility and nondeterministic choice in their behavior. The MAY Mechanism is an extension of ASSL emphasizing a special *MAY specification modifier* for some of the ASSL sub-tiers such as self-management policies, actions and service-level objectives. ASSL constructs specified with the *MAY* modifier are evaluated according the operational semantics of the MAY mechanism.

In this paper, we have also presented a case study where we applied the MAY Mechanism to demonstrate its applicability to cases where the conditions determining the self-managing behavior might change over time and the use of loose policies is more appropriate. Unfortunately, it is far easier to demonstrate validity of our approach than to demonstrate conclusively its completeness. In part, this is because completeness is at heart a relative rather than an absolute concept. Therefore, more experiments and results are needed and it is our intention to finish the code-generation part of the MAY Mechanism and perform tests with the generated loose policies under simulated conditions.

Moreover, we plan to extend the MAY Mechanism over other ASSL constructs such as metrics. Metrics defined with the *MAY* modifier shall be tolerant against threshold-class violations. Threshold classes specify a range of observable values for the ASSL metrics. The ASSL metrics are often used to sense the system's operational environment and thus, are very sensitive to changes.

ACKNOWLEDGMENT

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

[1] M. Salehie and L. Tahvildari, Self-adaptive software: Landscape and research challenges, *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol.4(2), pp.1-42, May 2009.

[2] J. Whittle, P. Sawyer , N. Bencomo and B. H. C. Cheng, A Language for Self-Adaptive System Requirements, In *Proceedings of the 2008 International Workshop on Service-Oriented Computing Consequences for Engineering Requirements*, 2008, pp.24-29.

[3] E. I. Vassev, *Towards a Framework for Specification and Code Generation of Autonomic Systems*, Ph.D. Thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, November 2008.

[4] J. O. Kephart, D. M. Chess, The vision of autonomic computing, *IEEE Computer*, vol. 36 (1), pp. 41-50, 2003.

[5] P. Horn, *Autonomic computing: IBM's perspective on the state of information technology*, Tech. rep., IBM T. J. Watson Laboratory, October 2001.

[6] E. Vassev, M. Hinchey, and J. Paquet, Towards an ASSL Specification Model for NASA Swarm-Based Exploration Missions, In *Proceedings of 23rd Annual ACM Symposium on Applied Computing (SAC2008) - AC Track*, ACM, 2008, pp.1652–1657.

[7] E. Vassev, M. Hinchey, Modeling the Image-processing Behavior of the NASA Voyager Mission with ASSL. In *Proceedings of the Third IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'09)*. IEEE Computer Society Press, 2009, pp. 246–253.

[8] S. A. Mokhov, E. Vassev, Autonomic Specification of Self-protection for Distributed MARF with ASSL. In *Proceedings of C3S2E'09*, ACM, 2009, pp. 175–183.

[9] IBM Corporation, *Autonomic Computing Policy Language*, Tutorial, IBM Tivoli, November 2005

[10] M. Hinchey, J. Rash, and C. Rouff, Requirements to Design to Code: Towards a Fully Formal Approach to Automatic Code Generation, *Technical Report TM-2005-212774*, NASA Goddard Space Flight Center, Greenbelt.

[11] J. Banatre, P. Fradet and Y. Radenac, Programming self-organizing systems with the higher-order chemical language, *International Journal of Unconventional Computing*, vol. 3(3), pp. 161-177, 2007.

[12] W. Truszkowski, M. Hinchey, J. Rash and C. Rouff, NASA's swarm missions: The challenge of building autonomous software, *IT Professional*, vol. 6(5), pp. 47-52, 2004.

[13] E. Vassev and M. Hinchey, ASSL Specification of Emergent Self-Adapting for NASA Swarm-Based Exploration Missions, In *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW 2008)*, IEEE Computer Society Press, 2008, pp. 13-18.