Instructions for MRS mission specification in the MutRoSe framework

Eric Bernd Gil - ericbgil@gmail.com
October 21, 2021

Abstract

This document aims at providing instructions for the specification of MRS missions in the **MutRoSe** (Multi-Robot systems Specification and decomposition) framework. We initially start with a brief introduction on what is needed for the specification to be used as input to the decomposition process, which generates the task instances to be executed. Afterwards, details on the structure of each necessary input are given in order to document the process for any interested user of the framework.

1 Introduction

The **MutRoSe** framework requires four inputs in order to work properly, which are: (i) the Goal Model for MRS missions, which is the high level description of the mission in a global perspective, (ii) the HDDL [1] domain definition, which defines the tasks, methods and actions in order to perform decomposition at the task level, (iii) the configuration file, which defines every aspect that is needed in order to link the Goal Model with the HDDL definition and also the paths of the world knowledge file and output file, and (iv) the world knowledge file. It is important to note that for now the only accepted format of the world knowledge is the file type, but other formats are future work. The overview of the specification and decomposition process of the **MutRoSe** framework is shown in Figure 1, where the necessary inputs are given to the Mission Decomposition process that generates an output file, which will be detailed in further sections.

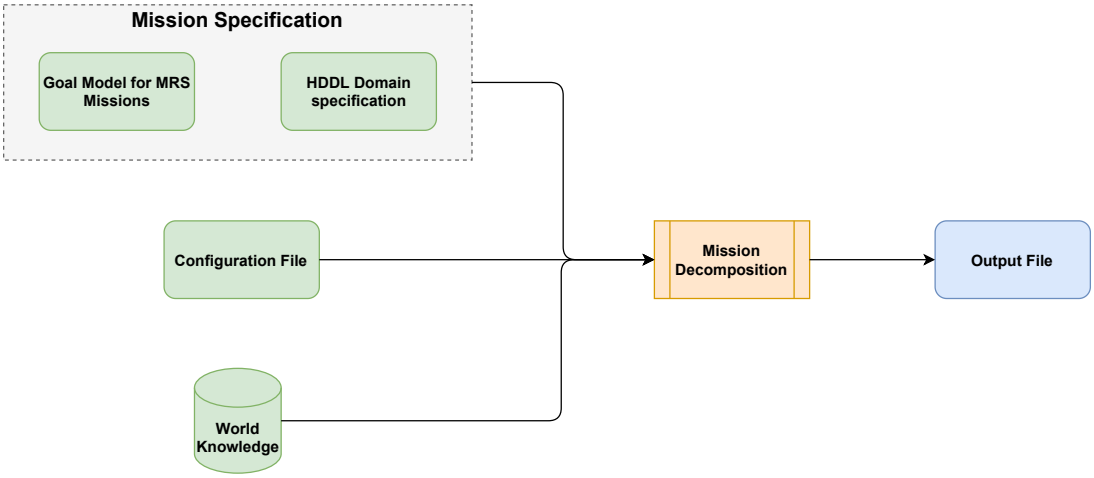

Figure 1: Overview of the specification and decomposition process

2 Goal Model for MRS Missions

The Goal Model for MRS missions is the model used to describe the mission in a high-level global view. The basic syntax is the CRGM syntax as described in [2]. Some additional properties can be defined, which will be further explained in this document. This model makes use of OCL [3] statements and parameters in order to allow it to be dynamic with respect to the current state of the world. This model can be made in the pistar-GODA framework[1] and saved in a JSON format, which will be the format used as input to the decomposition step. A conceptual model for the work is shown in Figure 2, where the structures that belong to the Goal Model are inside the red square. Each one of these important structures and concepts will be explained in detail in further subsections. The introduced features are: (i) Controls/Monitors syntax, similar to that proposed in [4], in order to define variables and maintain a dataflow between goals, (ii) goal types, as proposed in [5], (iii) OCL expressions, for variables and conditions definitions, (iv) the fact that tasks must map to tasks in the HDDL domain definition and (v) the addition of the specification of the group and divisible attributes for goals.

[1] http://pistar-goda.herokuapp.com/

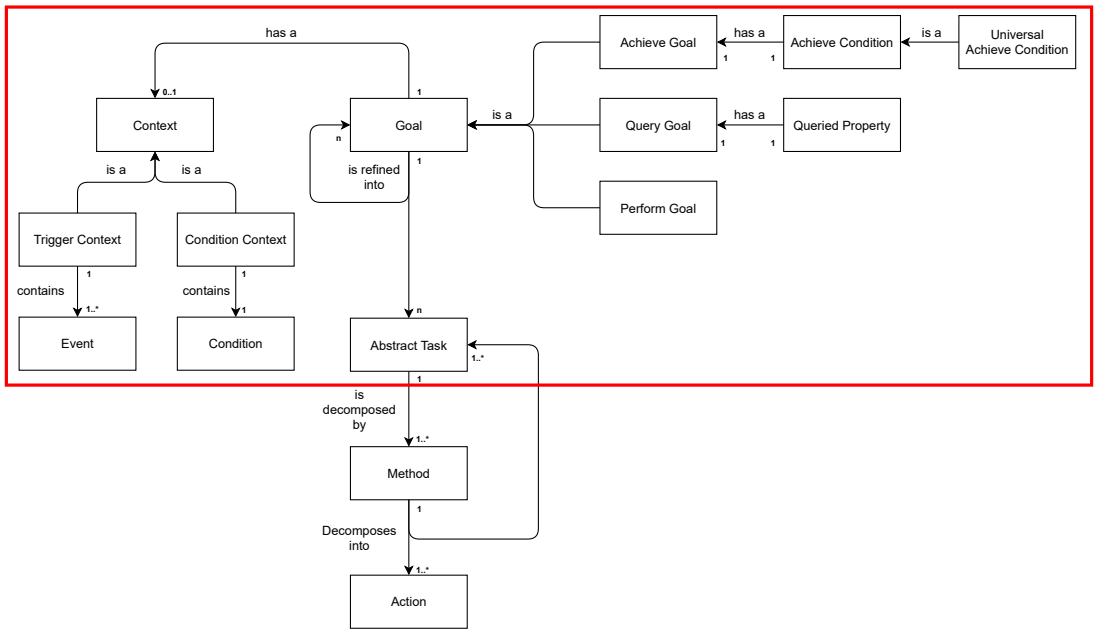

Figure 2: Conceptual model of the work (with GM structures highlighted)

2.1 Goal Attributes

Goals have several attributes in order to define variables, context conditions and other important features for mission specification, which will be explained in this section.

2.1.1 Context Conditions

Context conditions can be of two types: Condition and Trigger. These are instantiated by means of the *CreationCondition* property. Condition type contexts involve variables and their attributes and have the following structure on the value of the *CreationCondition* attribute:

assertion condition "[CONDITION]"

where [CONDITION] is over some variable attribute. As an example, let's suppose we want to assert that the *current_room* variable, which represents a *Room* type variable, must have its *is_clean* attribute equal to true. In this case, we would have the following *CreationCondition*:

assertion condition "current_room.is_clean"

Trigger type contexts are simply a list of event names that trigger the achievement of the goal. Suppose events E1 and E2 trigger some specific goal and are used in a trigger type *CreationCondition*. In this case, we would have:

assertion trigger "E1,E2"

Context Dependencies There are dependencies between tasks which are not explicitly defined, which are called *Context Dependencies*. This type of dependency happens when we have parallel decomposed nodes in which the context of one of them, which must not be enabled at the moment the node is checked in the decomposition process, is the effect of another one. Based on this definition, one can note that the node that enables the context must be a task and the one that has its context enables must be a goal. In the decomposition process, this kind of dependency generates a sequential constraint between the enabling task and all of the tasks that are children of the goal that owns the context. An illustration of a context dependency is shown in Figure 3, where task T1 enables the context of goal G3.

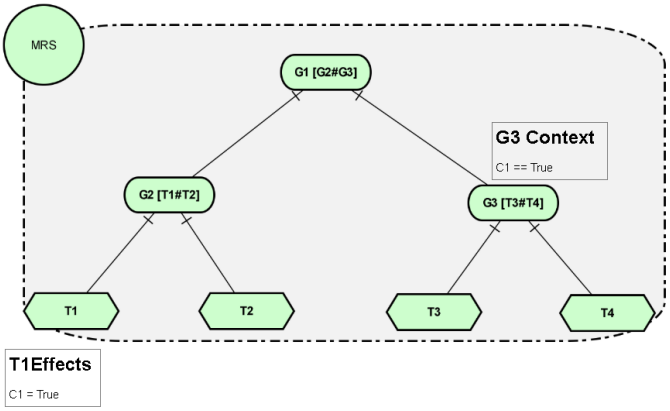

Figure 3: Example of a context dependency

2.1.2 Monitors and Controls Syntax

The *Monitors* and *Controls* properties are the means for variable declaration and dataflow definition in the Goal Model. The *Controls* property is used for variable initialization and its value must contain a list of variables and their types, where types are mandatory in this case. Let's suppose we want to declare two controlled variables *v1* and *v2* of types *T1* and *T2*, respectively. In this case, we would have the following value for the *Controls* property:

$$v1 : T1, v2 : T2$$

The *Monitors* property is what establishes the dataflow on the Goal Model, since it is used to reference variables that were previously controlled by some other goal. In this case, the declaration is a list similar to the *Controls* property, except for the fact that types are optional since we can infer them by their declaration. It is important to note that controlled variables are usually declared in Query or Achieve goals, which will be further explained. Also, in order for a goal G1 to be considered previous to another goal G2 it must be declared in a branch to the left of G2's branch and this branches must be involved in a sequential runtime annotation, which will also be further explained.

2.1.3 Group and Divisible Attributes

The *Group* and *Divisible* properties are related to constraints on the execution of tasks by robots. The concept of *group* and *non-group* goals was taken from [6] and had a slight change to its definition, with the further addition of the *divisible* concept. These properties are simple boolean properties, which by default are set to true (in the case they are not declared). The *Group* property defines one of two behaviors: (i) if set to *True* is states that the child branches of the goal can be executed by multiple robots, where the number of robots can be from 1 to N, and (ii) if set to *False* it states that the child branches of the goal can only be executed by a single robot. The *Divisible* property only has effect if the *Group* is *True* and defines one of two behaviors: (i) is set to *True* it states that each child branch can be executed by a different team of robots and (ii) if set to *False* it states that all child branches must be executed by the same team of robots. It is important to note that we have an order of priority to execution constraints, where:

- Goals with *Group* set to *False* have the higher priority
- Goals with *Group* set to *True* and *Divisible* set to *False* have the lowest priority
- Goals with *Group* set to *True* and *Divisible* set to *True* have no constraint

It is important to note that when a goal states constraints of the higher level priorities, further defined lower level priorities constraints are not considered. In general, higher priority constraints are always considered previously to lower priority ones. In order to illustrate this property let's use the Goal Model shown in Figure 4 as an example.

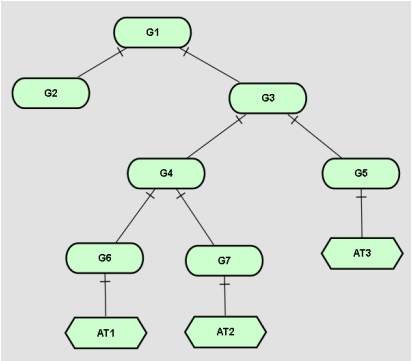

Figure 4: Example Goal Model to illustrate execution constraints properties

First, suppose that goal G3 has *Group* set to *False* and goal G4 has *Group* set to *True* and *Divisible* set to *False*. In this case we have that AT1, AT2 and AT3 are involved in non-group execution constraints with each other, since we have a non-group goal which is G3 which is a parent of all of the tasks. Now, let's suppose G3 has *Group* set to *True* and *Divisible* set to *False* and G4 has *Group* set to *False*. In this second case we would also have that AT1, AT2 and AT3 are involved in non-group execution constraints with each other. This happens because we have a constraint in G3, which is a parent of AT1, AT2 and AT3, and because we also have a higher priority execution constraint at G4, which is a child of G3. It would not make sense to have a group and non-divisible execution constraint of AT3 and the other two tasks given that these tasks are non-group, which implies in the same robot executing all of the tasks and thus a non-group execution constraint.

2.1.4 Goal Types

There are three possible Goal Types: (i) Achieve, where at the end of the goal's children execution there is some condition to be achieved, (ii) Query, where the knowledge is queried in order to instantiate variables, and (iii) Perform, which is the default one and simply performs the execution of the children with no specific condition to be achieved.

Query Goals As previously explained, Query goals query the knowledge in order to instantiate variables. In order to do so, there is the presence of a special property named *QueriedProperty* which makes use of a *select* OCL statement. A *select* OCL statement has the following syntax:

$$c- > select(x : xt \mid \phi)$$

In this syntax the variable c is of a collection type, x is a variable that represents elements in c, where the type of the elements in c can be specified, if desired, by the xt type specification, and ϕ is a boolean expression that evaluates to true or false for each x in c, in which elements where ϕ evaluates to true are selected. In this work we use the following nomenclature: (i) c is the *Queried Variable*, (ii) x is the *Query Variable*, (iii) xt is the *Queried Variable type*, and (iv) ϕ is the *Condition*.

It is important to note that the queried variable c can be either some previously defined variable or a special keyword *world_ db*, which represents the higher level of the world knowledge. Also, the variable that will be instantiated by the result of the select statement must be the first declared variable in the *Controls* property of the Query goal.

One last thing to mention about Query goals are the accepted formats for the *Condition* in the *select* statement, which are:

- Variable attributes, negative or not: (var.attr) / (!var.attr)
- Variable attributes equal or different than a string/number: (var.attr = "string") / (var.attr <> "string") / (var.attr = number) / (var.attr <> number)
- Variable attributes greater than, less than, greater or equal to, less than or equal to a number: (var.attr > number) / (var.attr < number) / (var.attr >= number) / (var.attr <= number)
- Variable attributes in a variable or variable attribute: (var.attr in var) / (var.attr in var.attr)
- Empty condition

It is important to note that a variable (or a variable + attribute) must follow the regex: [a-zA-Z]+[a-zA-z_.0-9]*. Also, numbers in conditions can be integers or floats.

ACHIEVE GOALS As previously explained, Achieve goals have a condition that must be achieved at the end of the children's execution. This is done by means of a property called *AchieveCondition* that can be of two types: (i) Universal, which makes use of a *forAll* OCL statement and (ii) Normal, which is just a simple condition. In the case of an universal *AchieveCondition*, we have the following syntax for the *forAll* statement:

$$c- > forAll(x : xt \mid \phi)$$

In this syntax we have a variable c which is of a collection type, a variable x which is of the same type as the elements in c, making the type xt an optional attribute, and a condition to be satisfied here called ϕ. In this work we use the following nomenclature: (i) c is the *Iterated Variable*, (ii) x is the *Iteration Variable*, (iii) xt is the *Iteration Variable type* and (iv) ϕ is the *Condition*.

It is important to note that c must be a previously instantiated collection variable, which must be in the variables list of the *Monitors* property of the Achieve goal, and x must be in the variables list of the *Controls* property of the Achieve goal. Also, the universal *AchieveCondition* on a collection variable with 2 or more elements generates multiple instances of the goal's children, as illustrated in Figure 5.

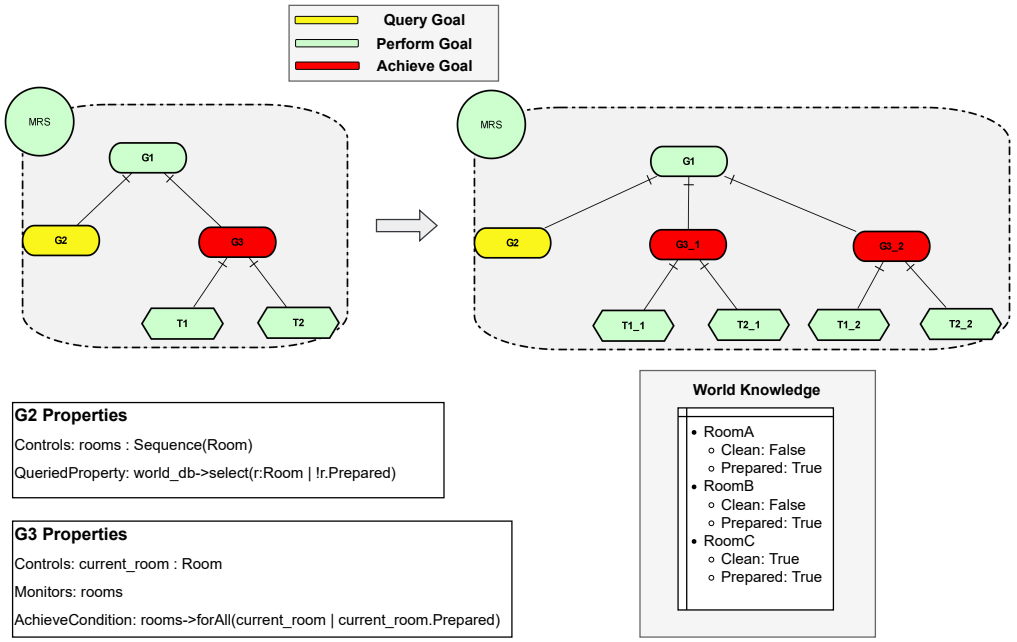

Figure 5: Example of the achieve goal effect on the goal model parsing

Similar to Query goals, for Achieve goals there are specific accepted formats for the *Condition*, being it in the *forAll* statement or as a single condition, which are:

- Variable attributes, negative or not: (var.attr) / (!var.attr)
- Variable attributes equal or different than a number: (var.attr = number) / (var.attr <> number)
- Variable attributes greater than, less than, greater or equal to, less than or equal to a number: (var.attr > number) / (var.attr < number) / (var.attr >= number) / (var.attr <= number)
- Empty condition

One last thing to note is that variables and attributes follow the same regex as in Query goals conditions.

2.2 TASK ATTRIBUTES

One important aspect of this work is that task names in the Goal Model must map to (abstract) tasks names in the HDDL Domain definition. Another important thing to mention is that the variables used in the task properties must be monitored by its parent goal. Besides that, there are 3 custom properties that a task can have:

- Location: This is the location in which the task will happen. Its value must be a variable name, which can be of a single type or of a collection type

- Params: This is a list of variables separated by commas, which will be mapped to HDDL variables
- RobotNumber: This defines the number of robots that will perform the task. This property can be a single number or a range in the format [n1,n2] where n1 is the minimum number of robots and n2 is the maximum number of robots.

2.3 Node Naming Convention

In the Goal Model, nodes must follow a specific naming convention which is explained in this subsection. The format for Goal nodes is as follows:

{GOAL_ID}: {GOAL_TEXT} {[{GOAL_RUNTIME_ANNOT}]}

where *GOAL_ID* is a unique ID consisting of a "G" followed by a number, which must increase in a depth first manner, *GOAL_TEXT* is a text explaining in what the goal consists in and *GOAL_RUNTIME_ANNOT* is a runtime annotation over the goal's children and can use three operators:

- Parallel operator "#": Describes that two expressions consisting of goal's children can be achieved in parallel. Its syntax consists in "exp1#exp2"
- Sequential operator ";": Describes that two expressions consisting of goal's children must be achieved sequentially from left to right. Its syntax consists in "exp1;exp2"
- Fallback operator "FALLBACK": Describes that multiple expressions consisting of goal's children happen in a fallback manner, where one will be executed in the case of failure of the expression to its left. Its syntax consists in "FALLBACK(exp1,exp2)". Note that this is a binary operator, so if it is desired to have a fallback between more than 2 goals you need to use it as follows "FALLBACK(FALLBACK(exp1,exp2),exp3)".

It is important to note that in the case of runtime annotations parenthesis must be used when of combining different operators. In the case of Task nodes the format is:

{TASK_ID}: {TASK_NAME}

where *TASK_ID* is a unique ID consisting of a "AT" followed by a number, which must increase in a depth first manner, and *TASK_NAME* is the name of the task which must map to a task defined in the HDDL Domain definition. An example of a Goal Model where these conventions are followed is shown in Figure 6.

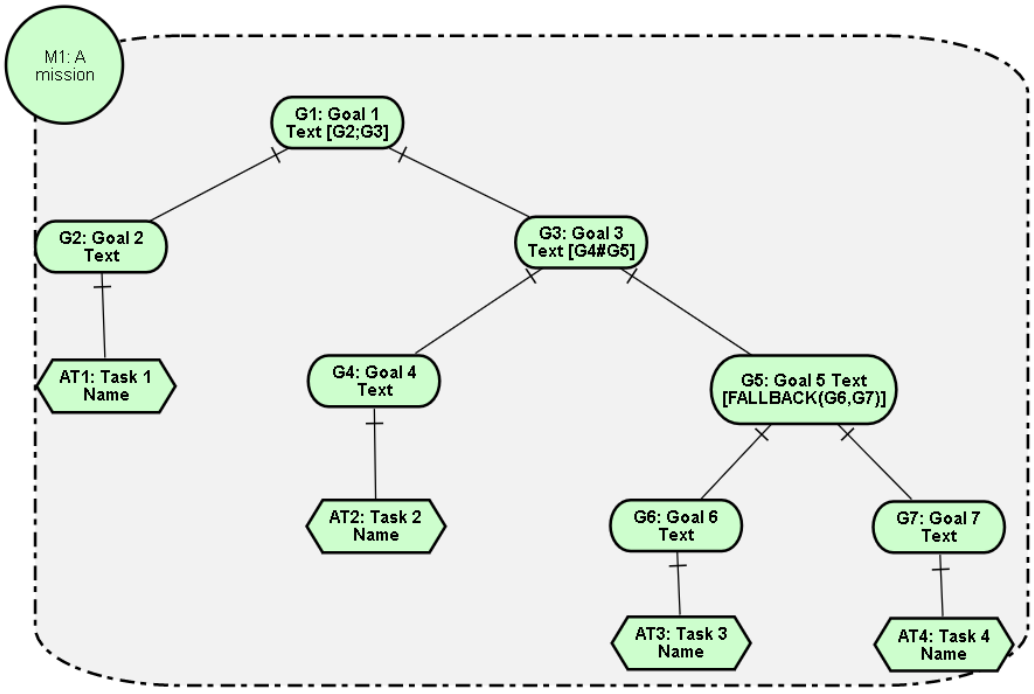

Figure 6: Example of a correctly named Goal Model

2.4 AND and OR operators in conditions

We have three types of conditions in the Goal Model: (i) the Condition of an AchieveCondition, (ii) the Condition of a QueriedProperty and (iii) the Condition of a condition type context. All of these conditions can have && (AND) and || (OR) operators to create statements on multiple conditions. The use of this operators follow their common syntax used in popular programming languages, where the use of parenthesis is advised.

3 HDDL Domain Definition

HDDL is an extension of PDDL for hierarchical planning which makes use of the HTN decomposition [1] theory. In this work we extend HDDL with some new concepts and concepts from PDDL itself which were not present in the work that proposed the language [1], such as functions. It is important to note that we only use the domain definition in this work, leaving the problem file unused.

A general extended HDDL domain definition is shown in Listing 1, where anything inside brackets is a custom name to be defined by the designer of the file. Every Domain must have a name, which is represented by DOMAIN_NAME. The other definitions will be further described.

Listing 1: HDDL Domain definition general structure

```
(:define (domain [DOMAIN_NAME])
        (:types
                [TYPE1] - [PARENT_TYPE1]
                [TYPE2] - [PARENT_TYPE2]
                                .
                                .
                                .
        )
        (:predicates
                ([PRED1] [ARG1] - [ARG_TYPE1] ...)
                                                .
                                                .
                                                .
        )
        (:functions
                ([FUNCTION_NAME1] [ARG1] - [ARG_TYPE1] ...)
                                                .
                                                .
                                                .
        )
        (:capabilities [CAPABILITY_NAME1] ...)
        (:constants [CONSTANT_NAME1] - [CONSTANT_TYPE1] ...)

        [TASK DEFINITIONS]

        [METHOD DEFINITIONS]

        [ACTION DEFINITIONS]
)
```

3.1 Types Definition

The types definition represents the accepted types and their respective parents, if any. There are three native types: (i) the object type, which is the only original native type and represents basically anything that can be considered an object, (ii) robot, which represents a single robot, and (iii) robotteam, which represents a variable number of robots.

3.2 Predicates and Functions Definition

Predicates and functions in HDDL are first-order logic (FOL) predicates and are user-defined. These FOL predicates consist simply in a name followed by one or more arguments, where each argument has its type. Arguments are declared in the form of variables, which consist in a '?' followed by the variable name. If arguments have the same type, they can be declared as [ARG_NAME1] [ARG_NAME2] - [ARG_TYPE12]. In this sense, the following two definitions are valid predicate/function definitions:

(at ?r - robot ?l - location)

(move ?l1 ?l2 - location)

3.3 Capabilities and Constants Definition

Capabilities are a concept introduced in this work. Their definition consist in a list of names separated by spaces like the following:

(:capabilities cap1 cap2 cap3)

3.4 (Abstract) Tasks Definition

Abstract tasks in HDDL are names followed by parameters, which are declared in the previously described variable format. The general structure of a task definition is shown in Listing 2.

Listing 2: HDDL task general structure

```
(:task [TASK_NAME] :parameters ([PARAM1] - [PARAM_TYPE1] ...))
```

3.5 Methods Definition

Methods in HDDL are responsible for decomposing tasks, thus being each method tied to a specific task. The general structure of a method definition is shown in Listing 3, where:

- Parameters is a list of variable names and types declared in the same fashion as arguments in the predicates/-functions definitions
- Task is the name of the task the method is tied to, followed by parameter names. Parameter names must be either a variable name as declared in the method parameters declaration or constants names as declared in the domain constants definition
- Preconditions are predicates in a format that will be further explained
- Subtasks: Subtasks define the task network generated by the method decomposition. There are two options:
  - If the keyword subtasks is used, the designer can additionally add the order (or ordering) keyword in order to give partial order of tasks. For this to happen, tasks must be declared with ids, where the ids are used in order to establish the ordering constraints in the form "(< [TASK_ID1] [TASK_ID2])". An example is as follows:

:subtasks (and
(t1 (task1 ?v1))
(t2 (task2 ?v2)))
:ordering (and
(< t1 t2))

  - If the keyword ordered-subtasks is used, the network is assumed to be totally ordered. In this sense the task ordering is the same as the declaration ordering

Listing 3: HDDL method general structure

```
(:method [METHOD_NAME]
        :parameters ([PARAM1] - [PARAM_TYPE1] ...)
        :task ([TASK_NAME] [PARAM1] ...)
        :precondition (and
                ([FUNCTION_PREC1] || [PREC1])
                                                .
                                                .
                                                .
        )
        :subtasks (and
```

```
                ([TASK_NAME1] [PARAM1] [PARAM2])
        )
)
```

3.6 Actions Definition

Actions are primitive tasks, in the sense that they cannot be decomposed. Their general structure is shown in Listing 4, where:

- Parameters is defined in the same way as in methods definitions
- Required-capabilities is a list of space separated names that must exist in the domain capabilities definition
- Preconditions and effects are predicates in a format that will be further explained

Listing 4: HDDL action general structure

```
(:action [ACTION_NAME]
        :parameters ([PARAM1] - [PARAM_TYPE1] ...)
        :required-capabilities ([CAPABILITY_NAME1] ...)
        :precondition (and
                ([FUNCTION_PREC1] || [PREC1])
                                                .
                                                .
                                                .
        )
        :effect (and
                ([FUNCTION_EFF1] || [EFF1])
                                                .
                                                .
                                                .
        )
)
```

3.7 Preconditions and Effects Definitions

Preconditions and effects are predicate or functions declarations, each with its own specificities.

3.7.1 Preconditions

If a precondition is a predicate it can be used either in a positive, using the predicate name, or negative, using the world not before the predicate name, fashion. Example declarations of predicate preconditions are as follows:

(pred1 ?v1 ?v2)
(not (pred1 ?v1 ?v2))

In the case of being a function precondition there are two possible options: (i) the equals to, using the '=' symbol, and (ii) the greater than, using the '>' symbol. Both of these conditions can compare the function value to an integer or to a float. Example declarations of function preconditions are as follows:

(= (func1 ?v1 ?v2) 1)
(> (func2 ?v1 ?v2) 1.2)

3.7.2 Effects

If an effect is a predicate it can be used in the same way as precondition definitions. In the case of being a function, there are three possible ways of declaring it: (i) using the increase keyword, (ii) using the decrease keyword or (iii) using the assign keyword. All of these conditions perform operations on the function using integer of float values. Example declarations of function effects are as follows:

(increase (func1 ?v1 ?v2) 1)
(decrease (func2 ?v1 ?v2) 1.5)
(assign (func3 ?v1) 0)

3.8 Declaration of Cyclic Decompositions

Cyclic decompositions are allowed in the decomposition process of the framework but there are some important details to consider first. Let's start with a simple example of a storage domain where object picking tasks are performed, shown in Listing 5.

Listing 5: HDDL storage domain example

```
(define (domain storage)
    (:types room - object)
    (:functions
        (objects ?rm - room)
    )
    (:capabilities pickobject)

    (:task PickObject :parameters (?r - robot ?rm - room))
    (:method object-pick-multiple
        :parameters (?r -robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (> (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (pick-object ?r ?rm)
            (PickObject ?r ?rm)
        )
    )
    (:method object-pick-single
        :parameters (?r - robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (= (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (pick-object ?r ?rm)
        )
    )
    (:action pick-object
        :parameters (?r - robot ?rm - room)
        :required-capabilities (pickobject)
        :precondition ()
        :effect (and
            (decrease (objects ?rm) 1)
        )
    )
)
```

One can verify that a decomposition of the task *PickObject* will depend on the number of objects to be picked in a storage room, represented by the (objects ?rm) function HDDL function. In the current state of the framework a specification must conform to certain conditions in order for the decomposition to be correctly performed. An expansion is detected if a method contains a greater than precondition and:

- Has a cycle with its parent task, as is the case of the method *object-pick-multiple* and the *PickObject* task
- All of the tasks that have effects on the function that created the need for an expansion. In the storage example, this is the *objects* function over the *?rm* variable of the *object-pick-multiple* method which maps to the *?rm* variable of the *PickObject* task. In this sense, We can verify that the only task that performs an operation over the *objects* on the storage example is the *pick-object* action, which is its direct children.
  - The *pick-object* which will be a child of the *object-pick-single* will not be present in the cycle, so this is not a problem
- Every effect related to the function that created the need for an expansion must be a decrease operator

The example shown in Listing 6 would configure a wrong specification of a cyclic decomposition, where the errors are highlighted in red.

Listing 6: HDDL storage domain example with specification errors

```
(define (domain storage)
    (:types room - object)
    (:functions
        (objects ?rm - room)
    )
    (:capabilities pickobject)

    (:task RetrieveObject :parameters (?r - robot ?rm - room))
    (:method retrieve-object
        :parameters (?r - robot ?rm - room)
        :task (RetrieveObject ?r ?rm)
        :precondition ()
        :ordered-subtasks (and
            (pick-object ?r ?rm)
        )
    )

    (:task PickObject :parameters (?r - robot ?rm - room))
    (:method object-pick-multiple
        :parameters (?r -robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (> (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (RetrieveObject ?r ?rm)
            (PickObject ?r ?rm)
        )
    )
    (:method object-pick-single
        :parameters (?r - robot ?rm - room)
        :task (PickObject ?r ?rm)
        :precondition (and
            (= (objects ?rm) 1)
        )
        :ordered-subtasks (and
            (pick-object ?r ?rm)
        )
    )

    (:action pick-object
        :parameters (?r - robot ?rm - room)
        :required-capabilities (pickobject)
        :precondition ()
        :effect (and
            (assign (objects ?rm) 1)
        )
    )
)
```

4 Configuration File

The configuration file can be either in JSON or XML format. This file contains information that is necessary in order for the decomposition process to work properly. The configuration file contains: (i) information of the world knowledge, (ii) information of the output, (iii) the high-level OCL types used as location types, (iv) the type mappings between OCL and HDDL types, (v) the variable mappings between OCL goal model variables and HDDL variables for each

task and (vi) the semantic mappings that establish relations between the knowledge and first-order logic predicates used in HDDL. All of this information will be further explained in the following subsections.

4.1 World Knowledge Information

The world knowledge information contains the information needed for the decomposition process to find and manipulate the knowledge of the system about the world. For now the only accepted type is the XML file type, but the structure was made in order for other types to be accepted. For example, when choosing the *file* type for the world knowledge one can choose the *file_type* attribute to be a specific file type, where at the moment we can only have *xml*. When having an XML file we need to give an XML root key, if any, where it can be user-defined for something like *world_db*. Finally, one needs to define the world knowledge unique ID, which is the attribute that is unique to each record. The default value for this unique ID is "name", in case no unique ID is defined by the user or the user leaves the unique ID blank. Listings 7 and 8 show the JSON and XML configuration file structure for the world knowledge information, respectively, where the *type* and *file_type* attributes are already set since this is the only accepted configuration for now.

Listing 7: World knowledge information in JSON

```
"world_db": {
    "type": "file",
    "file_type": "xml",
    "path": "",
    "xml_root": "",
    "unique_id": ""
}
```

Listing 8: World knowledge information in XML

```
<world_db>
    <type>file</type>
    <file_type>xml</file_type>
    <path></path>
    <xml_root></xml_root>
    <unique_id></unique_id>
</world_db>
```

4.2 Output Information

The output information contains the information needed for the output of the decomposition process to generate the output correctly. For now the only accepted type for the output is the file type, where it can be either an XML or a JSON file. Listings 9 and 10 show the JSON and XML configuration file structure for the output information, respectively, where the *output_type* attribute is already set since this is the only accepted configuration for now.

Listing 9: Output information in JSON

```
"output": {
    "output_type": "file",
    "file_path": "",
    "file_type": ""
}
```

Listing 10: Output information in XML

```
<output>
    <output_type>file</output_type>
    <file_path></file_path>
    <file_type></file_type>
</output>
```

4.3 High-Level Location Types

The high-level location types are the OCL types used in the Goal Model that can be used as locations for tasks. In the JSON configuration this declaration is simply a list of string values and in the XML configuration each location type is enclosed in a *type* tag. Listings 11 and 12 show the declaration in JSON and XML formats, respectively.

Listing 11: High-level location types declaration in JSON

```
"location_types": [""]
```

Listing 12: High-level location types declaration in XML

```
<location_types>
    <type></type>
</location_type>
```

4.4 High-Level Agent Types

The high-level agent types are the OCL types used in the Goal Model that can be used as agents for tasks. This declaration is useful if one wants to generate iHTN outputs but does not want to consider all types as agents. In the JSON configuration this declaration is simply a list of string values and in the XML configuration each agent type is enclosed in a *type* tag. Listings 13 and 14 show the declaration in JSON and XML formats, respectively.

Listing 13: High-level agent types declaration in JSON

```
"agent_types": [""]
```

Listing 14: High-level agent types declaration in XML

```
<agent_types>
    <type></type>
</agent_type>
```

4.5 Type Mappings

The type mappings are basically the relations between the OCL types, used in the Goal Model, and the HDDL types, where this is done in order to ensure type correctness when mapping variables. Listings 15 and 16 show the types mappings declarations in JSON and XML formats, respectively.

Listing 15: Type mappings declaration in JSON

```
"type_mapping": [
    {
        "hddl_type": "",
        "ocl_type": ""
    }
]
```

Listing 16: Type mappings declaration in XML

```
<type_mapping>
    <mapping>
        <hddl_type></hddl_type>
        <ocl_type></ocl_type>
    </mapping>
</type_mapping>
```

4.6 Variable Mappings

Variable mappings are simply the mappings between OCL variables and HDDL variables for each abstract task, being this the reason why for each mapping we establish one task ID. This is done in order to guarantee correct mappings in the decomposition process. Listings 17 and 18 show the variable mappings declarations in JSON and XML formats, respectively.

Listing 17: Variable mappings declaration in JSON

```
"var_mapping": [
    {
        "task_id": "",
        "map": [
            {
                "gm_var": "",
                "hddl_var": ""
            }
        ]
    }
]
```

Listing 18: Variable mappings declaration in XML

```
<var_mapping>
    <mapping>
        <task_id></task_id>
        <map gm_var="" hddl_var=""/>
    </mapping>
</var_mapping>
```

4.7 Semantic Mappings

Semantic mappings are relations between the world knowledge and first-order logic predicates used in HDDL. This is done for predicate initialization and conditions evaluations throughout the decomposition process. There are three accepted types of mappings for now: (i) attribute type mappings, (ii) ownership type mappings and (iii) relationship type mappings. These types of mappings will be further explained and also to which types of predicates they can be mapped to.

4.7.1 Attribute Semantic Mappings

Attribute semantic mappings are mappings between attributes of the knowledge and first-order logic predicates. Attributes are basically fields in some record that corresponds to a certain type. For example, if we had a record of a Room in our knowledge we could have an attribute called *is_ empty*. Listings 19 and 20 show the attribute semantic mappings in JSON and XML formats, respectively.

Listing 19: Attribute semantic mappings declaration in JSON

```
"semantic_mapping": [
    {
        "type": "attribute",
        "name": "",
        "relates_to": "",
        "belongs_to": "world_db",
        "mapped_type": "",
        "map": {
            "pred": "",
            "arg_sorts": [""]
        }
    }
]
```

Listing 20: Attribute semantic mappings declaration in XML

```
<semantic_mapping>
    <mapping>
        <type>attribute</type>
        <name></name>
        <relates_to></relates_to>
        <belongs_to>world_db</belongs_to>
        <mapped_type></mapped_type>
        <map>
            <pred></pred>
            <arg_sorts number=""></arg_sorts>
        </map>
    </mapping>
</semantic_mapping>
```

The fields of an attribute semantic mapping are as follows:

- type: This is present in every semantic mapping but is set to *attribute*
- name: The name of the attribute being considered
- relates_to: The type of the record that has this attribute
- belongs_to: This is the place to search for the knowledge. For now we can only search in the world knowledge, which is represented by the *world_db* tag. The only exception is if the *relates_to* attribute is set to robot, in which the *belongs_to* attribute must be set to *robots_db* that represents the robot knowledge, which is not used in the process right now
- mapped_type: This is the type to which we map the attribute which can be one in two values: *predicate* or *function*
- map: This is the predicate/function to which we map the attribute to. We need to define the first-order logic predicate name (pred) and the types of the arguments (arg_sorts)

There is an optional attribute that must be present when dealing with predicates related to collection objects, which the *predicate_type* attribute. This can be one of two types: (i) "Existential", when the predicate evaluates to true when one object inside the collection satisfies the condition, or (ii) "Universal", when the predicate evaluates to true only when all of the objects inside the collection satisfies the condition. This attribute can only exist when the *mapped_type* attribute is set to "predicate".

4.7.2 Relationship Semantic Mapping

Relationship semantic mappings are used when we have an attribute of a certain record that contains references to other records, which can be of the same type or even of different types (which probably is the most common use case). As an example, suppose we had a record of type Room which contains an attribute objects that contains a list of reference to Object type records. Then, this objects attribute would configure the relationship between a Room type record and one or multiple Object type records. Listings 21 and 22 show the relationship semantic mappings in JSON and XML formats, respectively.

The fields of a relationship semantic mapping are as follows:

- type: This is present in every semantic mapping but is set to *relationship*
- main_entity: The type of the record which contains the attribute
- related_entity: The type of the record which is referenced in the attribute
- relationship_type: At the moment, the only accepted value for a relationship is attribute
- attribute_name: This represents the name of the attribute that will establish the relationship and is only present when attribute relationship type is set
- belongs_to: Works in the same way as for the attribute semantic mapping

- mapped_type: For now the only accepted value is predicate, since relationship type mapping to functions is currently not allowed

Listing 21: Relationship semantic mappings declaration in JSON

```
"semantic_mapping": [
    {
        "type": "relationship",
        "main_entity": "",
        "related_entity": "",
        "relationship_type": "attribute",
        "attribute_name": "",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "",
            "arg_sorts": [""]
        }
    }
]
```

Listing 22: Relationship semantic mappings declaration in XML

```
<semantic_mapping>
    <mapping>
        <type>relationship</type>
        <main_entity></main_entity>
        <related_entity></related_entity>
        <relationship_type>
            attribute
        </relationship_type>
        <attribute_name></attribute_name>
        <belongs_to>world_db</belongs_to>
        <mapped_type>predicate</predicate>
        <map>
            <pred></pred>
            <arg_sorts number=""></arg_sorts>
        </map>
    </mapping>
</semantic_mapping>
```

4.7.3 Ownership Semantic Mapping

An ownership type semantic mapping is very similar to the relationship one, with the difference that the attribute that establishes the ownership does not contain references to other records but instead contains the other records, where attribute is the only allowed type for now. Listings 23 and 24 show the ownership semantic mappings in JSON and XML formats, respectively.

Listing 23: Ownership semantic mappings declaration in JSON

```
"semantic_mapping": [
    {
        "type": "ownership",
        "owner": "",
        "owned": "",
        "relationship_type": "attribute",
        "attribute_name": "",
        "belongs_to": "world_db",
        "mapped_type": "predicate",
        "map": {
            "pred": "",
            "arg_sorts": [""]
        }
    }
]
```

Listing 24: Ownership semantic mappings declaration in XML

```
<semantic_mapping>
    <mapping>
        <type>ownership</type>
        <owner></owner>
        <owned></owned>
        <relationship_type>
            attribute
        </relationship_type>
        <attribute_name></attribute_name>
        <belongs_to>world_db</belongs_to>
        <mapped_type>predicate</predicate>
        <map>
            <pred></pred>
            <arg_sorts number=""></arg_sorts>
        </map>
    </mapping>
</semantic_mapping>
```

The fields of an ownership semantic mapping are as follows:

- type: This is present in every semantic mapping but is set to *ownership*
- owner: The type of the record which contains the attribute
- owned: The type of the record which is contained in the attribute
- relationship_type: The same as in the relationship type semantic mapping
- belongs_to: The same as in the attribute type semantic mapping
- mapped_type: The same as in the relationship type semantic mapping

5 World Knowledge

The accepted format for the world knowledge at the moment is an XML file. In this sense, it is basically a collection of records with attributes that can be mapped to predicates using the semantic mapping. In orde to give an example of how this knowledge would look like, Listing 25 shows an example record of the knowledge of a hospital which contains rooms that need to be cleaned.

Listing 25: Example of a world knowledge record for a hospital domain

```
<world_db>
    <Room>
        <name>RoomA</name>
        <location>c3</location>
        <is_clean>False</is_clean>
        <door_open>False</door_open>
    </Room>
</world_db>
```

6 Important Considerations

There are some things that when done in the specification step generate errors in the decomposition step. General specification rules are listed below.

- OR decomposed goals can only contain parallel operators, since OR decompositions are treated as alternative decompositions where only one can be chosen
- Tasks without the RobotNumber attribute cannot have robotteam variables in the HDDL definition
- Non group tasks, which are children of non-group goals, must have 1 robot variable in its declaration or a RobotNumber attribute with 1 present in the range
- All predicates that are not of robot-related types, which are not resolved in this step, must have a semantic mapping defined in the configuration file
- If no runtime annotation is provided, the node is considered to be parallel. This can cause bad behavior if not taken into consideration.

The previously listed rules are the basic guidelines for the specification of missions in the framework and must be followed in order to avoid errors.

7 Using the Framework

With all the necessary inputs in hand, one needs the binary for the decomposition code in order to generate an output. In order to use the binary, just run the following command in a Linux terminal:

./MRSDecomposer [PATH_TO_HDDL_FILE] [PATH_TO_GM_JSON_FILE]
[PATH_TO_CONFIGURATION_FILE]

where:

- The -v option can be used in order to have a verbose terminal output.
- The -p option can be used in order to have a pretty-printed terminal output.
- The -h option can be used if instead of having the JSON file output information one wants to have an iHTN [7] JSON output

It important to note that -v and -p options generate the normal JSON output and can't be used together. Any questions refer to the author at the email given in the beginning of the document.

8 Example Specification

In this section will be shown an example specification of a mission called "Room Cleaning". In this simple example the mission consists of verifying all dirty rooms and cleaning them if they are not occupied, where there are two possible alternative ways of cleaning a room: (i) using a vacuum cleaner or (ii) using a UV light. In reality, these two ways of cleaning are not necessarily interchangeable but for the purposes of exemplification we consider them to be.

The Goal Model for the "Room Cleaning" example is shown in Figure 7. In this model we can verify that we have a Query goal in a sequential runtime annotation with an Achieve goal. The Query goal *G2* uses a select statement to fetch from the world knowledge all the *Room* type records where the *is_ clean* attribute is false and fills up the *rooms* variable which is of type *Sequence(Room)*. This *rooms* variable is the iterated variable for the Achieve goal *G3*, where the iteration variable is a *Room* type variable called *current_ room* and the condition is that for every room in *rooms* we have that the *is_ clean* attribute must be true. Also, goal *G4* has a context represented by the *CreationCondition* property which specifies that the room being cleaned must not be occupied. Finally, goal *G5* is has its *Group* property set to *True*, where the property *RobotNumber* of task *AT1* is equal to *[2,4*, and goal *G6* has its *Group* property set to *False*.

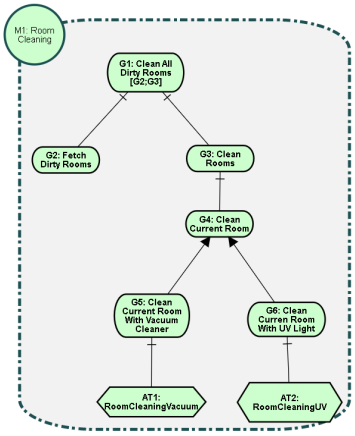

Figure 7: Goal Model for the "Room Cleaning" example

The select statement of goal G2 is as follows:

world_db->select(r:Room | !r.is_clean)

and the forAll statement of goal G3 is as follows:

rooms->forAll(current_room | current_room.is_clean)

Finally, the context for goal G4 is:

assertion condition "not current_room.is_occupied"

One important note to give here is that the use of a CreationCondition to express that rooms must not be occupied is a choice. If some room that is not clean turns out to be occupied, the mission as a whole will produce no valid decompositions. Another approach for dealing with this restriction would be to have the following select statement in goal G2:

world_db->select(r:Room | !r.is_clean && !r.is_occupied)

The HDDL Domain definition for the "Room Cleaning" example is shown in Listing 26. This is a very simple definition since each task only has one method and these methods decompose tasks in only one action. Notice that task RoomCleaningVacuum has a variable of type *robotteam*, which must be used since in the goal model this task has a variable number of robots to execute it (specified by the *RobotNumber* property).

Listing 26: HDDL Domain definition for the "Room Cleaning" example

```
(define (domain hospital)
    (:types room - object)
```

```
    (:predicates
        (clean ?rm - room)
        (occupied ?rm - room)
    )
    (:capabilities cleaningvacuum cleaninguv)

    (:task RoomCleaningVacuum :parameters (?rt - robotteam ?rm - room))
    (:method room-cleaning
        :parameters (?rt - robotteam ?rm - room)
        :task (RoomCleaningVacuum ?rt ?rm)
        :precondition (not (clean ?rm))
        :ordered-subtasks (and
            (clean-room-vacuum ?rt ?rm)
        )
    )
    (:action clean-room-vacuum
        :parameters (?rt - robot ?rm - room)
        :required-capabilities (cleaningvacuum)
        :precondition ()
        :effect (and
            (clean ?rm)
        )
    )

    (:task RoomCleaningUV :parameters (?r - robot ?rm - room))
    (:method room-cleaning-uv
        :parameters (?r - robot ?rm - room)
        :task (RoomCleaningUV ?r ?rm)
        :precondition (not (clean ?rm))
        :ordered-subtasks (and
            (clean-room-uv ?r ?rm)
        )
    )
    (:action clean-room-uv
        :parameters (?r - robot ?rm - room)
        :required-capabilities (cleaninguv)
        :precondition ()
        :effect (and
            (clean ?rm)
        )
    )
)
```

Finally, the JSON configuration file and the XML world knowledge are shown in Listings 27 and 28, respectively. In these files one can verify every aspect that was previously shown, in order to better illustrate the necessary inputs.

Listing 27: JSON configuration file for the "Room Cleaning" example

```
{
    "world_db": {
        "type": "file",
        "file_type": "xml",
        "path": "knowledge/World_db.xml",
        "xml_root": "world_db"
    },

    "output": {
        "output_type": "file",
        "file_path": "output/task_output.json",
        "file_type": "json"
    },

    "location_types": ["Room"],
```

```
    "type_mapping": [
        {
            "hddl_type": "room",
            "ocl_type": "Room"
        }
    ],

    "var_mapping": [
        {
            "task_id": "AT1",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                }
            ]
        },
        {
            "task_id": "AT2",
            "map": [
                {
                    "gm_var": "current_room",
                    "hddl_var": "?rm"
                }
            ]
        }
    ],

    "semantic_mapping": [
        {
            "type": "attribute",
            "name": "is_clean",
            "relates_to": "Room",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "clean",
                "arg_sorts": ["room"]
            }
        },
        {
            "type": "attribute",
            "name": "is_occupied",
            "relates_to": "Room",
            "belongs_to": "world_db",
            "mapped_type": "predicate",
            "map": {
                "pred": "occupied",
                "arg_sorts": ["room"]
            }
        }
    ]
}
```

Listing 28: XML knowledge file for the "Room Cleaning" example

```
<world_db>
    <Room>
        <name>RoomA</name>
        <is_clean>False</is_clean>
        <is_occupied>False</is_occupied>
    </Room>
    <Room>
```

```
            <name>RoomB</name>
            <is_clean>False</is_clean>
            <is_occupied>False</is_occupied>
        </Room>
        <Room>
            <name>RoomC</name>
            <is_clean>True</is_clean>
            <is_occupied>False</is_occupied>
        </Room>
</world_db>
```

The decomposition of the mission was performed running the following command on a linux terminal:

./MRSDecomposer hddl/Room Cleaning.hddl gm/Room Cleaning.txt configs/Room Cleaning Config.json

All of the files and the generated output can be found in the following link: `https://drive.google.com/drive/folders/1ATLdUgXQWdpl-CxrUNehKHEIqfgpOpOQ?usp=sharing`. With these files one can take a look at the goal model in more detail, in order to verify its properties.

References

[1] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, and R. Alford, "Hddl: An extension to pddl for expressing hierarchical planning problems," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 9883–9891, 2020.

[2] D. F. Mendonça, G. N. Rodrigues, R. Ali, V. Alves, and L. Baresi, "Goda: A goal-oriented requirements engineering framework for runtime dependability analysis," *Information and Software Technology*, vol. 80, pp. 245–264, 2016.

[3] O. OCL, "Object constraint language (ocl), version 2.4," 2014.

[4] A. Van Lamsweerde, "From system goals to software architecture," in *International School on Formal Methods for the Design of Computer, Communication and Software Systems*, pp. 25–43, Springer, 2003.

[5] L. Braubach, A. Pokahr, D. Moldt, and W. Lamersdorf, "Goal representation for bdi agent systems," in *International workshop on programming multi-agent systems*, pp. 44–65, Springer, 2004.

[6] A. Torreno, E. Onaindia, A. Komenda, and M. Štolba, "Cooperative multi-agent planning: a survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–32, 2017.

[7] C. Lesire, G. Infantes, T. Gateau, and M. Barbier, "A distributed architecture for supervision of autonomous multi-robot missions," *Autonomous Robots*, vol. 40, no. 7, pp. 1343–1362, 2016.