

**Development of SysML Executable Python Design Patterns for Ops Activity Models Simulation**

Wubing Ye

wye@caltech.edu

August 16, 2010

Space Grant Final Report

**Acknowledgements**

I would like to thank and acknowledge the following people for making this Space Grant project a great success:

My mentor, Leonard Reder, for bringing me on to this project, providing advice, and helping me with all the aspects of the project;

My co-mentor, Dr. Shang-Wen Cheng, for being the brilliant computer science guy;

The Ops Revitalization team especially Chris Delp, Maddalena Jackson, and Ryan Wollaehger;

Carol Casey and the SFP Office for answering all of my education program questions;

And of course friends and family.

**Table of Contents**

1. Introduction ……………………………….4
2. Background Information ………………….4
3. Design Pattern …………………………….5
4. Example Activity Diagrams ………………9
5. Summary ………………………………….10
6. Bibliography ………………………………11

**Introduction:**

The Operations Revitalization Project (Ops Rev.) is tasked to model flight operation processes using the System Modeling Language (SysML). The Ops Rev. project mainly uses the SysML to generate detailed documentation of existing processes, and desires the capability to develop simulations based on its models. These simulations will provide Ops Rev. with additional insights to how well existing processes work and where processes can be improved. To achieve such a simulator, it was desired to develop a Python design pattern that would capture the executable features of the SysML activity model. This pattern can then be used repeatedly in future implementations.

For this Space Grant project, the MagicDraw tool is used to generate a set of SysML activity models that illustrate particular activity features. The goal of this project is to, from a set of SysML activity example models, create a Python design pattern that can execute the activity diagrams and later be integrated into the JPL Statechart Autocoder.

**Background Information:**

An activity diagram is one of the nine diagrams of SysML. It is a behavior diagram, so it demonstrates system functionality and the flow of information (tokens) from action to action. Activity diagrams have several major features, of which we have implemented only a subset. The most basic features used in an activity model are shown in Figure 1.

Figure 1: Simple activity diagram called ProcedureExample with important features labeled.

The inputParameter box is where either the user or another program enters data tokens into the diagram – this is a SysML parameter box. From the inputParameter, tokens are parsed and propagated, via a data flow connection, to the input pin of the action (Step1). Each action is represented as an action node and it is within an action that a user-specified function processes tokens. Result tokens are then propagated to the output pins where they get passed on to the next action’s input pin per the model specification (e.g. data flow lines in the diagram). This cycle continues until all executable actions have finished and the resulting tokens are returned via the outputParameter box.

Other features of activity diagrams are the decision, merge, join, and fork nodes. Decision nodes take in tokens and transport them to different action nodes depending on which conditional test expressions are true. Merge nodes take in multiple tokens and funnel them to a single input pin. Join nodes also take in multiple tokens, but they wait until they have taken in all possible tokens before sending the tokens on to the input pin; this provides a synchronizing ability. Fork nodes take one token and copy it to multiple input pins.

A special feature of activity models is the “call action” which provides activities with a hierarchical feature. There is a top-level activity model that contains actions (known as call actions) that are actually embedded activity models. When the top-level’s call action is executed, it calls the embedded activity model which then proceeds to run as usual. The results of the called action are then passed back up to the top-level and then passed on to the next action node. This feature allows complex structures to be simplified for a better view of the entire system.

The activity diagrams are created using the MagicDraw Computer Aided Software Engineering (CASE) tool. A Jython plugin program annotates the MagicDraw model and prints the execution status to a MagicDraw console window while the Python executable activity code is running. As the activity model is stepped through, various executed actions are highlights. This was developed by Maddalena Jackson. The executable Python activity model code actually runs outside of MagicDraw. Dr. Shang-Wen Cheng developed an XML-RPC server that enables connection between the executable models and the MagicDraw Jython plugin. This provides a graphical display of the activity executing and was useful in illustrating that Python patterns executed properly.

**Design Pattern:**

The developed design pattern consists of 3 different classes. Classes describe different types of objects or instances in a system. Class diagrams are structure diagrams that show classes and their relationship with one another. A class diagram is actually a Unified Modeling Language (UML) diagram type and not a SysML type. The UML class diagram is useful here to describe the design and list attributes and methods used within the classes. Although class diagrams are not a part of the project, they are helpful in explaining the code structure. Figure 2 summarizes the Python design pattern using a class diagram scheme.

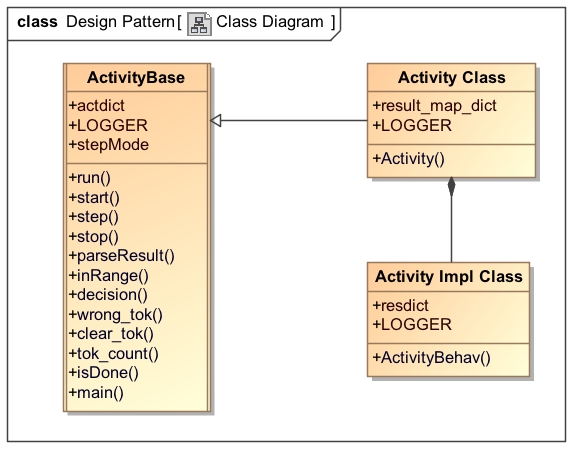


Figure 2: Class diagram of the Python design pattern.

The ActivityBase class is where the “actdict” dictionary attribute is accessed and read. In the Python language, dictionaries contain keys that are matched to definitions. The reason dictionaries are being used is because it is easy to map model trees in MagicDraw into dictionary form. The actdict stores all the action node and token information. Below is the actdict for the activity diagram, ProcedureExample, shown in Figure 1.

self.actdict = {“inputParameter” : {“stdin” : “no”,

“next\_step” : [“Step1”] },

“Step1” : {“argument” : {“source” : “inputParameter”,

“min\_tok” : 1,

“max\_tok” : 1,

“tok\_list” : [] },

“method” : self.Step1,

“next\_step” : “Step2”,

“exec\_status” : “Current” },

“Step2” : {“argument” : {“source” : “Step1”,

“min\_tok” : 1,

“max\_tok” : 1,

“tok\_list” : [] },

“method” : self.Step2,

“next\_step” : “return output parameters”,

“exec\_status” : “Current” },

“outputParameter”: {“stdout” : “no”,

“final” : {“source” : “Step2”,

“tok\_list” : [] } } }

Each action node is a key in the dictionary, and the corresponding value for the key contains the necessary data to determine if an action is ready to execute (e.g. fires). An input pin’s name is a key within the action name’s value. In the above example, “argument” is the input pin of Step1. Under “argument”, the token source and the minimum and maximum token requirements are specified. tok\_list is a list of all the tokens placed on the input pin. Aside from the input pin, action key values also include a method that connects the key name to the defined, coded function, a next\_step that specifies the following action, and an exec\_status that either allows the action to run once, multiple times, or not at all. In order for an action to execute, its tok\_list must satisfy the action input requirements (as specified initially by the model), and its exec\_status must be set to either “Current” to run once or “Continuous” to be able to run multiple times.

The ActivityBase class is also where common functions accessible to all coded activities are defined and stored. The first four functions listed under ActivityBase in Figure 2, run(), start(), step(), and stop(), are execution control functions. These control the execution of actions based on the state information contained within the actdict. The other functions, except for main(), are data control functions. They move tokens and return status messages based upon the tokens stored in actdict. Some examples are inRange() which determines whether there are sufficient tokens on an input pin for an action to fire, parseResult() which parses the tokens on an output pin to the next input pin, and wrong\_tok() which returns a list of input pins whose conditions are not satisfied. The following are lists of the major attributes and functions in ActivityBase.

|  |  |
| --- | --- |
| **Attribute Name** | **Purpose** |
| actdict | Contains action names as keys; each action name key has the following:  • criteria for an action’s input pins to be satisfied  • a tok\_list that stores input tokens  • exec\_status that shows whether an action has executed or not  • method that matches key name to a callable action |
| LOGGER | Catalogues and stores printed messages. |
| stepMode | Allows individual stepping through an activity if set to “True”. Otherwise, the activity will run to completion in one step. |

|  |  |
| --- | --- |
| **Function Name** | **Purpose** |
| run() | Runs in the background; calls step() when stepMode is False, otherwise terminates after mapping the input parameters. |
| start() | Collects input tokens and maps them to the correct input pins. |
| step() | A single step through the activity; executes one action. |
| stop() | Returns dictionary of action name and tok\_lists if “stdout” in actdict is set to “yes”. Otherwise, returns nothing. |
| parseResult() | Maps tokens on output pins to input pins of the next action. |
| inRange() | Returns “True” if input pin criteria are satisfied. |
| decision() | Decision node; maps token to input pin of the true condition. |
| wrong\_tok() | Returns list of input pins that are not satisfied. |
| clear\_tok() | Clears the tok\_list (input pins) of an action. |
| tok\_count() | Counts the number of tokens in the tok\_list (input pin). |
| main() | Utility main method for invoking activity executables. |
| isDone() | Returns “True” when all executable actions have finished. |

The Activity class is specific to each model and is what will later be auto-generated. For both the Activity and ActivityImpl classes, the “Activity” part of the name is replaced by the name of the model. The actdict is actually stored in the Activity class, but will be referenced by the ActivityBase class. Each Activity class contains a unique “result\_map\_dict” dictionary. The result\_map\_dict for ProcedureExample is shown below.

self.result\_map\_dict = {"inputParameter" : ["Step1.argument"],

"Step1.result" : ["Step2.argument"],

"Step2.result" : ["outputParameter.final"] }

The output pin (result) of Step1 is mapped to the input pin (argument) of Step2. The parseResult() function in ActivityBase uses this dictionary to map tokens from the output pins of an activity node to the input pins of the next activity node. Other than the result\_map\_dict, the Activity class is also where the ActivityImpl class is called during action execution.

The ActivityImpl class stands for activity implementation. Although a template of this class will be automatically generated it is intended for the user to tailor with implementation codes. The ActivityImpl class contains the user-coded (manually coded) functions that each action node executes. For example, if an integer token needed to be multiplied by two, then the ActivityImpl class is where the user would type out the multiplication code in Python. Once the function has completed, then the ActivityImpl class will return a “resdict” dictionary to the Activity class that maps the resulting tokens to output pins.

In the Python design pattern, tokens are treated as objects. step() loops through the actdict and executes whichever action has its input and exec\_status requirements satisfied. Once the action function has finished, then the implementation (ActivityImpl) class will return a resdict which is used to set tokens on the correct output pins. The tokens on the completed action node’s input pins are then cleared and the tokens on the output pins are passed to the next action’s inputs by parseResult() accessing the result\_map\_dict. Then the process repeats. Since dictionaries are not ordered, then if multiple actions can execute, any one of those actions will go first. A table listing the subset of activity model features implemented in our design pattern examples and their representations in Python is included below.

|  |  |
| --- | --- |
| **Feature of Model** | **Representation in Python** |
| Action node | Instance and key in actdict |
| Input pins | Definition key under action node entry in actdict |
| Output pins | Key in resdict |
| Tokens | Treated as objects |
| Mapping of tokens from action node to action node | result\_map\_dict |
| Decision node | Equivalent to action node |
| Merge node | Multiple output pins mapped to single input pin in result\_map\_dict |
| Fork node | Token mapped to multiple inputs in the result\_map\_dict |
| Join node | Equivalent to action node |
| Input/Output parameter | Key in actdict |
| Call actions | Instanced by top-level activity model |

The most prominent problem for generating the design pattern was keeping the code general. It is simple to hardwire a program to function for a specific activity diagram, but designing programming patterns that are flexible enough to accommodate various activity diagrams is more challenging. The approach used was to develop many example diagrams that include comprehensive subsets of SysML activity model features. Code snippets must be easy to implement in a template where variables are replaced with specific names so that the specific application code can be stamped out. By implementing the specialized activity model executable ActivityBase, Activity, and ActivityImpl classes, the design pattern is well suited for future automatic code generation. The pattern should be useful within various domains for all sorts of different scenarios.

**Example Activity Diagrams:**

Initially, there were four simple SysML activity diagrams drawn with the MagicDraw tool to be coded in Python. These examples illustrated the basic features and functionality of the SysML activity examples. It was later realized that several features, such as join and merge nodes, were not demonstrated in these examples, so more activity examples were added. Presently, seven example activity diagrams have been coded that form the basis for our executable activity model Python design pattern.

The simpler diagrams include ProcedureExample, ProcessExample, and Capability Example. ProcedureExample is shown in Figure 1, and was used to demonstrate that activity diagrams can be made executable in Python. ProcessExample contained an action node that had multiple input pins and addressed the issue of having multiple input/output pins on a single node. Multiple input pins resulted in multiple input pin keys under the action node key in actdict. CapabilityExample included a fork node. The solution was simply to change the result\_map\_dict so that the tokens on one output pin would be parsed to two input pins.

The higher-level diagrams are JoinExample, IncrementExample, MergeExample, and Process1. JoinExample is an extension of CapabilityExample in that it has both a fork and a join node. The join node is actually represented as another action node in the Activity class, so it has its own actdict key and definition. Once joinNode’s tok\_list is filled, the node appends all the tokens together and passes it on to the next node. IncrementExample has a decision node whose conditions are either less than or equal to ten or greater than ten. Decision nodes are also treated as action nodes, but their method is defined in ActivityBase. The decision node’s conditions are user-defined in the Impl class, and depending on which condition is true, the tokens will get passed on to different nodes. MergeExample has both a decision and merge node. The merge node is represented by having two output pins map to the same input pin in result\_map\_dict. Finally, there’s Process1. Process1 is a hierarchy structure whose action nodes are call actions. The called actions are treated like the functions in the Impl class and are instantiated by the top-level Activity class. The called action’s Impl class is called the same way. This recursive setup allows action calls to be nested.

An eighth activity diagram has also been coded and is able to animate in MagicDraw. The diagram was provided by Ops Revitalization team and is often used by the them on missions. No details were provided, so arbitrary methods were used for the action node functions. This demonstrates that the design pattern is capable of simulating diagrams that are actually being used by system engineers.

**Summary:**

Over the course of ten weeks, a Python design pattern was developed that executes the functionality modeled in SysML activity diagrams. The design pattern consists of an ActivityBase class that contains common functions that will be used across different models and that utilizes a central dictionary (named actdict). The actdict stores execution state information that is used to determine action execution. Model specific execution is implemented in the Activity class, and manually coded behavioral details are contained in an ActivityImpl class. Tokens are passed from action to action as execution occurs. The execution is orchestrated by the run() or (step()) methods in the ActivityBase class. This will continue until all executable actions have completed, and the user can view the final results. Execution can be run from within MagicDraw which highlights action nodes as they finish processing.

Improvements to the basic design pattern can be made, but are beyond the scope of the Space Grant project. For example, the currently implementation does not prioritize the execution of individual actions in any way. In the future, a plug-in interface will be added. This will allow implementation of custom priority execution rules within unique functions. Actions could be made to execute in the sorted order of action keys within the actdict. Execution priority could be ranked according to total number of tokens required. New stereotypes could be utilized to tailor the execution in the user’s activity model.

The basic Activity class pattern is ready for automatic code generation. The remaining goal is to integrate the developed Python design pattern into the JPL Statechart Autocoder this fall.

**Bibliography:**

Books:

1. Lutz, Mark and Ascher, David. Learning Python. Sebastopol: O’Reilly & Associates, Inc., 1999.
2. Friedenthal, Sanford et al. A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann, 2009.

Website:

1. Parkin, Tim. *Python Programming Language – Official Website*. Python Software Foundation. 30 Aug. 2010. <http://www.python.org/>.