COLLEGE OF INFORMATION SCIENCES AND TECHNOLOGY
THE PENNSYLVANIA STATE UNIVERSITY

A Gentle Introduction to XML within Herbal

Maik Friedrich, Mark A. Cohen, and Frank E. Ritter

Tutorial

July 15, 2007

---

friedrichm@gmx.de

Applied Cognitive Science Lab, School of Information Sciences and Technology,

323 IST Building, University Park, PA 16802

Maik Friedrich, friedrichm@gmx.de
Mark A. Cohen, mcohen@ist.psu.edu
Frank E. Ritter, ritter@ist.psu.edu

Applied Cognitive Science Lab

College of Information Sciences and Technology

Computer Science and Engineering Department

The Pennsylvania State University

University Park, PA 16802

Table of Content

1. Introduction ... 1
  1.1. Why use XML in Herbal ... 1
  1.2. Structure of XML in Herbal ... 1
2. The basics of the Herbal Interface ... 3
  2.1. Installing Herbal ... 3
  2.2. Starting Herbal and initializing the interface in Eclipse ... 3
  2.3. Creating a new Herbal project in Eclipse ... 3
  2.4. How to find and edit the XML files ... 3
  2.5. Building a Herbal project without Eclipse ... 4
3. Requirements for an agent ... 5
4. Types.XML ... 7
  4.1. Schema for Types (XSD) ... 7
  4.2. Lesson 1: Create the Types ... 8
5. Actions.XML ... 11
  5.1. Schema for Actions (XSD) ... 11
  5.2. Lesson 2: Create the Actions which Tom can execute ... 13
6. Conditions.XML ... 15
  6.1. Schema for Conditions (XSD) ... 15
  6.2. Lesson 3: Create the Conditions under which Tom has to work ... 17
7. Operators.XML ... 20
  7.1. Schema for Operators (XSD) ... 20
  7.2. Lesson 4: Create the Operators which can be used by Tom ... 21
8. Problemspace.XML ... 24
  8.1. Schema for Problemspace (XSD) ... 24
  8.2. Lesson 5: Create problem spaces for Tom ... 25
9. Models.XML ... 28
  9.1. Schema for Models (XSD) ... 28
  9.2. Lesson 6: Creating a vacuum cleaner agent named Tom ... 29
10. Running the Agent ... 32
11. Problems ... 32
  11.1. Name and type errors ... 32

11.2. Missing tags 33
References .... A

1. Introduction

Herbal is a high level behavior representation language that is realized through an integrated development environment consisting of a high-level language, a compiler, and a graphical editor that acts as a first step towards creating development tools that support the wide range of users of intelligent agents and cognitive models.

Chapter One to Three discusses how to get Herbal up and running and which structure the agent we want to create, has. Chapters Four to Nine contain two major parts. The first part of every chapter gives you an overview of the schema of all xml files used in Herbal and the second provides you with the major steps to create your own agent by using the xml interface. Chapter ten shows you examples of some problems which could come up while you programming with Herbal.

1.1. Why use XML in Herbal

The main objective of Herbal is to allow developers to focus on the architectural aspects of the cognitive agent while the detailed aspects of the programming nuances are managed by the Herbal compiler.

For Herbal beginners it is recommended to use the graphical interface to create their agents. For example the technical report "A Gentle Introduction to Herbal"[1] is a good help for beginners to get used to Herbal. It also helps if the user has no knowledge about cognitive modeling and just wants to get started.

The Herbal environment itself is based on two kinds of interfaces, the graphical editor, also called the Herbal GUI Editor, and the XML model files. Each interface is consistent to the other, which means when you change something in the GUI Editor it is automatically changed in the XML file and the other way around.

The reasons for using the XML file rather than the GUI Editor are very simple. If the user is accustomed to reading XML code he will faster be able to read and understand models. When the user knows how to create an agent in Herbal, it might slow him down if he uses the GUI Editor. At that point he can switch to the XML files to create his agent much faster. If the user knows the schema files for the XML files he knows exactly what Herbal can do and how to use it.

1.2. Structure of XML in Herbal

Herbal is based on the concept of self-explanation; the model is built with the capability of explaining itself. In Herbal, the topmost entity, the agent operates within a problem space which is driven by a global goal. The goal defines the reason for the agent's existence. Each problem space is a collection of several sub problem spaces which are also goal driven and, in turn, define smaller and more local goals in the service of the topmost problem space or other problem spaces.

[1] (Cohen, Ritter, & Bhandarkar, 2007)

A schematic view of the Herbal structure plus the responsible XML files is given in Figure 1. Every part of the model (agents, problem spaces,...) depends on a separate XML file and XSL file. This modular architecture makes it easy to keep an overview of the current project structure and it is helpful while debugging. The output files are completely independent from the XML files and even from each other.

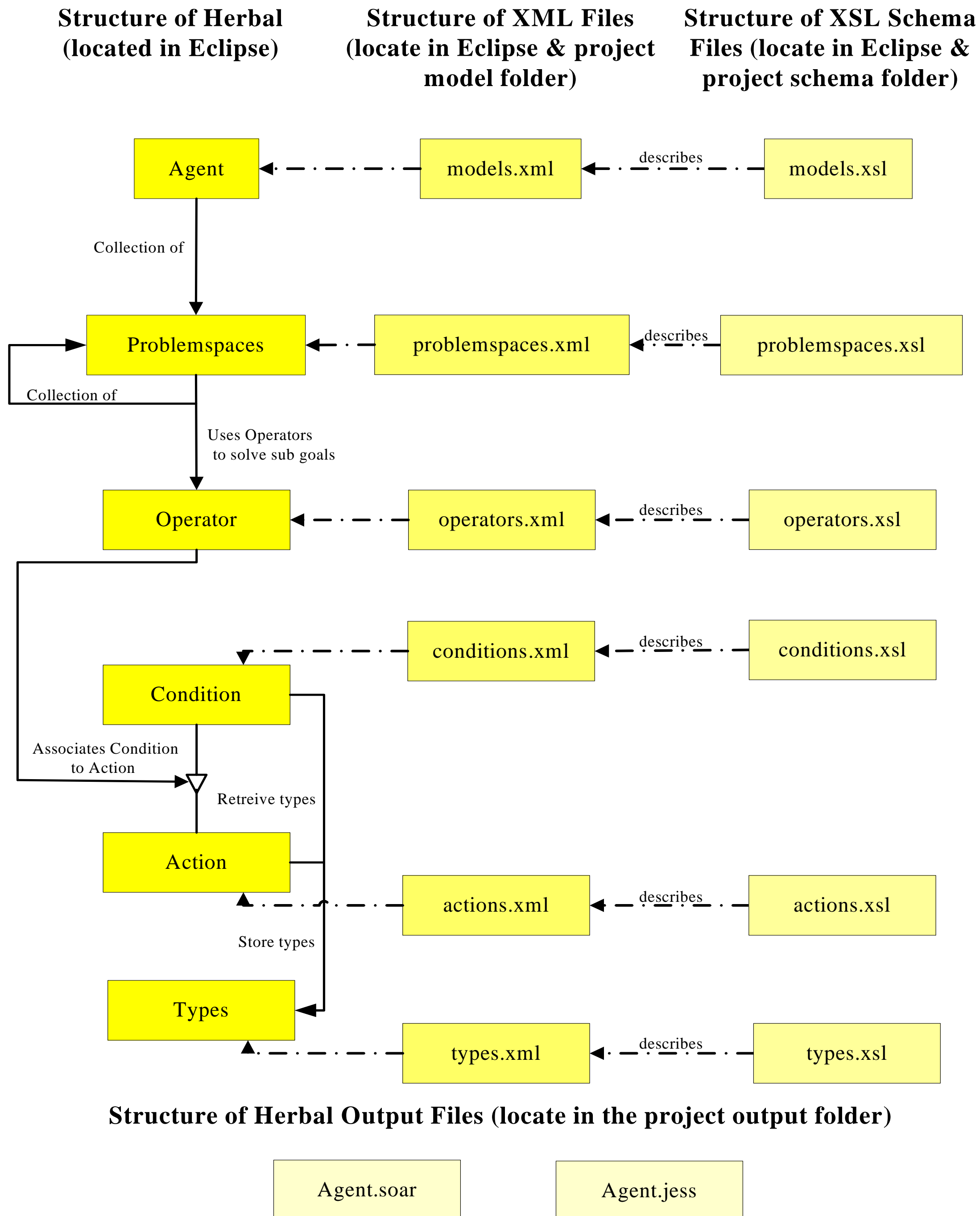

Figure 1: Herbal File Structure

2. The basics of the Herbal Interface

2.1. Installing Herbal

Reference to (Cohen, Ritter, & Bhandarkar, 2007) Chapter 2

2.2. Starting Herbal and initializing the interface in Eclipse

Reference to (Cohen, Ritter, & Bhandarkar, 2007) Section 3.1

2.3. Creating a new Herbal project in Eclipse

Reference to (Cohen, Ritter, & Bhandarkar, 2007) Section 4.1

2.4. How to find and edit the XML files

To find the XML files follow the sequence below.

1) Finding the XML files
   a) Browsing the Hard disk file structure. Open the *OS File Browser* and go to the Eclipse workspace directory and *open* it. The project folder has the same name as the created project.
   OR
   b) Using the Eclipse Navigator View. Use the *Navigator View* to open the project.
2) Within the project folder are three subfolders ("model","output","schema").
3) Open the *model* folder to find all XML files. Every XML file names are similar to his project compound, shown in Figure 1.

When your project is new, there are 6 XML files in the model folder. All of these files are connected to a Tab in the Herbal GUI Editor. Table 1 shows which GUI element depends on which XML file. If you use library files from other agents or projects these XML files should also be the model folder. Otherwise Herbal is not able to find them. The name of the library files depends on you, but they have to end with *".project part name"* and *".xml"*, for example a library file for types could look like this *"libraryforVacuum.types.xml"*.

Herbal GUI Element (Tab)	XML Files (if your project has the name “Tom”)
Agents	Tom.models.xml
ProblemSpaces	Tom.problemspaces.xml
Operators	Tom.operators.xml
Conditions	Tom.conditions.xml
Actions	Tom.actions.xml
Types	Tom.types.xml

Table 1: Herbal GUI Element and XML Files

To edit the XML files you can chose any program you prefer. Most people like to use a separate XML editor or have a special Eclipse plug-in for this task. Anyhow, this tutorial will show the use of the

standard Eclipse XML editor. To open an XML file in the Eclipse environment move the mouse over to the file you want to edit, press the *right mouse button* and select *Open with > Text Editor*.

Before you start editing the XML files there are some facts to consider. First, if the XML file has no correct syntax the GUI is completely empty. This means, it is not possible to switch between file editors and Herbal GUI while editing. Second, when encountering problems within this tutorial that cannot be solved, check your Herbal version because the structure of your scheme files could change between two versions.

2.5. Building a Herbal project without Eclipse

There are several ways to compile a Herbal project. The most common way is to use the Eclipse, but Herbal development can exist without the use of Eclipse and the GUI environment. The compiler can be run from the command line. This means that, the user has a choice of working with or without Eclipse.

To run the Herbal compiler without the Eclipse environment, follow the steps:

1. Setup your directory structure
    a. Create a folder with the name "myherbal", this represents your project name.
    b. Enter this folder and create tree folder called *"model"*, *"output"*, and *"schema"*.
    c. Go to another project, which you have created with eclipse and copy every file from the *"schema"* folder into the *"schema"* folder in your new project("actions.xsd"; "conditions.xsd"; "models.xsd"; "operators.xsd"; "problemspaces.xsd"; "types.xsd").
    d. Create all you model files and put them in the model folder.
2. Download the Herbal "jar" file from the Herbal website (described in section 2.1) and copy it into your new project folder.
3. Open a command window and make "myherbal" your working folder.
4. You can compile from the command line using this command, if you have the herbal version 2.0.10:
    "java -classpath .;edu.psu.ist.acs.Herbal_2.0.10.jar edu.psu.ist.herbal.Compiler -input:model"
    If you want to you can also refer to another output folder by adding the "*-output:output*" argument.
5. If you don't specify any output folder and you compiling was successful the soar and jess files are generated into the "*output*" folder.

3. Requirements for an agent

Every agent in Herbal consists of several components. These components are separated by the structure of Herbal (Figure 1). In order to create an agent, every developer needs at least one element of all the components. Before getting started with the design of any agent, let's thing about what problem is to be solved. This is very helpful to understand the practical lessons in this tutorial.

The vacuum cleaner task is relatively simple. There is a board which has several fields. Some of these fields are dirty and some are not. After initiation on one field a vacuum cleaner is randomly created. This vacuum has a sensor which allows him to see only one field ahead in every of the four directions. The task is to clean all the dirty parts on the board.

The next step is to think about a strategy to solve the task. In this tutorial we chose an easy solution. There are three different problem spaces in which the agent could be (Figure 2).

1. The *"clean"* problem space, which means that the vacuum cleaner is standing on a dirty field. In this situation he needs to suck up the dirt.
2. The *"pursuit"* problem space, which means that the vacuum cleaner is not directly on a dirty field, but there is one on his radar. So he needs to move in this particular direction.
3. The *"wander"* problem space, which means that the vacuum cleaner is not on a dirty field, and there is no dirty field nearby. Therefore he needs to move in a random direction to maybe find some dirt.

These three problem spaces depend on several operators. The operators depend on actions and conditions and the actions and conditions depend on types. The whole agent structure is shown in Figure 2. This Figure 2 is just for orientation. A better understanding will be given later in this tutorial.

The arrow on the right side of Figure 2 indicates the direction the tutorial takes. At the beginning stands the creation of the types and at the end the agent. This way there are several advantages, for example no pointing to elements that do not exist, or less problems with the Herbal GUI editor.

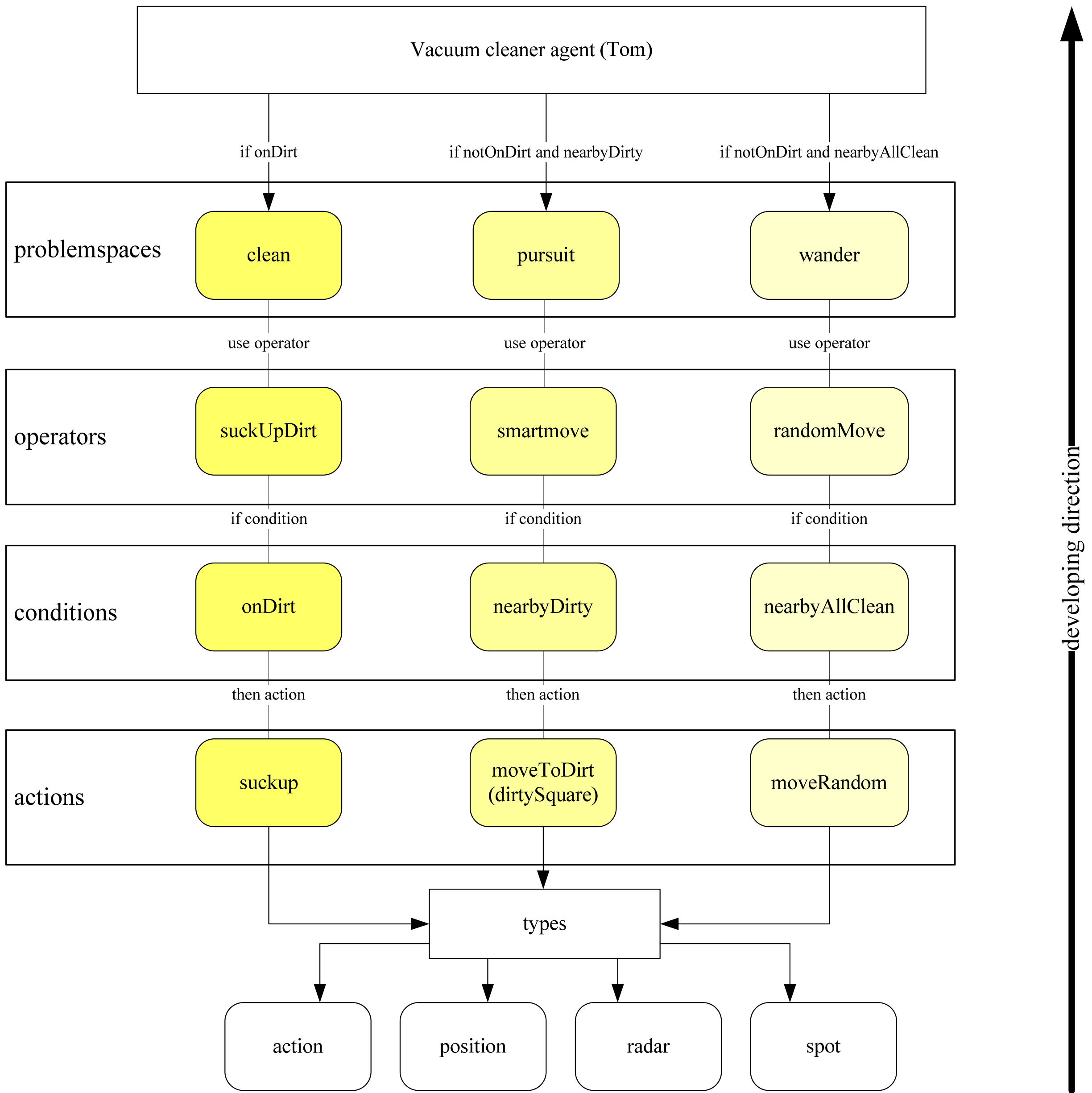

Figure 2: vacuum cleaner agent structure

4. Types.XML

As shown in Figure 2, the first step is creating the types. Figure 1 shows that every agent has a collection of types. The types determine the input and output data stream and the working memory of your agent. The "types.xml" files contain all existing types in your current project and all the field values.

If you are starting with a new environment it is a good idea to create a new types library for this particular environment. In this way you can reuse this library for other Herbal projects with the same environment. To get a better understanding of the "types.xml" file it is necessary to take a look at the "types.xsl" file.

4.1. Schema for Types (XSD)

The "types.xsl" file allows one to describe how "types.xml" files encoded in the XML standard are to be formatted. It works like a blueprint for the XML file structure. It shows you the rules for building the XML files that using this schema file. The structure of the "types.xsd" is shown in Figure 3. Every XML element which is created within the "types.xml" file has to follow the rules and regulations of this file. For example, every file starts with a root node *types* and this node has an unbounded number of *type* elements. The *type* node always has to have the attribute *name* of the type *xs:ID*.

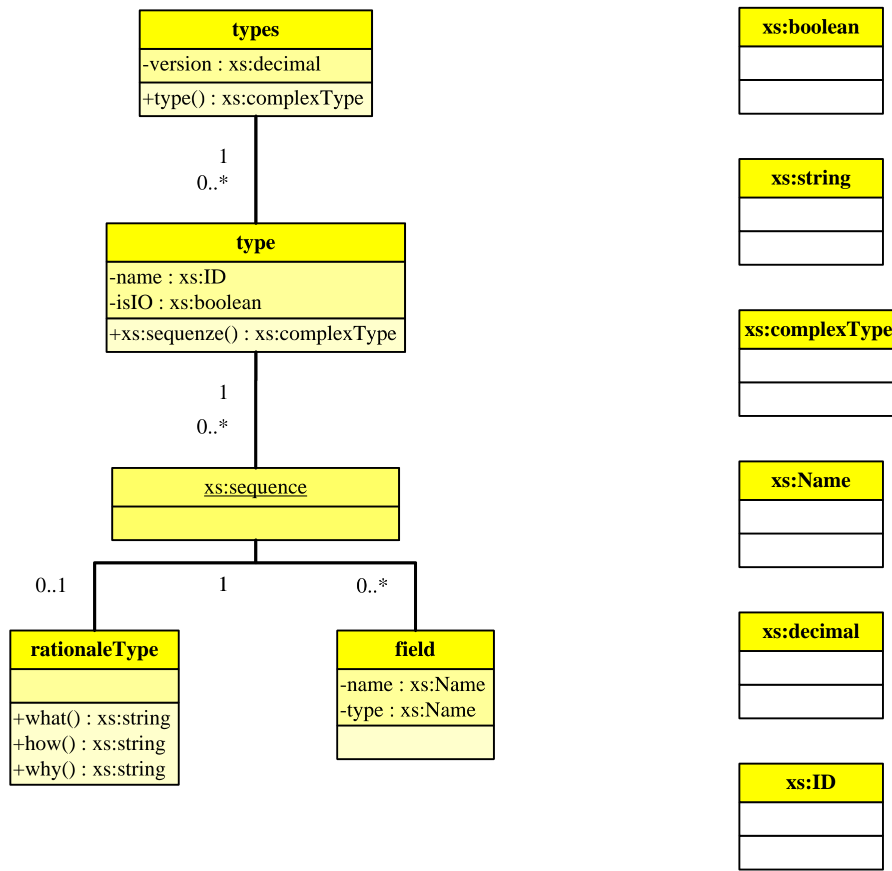

Figure 3: types.xsd structure

As you can see in Figure 3, every *type* element can have one rational element and an infinite number of field elements. This makes it easier to organize your *type* elements into groups to get a better overview of your input and output interfaces.

4.2. Lesson 1: Create the Types

1. Create a new project called *"VacuumCleaner"*, as described in Chapter 2.3.
2. Like mentioned earlier it is always a good thing to create a new library if you work with a new environment for the first time. So let's start by creating a library file.
    a. Make a right click on the model folder within your project and select *New > Others…*.
    b. From the *General* folder select *Untitled Text File* and press *Finish.*
    c. At the top select *File* and press *Save As…*.
    d. In the *Save As* frame select the *VacuumCleaner* project and the subfolder *models* will pop open. Set the *File name* to "vacuum.types.xml" and press *Ok*.
    e. Open the "vacuum.types.xml" with the Text Editor.
3. The new library file has to be completely empty and the first thing to do is add XML Code 1, so the file is in the same namespace as the rest of your Herbal project.

```
<?xml version='1.0'?>
<types version='1.0'
      xmlns='http://acs.ist.psu.edu/herbal'
```

```
	xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
	xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/types.xsd'>
</types>
```

XML Code 1: vacuum.types.xml at the beginning

4. The first type that you add is the *spot* type. It looks like XML Code 2. As you can see the *isIO* option is *true*. This means that this type is implemented as an interface, and can be used to communicate with the environment. This is essential for all types that are used as an input or output stream of data.

```
<type name='spot' isIO='true'>
  <rationale>
    <what>This type represents the condition of the current square (clean or dirty).</what>
    <how>It gets this information from the environment</how>
    <why>To interact with the environment</why>
  </rationale>
  <field name='status' type='VacuumCleaner.types.string'/>
</type>
```

XML Code 2: spot type

5. The next type you have to add is the *action* type. It looks like the spot type (XML Code 2), except the type attribute has changed and it has the XML Code 3 field element.

```
<field name='move' type='VacuumCleaner.types.string'/>
```

XML Code 3: action field

6. The next type you have to add is the *radar* type. It looks like the spot type (XML Code 2), except the type attribute has changed and it has the XML Code 4 field elements.

```
<field name='dir' type='VacuumCleaner.types.string'/>
<field name='reading' type='VacuumCleaner.types.string'/>
```

XML Code 4: radar fields

7. The last type you have to add is the *position* type. It looks like the spot type (XML Code 2), except the type attribute has changed and it has the XML Code 4 field elements.

```
<field name='x' type='VacuumCleaner.types.number'/>
<field name='y' type='VacuumCleaner.types.number'/>
```

XML Code 5: position fields

At the end, your “vacuum.types.xml” file should look like XML Code 6. As you may notice, for this tutorial the automatically generated “vacuumcleaner.types.xml” file is not used. This is no problem, because Herbal doesn’t need the automatically generated files to be filled with data. If you want to, you can create multiple libraries in your model folder. Herbal takes all files together that have the same ending (for example “vacuum.types.xml”, “vacuumcleaner.types.xml”, and “reactor.types.xml”) and compiles them in case there are no errors. It also shows you all the libraries in the GUI editor combined together.

```
<?xml version='1.0'?>
<types version='1.0'
	xmlns='http://acs.ist.psu.edu/herbal'
	xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
	xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/types.xsd'>

	<type name='position' isIO='true'>
```

```
            <rationale>
                  <what> This type represents the condition of the current position.</what>
                  <how> It gets this information from the environment </how>
                  <why> To interact with the environment </why>
            </rationale>
            <field name='x' type='VacuumCleaner.types.number'/>
            <field name='y' type='VacuumCleaner.types.number'/>
      </type>

      <type name='radar' isIO='true'>
            <rationale>
                  <what> This type represents the condition of the current radar.</what>
                  <how> It gets this information from the environment </how>
                  <why> To interact with the environment </why>
            </rationale>
            <field name='dir' type='VacuumCleaner.types.string'/>
            <field name='reading' type='VacuumCleaner.types.string'/>
      </type>

      <type name='action' isIO='true'>
            <rationale>
                  <what> This type represents the action you are able to perform.</what>
                  <how> It gets this information from the environment </how>
                  <why> To interact with the environment </why>
            </rationale>
            <field name='move' type='VacuumCleaner.types.string'/>
      </type>

      <type name='spot' isIO='true'>
            <rationale>
              <what>This type represents the condition of the current square (clean or dirty).</what>
               <how> It gets this information from the environment</how>
               <why>To interact with the environment</why>
            </rationale>
            <field name='status' type='VacuumCleaner.types.string'/>
      </type>
</types>
```

XML Code 6: vacuum.types.xml at the end

5. Actions.XML

The following step is by creating the actions (see Figure 2). As shown in Figure 1 every agent has a collection of actions which determine how the agent interacts with the environment. The files with the ending “actions.xml” in your “model” folder contain all existing actions in your current project and all their action clauses.

If you want to, you can also create your own library files for the different actions of your agent. This tutorial does not use this option, but if you want to you just have to repeat step two in section 4.2 to create separate “actions.xml” files. This creates a better reusability of your project.

5.1. Schema for Actions (XSD)

The "actions.xsl" file allows one to describe how "actions.xml" files encoded in the XML standard are to be formatted. The structure of the "actions.xsd" is shown in Figure 4. Every XML element which is created within the "actions.xml" files has to follow the rules and regulations of this file. For example, every *print* element is from type *printType* and has an attribute *order* from type *xs:string*.

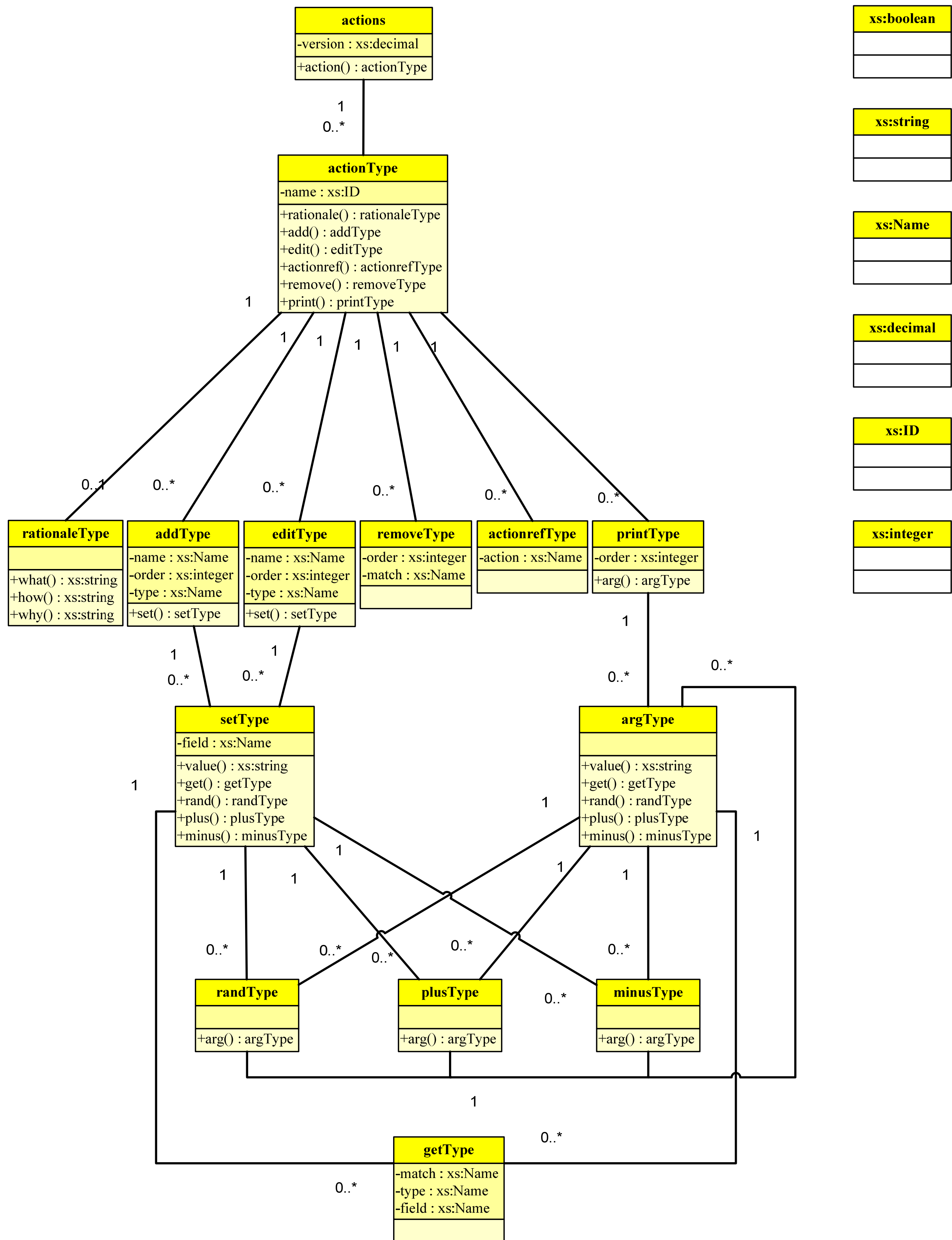

Figure 4: action.xsl structure

5.2. Lesson 2: Create the Actions that Tom can perform

1. Go to the model folder and open the “vacuumCleaner.action.xml” file. At the beginning the new automatically generated file looks like XML Code 12.

```
<?xml version='1.0'?>
<actions version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/actions.xsd'>

</actions>
```

XML Code 7: "vacuumClenaer.actions.xml" at the beginning

2. The first action you are going to create is the *suckup* action. As you can see in XML Code 8 the action is to add a value “suck” to the type “vacuum.types.action”, which you had defined in section 4.2 step 5. This “suck” tells the vacuumclener environment to clean the current field.

```
<action name='suckup'>
<rationale>
<what> This type represents the action to clean a square.</what>
<how>By setting the move value to suck</how>
<why>To clean up</why>
</rationale>
<add order='0' type='vacuum.types.action'>
<set field='move'><value>suck</value></set>
</add>
</action>
```

XML Code 8: *suckup* action

3. The next action you need is *movetoDirt.* It is similar to the *suckup* action, the only thing you need to change is the action attribute name and the *set* element. The *set* element is used to move the agent into a specified direction and it looks like XML Code 9. As you can see you can also use other types to set the new types. In this case you take the value from the radar and set the *move* value to it, this means your agent moves in the direction of a dirty field.

```
<set field='move'><get match='dirtySquare' type='vacuum.types.radar' field='dir'/></set>
```

XML Code 9: movetoDirt action

4. The next action you need is *randomMove*. It is also similar to the suckup action; the only thing you need to change is the action attribute name and the *set* element. The *set* element should look like XML Code 10. For this action the random function needs to be used to determine in what direction the agent should move next.

```
<set field='move'>
  <rand>
    <arg><value>"left"</value></arg><arg><value>"right"</value></arg>
    <arg><value>"up"</value></arg><arg><value>"down"</value></arg>
  </rand>
</set>
```

XML Code 10: randomMove action

At the end, your “vacuumcleaner.actions.xml” file should look like XML Code 11.

```xml
<?xml version='1.0'?>
<actions version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/actions.xsd'>

    <action name='suckup'>
        <what> This action represents the action to clean a square.</what>
         <how>By setting the move value to suck</how>
         <why>To clean up</why>
        <add order='0' type='vacuum.types.action'>
            <set field='move'><value>suck</value></set>
        </add>
    </action>

    <action name='moveToDirt'>
        <rationale>
            <what> This action represents the action to move to dirt</what>
            <how> Getting the information from the radar and move in this direction</how>
            <why> To make a smart move</why>
        </rationale>
        <add order='0' type='vacuum.types.action'>
            <set field='move'><get match='dirtySquare' type='vacuum.types.radar' field='dir'/></set>
        </add>
    </action>

    <action name='moveRandom'>
        <rationale>
            <what> This action represents the action to move into a random direction  </what>
            <how> Using a random generator to decide between the four different values</how>
            <why> To do something even if no dirt is nearby</why>
        </rationale>
        <add order='0' type='vacuum.types.action'>
         <set field='move'>
           <rand>
             <arg><value>"left"</value></arg><arg><value>"right"</value></arg>
             <arg><value>"up"</value></arg><arg><value>"down"</value></arg>
           </rand>
         </set>
        </add>
    </action>
</actions>
```

XML Code 11: vacuumCleaner.action.xml at the end

6. Conditions.XML

According to Figure 2, the conditions are the next part to create. As shown in Figure 1 every agent has a collection of conditions which determine when the agent has to enter a problem space or use an operator. The “conditions.xml” files contain all existing conditions in your current project. For a better understanding of the XML file it is necessary to take a look at the "conditions.xsl" file.

6.1. Schema for Conditions (XSD)

The "conditions.xsl" file allows one to describe how "condtions.xml" files encoded in the XML standard are to be formatted. The structure of the "conditions.xsd" is shown in Figure 5. Every XML element which is created within the "conditions.xml" file has to follow the rules and regulations of this file. For example, every *conditions* node has a unbounded number of *condition* and this node always has to have the attribute *name* of the type *xs:ID*.

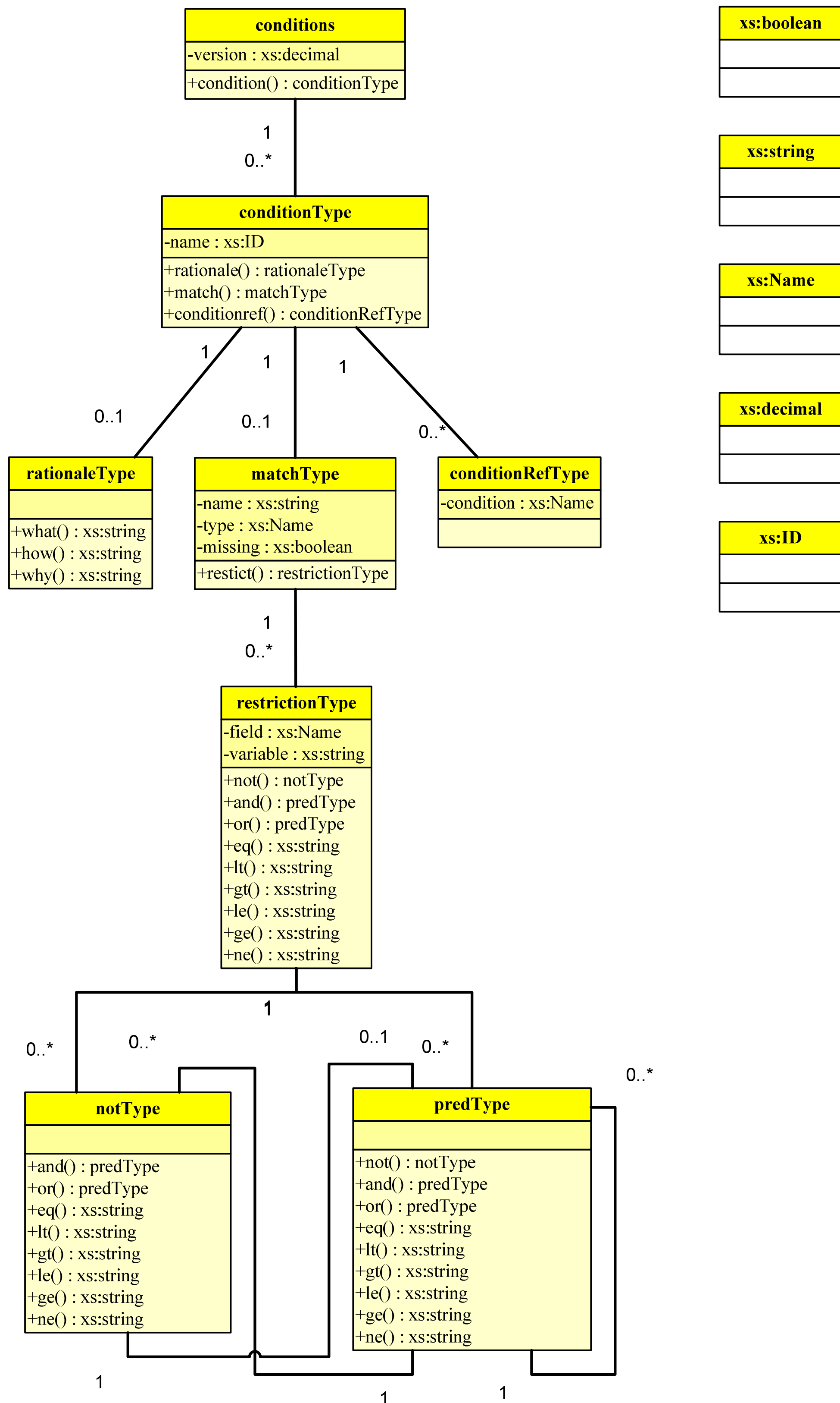

Figure 5: conditions.xsd structure

6.2. Lesson 3: Create the Conditions under which Tom has to work

1. Go to the model folder an open the conditions.xml file. At the beginning the new automatically generated file looks like XML Code 12.

```
<?xml version='1.0'?>
<conditions version='1.0'
     xmlns='http://acs.ist.psu.edu/herbal'
     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/conditions.xsd'>
</conditions>
```

XML Code 12: vacuumcleaner.conditions.xml at the beginning

2. The first thing you have to add is the *onDirt* condition. As you can see in XML Code 13, unlike in Figure 5 you don't need to add a *condionref* element. The *match* element says that if the *status* field of the type *vacuum.types.spot* has the value *"dirty"* the condition complies with the requirements.

```
<condition name='onDirt'>
        <rationale>
                <what> This condition represents the state of current position (dirty)  </what>
                <how>Getting information from the input type vacuum.types.spot</how>
                <why> To decide to move around </why>
        </rationale>
        <match name='StandNoOnDirt' type='vacuum.types.spot'>
                <restrict field='status'><eq>dirty</eq></restrict>
        </match>
</condition>
```

XML Code 13: onDirt condition

3. The next condition we need is the *NotDirty* condition. It looks similar to the XML Code 13, but the name of the condition needs to be changed and also the *match* value has to be changed into XML Code 14. As you can see, the same input variable is used to declare if the spot the cleaner is currently standing on is *clean*.

```
<match name='NotDirty' type='vacuum.types.spot'>
        <restrict field='status'><eq>clean</eq></restrict>
</match>
```

XML Code 14: NotDirty condition

4. Now you need a condition to support your *smartMove* operator called *nearbyDirty*. The *match* element for this condition looks like XML Code 15. This match is also used to get directions which of the nearby fields are *dirty*.

```
<match name='dirtySquare' type='vacuum.types.radar'>
        <restrict field='reading'><eq>dirty</eq></restrict>
        <restrict field='dir'></restrict>
</match>
```

XML Code 15: nearbyDirty

5. The last condition you need is the *nearbyAllClean* condition. This condition has four match elements with two possible states. There could be a *wall* or it could be *clean*. So you need to

check for every direction (up, down, right, left) and for the state in which the field is. XML Code 16 gives you the match rule for the up direction, which you can copy to create the missing three.

```
<match type='vacuum.types.radar'>
        <restrict field='dir'><eq>up</eq></restrict>
        <restrict field='reading'><or><eq>wall</eq><eq>clean</eq></or></restrict>
</match>
```

XML Code 16: nearbyAllClean match

After you have finished the steps one to five your "vacuumcleaner.conditions.xml" file should look similar to

```
<?xml version='1.0'?>
<conditions version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/conditions.xsd'>

    <condition name='onDirt'>
        <rationale>
            <what> This condition represents the state of current position (dirty)  </what>
             <how> Getting information from the input type vacuum.types.spot</how>
             <why> To decide to suck up</why>
        </rationale>
        <match name='StandNoOnDirt' type='vacuum.types.spot'>
            <restrict field='status'><eq>dirty</eq></restrict>
        </match>
    </condition>

   <condition name='NotOnDirt'>
        <rationale>
            <what> This condition represents the state of current position (clean)</what>
            <how>  Getting information from the input type vacuum.types.spot </how>
            <why> To decide to move around </why>
        </rationale>
        <match name='NotDirty' type='vacuum.types.spot'>
            <restrict field='status'><eq>clean</eq></restrict>
        </match>
    </condition>

    <condition name='nearbyDirty'>
        <rationale>
            <what> This condition represents the state of radar fields</what>
            <how> Getting information from the input type vacuum.types.radar</how>
            <why> To decide if smartMove is possible</why>
        </rationale>
        <match name='dirtySquare' type='vacuum.types.radar'>
            <restrict field='reading'><eq>dirty</eq></restrict>
            <restrict field='dir'></restrict>
        </match>
    </condition>

    <condition name='nearbyAllClean'>
        <rationale>
            <what> This condition represents the state of radar fields </what>
            <how> Getting information from the input type vacuum.types.radar </how>
```

```
            <why> To decide if randomMove is possible </why>
        </rationale>
        <match type='vacuum.types.radar'>
            <restrict field='dir'><eq>up</eq></restrict>
            <restrict field='reading'><or><eq>wall</eq><eq>clean</eq></or></restrict>
        </match>
        <match type='vacuum.types.radar'>
            <restrict field='reading'><or><eq>wall</eq><eq>clean</eq></or></restrict>
            <restrict field='dir'><eq>down</eq></restrict>
        </match>
        <match type='vacuum.types.radar'>
            <restrict field='reading'><or><eq>wall</eq><eq>clean</eq></or></restrict>
            <restrict field='dir'><eq>left</eq></restrict>
        </match>
        <match type='vacuum.types.radar'>
            <restrict field='reading'><or><eq>wall</eq><eq>clean</eq></or></restrict>
            <restrict field='dir'><eq>right</eq></restrict>
        </match>
    </condition>
</conditions>
```

XML Code 17: conditions.xml at the end

7. Operators.XML

According to Figure 2, the operators are the next part to be created. As shown in Figure 1 every agent has a collection of operators which determine the actions that are performed under certain conditions. The “operators.xml” files contain all existing operators in your current project. For a better understanding of the “operators.xml” files it is necessary to take a look at the "operators.xsl" file.

7.1. Schema for Operators (XSD)

The "operators.xsl" file allows one to describe how the "operators.xml" files encoded in the XML standard are to be formatted. The structure of the "operator.xsd" is shown in Figure 6. Every XML element which is created within the "operators.xml" files has to follow the rules and regulations of this file. For example, every *if* node is from the type *ifType* has a one or zero *init Elements* of the type *initType.*

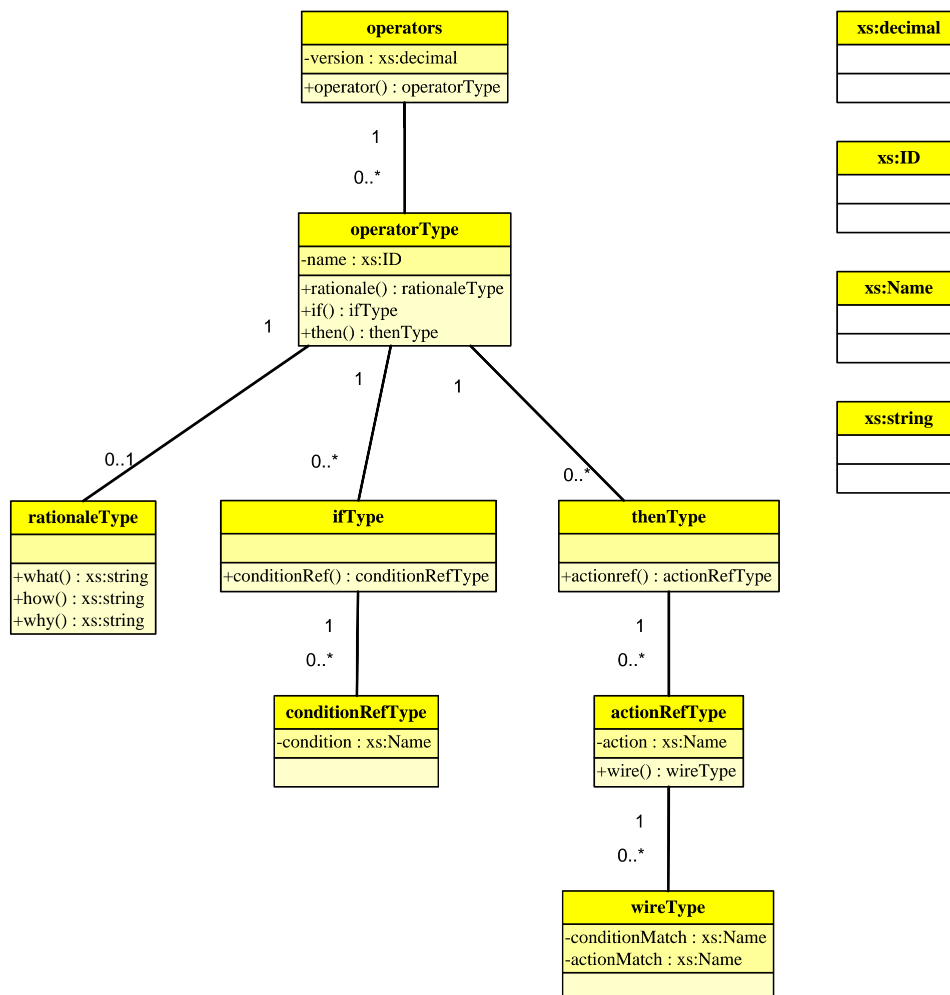

Figure 6: operator.xsl structure

7.2. Lesson 4: Create the Operators that can be used by Tom

1. Open the empty “vacuumcleaner.operators.xml” file. The file should look like this:

```
<?xml version='1.0'?>
<operators version='1.0'
     xmlns='http://acs.ist.psu.edu/herbal'
     xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
     xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/operators.xsd'>

</operators>
```

XML Code 18: "vacuumcleaner.operators.xml" at the beginning

2. The first operator you should create is “*suckupDirt”*. As you can see in XML Code 19, this operator combines the *onDirt* condition from lesson 3 with the *suckup* action from lesson 2.

```
<operator name='suckupDirt'>
          <rationale>
                    <what> This operator combines onDirt with suckup  </what>
                    <how>If onDirt then suckup</how>
                    <why> to bind a certain condition to an action</why>
          </rationale>
          <if>
                    <conditionref condition='onDirt'/>
          </if>
          <then>
                    <actionref action='suckup'>
                    </actionref>
          </then>
</operator>
```

XML Code 19: suckupDirt Operator

3. The next operator you create by binding the *nearbyDirty* condition to the *moveTodirt* action. To do this, use the XML Code 19 and change the operator attribute name to *smartMove*. Also you need to change the *if* and *then* elements to XML Code 20. As you can see you can use the wire element to pass values from the condition to the action.

```
<if>
          <conditionref condition='nearbyDirty'/>
</if>
<then>
          <actionref action='moveToDirt'>
                    <wire conditionMatch='dirtySquare' actionMatch='dirtySquare'/>
          </actionref>
</then>
```

XML Code 20: smartMove operator

4. The last operator you need to create is the *randomMove* operator. This operator connects the *nearbyAllClean* condition to the *moveRandom* action. To create this operator take the XML Code 19 and change the name to *randomMove* and the if and then elements to XML Code 21.

```
<if>
        <conditionref condition='nearbyAllClean'/>
</if>
<then>
        <actionref action='moveRandom'>
        </actionref>
</then>
```

XML Code 21: randomMove operator

After you have finished the steps one to five your “vacuumcleaner.operators.xml” file should look similar to

```
<?xml version='1.0'?>
<operators version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/operators.xsd'>

    <operator name='suckupDirt'>
        <rationale>
            <what> This operator combines onDirt with suckup  </what>
            <how>If onDirt then suckup</how>
            <why> to bind a certain condition to an action</why>
        </rationale>
        <if>
            <conditionref condition='onDirt'/>
        </if>
        <then>
            <actionref action='suckup'>
            </actionref>
        </then>
    </operator>

    <operator name='smartMove'>
        <rationale>
            <what> This operator combines nearbyDirty with moveToDirt  </what>
            <how>If nearbyDirty then moveToDirt  </how>
            <why> to bind a certain condition to an action</why>
        </rationale>
        <if>
            <conditionref condition='nearbyDirty'/>
        </if>
        <then>
            <actionref action='moveToDirt'>
                <wire conditionMatch='dirtySquare' actionMatch='dirtySquare'/>
            </actionref>
        </then>
    </operator>

    <operator name='randomMove'>
        <rationale>
            <what> This operator combines nearbyAllClean with moveRandom  </what>
            <how>If nearbyAllClean then moveRandom  </how>
            <why> to bind a certain condition to an action</why>
        </rationale>
        <if>
            <conditionref condition='nearbyAllClean'/>
```

```
            </if>
            <then>
                  <actionref action='moveRandom'>
                  </actionref>
            </then>
      </operator>
</operators>
```

XML Code 22: vacuumcleaner.operators.xml at the end

8. Problemspace.XML

The next thing we need to create are the problem spaces. As shown in Figure 1 every agent has a collection of problem spaces which determine how the agent reacts under certain initial actions and operators. This tab contains all existing problem spaces in your current project without any structural information of how the problem spaces are linked. This information can only be found in the "problemspaces.xml" files. For a better understanding of the "problemspaces.xml" files it is necessary to take a look at the "problemspaces.xsl" file.

8.1. Schema for Problemspace (XSD)

The "problemspaces.xsl" file allows one to describe how "problemspaces.xml" files encoded in the XML standard are to be formatted. The structure of the "problemspaces.xsd" is shown in Figure 7. Every XML element which is created within the "problemspaces.xml" file has to follow the rules and regulations of this file. For example, every *problemspace* node is from the type *problemspaceType* has a one or zero *init Elements* of the type *initType.*

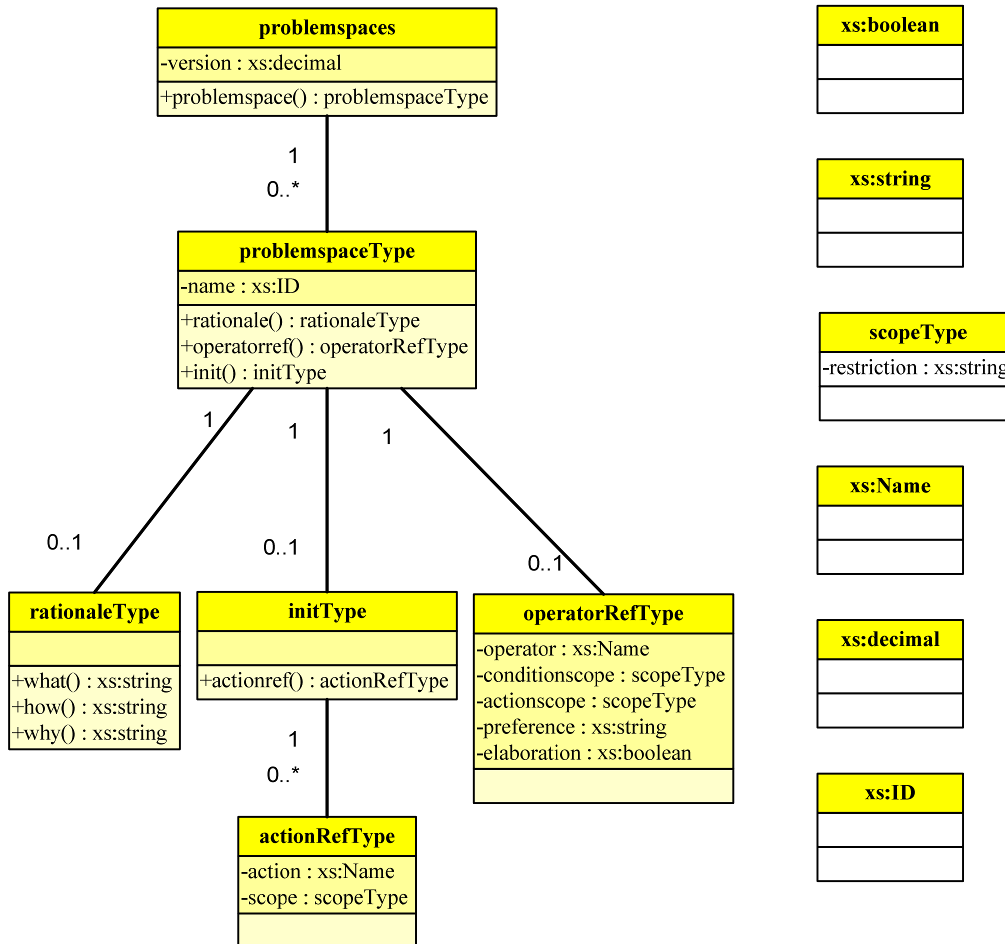

Figure 7: problemspaces.xsd structure

8.2. Lesson 5: Create problem spaces for Tom

Now we have to create the problem spaces which we will use in Section 9.2 to define the agent. This problem spaces are „*Top*“, „*Pursue*“, „*Clean*“ and „*Wander*“. To do this, go to the "model" folder and open the *"VacuumCleaner.problemspaces.xml"*, as described in chapter 2.4. Now follow these steps:

1. The *"VacuumCleaner.problemspaces.xml"* file should look like XML Code 23.

```
<?xml version='1.0'?>
<problemspaces version='1.0'
        xmlns='http://acs.ist.psu.edu/herbal'
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/problemspaces.xsd'>
</problemspaces>
```

XML Code 23: problemspaces.xml at the beginning

2. To define the *"Top"* problem space use XML Code 24.

```
<problemspace name='Top'>
            <rationale>
                        <what></what>
                        <how></how>
                        <why></why>
            </rationale>
            <init>
            </init>
</problemspace>
```
XML Code 24: top problem space

3. To define the *"Clean"* problem space, you can use the same structure like XML Code 24. Just change the name attribute to *"Clean"* and add the *operatorref* from XML Code 25.

```
<operatorref actionscope='top' conditionscope='top' operator='suckupDirt' elaboration='false'/>
```
XML Code 25: clean operatorref

4. To define the *"Pursue"* problem space, you can use the same structure like XML Code 24. Just change the name attribute to *"Pursue"* and add the *operatorref* from XML Code 26.

```
<operatorref actionscope='top' conditionscope='top' operator='smartMove' elaboration='false'/>
```
XML Code 26: pursue operatorref

5. To define the *"Wander"* problem space, you can use the same structure like XML Code 24. This time you have to add two more *operatorref* and also an *actionref* for the initial state. You can use XML Code 27 to do this.

```
<init>
            <actionref action='moveRandom' scope='top'/>
</init>
<operatorref actionscope='top' conditionscope='top' operator='randomMove' elaboration='false'/>
```
XML Code 27: wander actionref and operatorref

At the end, your problemspaces.xml file should look like XML Code 28.

```
<?xml version='1.0'?>
<problemspaces version='1.0'
      xmlns='http://acs.ist.psu.edu/herbal'
      xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
      xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/problemspaces.xsd'>

      <problemspace name='Clean'>
            <rationale>
                        <what> This operator combines nearbyAllClean with moveRandom  </what>
                        <how>If nearbyAllClean then moveRandom  </how>
                        <why> to bind a certain condition to an action</why>
            </rationale>
            <init>
            </init>
            <operatorref actionscope='top' conditionscope='top' operator='suckupDirt' elaboration='false'/>
      </problemspace>

      <problemspace name='Wander'>
            <rationale>
                  <what></what>
```

```
                <how></how>
                <why></why>
        </rationale>
        <init>
        <actionref action='moveRandom' scope='top'/>
        </init>
        <operatorref actionscope='top' conditionscope='top' operator='randomMove' elaboration='false'/>
    </problemspace>

    <problemspace name='Pursue'>
        <rationale>
                <what></what>
                <how></how>
                <why></why>
        </rationale>
        <init>
        </init>
        <operatorref actionscope='top' conditionscope='top' operator='smartMove' elaboration='false'/>
    </problemspace>

    <problemspace name='Top'>
        <rationale>
                <what></what>
                <how></how>
                <why></why>
        </rationale>
        <init>
        </init>
    </problemspace>
</problemspaces>
```

XML Code 28: problemspaces.xml at the end

9. Models.XML

As shown in Figure 1 the concept of agents forms the top level of any Herbal project and is positioned in the “model.xml” files. These files containing all existing agents in your current project and the problem spaces they use. For a better understanding of the XML file it is necessary to take a look at the "models.xsl" file.

9.1. Schema for Models (XSD)

The "models.xsl" file allows one to describe how "model.xml" files encoded in the XML standard are to be formatted. The structure of the "models.xsd" is shown in Figure 8. Every XML element that is created within the "model.xml" file has to follow the rules and regulations of this file. For example, every file has to start with the root node *models* and this node always has to have the attribute *version* of the type *xs:decimal.*

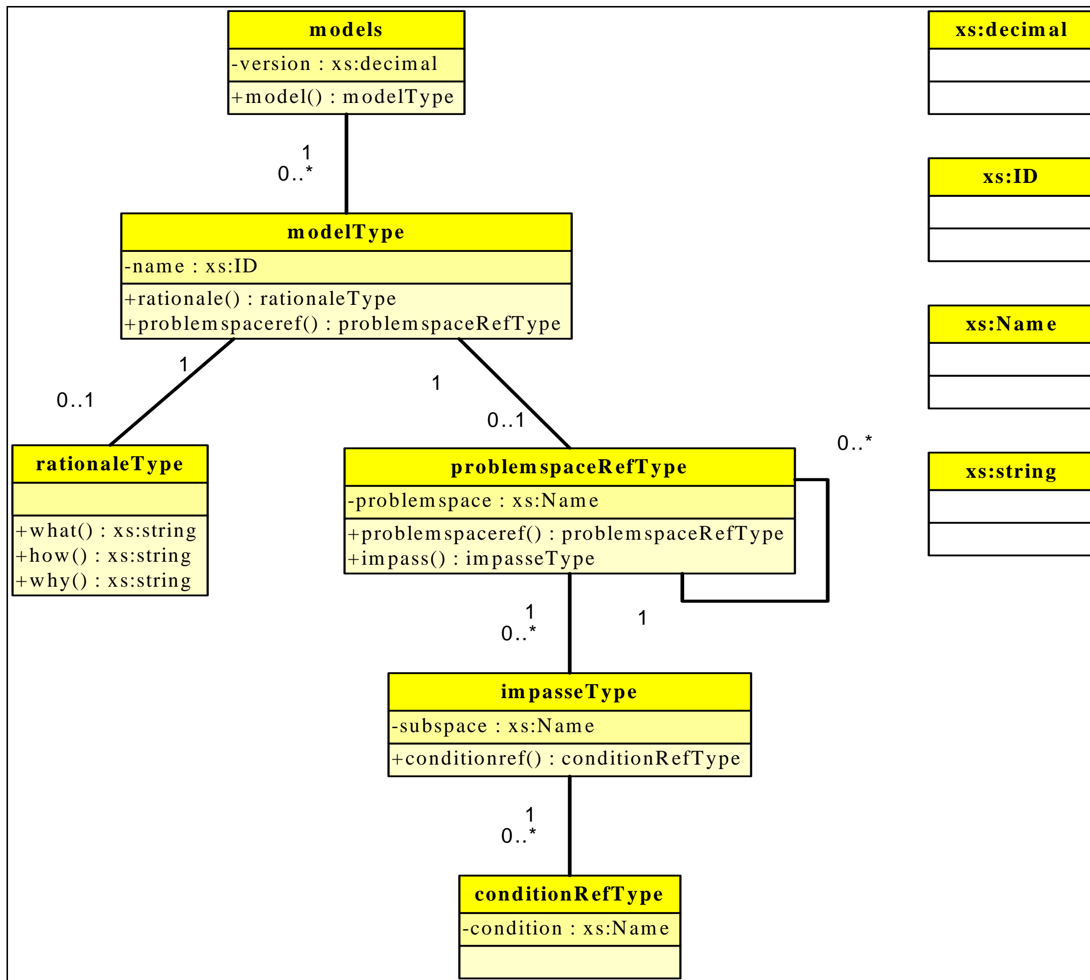

Figure 8: model.xsd structure

9.2. Lesson 6: Creating a vacuum cleaner agent named Tom

The last step in order to create a complete model is to build an agent which combines all the problem spaces. We will create an agent called Tom with the following steps:

1. Go to the "model" folder and open the *"VacuumCleaner.models.xml"*, as described in Chapter 2.4.
2. The file should look like XML Code 29.

```
<?xml version='1.0'?>
<models version='1.0'
        xmlns='http://acs.ist.psu.edu/herbal'
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/models.xsd'>
</models>
```

XML Code 29: model.xml at the beginning

3. The first thing we need is a *"model tag"* with an attribute *"name=Tom"*, see XML Code 30.

```
<model name='Tom'>
</model>
```

XML Code 30: model Tag

4. As shown in Figure 8, every *model* is allowed to have the sub tags *rationale* and *problemspaceref*.
   a. To generate a *rational* node with the *what* information *"This is the test agent for the vacuum cleaner environment"* use XML Code 31.

```
<rationale>
		<what>This is the test agent for the vacuum cleaner environment</what>
		<how></how>
		<why></why>
</rationale>
```

XML Code 31: model rational tag

   b. To generate a *problemspaceref* node with attribute *problemspace=top* use XML Code 32

```
<problemspaceref problemspace='Top'>
</problemspaceref>
```

XML Code 32: model problemspaceref tag

5. The next step is to create a sub problem space under the problem space "Top".
   a. To create a problem space *"Wander"* with the conditions *nearbyAllClean* and *NotOnDirt* use XML Code 33. As you can see the conditions are not directly under the problem space node. The reason for this is to keep the architecture (how the problemspaces are connected) independent from the impasses.

```
<problemspaceref problemspace='Wander'>
</problemspaceref>
<impasse subspace='Wander'>
		<conditionref condition='nearbyAllClean'/>
		<conditionref condition='NotOnDirt'/>
</impasse>
```

XML Code 33: wander problemspace

   b. In the next step we create the problemspace *"Pursue"* with the conditions *"nearbyDirty"* and *"NotOnDirt"*. XML Code 34 shows how this should look.

```
<problemspaceref problemspace='Pursue'>
</problemspaceref>
<impasse subspace='Pursue'>
		<conditionref condition='NotOnDirt'/>
		<conditionref condition='nearbyDirty'/>
</impasse>
```

XML Code 34: pursue problem space

   c. In the next step we create the problem space *"Clean"* with the condition *"onDirt"*. XML Code 35 shows how this should look.

```
<problemspaceref problemspace='Clean'>
 </problemspaceref>
<impasse subspace='Clean'>
		<conditionref condition='onDirt'/>
```

```
</impasse>
```

XML Code 35: clean problem space

In the end, your models.xml file should look like XML Code 36.

```
<?xml version='1.0'?>
<models version='1.0'
    xmlns='http://acs.ist.psu.edu/herbal'
    xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
    xsi:schemaLocation='http://acs.ist.psu.edu/herbal ../schema/models.xsd'>
    <model name='Tom'>
        <rationale>
            <what></what>
            <how></how>
            <why></why>
        </rationale>
        <problemspaceref problemspace='Top'>
            <problemspaceref problemspace='Clean'>
            </problemspaceref>
            <problemspaceref problemspace='Pursue'>
            </problemspaceref>
            <problemspaceref problemspace='Wander'>
            </problemspaceref>
            <impasse subspace='Clean'>
                <conditionref condition='onDirt'/>
            </impasse>
            <impasse subspace='Pursue'>
                <conditionref condition='NotOnDirt'/>
                <conditionref condition='nearbyDirty'/>
            </impasse>
            <impasse subspace='Wander'>
                <conditionref condition='nearbyAllClean'/>
                <conditionref condition='NotOnDirt'/>
            </impasse>
        </problemspaceref>
    </model>
</models>
```

XML Code 36: models.xml at the end

10. Running the Agent

Reference to (Cohen, Ritter, & Bhandarkar, 2007) Chapter 9.10

11. Problems

In this chapter you will get an overview of typical errors that can occur while you are programming your agent. If you have a problem, there are several things you should check before starting to rewrite your code. For example you may have made typing errors.

There are two ways to debug your project. The first is to use Eclipse. Whenever you make a mistake with the spelling an Eclipse error is created. It will tell you that you are using an "undefined model element". You can see this in the Problems tab of your Eclipse SDK. The second way is to use the command line compiler, described in section 2.5. For example, if you have a typing error in one of your actions the compiler will tell you that you try to use an action which does not exist within your "operators.xml" file. This could look like this:

"undefined model element: VacuumCleaner.actions.moveRandom
File:model\VacuumCleaner.operators.xml Line:35 Col:20 Model parse error: undefined model element: VacuumCleaner.actions.moveRandom"

As you can see the error that appears in the command line and the Eclipse SDK are essentially the same. For this reason the following section only refers to the Eclipse SDK errors.

11.1. Name and typing errors

Name errors could derive from several mistakes, but the most common ones are mistyping and wrong symbols.

Mistyping is not only typing the wrong letters, it is also using capital letters instead of lower case. As you can see, there are several mistakes in XML Code 38, if XML Code 37 is the corresponding type library.

```
<type name='action' isIO='true'>
        <field name='move' type='VacuumCleaner.types.string'/>
</type>
```

XML Code 37: type case 1

```
<action name='&moveRandom'>
        <add order='0' type='vacuum.type.Actions'>
            <set field='moveAround'><rand><arg><value>12</value></arg></rand></set>
        </add>
</action>
```

XML Code 38: action case 1

1. The name of the action itself is wrong. It is only allowed to use “a-z”,”A-Z”, “0-9”, and “-_” symbols to create name values. It is not allows to start the name with “-“or “0-9”. You correct this by removing the “&” from the beginning of the name attribute in the action element.
2. The next mistake is that the type name is “action” and not “Actions”. If the compiler would run over the action file he would not find any matching type and would produce an “undefined model element” error. And also the name of the library file is “types”. You correct this by changing the type attribute of the element value to “vacuum.types.action”.
3. The action type has only one field and this is named “move”. The action element instead tries to set values in a field named “moveAround”. This would result in an “undefined field” error. You correct this by changing the field attribute in the action element to “move”.
4. The last mistake is that the field move is from type string, and the action tries to add a numeric value. This is a severe mistake, because the compiler has no chance of finding these kinds of errors. This means there no error will occur while compiling your project to a soar file, but your agent will not work. To correct this, change 12 to “left”.

11.2. Missing tags

This section shows you some errors that could be called “schema violation” errors. XML Code 39 shows the part of an “agents.xml” file with three different problem spaces. The *Top* problemspace is on the top of the agent and it has a child problem space called *Pursue* and pursue has a child prolemspace *Clean.*

```
<model name='Tom'>
        <rationale>
                </what>
        </rationale>
        <problemspaceref problemspace='Top'>
                <problemspaceref problemspace='Pursue'>
                        <problemspaceref problemspace='Clean'>
                        </problemspaceref>
                        <impasse subspace='Clean'>
                                <conditionref condition='onDirt'>
                        </impasse>
                </problemspaceref>
                <impasse subspace='Pursue'>
                        <conditionref condition='nearbyDirty'/>
        </problemspaceref>
</model>
```

XML Code 39: schema violation errors

1. The schema error in XML Code 39 is on line 3, where the *what* element field is closed. If you close a tag you always need to open it up first. You could correct this problem by putting in an open tag for the *what* element.
2. The next error is at the *conditionref* element with the attribute *onDirt*. This element does not have any other elements so it could be closed at the end of the line. To fix this you have two options, putting down a hole tag that says “</conditionref>” or just putting a “/” at the end of the line in front of the “>” symbol.
3. The last error is that the *impass* element, with the subspace attribute *Pursue*, has no closing tag either. You just have to add a closing *impasse* element under the *conditionref* element of this impasse.

References

Cohen, M. A., Ritter, F. E., & Bhandarkar, D. (2007). *A Gentle Introduction to Herbal.* Retrieved from Welcome To Herbal: http://acs.ist.psu.edu/projects/herbal/tutorials.html

Cohen, M. (2007, 07 12). *Vacuum Cleaner Environment Tutorial*. Retrieved from http://acs.ist.psu.edu/herbal/vacuum-tutorial.html