Modeling Sensor Web Autonomy

Al Underbrink and Andrew Potter
Sentar, Inc.
315 Wynn Drive, Suite 1
Huntsville, AL 35805
256-430-0860
al.underbrink@sentar.com

Ken Witt and Jason Stanley
West Virginia High Technology Foundation Consortium
1000 Galliher Drive
Fairmont, WV 26554
304-363-5482
kwitt@wvhtf.org

Abstract—The research developed an architecture and experimental system for autonomously operating a web of Earth sensing devices. Example sensing devices include orbiting spacecraft, ground surface sensors, ocean surface sensors, ocean subsurface sensors, wheeled vehicles, fixed ground systems, weather balloons, and aircraft. A wide variety of sensors, sensor platforms, and operational characteristics (e.g., fixed versus mobile) required a common language for specifying and sharing data, information, and knowledge of the "sensor web". An ontology was developed and used for collaboration between multiple sensor systems. The ontology is used by distributed agents to autonomously operate a sensor web in a distributed testbed. The system was also deployed and demonstrated on an orbiting satellite. The ontology models the application domain and the agent-based control system for the sensor web. The ontology was also used to implement the distributed agent system and to guide the design and development of the sensor web control system. The result is comprehensive model not only of the application domain, but also the control system for the application domain.

TABLE OF CONTENTS

1. INTRODUCTION

The "Using Intelligent Agents to Form a Sensor Web for Autonomous Mission Operations" (SWAMO) project[1,2,3] developed an architecture and laboratory system for autonomously operating a web of Earth sensing devices [1,2]. Example Earth sensing devices include orbiting spacecraft, ground surface sensors, ocean surface sensors, ocean subsurface sensors, wheeled vehicles, fixed ground systems, weather balloons, and aircraft. A wide variety of sensors (e.g., temperature, wind speed, global position, seismic activity), sensor platforms (e.g., satellites, ground radar, ocean buoys), and operational characteristics (e.g., fixed position and orientation versus mobile or articulated) required a common language for specifying and sharing data, information, and knowledge of the "sensor web". An ontology was developed and used for collaboration between multiple sensor systems. The SWAMO ontology was used by distributed agents to autonomously operate a representative sensor web in a distributed testbed. The SWAMO system was also deployed and demonstrated on an orbiting satellite over a period of three months.

The ontology was developed to model the application domain and the agent-based control system for the sensor web. The ontology was then used to implement the distributed agent system and to guide the design and development of the sensor web control system.

A significant finding of the SWAMO project is the importance of comprehensively modeling not only the application domain (i.e., the sensor web), but the control system for such service-oriented systems. The degree of automation and autonomy can be greatly enhanced by reasoning about the control system and the dynamic capabilities of a sensor web. By adapting the control decision making in response to partial system degradation, human intervention can be limited, operational costs can be reduced, and new sensing services can be automatically discovered and engaged.

2. RELATED WORK

While SWAMO is unique in its use of ontology as an enabling satellite technology for automated decision making and response for the dynamic Sensor Web environment, there has been some work in applying ontological techniques to various aspects of the satellite domain. Sanchez-Gestido et al. [3] used a semantic grid architecture for deploying complex space domain applications involving multiple organizations and diverse shared data and computing elements. Their architecture was used for monitoring and data analysis in a satellite mission in nominal operations. One of the benefits of this approach was that, once the conceptualization of generic elements was complete, most of the features were reusable from one Satellite Mission to another. Wright et al. [4] also used semantic web and grid technologies for a system for

[1] Funded by the NASA Research Opportunities in Space and Earth Sciences (ROSES) project AIST-05-0061.

[3] IEEEAC paper #1101, Version 2, Updated 2010:10:27

monitoring and data analysis for satellite missions, and found similar benefits.

Pulvermacher, Brandsma, and Wilson [5] developed an XML-based ontology for capturing data structures, content, and semantics in the domain of military space surveillance. The ontology was proposed as a standard shared resource for data interoperability in multi-domain military space applications. Uszok et al. [6] used policy-based models to drive human-robotic teamwork and adjustable autonomy for highly-interactive autonomous systems such as the Personal Satellite Assistant (PSA), a softball-sized flying robot that was designed to operate onboard spacecraft in pressurized micro-gravity environments.

Zetocha and Ortiz [7] prototyped an executive controller based on intelligent agent technology for use in integrating reasoning mechanisms, taking into account the spacecraft environment as well as mission objectives to autonomously monitor and control a spacecraft. One of their prototypes was an on-board controller designed to monitor and control the aligning and pointing optical sensors based on sensor readings and input from other sources. Another prototype was developed to align inertial measurement units onboard the NASA Space Shuttle. The alignment agent was autonomous, self-aware, and goal-oriented: it ran without human intervention, maintained a representation of the operational environment, and it continuously pursued its goal of maintaining inertial measurement unit alignment. Pantoquilho [8] defined a spacecraft and space weather ontology. Because the ontology was developed for use in systems analysis rather than autonomous reasoning, it was implemented in UML, and presented as a high-level UML class diagram with XML based concept schemas. Thanapalan and Veres [9] used an ontology to support communication between the agents in a multi-agent architecture for satellite formation flying control, enabling satellites to maintain specified positions and attitudes relative to one another. The system worked successfully in experimental studies using laboratory testing hardware including multiple satellite models.

Similar approaches have been used in controlling autonomous land vehicles. For example, Schlenoff et al. [10] and Provine et al. [11] used ontologies to provide improved capabilities and performance of on-board route planning for autonomous vehicles. They developed an ontology of environmental objects, with rules for estimating the damage that would be incurred by collisions. Collision damage estimates were fed to a route planner for use in determining whether to avoid the object. More generally, Nilsson et al. [12] explored the use of ontologies as design tools for robotics and found that several factors would be critical for the ontological approach: standard device descriptions, distributed databases for both functional and device ontologies, and adoption of standard tools such as Protégé for ontology import and export, as well as for ease of use. To this extent, ontology for robotics differ little from other applications. Nilsson et al. also recommended that robotics ontologies consist of composition of devices, tasks, and skills, as that would enable machine reasoning on multiple levels of abstraction.

Dennis et al. [13] have noted that the space industry is currently moving away from monolithic platforms in favor of multiple, smaller satellites working in teams using distributed methods. A consequence of this is that we may see increased opportunities for ontology utilization in satellite control systems. Dennis et al. [14] describe a system based on rational hybrid agents. Their system uses an abstraction engine which generates a stream of incoming sensor and action abstractions which are represented ontologically. The aim of these researchers is to enable agents to make decisions by allowing them to reason about the outcome of possible actions.

3. BACKGROUND

The SWAMO ontology incorporates concepts from multiple web standards, including Open Geospatial Consortium (OGC) SensorML, OGC Web Services, and the Semantic Web for Earth and Environmental Terminology (SWEET) ontologies. The SWAMO ontology also includes conceptualizations of the control system (i.e., the distributed software agents) to better support autonomous operations. Platform, sensor, and software agent characteristics were modeled. The software agents perform decision making based on the sensor web concepts, entities, and relationships which describe the sensing capabilities within the sensor web. Automated reasoning determines, via Web Service descriptions, which sensor web platforms and sensors are capable and available for satisfying an end user science requirement. Sensing system products are dynamically composed to yield new science products fulfilling Web Service descriptions. The sensor web as a whole is able to autonomously adjust and adapt to partial or full system failure at the sensor platform and sensing system levels. The ability to autonomously operate a complex collection of sensing systems is a key result of the project.

The SWAMO ontology provides a semantic description of the fundamental concepts in the Sensor Web system and their inter-relationships. The overall SWAMO ontology describes physical devices, such as sensor platforms, sensors and detectors residing on the sensor platforms, any actuators and manipulators on the sensor platforms, ground control stations, and communications systems. Additionally, the overall ontology describes logical entities, such as experiment process models and methods, experiment process chains, the SWAMO software agents, current experiment plans and schedules, and sensor web system tasks and events. A prominent example of Sensor Web description is the Open Geospatial Consortium (OGC) Sensor Web Enablement (SWE) services such as the Sensor Observation Service (SOS) and the Sensor Planning Service (SPS). SOS and SPS are part of SWE which describes standards and services for allowing any sensing system conforming to the standards to interoperate.

The SWAMO ontology does not describe how the information and knowledge are used. This is left to implementations within the various SWAMO components (software agents, communications interfaces, experiment computational models, etc.). The concepts and interrelationships within the ontology are limited to those that must be processed by the SWAMO control components.

It is anticipated that, eventually, SensorML will be used as a common representation and storage language for the concepts and entities represented by the ontology. Future versions of the SWAMO system may be extended to ingest SensorML descriptions of Sensor Webs and use those descriptions within its own decision-making. Therefore, it seemed wise to enable compatibility between the two as much as possible, to which end we considered future SensorML migration in our design decisions, as will be described in the following discussion.

Aspects of SWAMO not directly supported in SensorML, however, are the concepts processed by the software agents and the knowledge and information they use in their operation. Intelligent software agents use resources such as plans and schedules, and include as well concepts such as events and time. For this reason, the software agent ontology does not quite fit into the SensorML partitioning presented below and has a separate ontology description.

4. THE SWAMO ONTOLOGY

The SWAMO ontology is presented in detail so that the overall results are available to other researchers. For presentation purposes, we divide the ontology into four parts. Note, however, that the parts are defined as a single ontology for the SWAMO distributed sensor web control system. The ontology follows the structure of SensorML and its major decomposition of concepts. In SensorML, all concepts are considered to be *processes*. Processes may be viewed as physical or logical, and may be aggregated or atomic. This decomposition of the SensorML view is depicted in Figure 1.

In SensorML, the view of a sensing system is rooted with the Process concept, which is a superclass to four major classes: System, Component, Process_Chain, and Process_Model. The System and Component classes represent composite and atomic physical devices, respectively. The Process_Chain and Process_Model classes represent composite and atomic logical processes, respectively.

An example of a composite physical system is a satellite platform. It may be composed of several subsystems, including several Earth sensors, a power subsystem, a communications bus, and a computing system. Each of these subsystems may be further decomposed into some set of atomic components. The granularity of decomposition is generally dependent upon the degree of control required for the control system on the platform.

Likewise, with the logical processes, process chains are composed of one or more subprocess chains or process models. These subprocesses represent logical procedures that must be performed on a physical device. Their degree of decomposition depends upon the required detail of sensing system planning and the sensing system service descriptions.

The SWAMO ontology, then, begins with a description of the Process Class. Referring to Figure 2, the Process class description has four slots defined: *inputs*, *outputs*, *parameters*, and *metadata*. The inputs, outputs, and parameters slots are similar to each other. All are of type Class; however, each is constrained to class instances of a different type. The value for the inputs slot must be an instance of type Input (described below), the value for the outputs slot must be an instance of type Output (described below), and the value for the parameters slot must be an instance of type Parameter. The cardinality is 1:* (one-to-many) since a valid process description must have at least one input and one output, but may have multiple inputs and outputs. The cardinality of parameters is 0:* (zero-to-many) since parameters are optional. There are no other constraints on these slots. The metadata slot is of type Class and is constrained to an instance of type Metadata (described below). Since SensorML metadata is optional, its cardinality is zero-to-one.

The Input, Output, and Parameter classes represent three distinct concepts, but their information structures are identical. An object-oriented implementation of these concepts may subclass them from a common superclass. These three concepts all have an identifying *name* slot, a *definition* slot, and a *unit_of_measure* slot.

The *name* and *definition* slots are constrained to type String, are required, and are singleton. These slots are not defined in further detail to allow for implementation-specific descriptions. The *unit_of_measure* slot is of type Class, is constrained to be an instance of type UCUM (described below), and has cardinality of 1:1.

The Metadata class describes the SensorML concept of extra information made available for implementation-specific uses. Such uses may include semantic searching for processes that produce results with specific characteristics or specifying requirements of a process for matching actors with processes. The Metadata class has two slots, *characteristics* and *capabilities*, which are optional, but if provided may only have a single value of type String. They serve as the placeholders for the SensorML specification of the information associated with processes and systems.

The UCUM class represents a Unified Code for Units of Measure[4] description. There are two slots included in the UCUM. The *dimensions* slot identifies the number of fields in the base unit description. For example, the base description of one Tesla is "Wb/m2", which has a dimension of three. The first base value is webers (Wb), the second is divided by (/), and the third is square meters (m^2). It is also important to distinguish the upper and lower case versions of a unit of measure symbol; for example, "t" represents mass in metric tonnes while "T" is a metric unit of measure for the strength of a magnetic field in teslas. This information is necessary for parsing units of measure, so the *dimensions* slot value is required and only one may be provided.

Lastly, the *base* slot captures the description of the unit of measure. Continuing with the example above, "Wb/m2" is the base value for the *teslas* position, commonly stated as "webers per square meter". The *base* slot is of type String and its cardinality if 1:1. With the values provided in these slots, a unique unit of measure can be extracted and used for conversions, comparisons, and other sensor web decision making.

The remaining sections will describe in detail the other parts of the SWAMO ontology. These are the Platform ontology, which represents physical systems; the Model ontology, which represents logical systems; the Agent ontology, which represents the reasoning and control aspects of SWAMO; and the SOS ontology, which represents the observations available from the sensor web. For SWAMO, there is actually only one ontology; the parts are presented separately only for convenience.

4.1 Platform Ontology

The Platform ontology is shown Figure 3. The Platform ontology represents the aspects of the SWAMO ontology which are consistent with the physical entities in a SensorML specification. There are placeholders for the System and Component classes shown in the top-level SWAMO ontology in the previous section. These classes, and others, shown in this diagram are described in detail in this section.

For compatibility with a SensorML specification of the Sensor Web, the Platform ontology includes a System class which corresponds to a standard system specification in a SensorML description. In SensorML, systems and subsystems are both types of the System class. For example, a satellite platform is a specialized System, as indicated by the "is-a" relationship between the Platform and System classes in the diagram. As another example, a detector is an atomic component as indicated by the is-a relationship between the Detector and Component classes in the diagram.

[4] The UCUM describes a standard way to specify and covert units of measure in a wide variety of systems. See http://aurora.rg.iupui.edu/UCUM/ for details.

From the perspective of ontology specification, a System class is a sub-class of a Component class. This is because the System class is more specific than the Component class. The specialization is due to the addition of a slot to the description of the Component class.

The Component class has four slots: the *capacity*, *availability*, *name*, and *reference_frame* slots allow decision making about the maximum performance of a system resource (the capacity slot), the current availability of that resource, the resource name/identifier, and the spatial reference for the physical device, respectively. The combination of *capacity* and *availability* allows the representation of degraded or failed systems and subsystems and enables autonomic operations. The *name* slot allows the association of an identifier for each system or subsystem. The *reference_ frame* describes how a device is spatially related to another, containing device. Since a component is atomic, it cannot be decomposed into other subsystems and thus has no slots to represent decomposition (as does the System class, described below).

The *capacity* slot is of type Integer, is required, and has a single value. The availability slot is of type Float, has a required single value, and is constrained between 0.0 and 100.0, inclusively. That is, the capacity is a percentage value. The *reference_frame* slot is of type Class and is constrained to instances of type Reference_Frame (described below). Each device must have one and only one reference frame. The *name* slot is a required single String value.

The Reference_Frame class description has a single slot, the *engineeringCRS*, whose SensorML specification uses the Geography Markup Language (GML) EngineeringCRS object. Lacking further definition, the *engineeringCRS* slot is a placeholder for a valid reference frame object that may be captured as a SensorML description. That is, the EngineeringCRS class description is TBD and is currently represented as a String value.

A System class is a subclass of the Component class and adds one additional slot to the Component superclass definition, the *subsystems* slot. The *subsystems* slot is consistent with SensorML in that a System class instance is defined by one or more of its subsystems. These subsystems may be made up of many devices, including sensors, actuators, satellite platforms, and on-board computers. Therefore, the *subsystems* slot is of type Class with allowed instances of type System. The cardinality of the *subsystems* slot is one or more subsystem component instances. This cardinality represents the required decomposability of a system. Since the *subsystems* slot is of type System class, any system can be recursively decomposed and described to any required level of detail.

There are six subclasses of the System/Component class shown in the Platform ontology diagram (Detector, Actuator, Sensor, Platform, Computer, and Bus), although

many more could be included. One of the more important system subclasses is the Platform class. A platform is a general system on which Sensor Web activities may be conducted. The general Platform class can be used to represent ground-based systems (e.g., an operator workstation), ocean surface sensors, orbiting satellites, etc., in a Sensor Web.

Each subclass of System inherits the slots defined in the System class. Additionally, the subclasses of System define their own slots. The Platform class has several slots that represent its description and characterize physical systems, which have sensing and data collecting attributes. Three slots combine to describe the characteristics of a platform. The *position* slot, the *capabilities* slot, and the *sensors* slot describe the spatial relationships with a platform to its reference frame, its general functional capabilities, and its Earth sensing capabilities, respectively.

The *position* slot is of type Class and it is constrained to be an instance of the Position class (described below). A value for the slot is optional, so the slot cardinality is zero or one. The *capabilities* slot describes the functional systems on the platform. Two examples of functional systems are a computer processor and a communications bus. Thus the *capabilities* slot is of type Class, whose values can be an instance of a Computer, Bus, or other Component class. One or more values are required.

The *sensors* slot is of type Class and constrained to instances of type Sensor (described next) with cardinality zero or more. This allows the representation of sensor platforms that may optionally have many sensors.

The Sensor class is an important class in the Platform ontology for the Sensor Web. A Sensor logically defines a collection of detectors, often in the form of an array of detectors, which collect and assimilate Earth science data. Therefore, the Sensor class description has a single slot, *detectors*, which is of type Class, and constrained to Detector class (described below) instances. A sensor will include at least one, but possibly many, *detectors*, so the cardinality of the detectors slot is one or more.

The Position class, shown in Figure 3, captures the spatial location, orientation, velocity, and acceleration of the platform with respect to time. Temporal information is necessary because the position information of a platform is applicable at a point in time. All of the spatial attributes may change over time. The position is intended to represent the spatial characteristics of a wide variety of Sensor Web platforms, such as orbiting satellites which can also be re-oriented for Earth sensing experiments, mobile ground-based systems such as trucks, and fixed ground stations with variable orientation such as radar. The position is intended to represent the spatial characteristics of a wide variety of Sensor Web platforms.

The *velocity* slot is of type Class and is constrained to instances of type Velocity (described below). The *location* slot is of type Class and is constrained to instances of type Location (described below). The *time* slot is of type Class and is constrained to instances of type Time (described in the Agent ontology section). The *acceleration* slot is of type Float. The *orientation* slot is of type Class and is constrained to instances of type Orientation. All of these slots have cardinality of 1:1; that is, a value is required and a single valued is allowed.

In the current Platform ontology description, the *velocity* and *acceleration* slots support the specification of spatial information in terms of Cartesian coordinate systems. While some systems may use polar coordinates instead, slots and/or subclasses of this type are as yet TBD. The *velocity* and *acceleration* slots, along with the *time* slot, may be used to locate a platform relative to Earth with respect to time through algebraic computation or by calculating the first derivative, respectively.

A Platform class will also typically have other capabilities. Previously mentioned and depicted in Figure 3 were the platform computer and communications bus. At this time, the only slot defined for either of these two classes is a *protocol* slot in the Bus class. The protocol is of type String and must contain a single value. Examples include the GMSEC bus and cFE/cFS bus. Clearly, attributes of these capabilities will be useful in decision making about the distribution of experiment processing and sharing of resources. Therefore, additional definition is required in order to have a more complete representation of the platforms.

The Position class collects several other classes used to represent the spatial relationship of the physical system to its environment. These classes, described below, are the Velocity class, the Location class, and the Orientation class. The Velocity class contains information for identifying the speed and direction of a platform. For fixed objects, values of zero are allowed. For moving objects, values are necessary to compute the current velocity.

The slots in the Velocity class support the specification of a velocity in terms of (x_0, y_0) and (x_1, y_1). The slots *x-start*, *y-start* and *x-end*, *y-end* corresponds to the vector specification. Each slot is of type Float, is required, and a single value is allowed.

The location information for the position of an Earth sensing system must currently be provided as a latitude/longitude and an altitude. Therefore, the Location class has slots corresponding to these values.

The *latitude* and *longitude* slots are both of type Integer and constrained between 0 and 360 degrees. A single value is required for both. The *altitude* slot is of type Float, its cardinality is 1:1, and its minimum and maximum values are -6.8 and 22,240.0, respectively, in statute miles. The

altitude constraints correspond to the deepest point in the Pacific Ocean (the Mariana Trench) and geosynchronous orbit, with 0.0 being sea level. This range may be expanded as needed in subsequent research and development. For a ground-based platform at sea level, a default value of 0.0 for the altitude applies.

The Orientation class describes the three-dimensional attitude of the spacecraft within its environment. These attributes may be specified as the yaw, the pitch, and the roll of a spacecraft or other physical system. Correspondingly, there are three slots in the Orientation class.

All three slots – the *yaw* slot, the *pitch* slot, and the *roll* slot – are of type Integer, have cardinality of 1:1, and are constrained to values between -180 and 180 in degrees. The *yaw* slot defaults to 0 degrees, representing a compass reading of north. The *pitch* slot and the *roll* slot also default to 0 degrees, representing a physical system that is perpendicular with the celestial body it is orbiting. For ground-based systems, the *yaw*/*pitch*/*roll* slot values of 0/0/0 are the default.

The Detector class, used to compose Sensor instances, is an ontological concept which closely follows the structure of its SensorML specification. A detector, unlike a general component, has a single input and a single output. Multiple detectors may be included in a more complex sensor description. The Detector class collects descriptive information about Earth sensing devices.

The *response_parameters* slot is also of type Class and must have one or more instances of type Response_Parameter[5]. The response parameters describe the details of how a detector performs its intended functions. Multiple response parameters are associated with a detector since a physical detector may have multiple modes of operation.

The *input* slot describes the single input value to the detector. Its type is Class of type Input (described previously). The *output* slot is similar, while being constrained to a Class of type Output (also described previously).

4.2 Example Instances

Examples of some of the instances associated with the platform classes are introduced here. Corresponding to the SWAMO testbed there are two FlatSats (satellite hardware simulators on flat table tops) for flight systems (ST-5 and the CHIPS), one satellite system (MidSTAR-1), and one ground-based mission operations center (MOC).

The MOC, MidSTAR-1, ST-5, and CHIPS are instances of the Platform class. The ST-5 and CHIPS satellites are examples of Platform instances which orbit the Earth. The FlatSats are breadboard implementations which duplicate their respective onboard systems. The MidSTAR-1 is an on-orbit flight system hosting the SWAMO software.

Since the MOC is a fixed, ground-based computer system, its direction and orientation information metadata are the default values. Its *sensors* slot is empty, but its *capabilities* slot has a list of computing resources available for planning, scheduling, and autonomic control.

The MidSTAR-1, ST-5, and CHIPS satellites have information about the orbital position, spacecraft orientation, and velocity vector to describe them spatially. They also have multiple sensors described. Similarly to a ground-based system, the satellites each have additional functional capabilities described.

While the complete details of these Platform instances are not provided here, Figure 4 shows how instance definitions fit into the overall ontology specification. In addition to the general platform characteristics, their spatial descriptions, sensor suites, and computing facilities are described with sufficient detail for an autonomic control system to manage these resources. This supports the intelligent agents with a detailed description of the capabilities for all platform types in the Sensor Web domain.

4.3 Model Ontology

The Model Ontology describes the computational models which are executed by SWAMO to complete the requested experiments on the Sensor Web. It is specified in such a way as to naturally translate to SensorML. This part of the SWAMO ontology specifies the logical aspects of the overall ontology.

The Process class shown in Figure 5 is specified in the overall SWAMO ontology above. The Model ontology diagram shows how this part of the SWAMO ontology connects and relates to the overall ontology.

The Process class is described in detail in the overall SWAMO ontology. It has two subclasses defined in detail here: the Process_Chain class and the Process_Model class. The Process_Chain class is a composite processing block with interconnected subprocesses. The interconnected subprocesses can be of the type Process_Model class (described below) or of the Process_Chain class. This allows for multiple layers of decomposition of the process chains. Also described in a process chain are process data sources for the inputs and outputs, which link other subprocesses.

The *inputs*, *outputs*, *parameters*, and *metadata* slots are defined in the Process class entity and are inherited by the Process_Chain class. Other slots in the Process_Chain class are the *system* slot, the *data_sources* slot, the *connections* slot, and the *subprocesses* slot.

[5] Details for the response parameters are provided in the SensorML weather station tutorial at http://www.botts-inc.net/vast.html.

The *system* slot associates a Process_Chain with a specific System class instance. There is a one-to-one correspondence between the two. The Process_Chain represents the logical description of a physical System instance. A particular process chain models a process (using a Process_Model) or a connection between two subprocesses (using a Link class instance). This means that a process chain is a combination of model steps and their interconnections. If the process chain models a Process_Model, it must also describe input and output values. The inputs and outputs may have optional parameters and/or data sources described. If the process chain models a connection between two subprocesses, there will be no input or output values; however, input and output process models representing the linked process chain must be described.

Other slots described in the Process_Chain class include the *data_sources* slot, which may be used to elaborate on the sources of information associated with the inputs and outputs. This information is useful for building process chains which were not originally intended to be combined. That is, this part of the SWAMO ontology supports the dynamic composition of process chains in order to collect and serve previously-undefined Earth science results.

If the Process_Chain describes a connection between two subprocesses, a *connections* slot of type Class and restricted to instances of type Link, is provided. This allows the representation of a process chain which represents the steps in a series of a process or the sequential relationships between the process steps.

Lastly, a *subprocesses* slot describes decomposition of the Process_Chain class. Its values are of type Class. If a process chain is decomposed, those values must be instances of type Process_Chain. This yields a recursive description of process chains. If a process chain is not further decomposed, it must have an associated Process_Model, which relates the inputs to the outputs of the Process_Chain. Thus, the Process_Chain description recursively defines the *operation* of a physical system to any required level of detail.

Next, the Process_Model class describes how the inputs are converted to outputs by the system being modeled. The inputs are converted to one or more outputs through the use of a process model. The process model may be in the form of a computational model implemented in MATLAB or some other modeling and simulation tool. This is captured very explicitly as the *process_method* slot.

The process method is a crucial part of computing the results of Earth sensing sensors and detectors. The *process_method* slot in the Process_Model class captures a simulation model of the computational model. Instances of the Process_Method class (described below) capture the information necessary for an intelligent agent to execute the model of the process chain at hand.

The Process_Method class has three slots used to describe the method used to model a logical process. The *implementations* slot, the *process_interface* slot, and the *algorithm* slot are all typed as String. There may optionally be multiple implementations of a single algorithm for a process method. However, there should only be a single interface to the process method. The allowed slot values are domain specific and thus not described in further detail at this time.

A process chain may be used to represent a computational function or a linkage between two computational functions. The Link class has two slots, the *input_process* and the *output_process*. Both are of type Class and both are constrained to instances of type Process. More specifically, the *input_process* and *output_process* slots identify *which* input and output of a process form the link. In this way, Link class instances, along with Process_Chain instances, form a higher-level process chain description. The subprocesses of the Process_Chain and the links define the process chain of interest.

While this class definition need not be defined in the Model ontology (it is essentially just another relationship between two arbitrary concept classes), it is necessary for a compatible SensorML specification of the model. Independent of SensorML, a Process_Chain instance would capture the input and output processes in the existing slots.

All of the Model ontology descriptions may be extended as subsequent SWAMO development warrants. Within the scope of the SWAMO project, however, this level of detail is sufficient for the intelligent agents to make decisions about executing models and managing resources.

4.4 Agent Ontology

The Agent ontology, shown in Figure 6, describes the intelligent agents of the SWAMO system and the concepts which they manage. This part of the SWAMO ontology represents the different software agents and their functionality, particularly with regard to planning, scheduling, executing, and monitoring the Earth sensing platforms, sensors, and detectors. A full description of the SWAMO architecture and its software agent components and their interactions is provided in [1].

The top-level Agent class is a superclass of the instantiable agents in the SWAMO system. The instantiable subclasses are the Executive Agent, the Planning Agent, the Repository Agent, the Interface Agent, the Resource Agent, the Execution Agent, and the Telemetry Agent. Each subclass of Agent inherits three descriptive slots from the superclass: an identifying name, the platform on which it resides, and whether or not the agent can be moved to another platform. The platform on which an agent resides may be a ground-based, aerial, or orbiting platform.

The *platform* slot has a type constraint for an instance of a Platform class (specified previously) and the value is required. The *name* slot is of type String and is required. The *mobile* slot is a Boolean value, defaulted to FALSE, and is also required. All of the slot values are limited to a single value.

The Executive Agent class defines slots for the end-user goals to be satisfied, a workflow for satisfying each goal, and a reference to a remote Agent for registering and looking up services. These slot values allow an Executive Agent to interact and coordinate with other Agents (i.e., remote Executive Agents and Repository Agents) in the Sensor Web. The *workflow* slot is constrained to be an instance of the Process_Chain class (described previously) and the *registry* slot is constrained to be an instance of the Agent class. Constraints on the *goals* slot are TBD, but are represented as a String type at this time.

The Telemetry Agent is responsible for communications between Sensor Web systems, including other flight systems and ground stations. This agent adds slots for tracking subscribers to data products, requests from other communications systems, and communications resource bridges. The *subscribers* slot is a list of one or more Agent instances, the *requests* slot is a list of one or more Task instances, and the *bridges* slot is a list of one or more communications bridges (i.e., an instance of a Bus class, a subclass of the Component class described previously). The bridges are used for inter-platform communications.

The Planning Agent class adds a slot, *schedule*, of type Class, and must point to a single instance of the Schedule class (described below). This effectively represents the experiment schedule for the specific platform. Additionally, the Planning Agent has a *time* slot, of type Time, which is a concept used to track the mission time.

The Repository Agent class adds to the Agent superclass a *models* slot with zero or more instances of a Process_Chain class (described previously). An instance of a Repository Agent must track and manage the models used on a specific platform and make the models available to Resource Agents and to Execution Agents. It is assumed that an experiment or process model description is compatible with a SensorML process chain description.

The Interface Agent is for operator monitoring and reporting. At this time, it has a lone slot, *external_interfaces*, of type String. Since at least one and possibly multiple interfaces may be supported, its cardinality is one-to-many.

The Resource Agent class adds an execution_agents slot with zero or more instances of an Execution Agent class. This slot effectively tracks and manages the potentially multiple Execution Agents on a given platform. Since Execution Agents are potentially mobile, this list may change over time and as mission conditions vary. Furthermore, an idle platform may have no active Execution Agents, so the slot value may be empty for periods of time.

The Execution Agent has the following additional slots with the associated properties: *mobile*, *current_model*, *current_task*, and *current_step*. These slots represent the fact that Execution Agents may be mobile, each Execution Agent manages one model at a time (a Process_Chain class instance), each Execution Agent may have a set of commands (a "task") which it is executing, and each Execution Agent has a specific task (a "step") within the set of tasks which it is currently executing. The intent is for the Execution Agent to track the task and step of the task being executed with the current model against the simulated execution of the task and step. Comparisons of the planned execution with the actual execution serve to monitor the overall progress of the experiment.

A Schedule class collects the activities planned for a platform. A Schedule class instance has a single *tasks* slot, of type Class, which may be populated with zero or more instances of type Task or of type Event. Zero tasks are possible if a platform completes all of its scheduled tasks and no more exist in a schedule or queue. In a Universal Daily Activity Plan (UDAP) schedule, tasks and events are intermixed. Since a UDAP schedule will be ingested by a Planning Agent and a full schedule created and managed, the Schedule for a Planning Agent must also manage both.

The Task class represents the information associated with a task in a UDAP schedule. That is, the slots defined for a Task correspond to those found in a UDAP schedule. The slots directly adapted from the UDAP schedule are for the platform identifier[6], the name of the activity or command being executed, the command type, a responsible application (essentially, a model identifier), and a times slot for the periods of time over which the command is to be executed. The responsible application is of type Class constrained to an instance of type Process_Chain. Multiple times may be specified on the task to allow for occasions in which an activity is repetitive.

An Event class is similar to a Task class, with one exception. Like the Task class, the Event class has slots for the spacecraft identifier, the name of the command generating the event, the command type, a responsible application, and a time for the event. The Event class treats the concept of time differently, however, than the Task class.

The *time* slot value is a singleton, since an event is not bounded by a starting and ending time, as is the case in a task. An event is generated in a point in time. Also, repeated events are separate instances, so multiple times are not parts of the single event concept.

[6] The slot name *spacecraft_id* is adopted from the UDAP nomenclature.

The Time_Duration class simply represents the concept of a period of time bounded by the starting and ending times. The concept of time duration is useful for planning and scheduling activities that may overlap and compete for platform resources. Some planned activities may be executed concurrently as long as they do not use the same fixed, or consumable, resources. Thus, we distinguish between the concept of time and the concept of a time duration.

The Time class has slot which corresponds to the year, Julian day, hour (in military time), minutes, and seconds. The granularity of time can be increased if necessary by adding slots for sub-second increments. Currently, one second time increments are sufficient for the SWAMO demonstrations. Also, a more comprehensive time ontology (e.g., from SWEET) can replace this part of the SWAMO ontology as required.

4.5 Sensor Observation Service Ontology

The Sensor Observation Service (SOS) is a SWE service and API for requesting and executing sensor observations. Since SOS may be used by the SWAMO agents to invoke sensing system services, its specification as part of the SWAMO ontology is useful. Other SWE services may be incorporated in subsequent research and development.

The SOS ontology, shown in Figure 7, specifies only the "mandatory" aspects of the SOS necessary for minimal compliance with the standard. The four concepts included are observations, measurements, phenomena, and observations offerings. The Observation Class describes the data recorded by an Earth sensor. It contains slots for an identifying name, a list of time intervals over which the observation(s) took place, a geographic region bounded by "from" and "to" values, a list of sensors used to capture the observation, and a method by which the observation was captured.

The identifying name is required and only one value is allowed. The *time_intervals* slot is constrained to be of type Class with allowed values of instances of the Time class. The *geographic_region_from* and *geographic_region_to* slots describe a two-dimensional area for the observation. At this time, the values are limited to strings; however, subsequent development may specify classes for an area of space. The *sensors* slot must be one or more of type Class with values of instances of the Sensor class. Lastly, the *method_reference* slot is constrained to be of type String and is required. A method reference is required for an observation to provide aspects of its provenance.

The Measurement Class relates an observation to a named phenomenon. Thus, it has two slots – *observed_value* and *phenomenon*. The *observed_value* slot is of type Class whose instances are the Observation class. The *phenomenon* slot is constrained to be of type Class of instances Phenomenon. The cardinality of both is 1:1, meaning that a single value is required.

The Phenomenon Class identifies the target of an observation. It has a lone slot, *name*, which may be used to identify some natural phenomenon to be observed. Its significance may be as an identifier for human operators and scientists. The name is required and only one is allowed.

The last class of the SOS is the Observation_Offering Class whose purpose is to logically group a series of observations into a single identifier. For example, multiple observations may be collected at the same time every day for a week. The series of observations would be grouped into an instance of an Observation_Offering. Therefore, the Observation_Offering Class has a single slot, *observations*, of type Class, and limited to instances of the Observation Class.

5. AUTONOMIC CONTROL

The SWAMO ontology provided a common representation for Sensor Web concepts and their interrelationships. The ontology describes the representation of sensor platforms, sensing devices, spatial information, capacity information, and status. Significantly, the SWAMO ontology describes the autonomic control system for the Sensor Web. The ontology includes descriptive information which is necessary for software agents to automatically compose Earth sensing services, plan the execution of those services, collect the sensor data and information, monitor the progress and results of the data collection processes, and deliver the results to science users. The SWAMO project identified the need to represent in the ontology the descriptions of the control system itself so that autonomic behavior could be provided for the Sensor Web as a whole and for individual platforms, sensor devices, and the control system agents.

The SWAMO agents make decisions based on the ontology. While the frame representation used to describe the ontology was not directly consumed and used by the software agents, the concepts were applied directly to the design and implementation of the SWAMO system. This approach facilitated a rapid development process and enabled testing of the SWAMO concepts and prototype on a flight system. Use of the ontology enabled the software agents to exchange high level information about the Sensor Web and the software agent operational status and availability.

The ontology concepts were translated into data structures within the software agents. The data structures are sharable and their interpretation is common among the SWAMO agents. The agents are able to adapt and respond to full or partial system and subsystem failures and provide autonomic behavior at all levels of the Sensor Web. These are the Sensor Web itself, sensor platforms, sensing systems, sensor devices, the SWAMO distributed control system, and the end user interactions.

6. SUMMARY OF RESULTS

The SWAMO project developed an architecture and experimental system for autonomously operating a web of sensing systems. An ontology was developed and used for collaboration between multiple, distributed sensor systems. The ontology was used by distributed agents to autonomously operate a sensor web in a laboratory testbed and an orbiting satellite. The ontology models the application domain and the agent-based control system for the sensor web. The ontology was also used to implement the distributed agent system and to guide the design and development of the sensor web control system. The result is comprehensive model not only of the application domain, but also the control system for the application domain.

A significant result is that automated systems based on semantic technologies such as ontology cannot solely rely on a representation of the application domain. In order to adapt to new missions and to recover from full or partial system or subsystem failures, a representation of the control system itself is required. The SWAMO ontology investigated how semantic technology could increase automation and enable autonomic control. By creating and sharing a common ontology, the Sensor Web is enabled to operate in an automated, distributed fashion. Additionally, as new sensing systems, Earth sensing capabilities, and service-oriented products become part of a Sensor Web, their inclusion does not have the consequence of propagating modifications throughout the entire Sensor Web. While the SWAMO ontology may not be a sensing system ontology that applies to other application domains (such as commercial air traffic control and military command and control systems), it lays the foundation for further development in semantics for autonomic control.

REFERENCES

[1] K. J. Witt, J. Stanley, D. Smithbauer, D. Mandl, V. Ly, A. Underbrink, and M. Metheny, "Enabling sensor webs by utilizing SWAMO for autonomous operations," in *Eighth Annual NASA Earth Science Technology Conference (ESTC2008)*, 2008.

[2] A. Underbrink, K. J. Witt, J. Stanley, and D. Mandl, "Autonomous mission operations for sensor webs," in *Fall Meeting of the American Geophysical Union*, 2008.

[3] M. Sánchez-Gestido, L. Blanco-Abruna, M. S. Perez-Hernandez, R. Gonzalez-Cabero, A. Gomez-Perez, and O. Corcho, "Complex data-intensive systems and semantic Grid: applications in satellite missions," in *Second IEEE International Conference on e-Science and Grid Computing (e-Science'06)*, 2006, pp. 158-158.

[4] R. Wright, M. Sánchez-Gestido, A. Gómez-Pérez, María S. Pérez-Hernández, R. González-Cabero, and O. Corcho, "A semantic data grid for satellite mission quality analysis " in *The Semantic Web (ISWC 2008)*, 2010.

[5] M. K. Pulvermacher, D. L. Brandsma, and J. R. Wilson, "A space surveillance ontology," MITRE Corporation, Bedford, Massachusetts 2000.

[6] A. Uszok, J. Bradshaw, R. Jeffers, N. Suri, P. Hayes, M. Breedy, L. Bunch, M. Johnson, S. Kulkarni, and J. Lott, "KAoS policy and domain services: Toward a description-logic approach to policy representation, deconfliction, and enforcement," *Proceedings of the IEEE Workshop on Policy 2003*, 2004.

[7] P. Zetocha and J. Ortiz, "PEACH - An agent for increased space operations automation," in *Proceedings of the SpaceOps 98 Conference* Tokyo, Japan, 1998.

[8] M. B. Pantoquilho, "A space environment information system for mission control purposes: System analysis and data integration design," Monte da Caparica, Portugal: Universidade Nova de Lisboa, 2005.

[9] K. K. T. Thanapalan and S. M. Veres, "Agent Based Controller for Satellite Formation Flying," in *Proceedings of the 2005 International Conference on Intelligent Sensors, Sensor Networks and Information Processing Conference*, 2005, pp. 385-389.

[10] C. Schlenoff, S. Balakirsky, M. Uschold, R. Provine, and S. Smith, "Using ontologies to aid navigation planning in autonomous vehicles," *The Knowledge Engineering Review*, vol. 18, pp. 243-255, 2003.

[11] R. Provine, C. Schlenoff, S. Balakirsky, S. Smith, and M. Uschold, "Ontology-based methods for enhancing autonomous vehicle path planning," *Robotics and Autonomous Systems*, vol. 49, pp. 123-133, 2004.

[12] A. Nilsson, R. Muradore, K. Nilsson, and P. Fiorini, "Ontology for robotics: A roadmap," in *International Conference on Advanced Robotics (ICAR 2009)*, 2009, pp. 1-6.

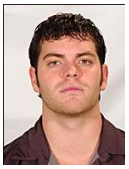
[13] L. Dennis, M. Fisher, A. Lisitsa, N. Lincoln, and S. Veres, "Satellite Control Using Rational Agent Programming," *IEEE Intelligent Systems*, vol. 25, pp. 92-97.

BIOGRAPHY

Mr. Al Underbrink has over eight years experience developing artificial intelligence applications for cyber security and 20 years experience in systems development. Mr. Underbrink has deep experience in artificial intelligence, expert systems, knowledge based systems, intelligent agents, robotics, automation, and autonomy. Mr. Underbrink is principal investigator for development the Sentar veriScan product, a tool for automated execution of multiple analysis tools to produce a risk assessment and recommendations on how to proceed. He holds BS and MS degrees in Computer Science from Texas A&M University.

Andrew Potter is Director of R&D with Sentar, Inc., in Huntsville, Alabama, USA. His work in ontologies includes development of a variety of technologies including multi-agent systems, cyber security, knowledge representation languages, and knowledge authoring tools. In a recent project he led development of a multi-agent system for cyber security situation awareness and policy-based automated response. Other research interests include mass collaboration, discourse-based reasoning theory, and asynchronous learning systems. Potter has PhD from Nova Southeastern University, an MLS from the University of Alabama, and an MAS and BA degrees from the University of Alabama in Huntsville.

Mr. Kenneth J. Witt is currently a Program Manager at the West Virginia High Technology Consortium Foundation in Fairmont, WV. He holds BS degrees in Electrical Engineering and in Computer Engineering from West Virginia University. He has been developing software in the commercial, DoD, and government areas for the last 20 years. Most recently, he has been developing applications for NASA's Goddard Space Flight Center and was the Principal Investigator for the SWAMO research grant. He has developed applications for immersive, scientific visualizations, mission operation center software, and Sensor Web enabling frameworks.



Mr. Jason Stanley is currently a Senior Software Engineer at the West Virginia High Technology Consortium Foundation in Fairmont, WV. Mr. Stanley holds a B.S. in Electrical Engineering, B.S. in Computer Engineering, and M.S. in Software Engineering from West Virginia University. Mr. Stanley has over nine years experience developing software systems for NASA and DoD programs. Most recently, he acted as the Technical Lead for SWAMO intelligent agent framework.

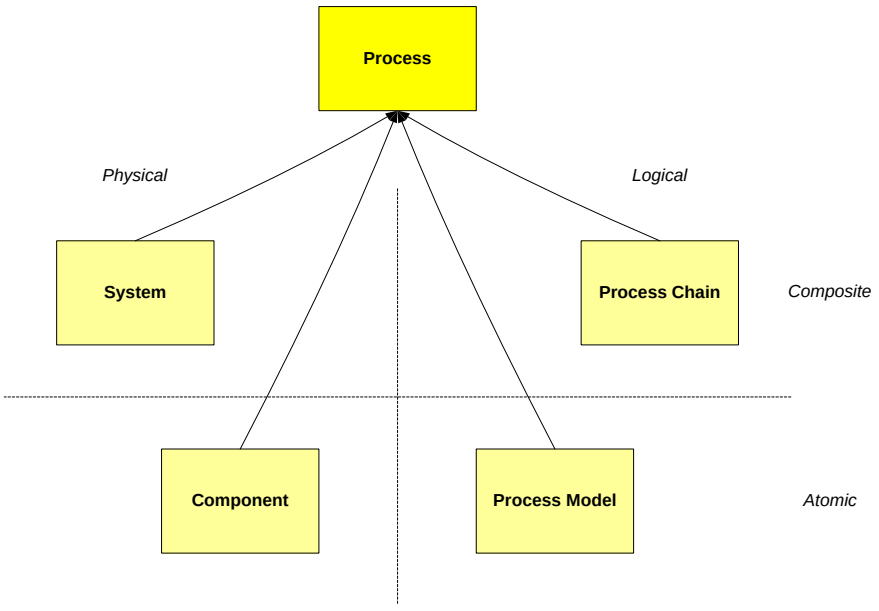


Figure 1. The SensorML view of sensing systems

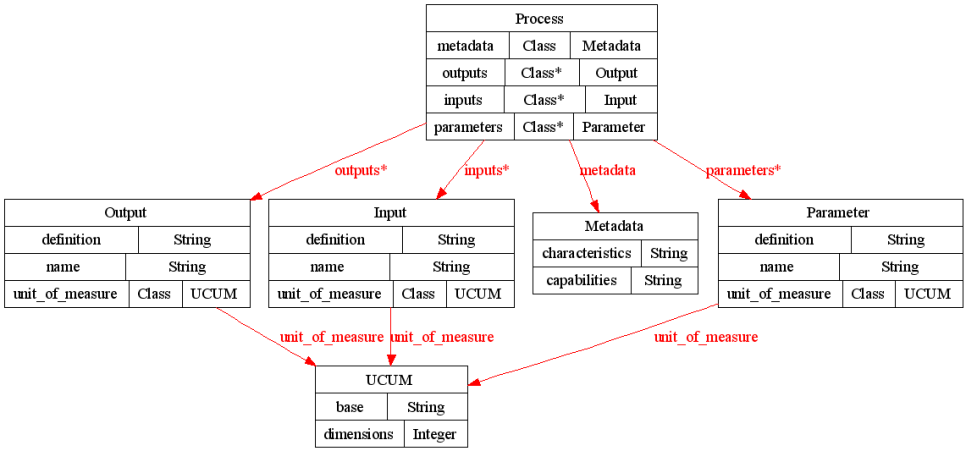


Figure 2. High level SWAMO ontology classes

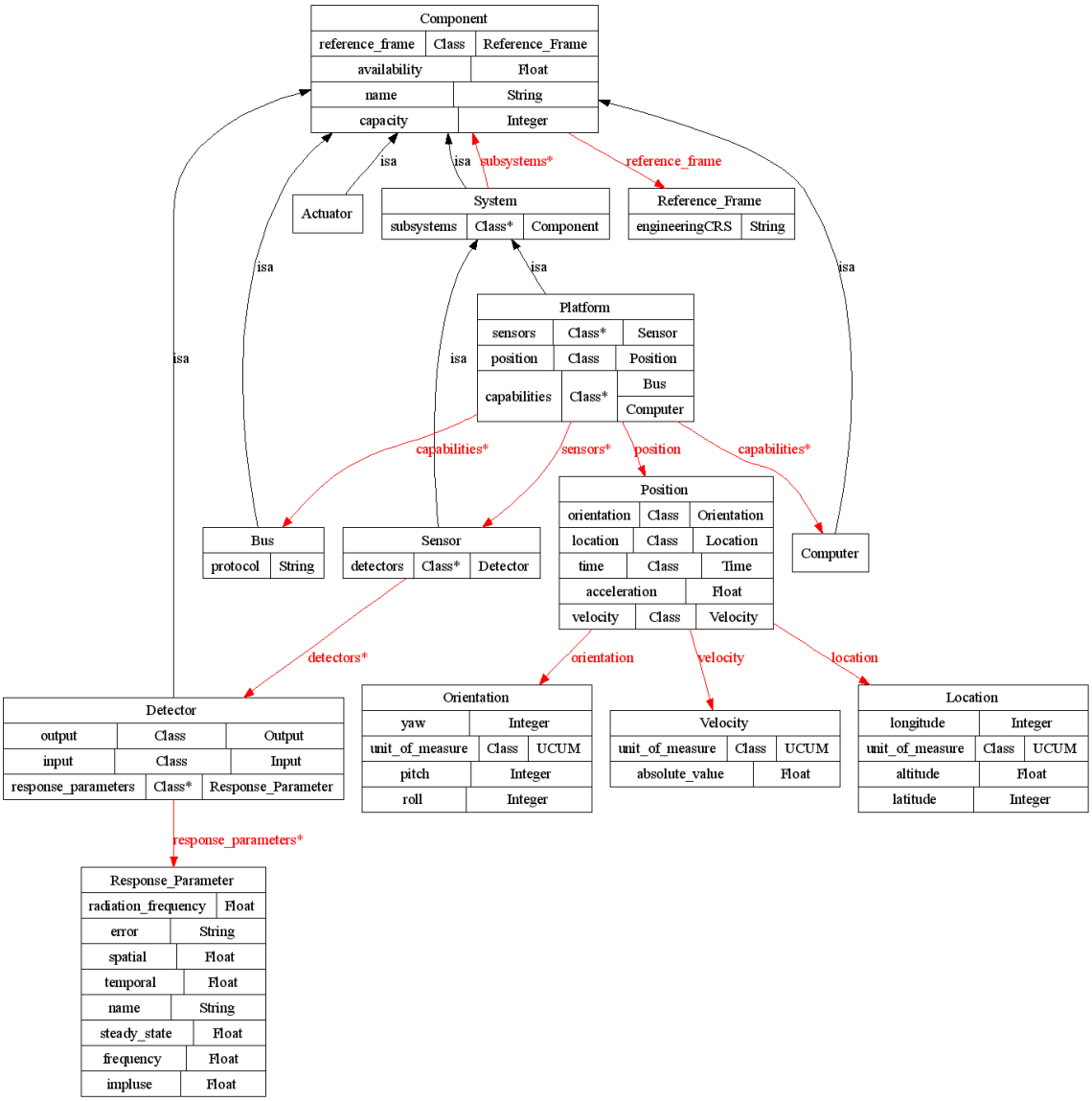


Figure 3. Platform ontology

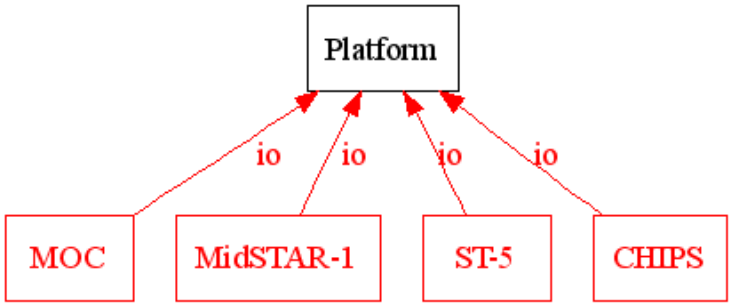


Figure 4. SWAMO platforms relating to the testbed

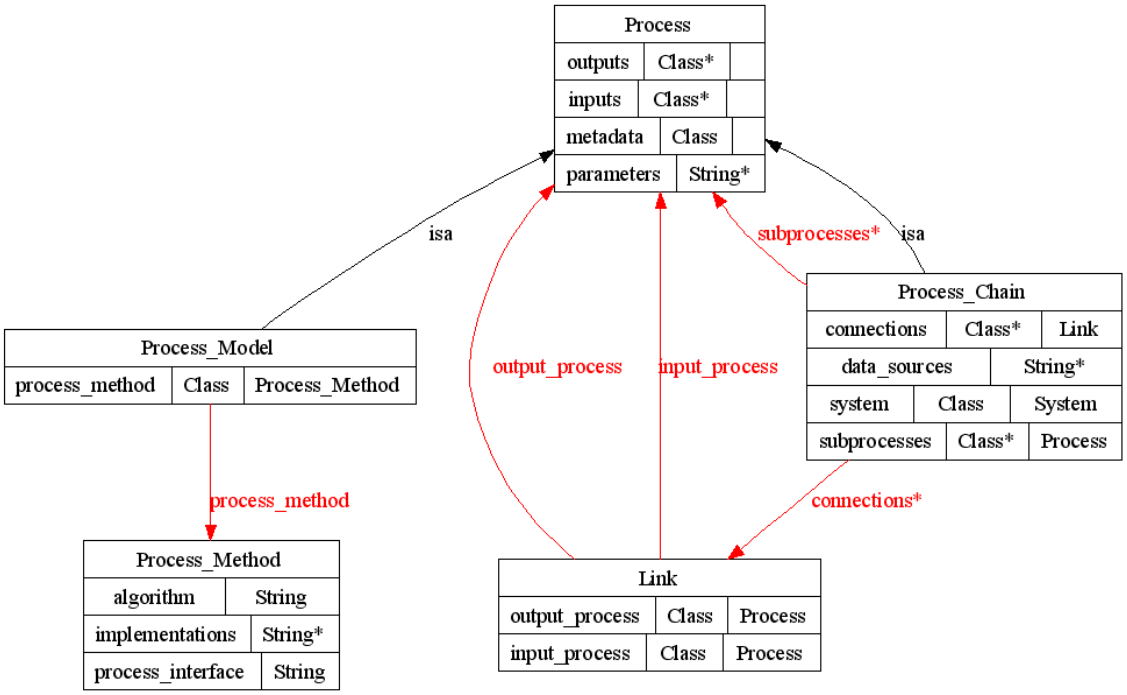


Figure 5. Model ontology

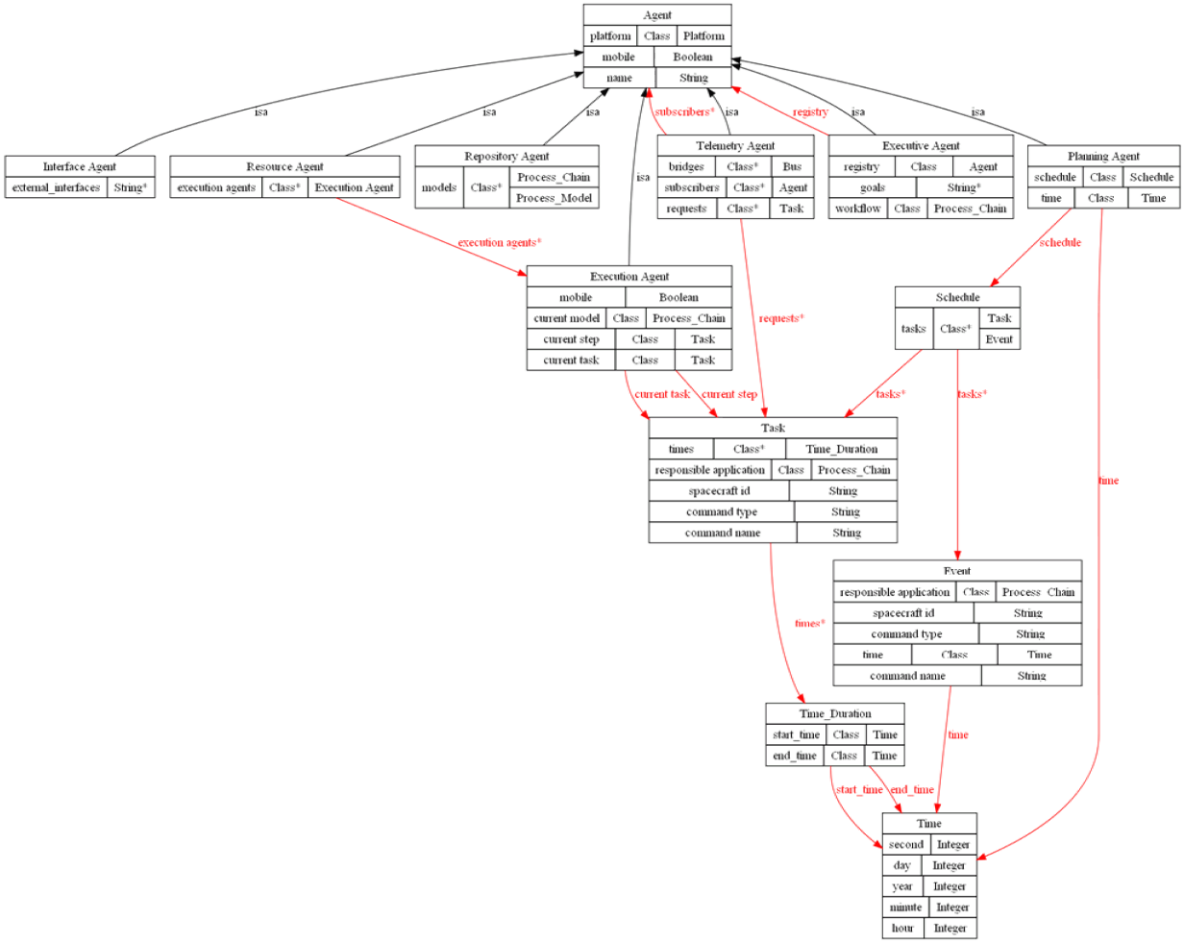


Figure 6. Agent ontology

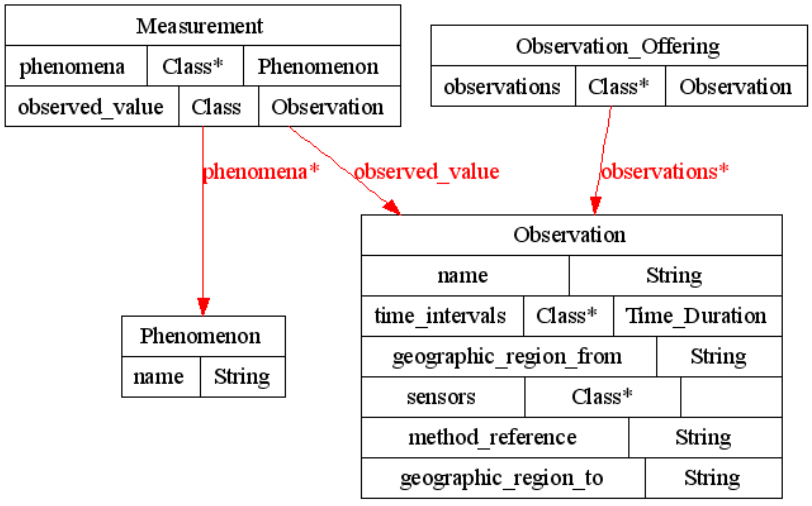


Figure 7. SOS ontology