The ASSL Approach to Formal Specification of Self-managing Systems

Emil Vassev[1] and Mike Hinchey[2(✉)]

[1] Concordia University, Montreal, Canada
emil@vassev.com
[2] Lero-The Irish Software Research Centre, University of Limerick, Limerick, Ireland
mike.hinchey@lero.ie

Abstract. ASSL (Autonomic System Specification Language) is a framework dedicated to the development of self-managing systems whereby developers are helped with problem formation, system design, system analysis and evaluation, and system implementation. The bottom line is a special multi-tier approach to specification exposing a rich set of constructs allowing a system to be modeled by emphasizing different key aspects, but centering the model around special self-management policies. This article presents in detail the aforementioned mechanism together with the underlying semantics. As a case study, we also present ASSL specifications of self-managing behavior of prospective autonomous NASA space exploration missions.

Keywords: Autonomic computing · Formal specification

1 Introduction

Complexity is widely recognized as one of the biggest challenges information technology faces today. To respond to this threat, many initiatives, such as Autonomic Computing (AC) [1–3], have been started to deal with complexity in contemporary computerized systems. AC is a rapidly growing field that promises a new approach to developing large-scale complex systems capable of self-management. The phrase "autonomic computing" came into the popular consciousness at the AGENDA 2001 conference where Paul Horn from IBM presented the new computing paradigm by likening computer systems to the human Autonomic Nervous System [1]. The idea behind this is that software systems must manage themselves, as the human body does, or they risk being crushed under their own complexity. Many major software vendors, such as IBM, HP, Sun, and Microsoft have started research programs to create self-managing computer systems. However, their main research efforts are mainly to make individual components of particular systems more self-managing rather than providing a complete solution to the problem of autonomic system development. As a result, ten years later after the AC initial announcement, there is still much to be done in making the transition to "autonomic culture" [4] and we still need programming techniques and technologies that emphasize the AC paradigm and provide us with programming concepts for implementing autonomic systems.

T. Margaria et al. (Eds.): Steffen Festschrift, LNCS 11200, pp. 268–296, 2019.
https://doi.org/10.1007/978-3-030-22348-9_17

This article presents the formalism of the Autonomic System Specification Language (ASSL) [5, 6], a dedicated to AC formal tool that emerges as a formal approach to developing autonomic systems. Providing both a formal notation and tools that support modeling and specification, validation and code generation of autonomic systems, ASSL has been successfully used in a variety of projects targeting functional prototypes of autonomous NASA space exploration missions [7, 8], autonomic pattern-recognition systems [9], home-automation sensor networks [10], etc. Note that a good understanding of the ASSL formalism and mastering the same were of major importance for the success of these endeavors. This paper gives an overview of the ASSL operational semantics and through formal semantics definitions it presents the operational behavior of some of the ASSL specifications for space exploration missions [7, 8]. This approach helps ASSL developers conceive an explicit understanding of the ASSL formalism.

The rest of this article is organized as follows. In Sect. 2, we discuss different formalisms for autonomic systems. Section 3 describes the ASSL specification model. Section 4 presents the ASSL operational semantics, which is used in Sect. 5 to enlighten ASSL specifications of NASA space exploration missions. Section 5 also presents some test results and Sect. 6 gives a brief overview of the ASSL's formal verification mechanisms. Finally, Sect. 7 concludes the article with summary remarks.

2 Formalism for Autonomic Systems

Conceptually, any formalism aims to assist the development of computer systems by providing formal notations that can be used to specify desirable system concepts (e.g. functionality). Usually, formal notations help developers precisely describe with the logical underpinning of mathematics features of the system under consideration at a higher level of abstraction than the one provided by implementation. However, a requirement is that developers should be able to move in a logical way from a formal specification of a system to implementation.

2.1 Formal Approaches to AC

Autonomic systems are special computer systems that emphasize *self-management* through *context awareness* and *self-awareness* [1–4]. Therefore, an AC formalism should not only provide a means of description of system behavior but also should tackle the vital for autonomic systems self-management and awareness issues. Moreover, an AC formalism should provide a well-defined semantics that makes the AC specifications a base from which developers may design, implement, and verify autonomic systems.

Formalisms dedicated to AC have been targeted by a variety of industrial and university projects. IBM Research developed a framework called Policy Management for Autonomic Computing (PMAC) [14, 15]. The PMAC formalism emphasizes the specification of self-management policies encompassing the scope under which those policies are applicable. A PMAC policy specification includes: (1) conditions to which a policy is in conformance (or not); (2) a set of resulting actions; (3) goals; and (4) decisions that need to be taken.

The so-called Specification and Description Language (SDL) is an object-oriented, formal language defined by the International Telecommunications Union – Telecommunications Standardization Sector (ITU-T) [16]. SDL is dedicated to real-time systems, distributed systems, and generic event-driven systems. The basic theoretical model of an SDL system consists of a set of extended finite state machines, running in parallel and communicating via discrete signals, thus making SDL suitable for the specification of self-management behavior.

Cheng et al. talk in [17] about a specification language for self-adaptation based on the ontology from system administration tasks and built over the underlying formalism of *utility theory* [18]. In this formalism, special self-adaptation actions are described as architectural operators, which are provided by the architectural style of the target system. A script of actions corresponds to a sequence of architectural operators. This sequence forms the so-called *adaptation tactic* defined in three parts: (1) the conditions of applicability; (2) a sequence of actions; and (3) a set of intended effects after execution. The definition of a tactic is similar to the "design by contract" interface definition [19].

Another formalism for Autonomic Systems (Ass) is provided by the *chemical programming* approach (represented by the Gamma Formalism [20]) which uses the chemical reaction metaphor to express the coordination of computations. The Gama Formalism describes computation in terms of chemical reactions (described as rules) in solutions (described as multisets of elements). When applied to AS specifications, 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.

In [21], Andrei and Kirchner present a biologically inspired formalism for AC called Higher-Order Graph Calculus (HOGC). This approach extends the Gama Formalism with high-level features by considering a graph structure for the molecules and permitting control on computations to combine rule applications. HOGC borrows various concepts from graph theory, in particular from *graph transformations* [22], and use representations for graphs that have been already intensively formalized.

2.2 The ASSL Formalism

ASSL is a *declarative* specification language for autonomic systems with well-defined semantics. It implements modern programming language concepts and constructs like inheritance, modularity, type system, and high abstract expressiveness. Being a formal language designed explicitly for specifying autonomic systems (ASs) ASSL copes well with many of the AS aspects [1–4]. Moreover, specifications written in ASSL present a view of the system under consideration, where specification and design are intertwined. Conceptually, ASSL is defined through formalization tiers (see Sect. 3). Over these tiers, ASSL provides a *multi-tier specification model* that is designed to be scalable and exposes a judicious selection and configuration of infrastructure elements and mechanisms needed by an AS. In order to determine the level of ASSL formalism, we investigated in the vast field of formal specification languages. Srivas and Miller in [11] refer to *constructive* versus *descriptive* style of specification (also known as

model-oriented versus *property-oriented*). The constructive or model-oriented style is typically associated with the use of definitions, whereas the descriptive or property-oriented style is generally associated with the use of axioms [12]. ASSL benefits from both styles, by using a property-oriented axiomatization as a top-level specification style and introducing a suitable number of specification layers with increasingly detailed model-oriented descriptions. As a formal language, ASSL defines a neutral, implementation-independent representation for ASs. Similar to many formal notations, ASSL enriches the underlying logic with modern programming concepts and constructs thereby increasing the expressiveness of the formal language while retaining the precise semantics of the underlying logic. For example, the ASSL formalism for self-management policies (see Sect. 3) is based on *event calculus* [13], whose formalism is enriched to fit in the ASSL mechanism for specifying self-management policies [5, 6].

To the best of our knowledge, the ASSL formalism is currently the only complete solution to the problem of AS specification. Although other solutions do exist, they emphasize individual AC aspects (e.g. self-management policies), which is far from what ASSL is proposing with its reach multi-tier specification model. Moreover, the ASSL framework together with the powerful formalism provides mature tools that allow ASSL specifications to be edited and formally validated. Finally, an operational Java application may be generated from any valid ASSL specification.

3 ASSL Specification Model

The ASSL formal notation is based on a specification model exposed over hierarchically organized *formalization tiers* [5, 6]. This specification model provides both infrastructure elements and mechanisms needed by an AS. ASSL defines ASs with special *self-managing policies*, *interaction protocols* (IPs), and *autonomic elements* (AEs), where the ASSL tiers and their sub-tiers describe different aspects of the AS under consideration.

Table 1 presents the ASSL specification model. As shown, it decomposes an AS in two directions - (1) into levels of functional abstraction; and (2) into functionally related sub-tiers. The first decomposition presents the system from three different perspectives (three major tiers) [5, 6]:

(1) a *general and global AS perspective*, where we define the general *system rules* (providing AC behavior), *architecture*, and global *actions*, *events*, and *metrics* applied in these rules;
(2) an *interaction protocol perspective*, where we define the means of communication between AEs within an AS;
(3) a *unit-level perspective*, where we define interacting sets of individual computing elements (AEs) with their own AC behavior rules, actions, events, metrics, etc.

The second decomposition presents the major tiers AS, ASIP, and AE as composed of functionally related sub-tiers, where new AS properties emerge at each sub-tier. This allows for different approaches to AS specification. For example, we may start with a global perspective of the system by specifying the AS service-level objectives and self-management policies and by digging down to find the needed metrics at the very detail

Table 1. 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 Behavior Models	
	AE Outcomes	
	AE Actions	
	AE Events	
	AE Metrics	

level of AE sub-tiers. Alternatively, we may start working at the detail level of AE sub-tiers and build our AS bottom-up. Finally, we can work on both abstract and detail level sides by constantly synchronizing their specification.

3.1 ASSL Tiers

The AS Tier specifies an AS in terms of *service-level objectives* (AS SLOs), *self-management policies*, *architecture topology*, *actions*, *events*, and *metrics* (see Table 1). The AS SLOs are a high-level form of behavioral specification that help developers establish system objectives such as performance. The self-management policies could be any of (but not restricted to) the four so-called self-CHOP policies defined by the AC IBM blueprint [2]: *self-configuring*, *self-healing*, *self-optimizing*, and *self-protecting*. These policies are driven by *events* and trigger the execution of *actions* driving an AS in critical situations. The metrics constitute a set of parameters and observables controllable by an AS. At the ASIP Tier, the ASSL framework helps developers specify an AS-level interaction protocol as a public communication interface, expressed with special *communication channels*, *communication functions*, and *communication messages*. At the AE Tier, the ASSL formal model exposes specification constructs for the specification of the system's AEs. Note that AEs are considered to be analogous to software agents able to manage their own behavior and their relationships with other AEs.

Note that ASSL targets only the AC features of a system and helps developers clearly distinguish the *AC features* from the *system-service features*. This is possible, because with ASSL we model and generate special *AC wrappers* in the form of ASs that embed the components of non-AC systems [5, 6]. The latter are considered as *managed elements* controlled by the AS in question. Conceptually, a managed element can be any software or hardware system (or sub-system) providing services. Managed elements are specified per AE (see Table 1) where the emphasis is on the *interface* needed to control a managed element. It is important also to mention that the ASSL tiers and sub-tiers are intended to specify different aspects of an AS, but it is not necessary to employ all of them in order to model such a system. For a simple AS we need to specify (1) the AEs providing self-managing behavior intended to control the managed elements associated with an AE; and (2) the communication interface. Here, self-management policies must be specified to provide such self-managing behavior at the level of AS (the AS Tier) and at the level of AE (AE Tier). The following sub-sections briefly present some of the ASSL sub-tiers.

Self-management Policies. The self-management behavior of an ASSL-developed AS is specified with the self-management policies. These policies are specified with special ASSL constructs termed *fluents* and *mappings* [5, 6]. 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. 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
```

ASSL Events. ASSL aims at event-driven autonomic behavior. Hence, to specify self-management policies, we need to specify appropriate events (see Sect. 3.1). Here, we rely on the reach set of event types exposed by ASSL [5, 6]. To specify ASSL events, one may use logical expressions over SLOs, or may relate events with metrics (see the ASSL code below), other events, actions, time, and messages. Moreover, ASSL allows for the specification of special conditions that must be stated before an event is prompted.

```
EVENT newAsteroidDetected {
    ACTIVATION {
        CHANGED { AS.METRICS.numberOfAsteroids }
    }
}
```

ASSL Metrics. For an AS, one of the most important success factors is the ability to sense the environment and react to sensed events. Together with the rich set of events, ASSL imposes metrics as a means of determining dynamic information about external and internal points of interest. Although four different types of metric are allowed [5, 6], the most important are the so-called *resource metrics* because those are intended to measure special *managed element* quantities. The following ASSL code demonstrates the ASSL specification of a resource metric (**noObstacle**) related to a managed element (**OBSTACLE_SENSOR**).

```
METRIC noObstacle {
    METRIC_TYPE { RESOURCE }
    METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.OBSTACLE_SENSOR.isClean }
    THRESHOLD_CLASS { Boolean [ true ] }
}
```

Managed Elements. An AE typically controls *managed elements*. In an ASSL-developed AS, a managed element is specified with a set of special interface functions intended to provide control functionality over that managed element. Note that ASSL can specify and generate interfaces controlling a managed element (generated as a stub), but not the real implementation of these interfaces. This is just fine for prototyping, however when deploying an AS prototype the generated interfaces must be manually programmed to deal with the controlled system (or sub-system).

```
MANAGED_ELEMENT meReceptor {
  INTERFACE_FUNCTION reset {}
  INTERFACE_FUNCTION getRadiationLevel {
    PARAMETERS { DECIMAL xCoord; DECIMAL yCoord; DECIMAL zCoord }
    RETURNS { DECIMAL }
    TRIGGERS { AS.EVENTS.newRadiationLevel }
    ONERR_TRIGGERS { AS.EVENTS.cannotGetRadiationLevel }
  }
}
```

Interaction Protocols. ASSL *interaction protocols* provide a means of communication interface expressed with *messages* that can be exchanged among AEs, *communication channels* and *communication functions*. Thus, by specifying an ASSL interaction protocol we develop an embedded messaging system needed to connect the AEs of an AS. In a basic communication process ongoing in such a system, an AE

relies on a communication function to receive a message over an *incoming* communication channel, changes its internal state and sends some new messages over an *outgoing* channel [5, 6].

```
ASIP {
  MESSAGES {  MESSAGE msgHello { SENDER { AES.ae2 }  RECEIVER { AES.ae1 } } }
  CHANNELS  {
     CHANNEL chnlIO { ACCETS { ANY }  ACCESS { SEQUENTIAL }  DIRECTION { INOUT } }
  }
  FUNCTIONS {
     FUNCTION sendHello {
        PARAMETERS { BOOLEAN hasSpeed; BOOLEAN hasDirection }
        DOES { MESSAGES.msgHello >> CHANNELS.chnlIO }
     }
  }
}
```

4 ASSL Notation and Semantics

ASSL is a declarative specification language for ASs with well-defined semantics [5, 6]. The language provides a powerful formal notation that enriches the underlying logic with modern programming language concepts and constructs such as *inheritance*, *modularity*, *type system*, and *abstract expressiveness*. As a formal language, ASSL defines a *neutral* (i.e., implementation-independent) representation for ASs described as a set of interacting AEs. The following is a generic meta-grammar in Extended Backus-Naur Form (BNF) [23] presenting the syntax rules for specifying ASSL tiers. Note that this meta-grammar is an abstraction of the ASSL grammar, which cannot be presented here due to the complex structure of the ASSL specification model (see Sect. 3), where each tier has its own syntax and semantic rules. The interested reader is advised to refer to [5] for the complete ASSL grammar expressed in BNF and for the semantics of the language.

GroupTier → ***FINAL****? ASSLGroupTierId* ***{*** *Tier+* ***}***
Tier → ***FINAL****? ASSLTierId TierName? {* *Data* TierClause+* ***}***
TierClause → ***FINAL****? ASSLClauseId ClauseName? {* *Data** ***}***
Data → *TypeDecl** | *VarDecl** | *CllctnDecl** | *Statement**
TypeDecl → *CustTypeIdentifier*
VarDecl → *Type VarIdentifier*
CllcntDecl → *Type CustCllctnIdentifier*
Type → *CustType* | *PredefType*
Statement → *Assign-Stmnt* | Loop | *If-Then-Else* | *Cllctn-Stmnt*
Loop → *Foreach-Stmnt* | *DoWhile-Stmnt* | *WhileDo-Stmnt*

As shown in the grammar above, an ASSL tier is syntactically specified with an ASSL *tier identifier*, an optional *name* and a *content block* bordered by curly braces.

Moreover, we distinguish two syntactical types of tier: *single tiers* (*Tier*) and *group tiers* (*GroupTier*), where the latter comprise a set of single tiers. Each single tier has an optional *name* (*TierName*) and comprises a set of special *tier clauses* (*TierClause*) and optional *data* (*Data*). The latter is a set of *data declarations* and *statements*. Data declarations could be: (1) *type declarations*; (2) *variable declarations*; and (3) *collection declarations*. Statements could be: (1) *loop statements*; (2) *assignment statements*; (3) *if-then-else statements*; and (4) *collection statements*. Statements can comprise *Boolean* and *numeric expressions*. In addition, although not shown in the grammar above, note that identifiers participating in ASSL expressions are either simple, consisting of a single identifier, or qualified, consisting of a sequence of identifiers separated by "." tokens.

4.1 ASSL Operational Semantics

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 [5]. 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:

$$\sigma \xrightarrow{Op(x_1,x_2,\ldots,x_n)} \sigma'$$

where the operation $Op(x_1, x_2, \ldots, x_n)$ is an abstraction of a *transition operation* performed by the framework which potentially takes n arguments. All the arguments are evaluated to their expression value first, and then the operation is performed. Here, *Op* is a transition operation of type O^{trans} (see the set definition below).

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

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. Thus, we write $\rho \vdash_\sigma$ to mean that ρ is evaluated in context σ and $\rho \vdash_\sigma e \rightarrow e'$ 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 where they are presented by their *TierName*. 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.

The following subsections present two algorithms implemented by the ASSL framework for operational evaluation of ASSL actions and self-management policies.

4.2 Operational Evaluation of ASSL Actions

From operational semantics perspective, the AS/AE Action tier is the most important and the most complex ASSL tier. The following is a partial EBNF grammar presenting syntactically that tier.

Action-Decl → ***ACTION IMPL****? Action-Name* ***{*** *Action-Decl-Seqnce* ***}***
Action-Decl-Seqnce → *Params-Decl? Returns-Decl? Guards-Decl? Ensures-Decl? Var-Decl-Seqnce? Does-Decl OnErr-Does-Decl? Trigs-Decl? OnErr-Trigs-Decl?*

ASSL actions comprise the tier clauses: *PARAMETERS {…}*, *RETURNS {…}*, *GUARDS {…}*, *ENSURES {…}*, *DOES {…}*, *ONERR_DOES {…}*, *TRIGGERS {…}*, and *ONERR_TRIGGERS {…}*. Note that only the *DOES {…}* clause is mandatory. The operational evaluation of an ASSL action follows the following algorithm:

I. Map the arguments, if any, from the action call to the parameters (*PARAMETERS {…}* clause).
II. Process the action guards, if any (*GUARDS {…}* clause):
    - If the guards are held then perform the action.
    - Otherwise, deny the action.
III. Evaluate the variable declarations, if any.
IV. 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.
V. 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.
VI. 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.

4.3 Operational Evaluation of ASSL Policies

ASSL specifies policies with *fluents* and *mappings* (see Sect. 3.1). Whereas the former are considered as specific policy conditions, the latter map these conditions to appropriate actions. A partial presentation of the fluent grammar is the following:

Fluent-Decl → ***FLUENT*** *Fluent-Name* ***{*** *Fluent-Inner-Decl* ***}***
Fluent-Name → ***Bool-Identifier***
Fluent-Inner-Decl → *Initiates-Sqnce Terminates-Sqnce*
Fluent-Inner-Decl → *Initiates-Sqnce*
Initiates-Sqnce → ***INITIATED_BY*** ***{*** *Event-Names* ***}***
Terminates-Sqnce → ***TERMINATED_BY*** ***{*** *Event-Names* ***}***

Map-Decl → ***MAPPING*** ***{*** *Mapping-Inner-Decl* ***}***
Mapping-Inner-Decl → *Condition-Sqnce Action-Sqnce*
Condition-Sqnce → ***CONDITIONS*** *{ Fluent-Names* ***}***
Action-Sqnce → ***DO_ACTIONS*** ***{*** *Action-Calls ; Action-Calls-Forall* ***}***

An ASSL policy is evaluated based on its fluents. The operational evaluation of a fluent follows the following algorithm:

If an event has occurred in the system then:

I. 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 {....}* clause and if the stated conditions are held then evaluate the *DO_ACTIONS {....}* clause to perform the actions listed there.

II. Process the *TERMINATED_BY {...}* clause to check if that event can terminate the previously-initiated policy fluent f and if so, terminate it.

The semantic rules 1 through to 2 present the operational semantics that cope with the algorithm stated above. In these rules, each premise is a system transition operation (see Sect. 4.1) such as $Event(ev)$, $FluentIn(f, ev)$, $FluentOut(f, ev)$, and $ActionMap(f, a)$.

$$(1)\quad \frac{\sigma \xrightarrow{Event(ev)} \sigma'}{f \vdash_{\sigma'} \textbf{\textit{INITIATED\_BY}}\{ev_1, \ldots, ev_n\} \xrightarrow{FluentIn(f,ev)} \sigma''} ev \in \{ev_1, \ldots, ev_n\}$$

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

$$(3)\quad \frac{\sigma \xrightarrow{FluentIn(f,ev)} \sigma'}{map \vdash_{\sigma'} \textbf{\textit{CONDITIONS}}\{f_1, \ldots, f_n\} \xrightarrow{ActionMap(f,a)} \sigma''} f \in \{f_1, \ldots, f_n\}$$

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

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

5 Case Study - ASSL Specifications for NASA ANTS

5.1 Nasa Ants

The Autonomous Nano-Technology Swarm (ANTS) concept sub-mission PAM (Prospecting Asteroids Mission) is a novel approach to asteroid belt resource exploration that provides for extremely high autonomy, minimal communication requirements with Earth, and a set of very small explorers with a few consumables [24]. These explorers forming the swarm are pico-class, low-power, and low-weight spacecraft, yet capable of operating as fully autonomous and adaptable units. The units in a swarm are able to interact with each other, thus helping them to self-organize based on the emergent behavior of the simple interactions. Figure 1 depicts the ANTS concept mission. A transport spacecraft launched from Earth to carries a laboratory that assembles tiny spacecraft. Once it reaches a point in space, termed L1 (the Earth-Moon Lagrangian point), where gravitational forces on small bodies are balanced, the transport releases the assembled swarm, which will head for the asteroid belt. Each spacecraft is equipped with a solar sail for power; thus it relies primarily on power from the sun, using only tiny thrusters to navigate independently. Moreover, each spacecraft also has onboard computation, artificial intelligence, and heuristics systems for control at the individual and team levels.

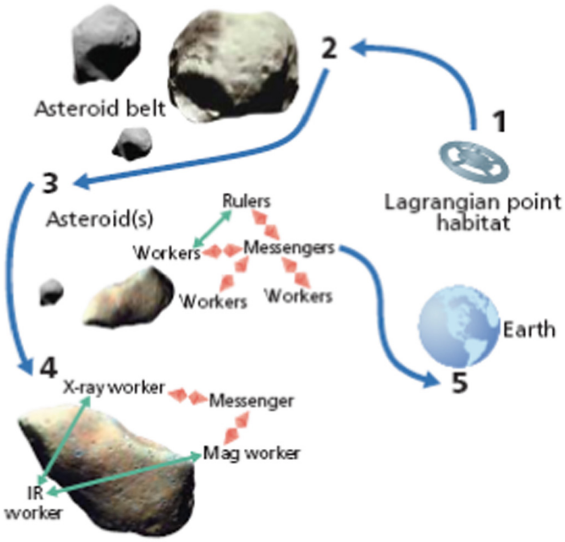

Fig. 1. ANTS mission concept [24]

As Fig. 1 shows, there are three classes of spacecraft—*rulers*, *messengers* and *workers*. Sub-swarms are formed to explore particular asteroids in an ant colony analogy. 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, due to a long distance or a barrier.

5.2 Specifying ANTS with ASSL

In our endeavor to specify ANTS with ASSL, we emphasized modeling ANTS self-management policies such as self-configuring [8], self-healing [25], self-scheduling [26] and emergent self-adapting [27]. In addition, we proposed a specification model for the ANTS safety requirements [8]. To specify the ANTS safety requirements, we used the AS/AE SLO tier specification structures, and to specify the self-management policies we used ASSL tiers and clauses as following:

- self-management policy tiers to specify the self-management policies under consideration through a finite set of *fluents* and *mappings*.
- *actions*—a finite set of actions that can be undertaken by ANTS in response to certain conditions, and according to the self-management policies.
- *events*—a set of events that initiate fluents and are prompted by the actions according to the policies.
- *metrics*—a set of metrics needed by the events and actions.

The following subsections present some ASSL specification models together with a formal presentation of their operational behavior. Note that the specifications presented here are partial, because we omitted some of the aspects that were specified due to space limitations. The operational behavior of the presented specifications is presented in a Structural Operational Semantics style [28]. Thus, we define semantics definitions formed by *inference rules*. An inference rule is presented as a set of premises deducting a conclusion, possibly under control of some condition.

5.3 Self-configuring

Figure 2 presents a partial specification of the self-configuring behavior in ANTS when a new asteroid has been detected [8]. This policy specifies the "on the fly" team configuration of ANTS spacecraft, to explore asteroids. The key features of the proposed model are:

- a *numberOfAsteroids* metric that counts the number of detected asteroids;
- an *inANTSReconfigurationForNewAsteroid* fluent that takes place when the swarm detects a new asteroid;

- a *reconfigureANTS* action that performs the ANTS reconfiguration;
- a *newAsteroidDetected* event that initiates the fluent above and is prompted by the *numberOfAsteroids* metric when the latter changes its value.

```
AS ANTS {
  ASSELF_MANAGEMENT {
    SELF_CONFIGURING {
      FLUENT inANTSReconfigurationForNewAsteroid {
        INITIATED_BY { EVENTS.newAsteroidDetected }
        TERMINATED_BY {EVENTS.reconfigurationForNewAsteroidDone }
      }
      MAPPING { // force ANTS reconfiguration
        CONDITIONS { inANTSReconfigurationForNewAsteroid }
        DO_ACTIONS { ACTIONS.reconfigureANTS }
      }
    }
  } // ASSELF_MANAGEMENT
  ACTIONS {
    ACTION IMPL reconfigurationForNewAsteroid { TRIGGERS { EVENTS.reconfigurationForNewAsteroidDone }
}
    ACTION reconfigureANTS {
      GUARDS { ASSELF_MANAGEMENT.SELF_CONFIGURING.inANTSReconfigurationForNewAsteroid }
      ENSURES { EVENTS.reconfigurationForNewAsteroidDone }
      DOES { call IMPL ACTIONS.reconfigurationForNewAsteroid }
      ONERR_TRIGGERS { EVENTS.reconfigurationForNewAsteroidDenied }
    }
  } // ACTIONS
  EVENTS {
    EVENT newAsteroidDetected {  ACTIVATION { CHANGED { AS.METRICS.numberOfAsteroids } } }
    EVENT reconfigurationForNewAsteroidDone { }
    EVENT reconfigurationForNewAsteroidDenied { }
  }
  METRICS {
    METRIC numberOfAsteroids {
      METRIC_TYPE { RESOURCE }
      DESCRIPTION { "the number of detected asteroids during the ANTS lifecycle" }
      THRESHOLD_CLASS { DECIMAL [0 ~ ) } //open range: from 0 to ....
    }
  }
} // AS ANTS
```

Fig. 2. ASSL specification: self-configuring

Operational Behavior. We consider two main states in this specification model—ANTS "in" and ANTS "not in" the *inANTSReconfigurationForNewAsteroid* fluent; i.e. ANTS performing self-configuring and ANTS not performing self-configuring. While operating, ANTS workers can discover a new asteroid. This increases the number of detected asteroids; i.e. the metric *numberOfAsteroids* changes its value, this causing the framework to perform the *Metric*(numberOfAsteroids) *transition operation*. The latter consecutively prompts the *newAsteroidDetected* event, which is attached to that metric (the event is prompted when the metric has changed its value). Rule 5 presents the operational evaluation of the *newAsteroidDetected* event:

$$(5)\quad \frac{\langle ANTS\rangle \xrightarrow{Metric(\text{numberOfAsteroids})} \langle ANTS\rangle'}{ev \vdash_{\sigma} \textbf{\textit{CHANGED}}\ \{\text{numberOfAsteroids}\} \xrightarrow{Event(\text{newAsteroidDetected})} \sigma'}$$

where *ev* is the tier environment exposed by that event and the transition operation *Event*(newAsteroidDetected) denotes that the event has been prompted. Subsequently, that transition operation initiates the *inANTSReconfigurationForNewAsteroid* fluent. Inference rules 6 through 9 enforce a definite strategy for evaluating that fluent's clauses in their host tier context σ and in the context π of the ***SELF_CONFIGURING*** policy. These semantic rules follow the algorithm presented in Sect. 4.3. Thus,

$$(6)\quad \frac{\langle ANTS\rangle \xrightarrow{Event(\text{newAsteroidDetected})} \langle ANTS\rangle'}{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{newAsteroidDetected}\} \xrightarrow{FluentIn(f,\text{newAsteroidDetected})} \sigma',\pi'}$$

$$(7)\quad \frac{ANTS \xrightarrow{Event(\text{reconfigurationForNewAsteroidDone})} ANTS'\ \ \pi \vdash_{\sigma} \text{inANTSReconfigurationForNewAsteroid} \rightarrow \textbf{\textit{true}}}{f \vdash_{\sigma,\pi} \textbf{\textit{TERMINATED\_BY}}\{\text{reconfigurationForNewAsteroidDone}\} \xrightarrow{FluentOut(f,\text{reconfigurationForNewAsteroidDone})} \sigma',\pi'}$$

$$(8)\quad \frac{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{newAsteroidDetected}\} \xrightarrow{FluentIn(f,\text{newAsteroidDetected})} \sigma',\pi'}{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inANTSReconfigurationForNewAsteroid}\} \xrightarrow{ActionMap(f,\text{reconfigureANTS})} \sigma'',\pi''}$$

$$(9)\quad \frac{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inANTSReconfigurationForNewAsteroid}\} \xrightarrow{ActionMap(f,\text{reconfigureANTS})} \sigma'',\pi''}{map \vdash_{\sigma'',\pi''} \textbf{\textit{DO\_ACTIONS}}\{\text{reconfigureANTS}\} \xrightarrow{Action(\text{reconfigureANTS})} \sigma''',\pi'''}$$

where *f* is the tier environment exposed by the *inANTSReconfigurationForNewAsteroid* fluent and *map* is the tier environment exposed by the ***MAPPING {…}*** clause (see Fig. 2). Here, *FluentIn*(*f*, newAsteroidDetected) is a transition operation denoting that the ***SELF_CONFIGURING*** policy has entered that fluent (initiated by the *newAsteroidDetected* event) and *FluentOut*(*f*, reconfigurationForNew *AsteroidDone*) is a transition operation denoting that the ***SELF_CONFIGURING*** policy has exited the same fluent (terminated by the *reconfigurationForNew AsteroidDone* event) (see rules 6 and 7). In addition, *ActionMap*(*f*, reconfigureANTS) is a transition operation denoting that the ***SELF_CONFIGURING*** policy has mapped the *reconfigureANTS* action to that fluent.

Rules 10 through 17 present the operational evaluation of the *reconfigureANTS* action, thus following the algorithm presented in Sect. 4.2. This evaluation is triggered by the *Action*(reconfigureANTS) transition operation, which is performed by the framework when the *inANTSReconfigurationForNewAsteroid* fluent is mapped to the

reconfigureANTS action (see Rule 9). This causes the state transition $\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle'$. Thus, in the given *reconfigureANTS* action tier environment a defined in the tier context σ we evaluate the operational action clauses.

$$(10)\quad \frac{ANTS \xrightarrow{Action(\text{reconfigureANTS})} ANTS' \;\; a\vdash_\sigma \text{newAsteroidDetected} \;\rightarrow \textbf{\textit{true}}}{a\vdash_\sigma \textbf{\textit{GUARDS}}\{\text{newAsteroidDetected}\} \rightarrow \textbf{\textit{perform}}(\text{reconfigureANTS})}$$

$$(11)\quad \frac{\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle' \; a\vdash_\sigma \text{newAsteroidDetected} \rightarrow \textbf{\textit{false}}}{a\vdash_\sigma \textbf{\textit{GUARDS}}\{\text{newAsteroidDetected}\} \rightarrow \neg\textbf{\textit{perform}}(\text{reconfigureANTS})}$$

$$(12)\quad \frac{\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle'}{a\vdash_\sigma \textbf{\textit{DOES}}\{\textbf{\textit{CALL IMPL}}\,\text{ReconfigurationForNewAsteroid}\} \xrightarrow{Action(\text{ReconfigurationForNewAsteroid})} \sigma'}$$

$$(13)\quad \frac{\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle'}{a\vdash_\sigma \textbf{\textit{DOES}}\{\textbf{\textit{CALLIMPL}}\text{ReconfigurationForNewAsteroid}\} \xrightarrow{\neg Action(\text{ReconfigurationForNewAsteroid})} \sigma'[err]}$$

$$(14)\quad \frac{a\vdash_\sigma DOES\{\textbf{\textit{CALL IMPL}}\,\text{ReconfigurationForNewAsteroid}\} \xrightarrow{Action(\text{ReconfigurationForNewAsteroid})} \sigma' \quad \langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle'}{a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{reconfigurationForNewAsteroidDone}\} \xrightarrow{Event(\text{reconfigurationForNewAsteroidDone})} \sigma''}$$

$$(15)\quad \frac{\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle' \; a\vdash_\sigma \text{reconfigurationForNewAsteroidDone} \rightarrow \textbf{\textit{true}}}{a\vdash_\sigma \textbf{\textit{ENSURES}}\{\text{reconfigurationForNewAsteroidDone}\} \rightarrow \sigma}\; Err^a = \emptyset$$

$$(16)\quad \frac{\langle ANTS\rangle \xrightarrow{Action(\text{reconfigureANTS})} \langle ANTS\rangle' \; a\vdash_\sigma \text{reconfigurationForNewAsteroidDone} \rightarrow \textbf{\textit{false}}}{a\vdash_\sigma \textbf{\textit{ENSURES}}\{\text{reconfigurationForNewAsteroidDone}\} \rightarrow \sigma'[err]}\; Err^a = \emptyset$$

$$(17)\quad \frac{ANTS \xrightarrow{Action(\text{reconfigureANTS})} ANTS'}{a\vdash_\sigma \textbf{\textit{ONERR\_TRIGGERS}}\{\text{reconfigurationForNewAsteroidDenied}\} \xrightarrow{Event(\text{reconfigurationForNewAsteroidDenied})} \sigma'}\; Err^a \neq \emptyset$$

where a is the tier environment exposed by the *reconfigureANTS* action and Err^a is the finite set of errors produced by that action in a single performance of the $Action(\text{reconfigureANTS})$ transition operation. In addition, in rules 10 through 17 we use transition operations $Action(\ldots)$ and $Event(\ldots)$ to denote state transitions that occur during the evaluation of the action tier clauses. Moreover, we use the abstract function $perform(a)$ (see rules 10 and 11) to denote *continuation* of the *reconfigureANTS* action.

5.4 Self-healing

Figure 3 presents a partial specification of the self-healing policy for ANTS. In our approach, we assume that each worker sends, on a regular basis, heartbeat messages to the ruler [25]. The latter uses these messages to determine when a worker is not able to continue its operation, due to a crash or malfunction in its communication device or instrument. The specification snippet shows only fluents and mappings forming the

```
AE ANT_Worker {
  AESELF_MANAGEMENT {
    SELF_HEALING {
      FLUENT inCollision {
        INITIATED_BY { EVENTS.collisionHappen } TERMINATED_BY { EVENTS.instrumentChecked } }
      FLUENT inInstrumentBroken {
        INITIATED_BY { EVENTS.instrumentBroken } TERMINATED_BY { EVENTS.isMsgInstrumentBrokenSent
} }
      FLUENT inHeartbeatNotification {
        INITIATED_BY { EVENTS.timeToSendHeartbeatMsg } TERMINATED_BY { EVENTS.isMsgHeartbeatSent
} }
      MAPPING { // if collision then check if the instrument is still operational
        CONDITIONS { inCollision } DO_ACTIONS { ACTIONS.checkANTInstrument } }
      MAPPING { // if the instrument is broken then notify the group leader
        CONDITIONS { inInstrumentBroken } DO_ACTIONS { ACTIONS.notifyForBrokenInstrument } }
      MAPPING { // time to send a heartbeat message has come
        CONDITIONS { inHeartbeatNotification } DO_ACTIONS { ACTIONS.notifyForHeartbeat } }
    }
  }
  ....
  ACTIONS {
    ACTION IMPL checkInstrument { RETURNS { BOOLEAN } TRIGGERS { EVENTS.instrumentChecked } }
    ACTION checkANTInstrument {
      GUARDS { AESELF_MANAGEMENT.SELF_HEALING.inCollision }
      ENSURES { EVENTS.instrumentChecked }
      VARS { BOOLEAN canOperate }
      DOES { canOperate = CALL ACTIONS.checkInstrument }
      TRIGGERS { IF (not canOperate) THEN EVENTS.instrumentBroken END }
      ONERR_TRIGGERS { IF (not canOperate) THEN EVENTS.instrumentBroken END }
    }
  ....
  }
  ....
  EVENTS {
    EVENT collisionHappen {
      GUARDS { not METRICS.distanceToNearestObject }
      ACTIVATION { CHANGED { METRICS.distanceToNearestObject } } }
    EVENT timeToSendHeartbeatMsg { ACTIVATION { PERIOD { 1 min } } } }
  }
  ....
  METRICS {
    METRIC distanceToNearestObject {
      METRIC_TYPE { RESOURCE }
      METRIC_SOURCE { AEIP.MANAGED_ELEMENTS.worker.getDistanceToNearestObject }
      THRESHOLD_CLASS { DECIMAL [0.001 ~ ) } } }
  }// METRICS
} // ANT_Worker
```

Fig. 3. ASSL specification: self-healing

specification for the self-healing policy for an ANTS Worker. Here, the key features are:

- an *inCollision* fluent that takes place when the worker crashes into an asteroid or into another spacecraft, but it is still able to perform self-checking operations;
- an *inInstrumentBroken* fluent that takes place when the self-checking operation reports that the instrument is not operational anymore;
- an *inHeartbeatNotification* fluent that is initiated on a regular basis by a timed event to send the *heartbeat* message to the ruler;
- a *checkANTInstrument* action that performs operational checking on the carried instrument.
- a *distanceToNearestObject* metric that measures the distance to the nearest object in space (not presented here).
- a *collisionHappened* event prompted by the *distanceToNearestObject* metric when the latter changes its value and the same does not satisfy the metric's threshold class.

Operational Behavior

A self-management policy is evaluated as "held" if the policy is not in either one of its specified fluents, and as "not held" if there is at least one initiated fluent for that policy (the policy is currently in that fluent) [5, 6]. The ***SELF_HEALING*** policy (see Fig. 3) has three fluents: *inCollision*, *inInstrumentBroken*, and *inHeartbeatNotification*, i.e., the policy is evaluated as held when the policy is at least in one of these three fluents. Inference rules 18 through 48 enforce a definite strategy for evaluating the ***SELF_HEALING*** policy. The policy clauses (fluents and mappings) are evaluated in the context π of the ***SELF_HEALING*** policy, and the actions, events, and metrics are evaluated in the context of the ***ANT_Worker*** autonomic element (see Fig. 3). Inference rule 18 presents the operational evaluation of the *timeToSendHeartbeatMsg* timed event initiating the *inHeartbeatNotification* fluent (see rules 22 through 25). Thus,

$$(18)\quad \frac{\sigma \vdash \textbf{\textit{systemclock}}() \rightarrow t_{actv}}{ev \vdash_{\sigma} \textbf{\textit{ACTIV\_TIME}}\{t_{actv}\} \xrightarrow{Event(\text{timeToSendHeartbeatMsg})} \sigma'}$$

where ev is the tier environment exposed by the timed event, *systemclock*() is an abstract function returning the current time in the context σ, t_{actv} is the time at which the timed event is specified to occur.

Inference rules 19 through 21 present the operational evaluation of the *collisionHappened* event, which initiates the *inCollision* fluent (see rules 26–30). Thus,

$$(19)\quad \frac{\langle AE\rangle \xrightarrow{Metric(\text{distanceToNearestObject})} \langle AE\rangle' \; ev \vdash_{\sigma} \text{distanceToNearestObject} \rightarrow \textbf{\textit{true}}}{ev \vdash_{\sigma} \textbf{\textit{GUARDS}}\{\text{distanceToNearestObject}\} \rightarrow \textbf{\textit{prompt}}(\text{collisionHappened})}$$

$$(20)\quad \frac{\langle AE\rangle \xrightarrow{Metric(\text{distanceToNearestObject})} \langle AE\rangle' \; ev \vdash_{\sigma} \text{distanceToNearestObject} \rightarrow \textbf{\textit{false}}}{ev \vdash_{\sigma} \textbf{\textit{GUARDS}}\{\text{distanceToNearestObject}\} \rightarrow \neg\textbf{\textit{prompt}}(\text{collisionHappened})}$$

$$(21)\quad \frac{\langle AE\rangle \xrightarrow{Metric(\text{distanceToNearestObject})} \langle AE\rangle'}{ev \vdash_{\sigma} \textbf{\textit{CHANGED}}\{\text{distanceToNearestObject}\} \xrightarrow{Event(\text{collisionHappened})} \sigma'}$$

where *ev* is the tier environment exposed by the *collisionHappen* event. In rules 19 and 20 we use the transition operation *Metric*(distanceToNearestobject) to denote a state transition that occurs when the *distanceToNearestObject* metric changes its value, thus possibly prompting the *collisionHappened* event. Note that by operational semantic definition, an ASSL metric is evaluated as Boolean and is "true" only if the value it holds falls in the range determined by the metric's threshold class [5, 6] (see ***THRESHOLD_CLASS*** in Fig. 3). Here, rules 19 and 20 evaluate the ***GUARDS {…}*** clause, which verifies whether that metric is still valid after changing its value.

Inference rules 22 through 25 present the operational evaluation of the *inHeartbeatNotification* fluent together with the ***MAPPING {…}*** clause mapping that fluent to the *notifyForHeartbeat* action. Thus,

(22)

$$\frac{\langle AE\rangle \xrightarrow{Event(\text{timeToSendHeartbeatMsg})} \langle AE\rangle'}{f \vdash_{\sigma',\pi'} \textbf{\textit{INITIATED\_BY}}\{\text{timeToSendHeartbeatMsg}\} \xrightarrow{FluentIn(f,\text{timeToSendHeartbeatMsg})} \sigma',\pi'}$$

$$(23)\quad \frac{\langle AE\rangle \xrightarrow{Event(\text{msgHeartbeatSent})} \langle AE\rangle'\pi \vdash_{\sigma} \text{inHeartbeatNotification} \rightarrow \textbf{\textit{true}}}{f \vdash \sigma,\pi\, \textbf{\textit{TERMINATED\_BY}}\{\text{msgHeartbeatSent}\} \xrightarrow{FluentOut(f,\text{msgHeartbeatSent})} \sigma',\pi'}$$

(24)

$$\frac{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{timeToSendHeartbeatMsg}\} \xrightarrow{FluentIn(f,\text{timeToSendHeartbeatMsg})} \sigma',\pi'}{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inHeartbeatNotification }\} \xrightarrow{ActionMap(f,\text{notifyForHeartbeat})} \sigma'',\pi''}$$

(25)

$$\frac{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inHeartbeatNotification }\} \xrightarrow{ActionMap(f,\text{notifyForHeartbeat})} \sigma'',\pi''}{map \vdash_{\sigma'',\pi''} \textbf{\textit{DO\_ACTIONS}}\{\text{notifyForHeartbeat}\} \xrightarrow{Action(\text{notifyForHeartbeat})} \sigma''',\pi'''}$$

where f is the tier environment exposed by the *inHeartbeatNotification* fluent, π is the tier environment (and context) exposed by the ***SELF_HEALING*** policy, and *map* is the tier environment exposed by the ***MAPPING {…}*** clause (see Fig. 3).

Inference rules 26 through 30 present the operational evaluation of the *inCollision* fluent.

$$(26)\quad \frac{\langle AE\rangle \xrightarrow{Event(\text{collisionHappened})} \langle AE\rangle'}{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{collisionHappened}\} \xrightarrow{FluentIn(f,\text{collisionHappened})} \sigma',\pi'}$$

(27)

$$\frac{\langle AE\rangle \xrightarrow{Event(\text{instrumentChecked})} \langle AE\rangle'\pi \vdash_{\sigma} \text{inCollision} \rightarrow \textbf{\textit{true}}}{f \vdash_{\sigma,\pi} \textbf{\textit{TERMINATED\_BY}}\{\text{instrumentChecked}\} \xrightarrow{FluentOut(f,\text{instrumentChecked})} \sigma',\pi'}$$

(28)

$$\frac{\langle AE\rangle \xrightarrow{Event(\text{cannotCheckInstrument})} \langle AE\rangle' \pi \vdash_{\sigma} \text{inCollision} \rightarrow \textbf{\textit{true}}}{f \vdash_{\sigma,\pi} \textbf{\textit{TERMINATED\_BY}}\{\text{cannotCheckInstrument}\} \xrightarrow{FluentOut(f,\text{cannotCheckInstrument})} \sigma', \pi'}$$

(29)

$$\frac{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{collisionHappened}\} \xrightarrow{FluentIn(f,\text{collisionHappened})} \sigma', \pi'}{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inCollision }\} \xrightarrow{ActionMap(f,\text{checkANTInstrument})} \sigma'', \pi''}$$

(30)

$$\frac{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inCollision }\} \xrightarrow{ActionMap(f,\text{checkANTInstrument})} \sigma'', \pi''}{map \vdash_{\sigma'',\pi''} \textbf{\textit{DO\_ACTIONS}}\{\text{checkANTInstrument}\} \xrightarrow{Action(\text{checkANTInstrument})} \sigma''', \pi'''}$$

Inference rules 31 through 34 present the operational evaluation of the *inInstrumentBroken* fluent (f is the tier environment exposed by that fluent).

(31)

$$\frac{\langle AE\rangle \xrightarrow{Event(\text{instrumentBroken})} \langle AE\rangle'}{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{instrumentBroken}\} \xrightarrow{FluentIn(f,\text{instrumentBroken})} \sigma', \pi'}$$

(32)

$$\frac{\langle AE\rangle \xrightarrow{Event(\text{msgInstrumentBrokenSent})} \langle AE\rangle' \ \pi \vdash_{\sigma} \text{inInstrumentBroken} \rightarrow \textbf{\textit{true}}}{f \vdash_{\sigma,\pi} \textbf{\textit{TERMINATED\_BY}}\{\text{msgInstrumentBrokenSent}\} \xrightarrow{FluentOut(f,\text{msgInstrumentBrokenSent})} \sigma', \pi'}$$

(33)

$$\frac{f \vdash_{\sigma,\pi} \textbf{\textit{INITIATED\_BY}}\{\text{instrumentBroken}\} \xrightarrow{FluentIn(f,\text{instrumentBroken})} \sigma', \pi'}{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inInstrumentBroken }\} \xrightarrow{ActionMap(f,\text{notifyForBrokenInstrument})} \sigma'', \pi''}$$

(34)

$$\frac{map \vdash_{\sigma',\pi'} \textbf{\textit{CONDITIONS}}\{\text{inInstrumentBroken }\} \xrightarrow{ActionMap(f,\text{notifyForBrokenInstrument})} \sigma'', \pi''}{map \vdash_{\sigma'',\pi''} \textbf{\textit{DO\_ACTIONS}}\{\text{notifyForBrokenInstrument}\} \xrightarrow{Action(\text{notifyForBrokenInstrument})} \sigma''', \pi'''}$$

Note that the *inInstrumentBroken* fluent is initiated by the *instrumentBroken* event (see Rule 31), which is triggered by the *checkANTInstrument* action (see Rule 40).

Inference rules 35 through 44 present the stepwise operational evaluation of the clauses of the *checkANTInstrument* action. Thus,

(35)

$$\frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' a \vdash_{\sigma} \text{collisionHappend} \rightarrow \textbf{\textit{true}}}{a \vdash_{\sigma} \textbf{\textit{GUARDS}}\{\text{collisionHappend}\} \rightarrow \textbf{\textit{perform}}(\text{checkANTInstrument})}$$

(36)

$$\frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' a \vdash_{\sigma} \text{collisionHappend} \rightarrow \textbf{\textit{false}}}{a \vdash_{\sigma} \textbf{\textit{GUARDS}}\{\text{collisionHappend}\} \rightarrow \textbf{\textit{perform}}(\text{checkANTInstrument})}$$

(37)

$$\frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle'}{a \vdash_{\sigma} \textbf{\textit{DOES}}\{\text{canOperate} = \textbf{\textit{CALL}}\text{CheckInstrument}\} \xrightarrow{Action(\text{checkInstrument})} \sigma'}$$

(38)

$$\frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle'}{a \vdash_{\sigma} \textbf{\textit{DOES}}\{\text{canOperate} = \textbf{\textit{CALL}}\,\text{CheckInstrument}\} \xrightarrow{\neg Action(\text{checkInstrument})} \sigma'[err]}$$

$$(39)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' \quad a\vdash_{\sigma'} \textbf{\textit{DOES}}\ \{\text{canOperate} = \textbf{\textit{CALL}}\ \text{CheckInstrument}\} \xrightarrow{Action(\text{checkInstrument})} \sigma' \quad a\vdash_{\sigma'} \textbf{\textit{ENSURES}}\{\text{instrumentChecked}\} \xrightarrow{Event(\text{instrumentChecked})} \sigma''}{a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{instrumentChecked}\} \xrightarrow{Event(\text{instrumentChecked})} \sigma''}$$

$$(40)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' \quad a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{instrumentChecked}\} \xrightarrow{Event(\text{instrumentChecked})} \sigma'' \quad a\vdash_{\sigma''} be \rightarrow \textbf{\textit{true}}}{a\vdash_{\sigma''} \textbf{\textit{TRIGGERS}}\{\textbf{\textit{IF}}\ be\ \textbf{\textit{THEN}}\ \text{instrumentBroken}\ \textbf{\textit{END}}\} \xrightarrow{Event(\text{instrumentBroken})} \sigma'''}$$

$$(41)\quad \frac{a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{instrumentChecked}\} \xrightarrow{Event(\text{instrumentChecked})} \sigma'' \quad a\vdash_{\sigma''} be \rightarrow \textbf{\textit{false}} \quad \langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle'}{a\vdash_{\sigma''} \textbf{\textit{TRIGGERS}}\{\textbf{\textit{IF}}be\textbf{\textit{THEN}}\text{instrumentBroken}\,\textbf{\textit{END}}\} \rightarrow \sigma''}$$

$$(42)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' a\vdash_{\sigma} \text{instrumentChecked} \rightarrow \textbf{\textit{true}}}{a\vdash_{\sigma} \textbf{\textit{ENSURES}}\{\text{instrumentChecked}\} \rightarrow \sigma}$$

$$(43)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' a\vdash_{\sigma} \text{instrumentChecked} \rightarrow \textbf{\textit{false}}}{a\vdash_{\sigma} \textbf{\textit{ENSURES}}\{\text{instrumentChecked}\} \rightarrow \sigma'[err]}$$

$$(44)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{checkANTInstrument})} \langle AE\rangle' \quad Err^{a} \neq \emptyset}{a\vdash_{\sigma} \textbf{\textit{ONERR\_TRIGGERS}}\{\text{cannotCheckInstrument}\} \xrightarrow{Event(\text{cannotCheckInstrument})} \sigma'}$$

where a is the tier environment exposed by the *checkANTInstrument* action and *be* states for a Boolean expression (evaluated in a single step). In addition, Err^{a} and $perform(a)$ have the same meaning as in Sect. 5.3.1, but are addressed to the *checkANTInstrument* action.

Inference rules 45 through 46 and rules 47 through 48 present the operational evaluation of *notifyForHeartbeat* and *checkANTInstrument* actions respectively. Note, that 1) the ASSL specification of these actions is not presented in Fig. 3 due to space limitations; 2) we present only the evaluation of their ***DOES {…}*** and ***TRIGGERS {…}*** clauses.

$$(45)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{notifyForHeartbeat})} \langle AE\rangle'}{a\vdash_{\sigma} \textbf{\textit{DOES}}\{\textbf{\textit{CALL}}\text{sendHeartbeat}\} \xrightarrow{Function(\text{sendHeartbeat})} \sigma'}$$

$$(46)\quad \frac{a\vdash_{\sigma} \textbf{\textit{DOES}}\{\textbf{\textit{CALL}}\text{sendHeartbeat}\} \xrightarrow{Function(\text{sendHeartbeat})} \sigma\prime \quad \langle AE\rangle \xrightarrow{Action(\text{notifyForHeartbeat})} \langle AE\rangle'}{a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{msgHeartbeatSent}\} \xrightarrow{Event(\text{msgHeartbeatSent})} \sigma''}$$

$$(47)\quad \frac{\langle AE\rangle \xrightarrow{Action(\text{notifyForBrokenInstrument})} \langle AE\rangle'}{a\vdash_{\sigma} \textbf{\textit{DOES}}\{\textbf{\textit{CALL}}\text{sendInstrumentBroken}\} \xrightarrow{Function(\text{sendInstrumentBroken})} \sigma'}$$

$$(48)\quad \frac{a\vdash_{\sigma} \textbf{\textit{DOES}}\{\textbf{\textit{CALL}}\text{sendInstrumentBroken}\} \xrightarrow{Function(\text{sendInstrumentBroken})} \sigma' \quad \langle AE\rangle \xrightarrow{Action(\text{notifyForBrokenInstrument})} \langle AE\rangle'}{a\vdash_{\sigma'} \textbf{\textit{TRIGGERS}}\{\text{msgInstrumentBrokenSent}\} \xrightarrow{Event(\text{msgInstrumentBrokenSent})} \sigma''}$$

Testing the Self-healing Behavior

In this example, we experimented with the Java generated code for the ASSL self-healing specification for ANTS [25]. Note that by default, any Java application generated with the framework generates run-time log records that show important state-transition operations ongoing in the system at runtime. Thus, we can easily trace the behavior of the generated system by following the log records generated by the same. In this test, we generated the Java application for the ASSL self-healing specification model for ANTS, compiled the same with Java 1.6.0, and ran the compiled code. The application ran smoothly with no errors.

First, it started all system threads as it is shown in the following log records. Note that starting all system threads first is a standard running procedure for all Java application skeletons generated with the ASSL framework.

Log Records "Starting System Threads"

```
************************************************************

********** INIT ALL TIERS **********

************************************************************

********** START AS THREADS **********

************************************************************
1)   METRIC 'generatedbyassl.as.aes.ant_ruler.metrics.DISTANCETONEARESTOBJECT': started
2)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.INSTRUMENTLOST': started
3)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.MSGINSTRUMENTBROKENRECEIVED': started
4)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.SPACECRAFTCHECKED': started
5)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.TIMETORECEIVEHEARTBEATMSG': started
6)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.INSTRUMENTOK': started
7)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.MSGHEARTBEATRECEIVED': started
8)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.RECONFIGURATIONDONE': started
9)   EVENT 'generatedbyassl.as.aes.ant_ruler.events.RECONFIGURATIONFAILED': started
10)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.COLLISIONHAPPEN': started
11)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INHEARTBEATNOTIFICATION': started
12)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCOLLISION': started
13)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INTEAMRECONFIGURATION': started
14)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCHECKINGWORKERINSTRUMENT': started
15)  POLICY 'generatedbyassl.as.aes.ant_ruler.aeself_management.SELF_HEALING': started
16)  AE 'generatedbyassl.as.aes.ANT_RULER': started

************************************************************
17)  METRIC 'generatedbyassl.as.aes.ant_worker.metrics.DISTANCETONEARESTOBJECT': started
18)  EVENT 'generatedbyassl.as.aes.ant_worker.events.ISMSGHEARTBEATSENT': started
19)  EVENT 'generatedbyassl.as.aes.ant_worker.events.INSTRUMENTCHECKED': started
20)  EVENT 'generatedbyassl.as.aes.ant_worker.events.ISMSGINSTRUMENTBROKENSENT': started
21)  EVENT 'generatedbyassl.as.aes.ant_worker.events.COLLISIONHAPPEN': started
22)  EVENT 'generatedbyassl.as.aes.ant_worker.events.INSTRUMENTBROKEN': started
23)  EVENT 'generatedbyassl.as.aes.ant_worker.events.TIMETOSENDHEARTBEATMSG': started
24)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': started
25)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.ININSTRUMENTBROKEN': started
26)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INCOLLISION': started
27)  POLICY 'generatedbyassl.as.aes.ant_worker.aeself_management.SELF_HEALING': started
28)  AE 'generatedbyassl.as.aes.ANT_WORKER': started

************************************************************
29)  EVENT 'generatedbyassl.as.ants.events.SPACECRAFTLOST': started
30)  EVENT 'generatedbyassl.as.ants.events.EARTHNOTIFIED': started
31)  FLUENT 'generatedbyassl.as.ants.asself_management.self_healing.INLOSINGSPACECRAFT': started
32)  POLICY 'generatedbyassl.as.ants.asself_management.SELF_HEALING': started
33)  AS 'generatedbyassl.as.ANTS': started

************************************************************

********** AS STARTED SUCCESSFULLY **********

************************************************************
```

Here, records 1 through to 16 show the ANT_RULER autonomic element startup, records 17 through to 28 show the ANT_WORKER autonomic element startup, and records 29 through to 33 show the last startup steps of the ANTS autonomic system. After starting up all the threads, the system ran in idle mode for 60 s, when the timed

event timeToSendHeartbeatMsg occurred. This event is specified in the ***ANT_Worker*** to run on a regular time basis every 60 s (see below). The occurrence of this event activated the self-healing mechanism as shown in the following log records.

Log Records "Self-healing Behavior"

```
**********************************************************

********** AS STARTED SUCCESSFULLY **********

**********************************************************
34)  EVENT 'generatedbyassl.as.aes.ant_worker.events.TIMETOSENDHEARTBEATMSG': has occurred
35)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been initiated
36)  ACTION 'generatedbyassl.as.aes.ant_worker.actions.NOTIFYFORHEARTBEAT': has been performed
37)  EVENT 'generatedbyassl.as.aes.ant_worker.events.ISMSGHEARTBEATSENT': has occurred
38)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been terminated

39)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.TIMETORECEIVEHEARTBEATMSG': has occurred
40)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been initiated
41)  ACTION 'generatedbyassl.as.aes.ant_ruler.actions.CONFIRMHEARTBEAT': has been performed
42)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.MSGHEARTBEATRECEIVED': has occurred
43)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been terminated

44)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCHECKINGWORKERINSTRUMENT': has been initiated
45)  ACTION 'generatedbyassl.as.aes.ant_ruler.actions.CHECKWORKERINSTRSTATUS': has been performed
46)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.INSTRUMENTOK': has occurred
47)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCHECKINGWORKERINSTRUMENT': has been terminated

48)  EVENT 'generatedbyassl.as.aes.ant_worker.events.TIMETOSENDHEARTBEATMSG': has occurred
49)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been initiated
50)  ACTION 'generatedbyassl.as.aes.ant_worker.actions.NOTIFYFORHEARTBEAT': has been performed
51)  EVENT 'generatedbyassl.as.aes.ant_worker.events.ISMSGHEARTBEATSENT': has occurred
52)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been terminated

53)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.TIMETORECEIVEHEARTBEATMSG': has occurred
54)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been initiated
55)  EVENT 'generatedbyassl.as.aes.ant_worker.events.TIMETOSENDHEARTBEATMSG': has occurred

56)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been initiated
57)  ACTION 'generatedbyassl.as.aes.ant_ruler.actions.CONFIRMHEARTBEAT': has been performed
58)  ACTION 'generatedbyassl.as.aes.ant_worker.actions.NOTIFYFORHEARTBEAT': has been performed
59)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.MSGHEARTBEATRECEIVED': has occurred
60)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been terminated

61)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCHECKINGWORKERINSTRUMENT': has been initiated

62)  EVENT 'generatedbyassl.as.aes.ant_worker.events.ISMSGHEARTBEATSENT': has occurred
63)  FLUENT 'generatedbyassl.as.aes.ant_worker.aeself_management.self_healing.INHEARTBEATNOTIFICATION': has been terminated

64)  ACTION 'generatedbyassl.as.aes.ant_ruler.actions.CHECKWORKERINSTRSTATUS': has been performed
65)  EVENT 'generatedbyassl.as.aes.ant_ruler.events.INSTRUMENTOK': has occurred
66)  FLUENT 'generatedbyassl.as.aes.ant_ruler.aeself_management.self_healing.INCHECKINGWORKERINSTRUMENT': has been terminated
```

As we see from the log records, the self-healing behavior correctly followed the specification model. Records 34 through to 38 show the initiation and termination of the INHEARTBEATNOTIFICATION fluent. This resulted in the execution of the NOTIFYFORHEARTBEAT action (see record 36) that sends a heartbeat message to ***ANT_Ruler***[1] (see record 37). Records 39 through to 43 show how this message is handled by the ***ANT_Ruler***. Records 44 through to 47 show how the INCHECKINGWORKERINSTRUMENT fluent is handled by the system. This fluent is initiated by the MSGHEARTBEATRECEIVED event. Next the CHECKWORKERINSTRSTATUS action is performed (see record 45), which resulted into the

[1] The ASSL specification of ANT_Ruler is not presented here. The interested reader is advised to refer to [25].

INSTRUMENTOK event (see record 46). The latter terminated the INCHECKINGWORKERINSTRUMENT fluent (see record 47). Records 48 through to 66 show that the system continued repeating the steps shown in records 34 though to 47. This is because the policy-triggering events are periodic timed events and the system did not encounter any problems while performing the executed actions, which could possibly branch the program execution.

This experiment demonstrated that the generated code had correctly followed the specified self-healing policy by reacting to the occurring self-healing events and, thus, providing appropriate self-healing behavior.

6 Formal Verification with ASSL

Due to the synthesis approach of automatic code generation, ASSL guarantees consistency between a specification and the corresponding implementation. Moreover, the framework provides mechanisms for formal verification of the ASSL specifications.

6.1 Consistency Checking

The ASSL Consistency Checker (see Fig. 3) is a framework mechanism for verifying ASSL specifications by performing exhaustive traversing. In general, the Consistency Checker performs two kinds of consistency-checking operations: (1) light - checks for type consistency, ambiguous definitions, etc.; and (2) heavy - checks whether the specification model conforms to special correctness properties. The ASSL correctness properties are special ASSL semantic definitions [5, 6] defining tier-specific rules that make it possible to reason about the properties of the specifications created with ASSL. They are expressed in First-Order Linear Temporal Logic (FOLTL)[2] [29], which allows for formalization of rules related to system evolution over time. An example of a semantic rule defined for the AS/AE *Self-management Policies Tier* (see Table 1) is related to policy initiation [5, 6]:

> *"Every policy is triggered by a finite non-empty set of fluents, and performs actions associated with these fluents".*

$$\forall \boldsymbol{\pi} \in \boldsymbol{\Pi} \bullet ((\mathcal{F} \not\equiv \emptyset \wedge \boldsymbol{A} \not\equiv \emptyset) \Rightarrow (\forall \boldsymbol{f} \in \mathcal{F} \bullet \exists \boldsymbol{a} \in \boldsymbol{A} \bullet (\boldsymbol{trigger}(\boldsymbol{f}, \boldsymbol{\pi}) \Rightarrow \boldsymbol{perform}(\boldsymbol{a}))))$$

where:

- $\boldsymbol{\Pi}$ is the universe of self-management policies in the AS;
- $\mathcal{F}$ is a finite set of fluents specified by the policy π;
- $\boldsymbol{A}$ is a finite set of actions mapped to the fluents specified by the policy π.

[2] In general, FOLTL can be seen as a quantified version of linear temporal logic. FOLTL is obtained by taking propositional linear temporal logic and adding a first order language to it.

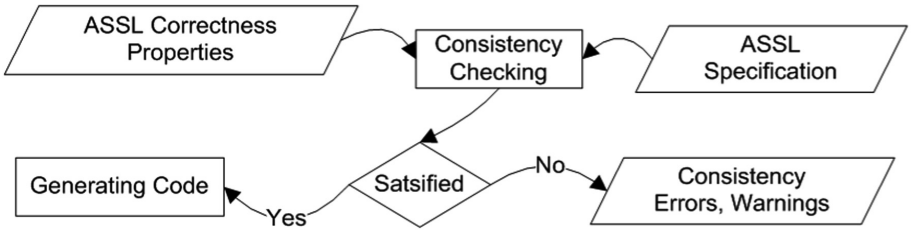

Fig. 4. Consistency checking with ASSL

It is important to mention that the consistency checking mechanism generates *consistency errors* and *warnings* (see Fig. 4). Warnings are specific situations, where the specification does not contradict the correctness properties, but rather introduces uncertainty as to how the code generator will handle it.

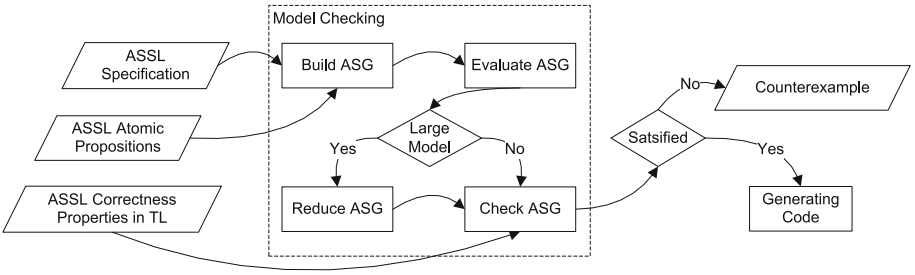

Fig. 5. Model checking with ASSL

6.2 Model Checking

Although the ASSL Consistency Checker tool takes care of syntax and consistency errors, it still cannot handle logical errors and thus, cannot assert safety (e.g., freedom from deadlock) or liveness properties. Therefore, to ensure the correctness of the ASSL specifications, and that of the generated ASs, at the time of writing, there was ongoing research on model checking with ASSL:

- The main trend influencing this research is on a *model-checking mechanism* that takes an ASSL specification as input and produces as output a *finite state-transition system* (called ASSL State Graph (ASG) or state machine) such that a specific correctness property in question is satisfied if and only if the original ASSL specification satisfies that property [30].
- Another research direction is towards mapping ASSL specifications to special service logic graphs, which support the so-called reverse model checking [31].

Figure 5 depicts the first approach to model checking in ASSL. As shown, the Model Checker tool builds the ASG for the AS in question by using its ASSL specification to derive the system states and associates with each derived state special atomic propositions (defined in FOLTL) true in that state [30].

The notion of state in ASSL is related to tiers. The *ASSL Operational Semantics* (see Sect. 4) considers a state-transition model where tier instances can be in different *tier states*. Formally, an ASG is presented as a tuple **(S; Op; R; S_0; AP; L)** [30] where: **S** is the set of all possible ASSL tier states; **Op** is the set of special ASSL state-transition operations (see Sect. 4.1); $\mathbf{R} \subseteq \mathbf{S} \times \mathbf{Op} \times \mathbf{S}$ are the possible transitions; $\mathbf{S}_0 \subseteq \mathbf{S}$ is a set of initial tier states; **AP** is a set of atomic propositions; $\mathbf{L} : \mathbf{S} \rightarrow 2^{\mathbf{AP}}$ is a labeling function relating a set $\mathbf{L}(\mathbf{s}) \in 2^{\mathbf{AP}}$ of atomic propositions to any state **s**, i.e., a set of atomic propositions that hold in that state. The ASSL model-checking mechanism uses *correctness properties* (see Sect. 6.1) to check if these are held over the system's **ASG** by matching for each state the correctness properties with the atomic propositions **AP**. This helps the ASSL framework trace the execution state paths in ASG and produce counterexamples of such paths that lead to violation of the correctness properties. Moreover, the so-called state explosion problem [29] is considered when the size of the ASG must be reduced in order to perform efficient model checking [30].

7 Summary

This article has presented the formalism of ASSL (Autonomic System Specification Language) in terms of notation and operational semantics. ASSL is a domain-specific formal approach providing both formalism and tool support that help developers implement autonomic systems. It has been successfully used to develop prototype models for a variety of systems incorporating AC features and proven to be a valuable approach to problem formation, modeling, verification and implementation of autonomic systems. With ASSL, the formal specifications are automatically verified for consistency flaws and the provided synthesis approach of automatic code generation, guarantees consistency between a specification and the corresponding implementation. Moreover, to enhance the software verification capabilities of the framework, a model checking mechanism is under development.

ASSL implies a complex multi-tier hierarchy of *specification constructs* categorized as ASSL *tiers*, *sub-tiers* and *clauses*. Both *structural* and *functional relationships* form the semantic relations between the ASSL specification constructs. Whereas the ASSL multi-tier specification model imposes the structural relationships between tiers, sub-tiers and clauses, the ASSL operational semantics forms the functional relationships of the same. Conceptually, the ASSL operational semantics is driven by special state-transition operations and tier states. The operational evaluation of ASSL specifications is a stepwise evaluation of the specified ASSL tiers, sub-tiers and clauses, which are evaluated as state transition models where state-transition operations cause a current state to evolve to a new one.

Specifying with ASSL requires a good understanding of the ASSL formalism. This article tackles this problem by introducing the ASSL formalism from both structural and operational perspectives. In addition, to demonstrate the theoretical concepts and

flavor of the ASSL formalism, case study examples have presented ASSL specifications and their operational evaluation.

In conclusion, it should be noted that ASSL provides the IT community with an extremely needed and powerful framework for development of autonomic systems. Overall, ASSL is sufficiently generic and adaptable to accommodate most of the AC development aspects.

Acknowledgement. This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software Research Centre (www.lero.ie).

References

1. Horn, P.: Autonomic computing: IBM's perspective on the state of information technology, Technical report, IBM T. J. Watson Laboratory, 15 October 2001
2. IBM Corporation: An architectural blueprint for autonomic computing, white paper, Fourth edition, IBM Corporation (2006)
3. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
4. Murch, R.: Autonomic Computing: On Demand Series. IBM Press, Prentice Hall (2004)
5. Vassev, E.: 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
6. Vassev, E.: ASSL: autonomic system specification language - a framework for specification and code generation of autonomic systems, LAP Lambert Academic Publishing, Germany, November 2009
7. Vassev, E., Hinchey, M.: Modeling the image-processing behavior of the NASA Voyager mission with ASSL. In: Proceedings of the 3rd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT 2009), pp. 246–253 IEEE Computer Society (2009)
8. Vassev, E., Hinchey, M., Paquet, J.: Towards an ASSL specification model for NASA swarm-based exploration missions. In: Proceedings of the 23rd Annual ACM Symposium on Applied Computing (SAC 2008) - AC Track, pp. 1652–1657. ACM (2008)
9. Vassev, E., Mokhov, S.A.: Towards autonomic specification of distributed MARF with ASSL: self-healing. In: Lee, R., Ormandjieva, O., Abran, A., Constantinides, C. (eds.) Software Engineering Research, Management and Applications 2010. Studies in Computational Intelligence, vol. 296, pp. 1–15. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13273-5_1
10. Vassev, E., Hinchey, M., Nixon, P.: Prototyping home automation wireless sensor networks with ASSL. In: Proceedings of the 7th IEEE International Conference on Autonomic Computing and Communications (ICAC2010). IEEE Computer Society (2010 to appear)
11. Srivas, M., Miller, S.: Formal verification of the AAMP5 microprocessor: a case study in the industrial use of formal methods. In: Proceedings of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT 1995), pp. 2–6. IEEE Computer Society (1995)
12. National Aeronautics and Space Administration: Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, I: Planning and Technology Insertion. NASA, Washington, DC (1998)

13. Kowalsky, R., Sergot, M.: A logic-based calculus of events. New Gener. Comput. **4**(1), 67–95 (1986)
14. IBM Corporation. Defining service-level objectives, Tivoli Software. IBM Tivoli. http://publib.boulder.ibm.com/tividd/td/TDS390/SH19-6818-08/en_US/HTML/DRLM9mst27.htm. Accessed 19 Aug 2009
15. IBM Corporation: Policy Management for Autonomic Computing - Version 1.2, Tutorial. IBM Tivoli (2005)
16. The International Engineering Consortium, Specification and Description Language (SDL), Web ProForum Tutorials. http://www.iec.org. Accessed 2 Feb 2009
17. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of ICSE 2006 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2006), China (2006)
18. Read, D.: Utility theory from Jeremy Bentham to Daniel Kahneman, Working Paper No: LSEOR 04-64, Department of Operational Research, London School of Economics, London (2004)
19. Leavens, G.T., Cheon, Y.: Design by contract with JML, Technical report, Formal Systems Laboratory (FSL) at UIUC (2006)
20. Banatre, J.P., Fradet, P., Radenac, Y.: Programming self-organizing systems with the higher-order chemical language. Int. J. Unconv. Comput. **3**(3), 161–177 (2007)
21. Andrei, O., Kirchner, H.: A higher-order graph calculus for autonomic computing. In: Lipshteyn, M., Levit, Vadim E., McConnell, Ross M. (eds.) Graph Theory, Computational Intelligence and Thought. LNCS, vol. 5420, pp. 15–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02029-2_2
22. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Lowe, M.: Algebraic approaches to graph transformation - Part I: basic concepts and double pushout approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1, pp. 163–246. World Scientific, Singapore (1997)
23. Knuth, D.E.: Backus normal form vs. Backus Naur form. Commun. ACM **7**(12), 735–773 (1964)
24. Truszkowski, W., Hinchey, M., Rash, J., Rouff, C.: NASA's swarm missions: the challenge of building autonomous software. IT Prof. **6**(5), 47–52 (2004)
25. Vassev, E., Hinchey, M.: ASSL specification and code generation of self-healing behavior for NASA swarm-based systems. In: Proceedings of the 6th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2009), pp. 77–86. IEEE Computer Society (2009)
26. Vassev, E., Hinchey, M., Paquet, J.: A self-scheduling model for NASA swarm-based exploration missions using ASSL. In: Proceedings of the 5th IEEE International Workshop on Engineering of Autonomic and Autonomous Systems (EASe 2008), pp. 54–64. IEEE Computer Society (2008)
27. Vassev, E., Hinchey, M.: 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), pp. 13–18. IEEE Computer Society (2008)
28. Plotkin, G.D.: A structural approach to operational semantics, Report DAIMI FN-19, Computer Science Department, Aarhus University, Aarhus, Denmark (1981)
29. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)

30. Vassev, E, Hinchey, M., Quigley, A: Model checking for autonomic systems specified with ASSL. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), pp. 16–25. NASA (2009)
31. Bakera, M., Wagner, C., Margaria, T., Vassev, E., Hinchey, M., Steffen, B.: Component-oriented behavior extraction for autonomic system design. In: Proceedings of the First NASA Formal Methods Symposium (NFM 2009), pp. 66–75. NASA (2009)