Proceedings of the 2020 Winter Simulation Conference
K.-H. Bae, B. Feng, S. Kim, S. Lazarova-Molnar, Z. Zheng, T. Roeder, and R. Thiesing, eds.

SIMULATING RE-CONFIGURABLE MULTI-ROVERS FOR PLANETARY EXPLORATION USING BEHAVIOR-BASED ONTOLOGY

Justin Jose
Divye Singh
Amit Patel
Harshal G. Hayatnagarkar

Engineering for Research,
ThoughtWorks Technologies India,
6th Floor, Binarius Building, Deepak Complex
Pune, Maharashtra 411006, INDIA

ABSTRACT

For planetary explorations, the space agencies have usually sent single robotic rovers to complete missions. An alternative approach is to send multiple rovers, which can insure against failure of one or more rovers. Planning for a multi-rover mission has its own challenges, and simulations can aid in identifying and addressing such challenges. In this paper, we present an ontology-based approach to simulate a multi-rover planetary exploration mission, with a focus on resilience, adaptation, heterogeneity, and reconfigurability. We present an ontology that describes multiple rovers along with an inventory of their parts shipped with a lander. Our approach shows that having the ontology-based simulations help in complex scenarios such as to loan parts from inventory, and salvaging a damaged rover for good parts.

1 INTRODUCTION

Planetary exploration missions designed for landing expeditions typically consist of an orbiter, a lander and a rover. These systems are remotely controlled from an Earth station. The current state-of-the-art in remote robotic exploration is a single large rover deployed at a site of interest. A rover is usually built as a large, feature-rich, robust, heavy, and extremely expensive cyber-physical system (Seeni et al. 2010). These rovers move slowly owing to their large sizes, and to the unknown and uneven terrains, and try to explore only relatively large and open areas.

Another factor that restricts speedy exploration is the lack of autonomy. Mars Space Laboratory, sent to Mars in 2011, requires humans-in-the-loop to plan sequences of actions for each sol based on down-linked data (Grotzinger et al. 2012; Gaines et al. 2016). These sequences need to be synchronized over long interplanetary distances, and have a latency running in minutes (Gaines et al. 2016). Some may argue in favor of granting some degree of autonomy to a rover (Fink et al. 2015), whereas others may argue against it as the only rover in charge of an exploration is too precious to be left autonomous. Can this trade-off be salvaged by sending multiple rovers? In addition, we ask questions such as:

1. Will all rovers be identical, or will some rovers have some heterogeneity and unique skills?
2. How can we handle damage to the rovers? Can we repair a rover by replacing parts from an inventory with the lander? Can parts be borrowed from other nearby rovers?
3. If a rover receives irreparable damage, can we salvage the usable parts of that rover?

Previously, (Vaquero et al. 2018; Estlin et al. 1999; Skelton and Pang 2008; Fink et al. 2015) discussed the architectures of multi-rover systems for space exploration, their benefits, challenges and approaches. While multi-rover systems solve a lot of these challenges, they bring in some of their own, especially for coordination and knowledge sharing among the rovers (Isarabhakdee and Gao 2009; Tumer and Agogino 2005; Skelton and Pang 2009). Knowledge sharing using a common ontology enables agents to communicate via a shared and disambiguated vocabulary, and perhaps even reason on top of this knowledge (Torres and Wijnands 2011). We presume that each mission has different goals, and thus has its own coordination and knowledge sharing needs. Therefore, we see a merit in building an agent-based simulation of a prospective mission to discover those needs and accordingly to help build a system (Shafto et al. 2012).

Ability to reconfigure a robot is another desirable quality to be had as an insurance against failure of a part, which may render an entire robot unusable. Earlier work (Patil et al. 2013) has explored this capability in the physical robots, and mentions a need to work on software architecture. An essential component of this architecture is the description of robot parts, assemblies, and the process of configuration. Description logics such as Web Ontology Language (OWL) help to capture and reason about the description. While ontology for autonomous robots have been explored (Olszewska et al. 2017) before, it still remains a good problem to tackle in different application areas. In this paper, we focus on agent-based simulation of heterogeneous multi-rover system designated for planetary exploration, with a focus on achieving ability to re-configure via behavior-based ontology. Based on this ontology, we demonstrate rovers completing assigned tasks even when a rover gets damaged, in following ways:

- A rover gets a damaged part replaced from the inventory.
- An irreparable rover is salvaged for its working parts.
- A rover loans a functioning part from another rover to complete its assigned task. However, in this paper we do not loan parts from fully functioning rovers.

For these goals, the common ontology mentioned above, need to be further augmented to capture descriptions of rover parts, their assemblies, capabilities associated to specific parts and their tasks, mission details, and so on. According to (Tolk 2013), a simulation model needs to capture the semantics of the systems and processes using ontology. *DISim*, an ontology for modeling and simulation of data integration processes in the biomedical domain, focuses on modelling and simulating web integration tasks from heterogeneous resources (Sernadela et al. 2015). Automated adaptation is another complex problem in this space, and (Lattner et al. 2010) presented a knowledge-based approach to automated adaptation in a simulation model. For ontology and domain modeling in general, (Clancey 1993) observes that knowledge representation must not restrict the knowledge of mechanisms of the system. Complete knowledge modeling should be a result of domain knowledge along with the behavior of an intelligent agent in its environment. Following this, (An and Christley 2011) have shown how agent-based modeling framework can go beyond structural knowledge and capture behaviors as well. In their work, they have mentioned that in order to achieve dynamic knowledge representation, there is a need to extend the relational predicate logic system to *actionable predicate logic systems*. (Guerrero et al. 2005) in their work have demonstrated the use of OWL and SWRL to capture the management information and behavior in a single network management information model. Further, when it comes to modeling processes, using the idea of *Function-Behavior-Structure Ontology* can provide higher-level semantics in their representation (Gero and Kannengiesser 2007).

In the rest of this paper, we discuss the high-level design of the simulation, followed by the ontology to describe rover parts, detailed rover assembly, inventory, mission plans, tasks, and matching parts to tasks. Thereafter we describe simulation scenarios to demonstrate how the ontology design helps in re-configuration of rovers during a mission execution. We share a few observations related to decisions in modeling and technical design, and finally conclude with a note about future work.

2 SYSTEM DESIGN

An agent-based simulation model requires description of various agents, their structural and behavioral composition, and behavioral rules for invoking a specific behavior. It is a non-trivial exercise to model agents with heterogeneous skills and capabilities, especially towards reconfigurability. In our simulation model, we have heterogeneous agents such as rovers of different skills, the lander, and other robots for management and reconfiguration. The structure of agents including their skills, parts, and assemblies being a declarative description is captured in the ontology. The agent behaviors being imperative in nature are part of the simulation engine code in the Python language. The ontology bridges the references of these behaviors in the source code to the respective agents, to be later consumed during the simulation process. Currently, the missions are also modeled in the ontology because we see them as simple sequences of tasks.

In the rest of this section, we will discuss design of simulation engine components including ontology. The simulation process and scenarios are discussed in section 3.

2.1 Simulation Engine Design

The simulation engine is designed as a state transition machine as shown in Figure 1, where each state consists of the triple —`<agent, behavior,executable>`. The `executable` represents a scenario or a task, and the `agent` executes each `executable` using the `behavior` referred in the ontology and defined in the Python source code.

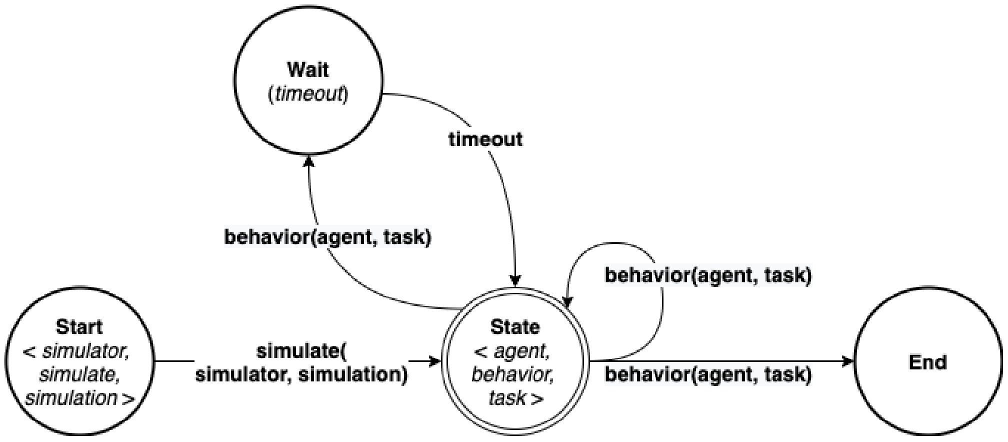


Figure 1: Simulation engine state transition.

The simulation engine design comprises of four classes: `Simulator`, `Reasoner`, `Behavior` and `State`. The `Simulator` class orchestrates simulation using `State`, `Reasoner` and `Behavior` instances as shown in Figure 2.

The class `Behavior` is responsible for managing behaviors defined as functions in Python language, and for linking these definitions to the behavior instances in ontology. The `Behavior` class in Python mirrors the `Behavior` class in the ontology. A developer must ensure the mirroring by explicitly capturing a behavior's identifier in the ontology and the Python function as shown in Figure 3 and Figure 4. Then the simulation engine knows which two `Behavior` instances are equivalent, and then using Python's annotations combined with *Decorator* design pattern (Gamma 1995), it keeps a mapping of an identifier to the Python function. Decorator is a design pattern that allows adding new behaviors to objects dynamically by placing them inside special wrapper objects. This approach helps in externalizing the behaviors via annotated function definitions as concrete implementations of their respective ontological behavior. These annotations add meta definitions to the function, thus allowing them to be redefined in the context of the simulation engine. Currently, the decorator accepts the identifier of a behavior service in a URI format (Masinter et al. 2005). The URI acts as key in a dictionary of behaviors that the `Simulator` looks for.

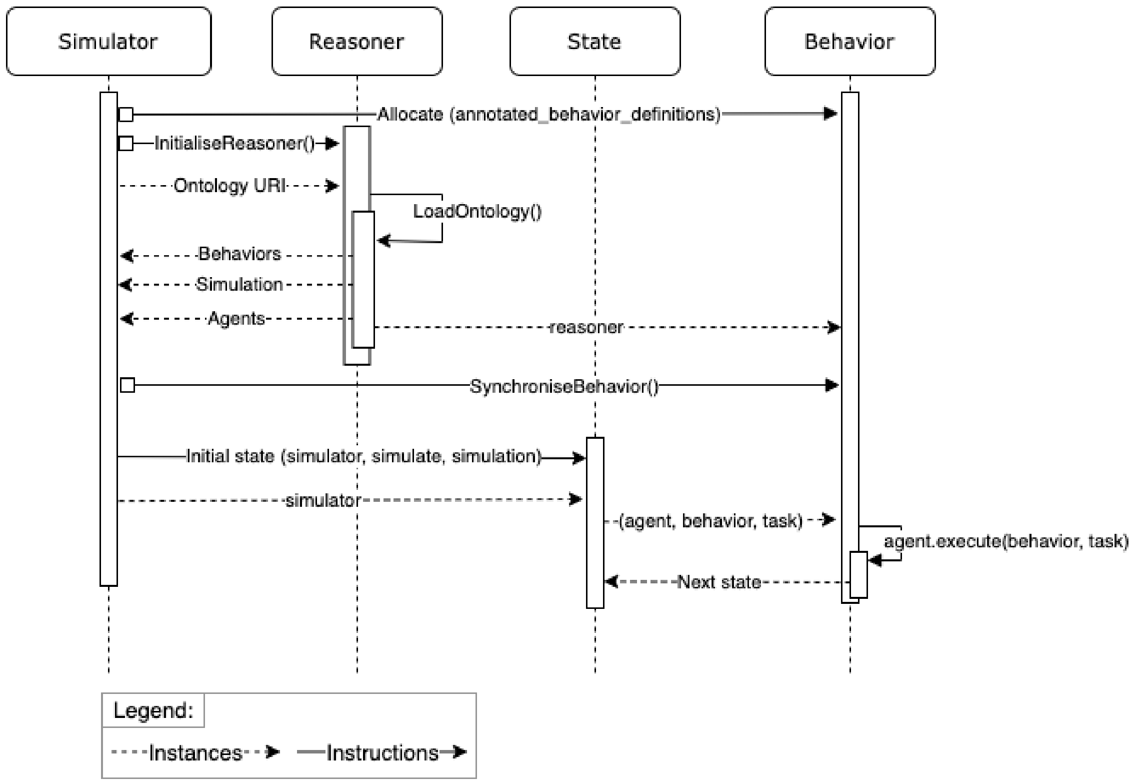


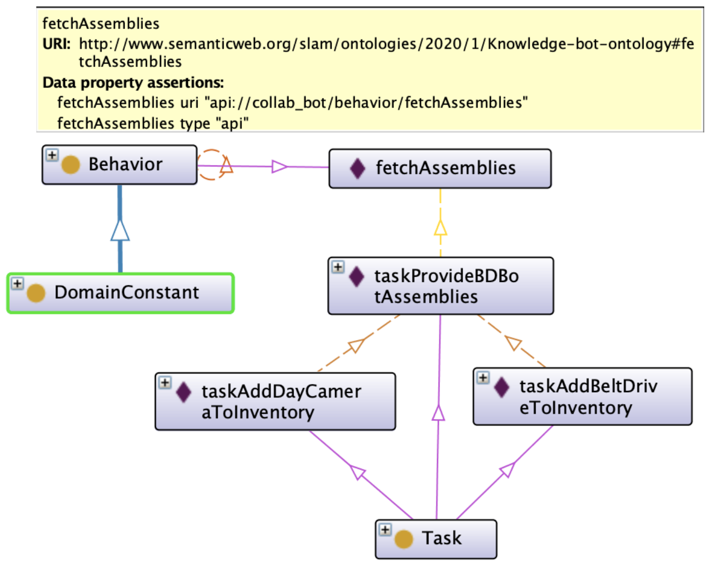
Figure 2: Simulation engine control flow.

Once found in the dictionary, the `Simulator` then invokes the service to actually perform the expected `Behavior`. A URI offers benefits such as encoding the protocol of invocation (such as HTTP, RPC, or a native API call), owner or host of the service, and the service along with arguments. Currently, the framework supports only the API call invocation.

The `Reasoner` class is responsible for managing, reasoning, and querying the ontology with the help of `owlready2` (Lamy 2017) and the `pellet` reasoner (Sirin et al. 2007). *Owlready2* is a Python package for ontology-oriented programming. It allows access to the entities of an OWL ontology as if they were objects in the python programming language. It provides an out-of-the-box support for the reasoner which is very helpful while doing OWL reasoning and also provides support for running SPARQL queries. SPARQL (a recursive acronym for SPARQL Protocol and RDF Query Language) is a language to query and manipulate data store in the form of a directed, labeled graph known as Resource Description Framework (World Wide Web Consortium 2013). SPARQL is inspired from the popular database query language SQL.

The `Simulator` initiates a simulation with the triple `<simulator, simulate, simulation>`. The `simulate` behavior is an externally supplied behavior to set the course of the simulation. *State* of the simulation is affected by a behavior, and each behavior returns a set of next states. The `State` class converts the simulation into a directed acyclic graph (DAG) of states, which is traversed in a breadth-first manner. Newer states obtained after execution of the current state are added to the pareto front. This pareto front serves as the stack of next states, and gets modified through the course of the simulation. The `State` class provides two special states, `NIL` state, and `WAIT` state. The `NIL` state marks the end of a simulation path. The `WAIT` state puts the execution of `<agent, behavior, executable>` on hold for a timeout duration.

The simulation engine is written on top of `SimPy` library (Muller, Vignaux, and Chui 2020), which is used for resource management and clock synchronization.


fetchAssemblies
URI: http://www.semanticweb.org/slam/ontologies/2020/1/Knowledge-bot-ontology#fetchAssemblies
Data property assertions:
fetchAssemblies uri "api://collab_bot/behavior/fetchAssemblies"
fetchAssemblies type "api"

Behavior
fetchAssemblies
DomainConstant
taskProvideBDBotAssemblies
taskAddDayCameraToInventory
taskAddBeltDriveToInventory
Task

Figure 3: Relationships (with `uri` data-property value in tooltip).

```python
@Simulator.behavior('api://collab_bot/behavior/fetchAssemblies')
def fetchAssemblies(manager, task, simulator=None, *args, **kwargs):
    dependent_tasks = task.hasPreviousTask
    if not all(dep_task.done for dep_task in dependent_tasks):
        return State.WAIT

    if task.done:
        return State.NILL

    queries = task.query
    namespace = task.name_space
    query_result = [
        simulator.reasoner.run_query(query, namespace)
        for query in queries
    ]
    assemblies = [
        result[0] for results in query_result for result in results
    ]

    for next_tasks in task.hasNextTask:
        next_tasks.assemblies = assemblies
    task.done = True
    return get_next_tasks(task)
```

Figure 4: Python implementation.

2.2 Ontology Design

In this section, we discuss how the ontology caters to structural and behavioral descriptions of the simulation of planetary exploration. For this arrangement, we have defined an ontology having two logical parts, namely *Simulation Ontology* and *Planetary Exploration Ontology*. The ontology is implemented using *Descriptive Logic (DL)* profile of Web Ontology Language 2 (OWL2) (World Wide Web Consortium 2012).

2.2.1 Simulation Ontology

The simulation ontology is the ontological construct of the simulations and consists of two entities:

- **Simulation** class defines the simulations to be run. It is the bridge between the domain ontology and the simulation ontology. The instances of `Simulation` are stimulated by the `Simulator` instance.
- **Simulator** is a singleton class (which means it has only one instance), and its instance becomes the simulator agent with the `simulate` behavior.

2.2.2 Planetary Exploration Ontology

To describe the domain of planetary exploration, especially using multiple rovers, we define following classes.

- **DomainConstant** represents the domain-specific constants and singleton classes. These components describe the environment for planetary exploration, and help in defining simulation scenarios. Following are the domain constants, schematically shown in Figure 5.
  - **Action** defines actions that `Bot` instances perform during simulation. Examples are excavation and exploration.
  - **Terrain** represents different terrains along with their properties encountered during the planetary exploration and excavation.
  - **TimeOfDay** describes the daytime and nighttime luminosity.
  - **Skill** represents various capabilities that a `Bot` requires to perform an `Action`, to traverse on a `Terrain`, or to view during a `TimeOfDay`.
  - **Behavior** describes an agent behavior as discussed earlier Section 2.1.

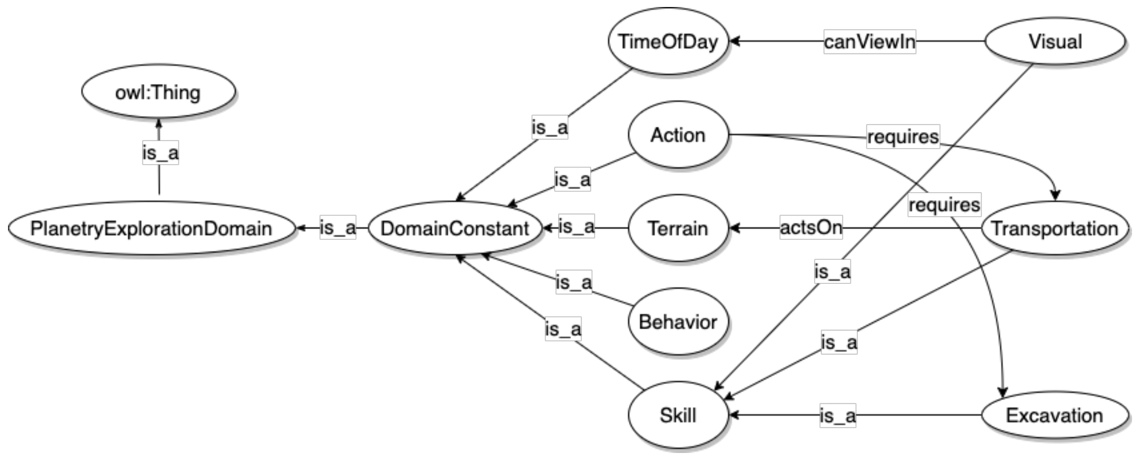


Figure 5: Taxonomy of `DomainConstants`.

- **DomainThing** subclasses are instantiated classes. The following are the subclasses of `Domain-Things` as shown in Figure 6.

- **Containable** is an abstraction for the holding areas namely, `inventory` and `parkingLot`, which have the capability to hold `Part`, `Assembly` and `Bot` instances. These are managed by `Managers`. The `inventory` contains workable atomic components - available in large finite quantity, required for assembling any `Bot` from a predefined schema. The `parkingLot` holds the assembled `Bots` awaiting tasks.
- **Agent** is the super-class for the `Bot`, the `Manager` and the `Orchestrator` classes.
  - **Bot** consists of a collection of specialized rover definitions. `Bot` instances are responsible for undertaking planetary exploration tasks.
  - **Orchestrator** gets assigned to a `Mission`. The `Orchestrator` has behavior `execute` which enables it to extract the `Task` instances from the mission, and return a set of next States composed of - an agent assigned to the task, a behavior required for the task, and the task itself.
  - **FactoryManager** has behavior `createAssembly` and `createBot` to create new instances of `Assembly` and `Bot` respectively.
  - **InventoryManager** manages the `inventory` based on the addition or removal of parts or assemblies. It also has behaviors required to provide parts and assemblies to the `FactoryManager`.
  - **ParkingLotManager** is responsible for managing a `parkingLot` and has behaviors required in order to add `Bots` to a `parkingLot`, manage `Task` allocation, perform health diagnosis, repairs and part-salvaging on the `Bot`.
- **Part** and **Assembly** classes represent individual parts and their assemblies respectively. `Bot` consists of multiple `Assembly` instances, and in turn each `Assembly` instance consists of multiple `Part` instances.
- **Executable** is the super-class for `Mission` and `Task` classes. Each `executable` instance represents an atomic operation that an agent can perform. Many instances of `executable` can be linked together to form a control flow.

3 SIMULATIONS AND RESULTS

In this section we will discuss the overall simulation flow along with two different simulation scenarios for planetary exploration, namely *repairing damaged rovers*, and *salvaging parts and assemblies from irreparable damaged rovers*.

3.1 Simulation Flow

The simulation scenarios are designed as missions, and each mission has multiple tasks. The tasks are assigned to an agent, and are defined by a behavior. The simulation works in two phases: First phase is to build a rover, and to perform the excavation/exploration task, and the second phase is to handle a damaged rover. Based on how the damaged rover is dealt, we describe here two simulation scenarios, which are instantiated with the `landerMissionControl` as an instance of the Orchestrator, `factoryManager`, `inventoryManager` and `parkingLotManager` as instances of the `Manager`, and parts required for assembling a rover. Figure 7 and Figure 8 represent the control flow for building a rover, and performing an exploration or excavation task, with *Repair* and *Salvage* boxes representing the repair and salvage activities respectively.

1. The simulator agent starts the simulation by assigning the mission to the `landerMissionControl`. The agent achieves this by preparing the state triple – `<landerMissionControl, execute, missionCreateBDBotDay>`, which triggers the `execute` behavior of the `landerMission-Control` on the mission instance.

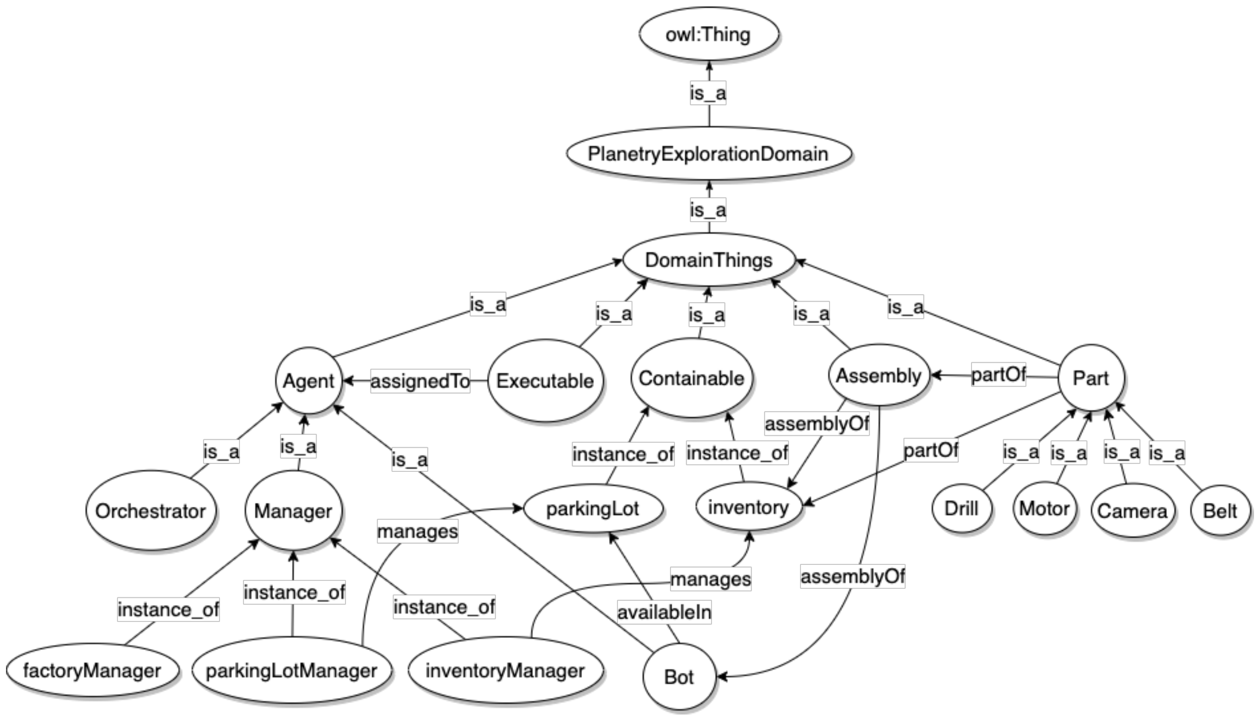


Figure 6: Taxonomy of DomainThings.

2. The `landerMissionControl` executes the mission by extracting the next tasks from the mission. The mission consists of two tasks, both assigned to the `inventoryManager`. The `landerMissionControl` hands over the tasks to `inventoryManager`.
3. The `inventoryManager` obtains the parts required to build `BeltDrive`, `DayCameraRig` and `DrillerArm` assemblies as requirements for the building a belt drive bot having the capabilities to excavate rocky terrain during day time. These parts include motors, belt drives, cameras and drills. Once the task of obtaining the parts is completed, the `inventoryManager` obtains the next task, which is assigned to the `factoryManager`.
4. The `inventoryManager` provides the factory manager with the parts along with the task, which directs the `factoryManager` to build the assemblies - `BeltDrive`, `DayCameraRig`, and the `DrillerArm`. The `factoryManager` proceeds to build these assemblies and returns it to the `inventoryManager` as directed by the next task.
5. inventoryManager adds the newly created assemblies to the inventory and makes them available for rover construction. The next task directs the `inventoryManager` to provide the assemblies required for Belt Drive bot to the `factoryManager`.
6. With assemblies required for the `BeltDriveBot`, the `factoryManager` proceeds to build the `BeltDriveBot`.
7. The newly constructed bot is sent to the `parkingLot`, where it undergoes a health diagnosis. As the rover is in pristine condition, the `parkingLotManager` marks it as available for task allocation. This marks the end of the first mission.

In a hostile environment the rover runs the risk of damaging parts during the execution of a task. We represent this by providing a `part_life` property to the rover parts. This brings us to the second part of the simulation scenario which would be discussed below.

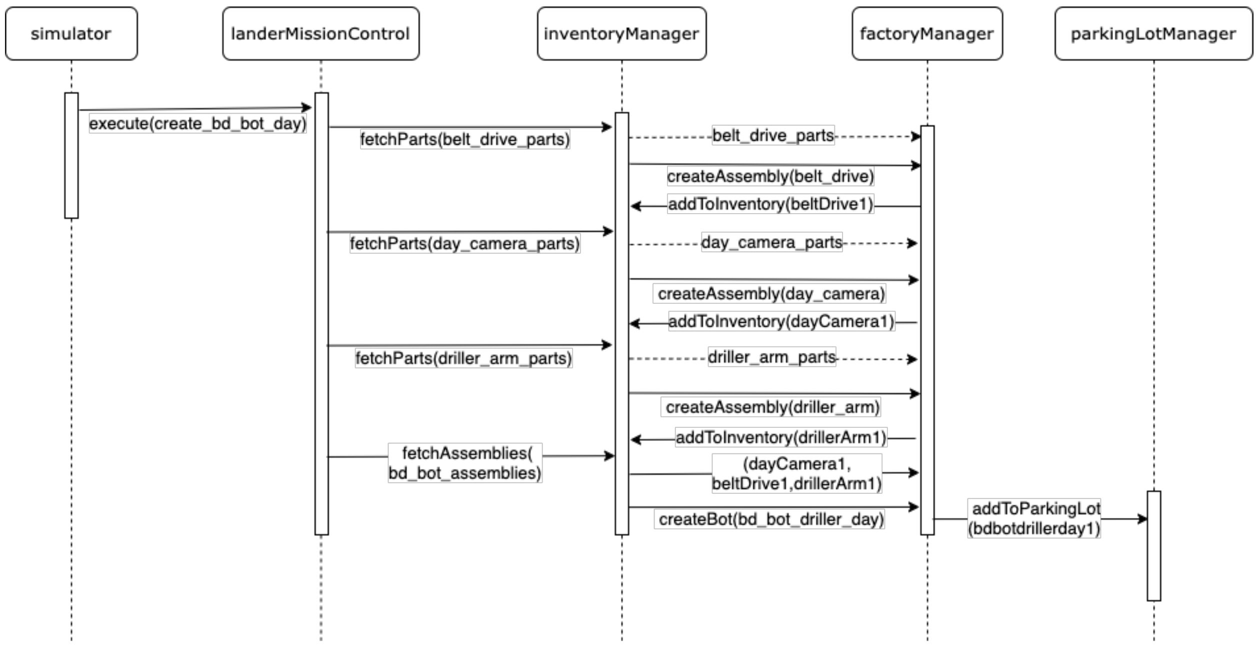

simulator
landerMissionControl
inventoryManager
factoryManager
parkingLotManager
execute(create_bd_bot_day)
fetchParts(belt_drive_parts)
belt_drive_parts
createAssembly(belt_drive)
addToInventory(beltDrive1)
fetchParts(day_camera_parts)
day_camera_parts
createAssembly(day_camera)
addToInventory(dayCamera1)
fetchParts(driller_arm_parts)
driller_arm_parts
createAssembly(driller_arm)
addToInventory(drillerArm1)
fetchAssemblies(bd_bot_assemblies)
(dayCamera1, beltDrive1,drillerArm1)
createBot(bd_bot_driller_day)
addToParkingLot (bdbotdrillerday1)

Figure 7: Simulation flow - building bot.

3.1.1 Repairing Damaged Rovers

In the second part of the simulation as represented in Figure 8, the newly constructed rover is allocated an exploration task, which is handed over by the `parkingLotManager`. The rover dissociates itself from the `parkingLot` in order to perform the task, during which the parts experience wear and tear. The rover returns back to the `parkingLot` after completing the task, where the `parkingLotManager` performs some health diagnosis on it. The `healthDiagnosis` behavior of the `parkingLotManager` identifies that one of the parts of the rover has been damaged and requires repairs. The `parkingLotManager` proceeds with the repair on the rover by first identifying the damaged parts and assemblies, and then obtaining a new set of parts and assemblies in lieu of the damaged ones with the help of the inventory manager. The newly obtained parts and assemblies replace the damaged parts and assemblies in the rover. The repaired rover returns back to the `parkingLot` where it awaits new tasks. This marks the end of the first simulation scenario.

3.1.2 Salvaging Parts and Assemblies from Damaged Rovers

Similar to the previous section, the rover is allocated an excavation task. Rover parts and assemblies undergo wear and tear during the execution of the task. Once the task is completed the rover returns to the `parkingLot` where the `parkingLotManager` performs a health diagnosis on the rover. The `healthDiagnosis` behavior of the `parkingLotManager` labels the rover damaged beyond repair. The `parkingLotManager` then proceeds to salvage the good parts and assemblies from the damaged rover, and returns these salvaged parts and assemblies to the inventory.

3.2 Reproducing the Results

These results can be reproduced by cloning the following Git repository and using the tag *WSC_2020_V1.0* https://github.com/t3pleni9/KnowledgeSimulator

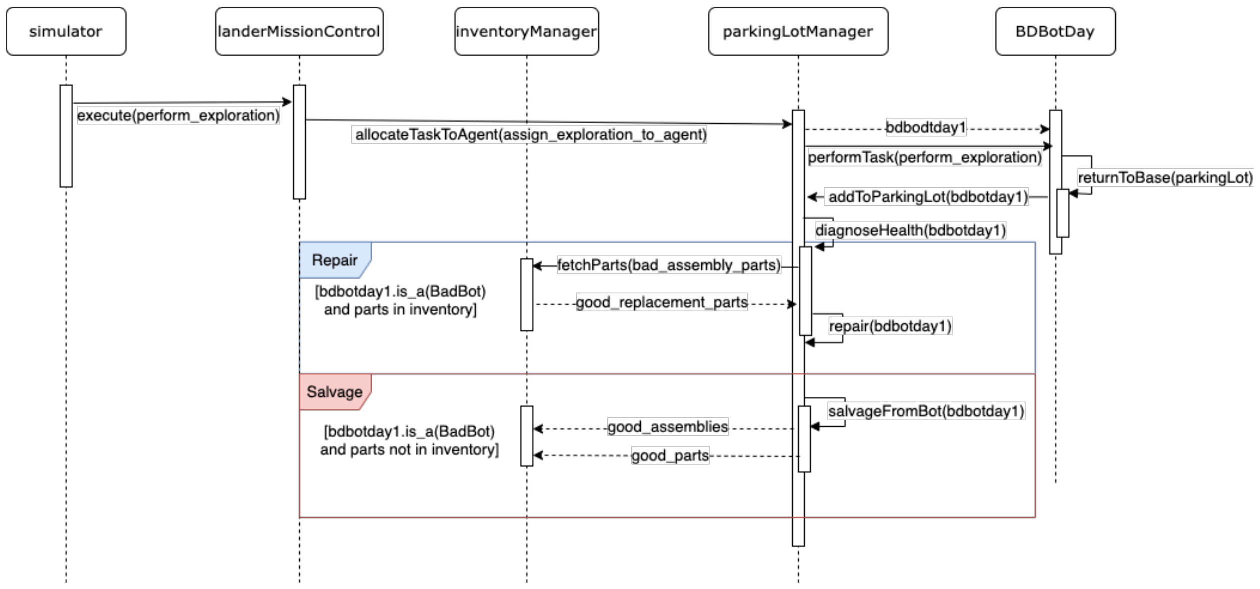

simulator
landerMissionControl
inventoryManager
parkingLotManager
BDBotDay
execute(perform_exploration)
allocateTaskToAgent(assign_exploration_to_agent)
bdbodtday1
performTask(perform_exploration)
returnToBase(parkingLot)
addToParkingLot(bdbotday1)
diagnoseHealth(bdbotday1)
Repair
[bdbotday1.is_a(BadBot) and parts in inventory]
fetchParts(bad_assembly_parts)
good_replacement_parts
repair(bdbotday1)
Salvage
[bdbotday1.is_a(BadBot) and parts not in inventory]
salvageFromBot(bdbotday1)
good_assemblies
good_parts

Figure 8: Simulation flow - repair and salvage.

4 OBSERVATIONS

During the design and development of the ontology model and the overall simulator system, we made a few observations. We observed that the multi-agent simulation required a non-monotonic logic reasoning, as addition of newer ontology nodes during the course of the simulation modified a previously held notion. Semantic Web Rule Language (SWRL) supports monotonic inferences only, and this constraint hindered the simulation and drove the ontology into an inconsistent state. We circumvented the need for SWRL rules, by capturing the transition rules as behaviors defined as external APIs.

We observed that SPARQL queries were easier to operate upon as compared to DL queries, as the string representation of the SPARQL query could easily be stored as a data property of an ontological instance and presented to *owlready2* library for execution. This design choice made the simulation more robust and dynamic in nature.

During the course of the simulation, the simulation engine updates the in-memory ontology, which the modeler can choose to persist in a filesystem. We observed that the persisted ontology has the following benefits: 1. It serves as a checkpoint of the last good state of a simulation and to resume from this point, 2. It serves as a starting point for multiple simulations branching out, and 3. It allows a large simulation to be broken into multiple smaller simulations such that each simulation feeds its output to be the input for the next one. However, this is not unique to ontology-based approach, and can be achieved via serialization of an in-memory state to be persisted using any other mechanism such as a database.

During the process of simulation model development, we had introduced design inconsistencies in the ontology of simulation models (for example, a mission's description), and some of which later appeared as run-time inconsistencies. We found that reasoner-based inference was of substantial help in tracing the sources of these inconsistencies and fixing them quickly. For large simulation models, we believe this traceability is very critical.

5 CONCLUSION AND FUTURE WORK

A multi-rover system for planetary exploration was described using an ontology, which also served as the description for the simulation. Since the ontology was supported with underlying class behaviors, the simulation behavior could be prototyped with the help of a higher-level programming language. From our experimentation we were able to define an ontology, which acted as the simulation itself. Further it was demonstrated that a new simulation scenario could be defined within the ontology definition. This experimentation also demonstrated how ontology could play a crucial role in collaboration and knowledge sharing in a multi-agent system.

A minimal simulation engine was also designed as part of the experimentation. The input to the simulation engine was the ontology, which held the domain definition, along with the simulation scenario definition. The engine simulated the ontology through state transition. The final simulation engine produced as a result is highly modular.

The current multi-agent-based simulation is written from the assumption that the agents and rovers with specialized capabilities are already initialized and present in the simulation environment. As part of continued improvement, we plan to create a simulation environment which would also be able to create new rovers on demand with task scheduling capabilities.

REFERENCES

An, G., and S. Christley. 2011. “Agent-based Modeling and Biomedical Ontologies: A Roadmap”. *Wiley Interdisciplinary Reviews: Computational Statistics* 3(4):343–356.

Clancey, W. J. 1993. “The Knowledge Level Reinterpreted: Modeling Socio-Technical Systems”. *International Journal of Intelligent Systems* 8:33–49.

Estlin, T., A. Gray, T. Mann, G. Rabideau, R. Castaño, S. Chien, and E. Mjolsness. 1999. “An Integrated System for Multi-Rover Scientific Exploration”. In *Proceedings of the sixteenth national conference on Artificial intelligence and the eleventh Innovative applications of artificial intelligence conference innovative applications of artificial intelligence*. July 18th–22th, Orlando, Florida, 613–620.

Fink, W., V. R. Baker, D. Schulze-Makuch, C. W. Hamilton, and M. A. Tarbell. 2015. “Autonomous Exploration of Planetary Lava Tubes Using a Multi-Rover Framework”. In *2015 Institute of Electrical and Electronics Engineers Aerospace Conference*. March 7th–14th, Big Sky, Montana, 1–9.

Gaines, D., G. Doran, H. Justice, G. Rabideau, S. Schaffer, V. Verma, K. Wagstaff, V. Vasavada, W. Huffman, R. Anderson, R. Mackey, and T. Estlin. 2016. “Productivity Challenges for Mars Rover Operations: A Case Study of Mars Science Laboratory Operations”. Technical Report No.: D-97908, Jet Propulsion Laboratory, Pasadena, California.

Gamma, E. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Noida: Pearson Education India.

Gero, J. S., and U. Kannengiesser. 2007. “A Function–Behavior–Structure Ontology of Processes”. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 21:379–391.

Grotzinger, J. P., J. Crisp, A. R. Vasavada, R. C. Anderson, C. J. Baker, R. Barry, D. F. Blake, P. Conrad, K. S. Edgett, B. Ferdowski, R. Gellert, J. B. Gilbert, M. Golombek, J. Gómez-Elvira, D. M. Hassler, L. Jandura, M. Litvak, P. Mahaffy, J. Maki, M. Meyer, M. C. Malin, I. Mitrofanov, J. J. Simmonds, D. Vaniman, R. V. Welch, and R. C. Wiens. 2012. “Mars Science Laboratory Mission and Science Investigation”. *Space Science Reviews* 170(1):5–56.

Guerrero, A., V. Villagra, J. López de Vergara Méndez, and J. Berrocal. 2005. “Including Management Behavior Defined with SWRL Rules in an Ontology-Based Management Framework”. In *Proceedings of the 12th Annual Workshop of the HP Openview University Association*. July 10th-13th, Porto, Portugal.

Isarabhakdee, P., and Y. Gao. 2009. “Cooperative Control of a Multi-tier Multi-agent Robotic System for Planetary Exploration”. In *Proceedings of International Joint Conference on Artificial Intelligence - Workshop on Artificial Intelligence in Space*. July 17th-18th, Pasadena, California.

Lamy, J.-B. 2017. “Owlready: Ontology-Oriented Programming in Python with Automatic Classification and High Level Constructs for Biomedical Ontologies”. *Artificial Intelligence in Medicine* 80:11–28.

Lattner, A. D., T. Bogon, Y. Lorion, and I. J. Timm. 2010. “A Knowledge-Based Approach to Automated Simulation Model Adaptation”. In *Proceedings of the 2010 Spring Simulation Multiconference*. April 11th-15th, Orlando, Florida: Society for Computer Simulation International.

Masinter, L., T. Berners-Lee, and R. T. Fielding. 2005. *Uniform Resource Identifier (URI): Generic Syntax*. Network Working Group: Fremont, CA, USA. https://tools.ietf.org/html/rfc3986, accessed 16th August 2020.

Muller, K., T. Vignaux, and C. Chui. 2020. *SimPy: Discrete-Event Simulation in Python*. SimPy Development Team. https://simpy.readthedocs.io/en/latest/, accessed 16th August 2020.

Olszewska, J. I., M. E. Barreto, J. Bermejo-Alonso, J. L. Carbonera, A. Chibani, S. R. Fiorini, P. J. S. Gonçalves, M. K. Habib, A. M. Khamis, A. O. Alarcos, E. P. de Freitas, E. P. e Silva, S. V. Ragavan, S. A. Redfield, R. Sanz, B. Spencer, and H. Li. 2017. “Ontology for Autonomous Robotics”. In *26th Institute of Electrical and Electronics Engineers International Symposium on Robot and Human Interactive Communication*. August 28th-31st, Lisbon, Portugal, 189–194.

Patil, M., T. Abukhalil, and T. Sobh. 2013. “Hardware Architecture Review of Swarm Robotics System: Self-Reconfigurability, Self-Reassembly, and Self-Replication”. *International Scholarly Research Notices Robotics* 2013:255–268.

Seeni, A., B. Schäfer, and G. Hirzinger. 2010. "Robot Mobility Systems for Planetary Surface Exploration–State-of-the-Art and Future Outlook: a Literature Survey". *Aerospace Technologies Advancements* 492:189–208.

Sernadela, P., A. Pereira, and R. Rossetti. 2015. "DISim: Ontology-Driven Simulation of Biomedical Data Integration Tasks". In *2015 10th Iberian Conference on Information Systems and Technologies*. June 17th-20th, Aveiro, Portugal, 1–4.

Shafto, M., M. Conroy, R. Doyle, E. Glaessgen, C. Kemp, J. LeMoigne, and L. Wang. 2012. "Modeling, Simulation, Information Technology & Processing Roadmap". Technical report, National Aeronautics and Space Administration, Washington, DC.

Sirin, E., B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. 2007. "Pellet: A Practical OWL-DL Reasoner". *Journal of Web Semantics* 5(2):51–53.

Skelton, G. W., and Q. Pang. 2008. "Hierarchical Architecture for a Multi-Rover System". In *Institute of Electrical and Electronics Engineers SoutheastCon 2008*. April 3rd-6th, Huntsville, Alabama, 346–350.

Skelton, G. W., and Q. Pang. 2009. "Multi-Rover Collaboration". In *Institute of Electrical and Electronics Engineers SoutheastCon 2009*. March 5th–8th, Atlanta, Georgia, 284–288.

Tolk, A. 2013. *Ontology, Epistemology, and Teleology for Modeling and Simulation - Philosophical Foundations for Intelligent M&S Applications*. Springer-Verlag Berlin Heidelberg.

Torres, J. M., and Q. Wijnands. 2011. "Distributed Agents for Multi-Rover Autonomy". In *2011 Institute of Electrical and Electronics Engineers 20th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*. June 27th-29th, Paris, France, 53–58.

Tumer, K., and A. Agogino. 2005. "Coordinating Multi-Rover Systems: Evaluation Functions for Dynamic and Noisy Environments". In *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation*. June 25th-29th, Washington DC, 591–598.

Vaquero, T., M. Troesch, and S. Chien. 2018. "An Approach for Autonomous Multi-Rover Collaboration for Mars Cave Exploration: Preliminary Results". In *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*. June 4th-6th, Madrid, Spain.

World Wide Web Consortium 2012. *OWL 2 Web Ontology Language Document Overview (Second Edition)*. World Wide Web Consortium. http://www.w3.org/TR/2012/REC-owl2-overview-20121211/, accessed 16th August 2020.

World Wide Web Consortium 2013. *SPARQL 1.1 Overview*. World Wide Web Consortium. http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/, accessed 16th August 2020.

AUTHOR BIOGRAPHIES

JUSTIN JOSE is a researcher in Engineering for Research initiative of ThoughtWorks Technologies India. He has finished his Master's in Technology in Computer Science from University of Calicut, Kerala. His research interests include artificial intelligence, human behavior modelling, and knowledge representation and processing. His e-mail address is justinj@thoughtworks.com.

DIVYE SINGH is a modelling and simulation researcher in Engineering for Research initiative of ThoughtWork Technologies India. He has finished his Master's in Technology in Mathematical Modeling and Simulation from Centre for Modeling and Simulation, SPPU, Pune. His research interests include agent-based simulation, artificial intelligence, learning-based systems, machine learning, and stochastic optimization. His e-mail address is divye.singh@thoughtworks.com.

AMIT PATEL is a cyber-physical systems engineer in Engineering for Research initiative of ThoughtWorks Technologies India. He has finished his Master's in Technology (in Mechanical Engineering specialized in CAD and Automation) from Indian Institute of Technology Bombay. His e-mail address is pamit@thoughtworks.com.

HARSHAL G. HAYATNAGARKAR is a computer scientist in Engineering for Research initiative of ThoughtWorks Technologies India. His research interests include agent-based simulation, artificial intelligence, high performance computing, and data-intensive computing. He is currently focusing on large-scale simulations to aid public health and economic policy making. His e-mail address is harshalh@thoughtworks.com.