Quantitative Evaluation of Stochastic Models Project Assignment

Filino-Damerini-Macaluso

Context:

We want to develop a monolithic Java domain model that allows defining object Directed Acyclic Graphs (DAGs) through the Composite design pattern. It is important in this context to develop everything by implementing the Reflection architectural pattern, which allows defining both a knowledge and an operational level of the mentioned DAGs.

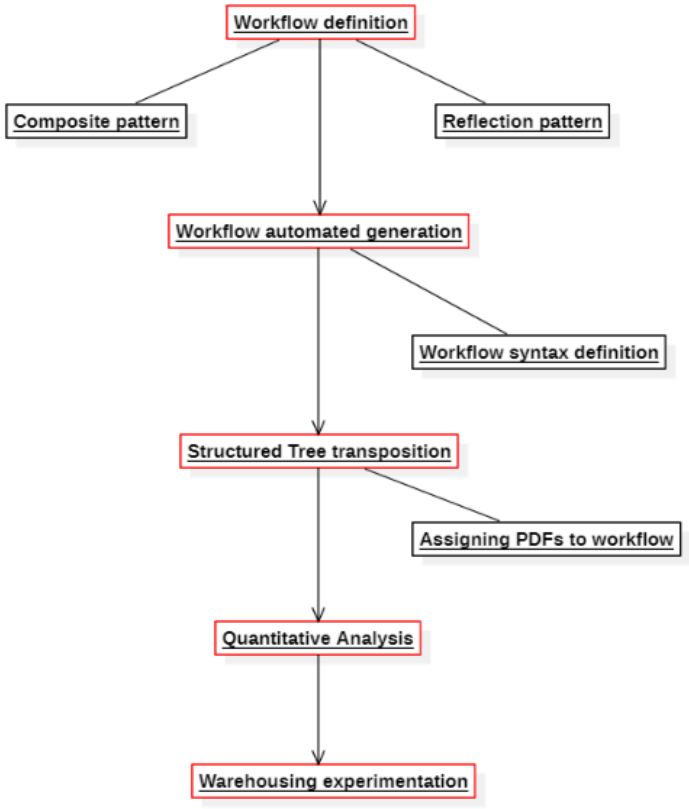
Subsequently, an automatic DAG generation system is desired at the knowledge level, providing the degree of parallelism and the maximum depth of the DAG as parameters.

Another objective is represented by linking the generated DAGs to EULERO/ROSPO , a Java library implementing a compositional approach to safe approximation of the end-to-end response time Probability Density Function (PDF) of complex workflows, made of activities with general (i.e., non-Exponential) duration possibly with bounded support, composed through sequence, choice/merge, and split/join blocks, an unbalanced split and join constructs that may break the structure of well-formed nesting.

Roadmap:

- **Workflow Definition**: The first step is to define the classes that represent DAGs using the Composite Pattern to define Knowledge Level of the Reflection. After that it is necessary to define Operational Level, allowing an operational workflow allocation.

- **Workflow automated generation**: Another important objective is represented by the definition of a set of functions that allows the automated generation of workflows using hyper-parameters, such as:
    - Max parallelism degree.
    - Max/min depth.
    - Max/Min width.
    - …



- **Structured Tree Transposition:** After creating the module for the automated generation of workflows, it becomes crucial at this point to be able to associate a PDF with each node to characterize its execution/production time.
  This is then extended by translating the resulting workflows into Structured Tree form to be used in the next step for quantitative evaluation.

- **Quantitative Analysis:** Once the analysis functions have been identified, it is necessary to define functions that enable the analysis through Eulero, providing Directed Acyclic Graphs (DAGs) among those generated and translated into structured trees with associated PDFs.
  Subsequently, carry out a system test suite with the generation of non-trivial DAG and analyze the completion time in various cases.

- **Warehousing experimentation:**
  The goal is to enhance the model by incorporating a Warehouse module to track available Products in both current and future storage. The immediate availability of products would replace the usual processing times, while future storage would track ongoing productions that will soon become available. This enhancement allows for experimentation with various scenarios, including immediate availability, just-in-time availability, and faster-than-processing availability.

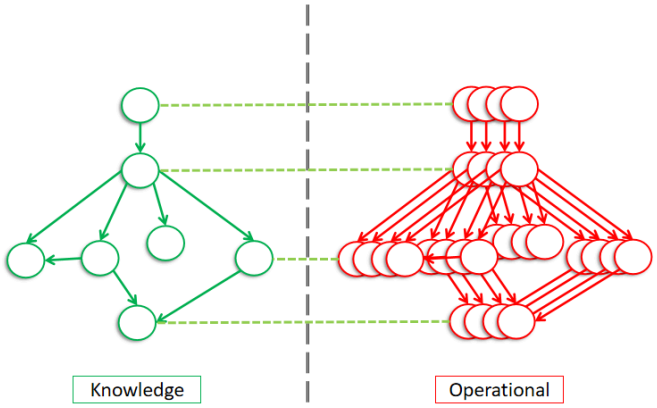
Important Concept and Design And Architectural Patterns:

Composite - Design Pattern:

- https://www.baeldung.com/java-composite-pattern
- https://dellabate.wordpress.com/2011/02/21/gof-patterns-composite/

Reflection - Architectural Pattern:

The Reflection architectural pattern is then introduced to express the concept of meta-model / ontological-model as the pattern enables a system to dynamically examine its own structure and behaviors at runtime, providing flexibility and extensibility. Reflection architectural pattern defines knowledge and operational levels: a template on the knowledge level can be instanced multiple times on the Operational level. This creates a structural constraint between the abstract and concrete objects, with the latter forced to follow the first's characteristics.



For a better understanding of Reflection Architecture Pattern:

- **Pattern-Oriented Software Architecture (pag 193 - 212 on pdf):** https://github.com/ppizarro/coursera/blob/master/POSA/Books/Pattern-Oriented%20Software%20Architecture/Pattern-Oriented%20Software%20Architecture%2C%20Volume%201%20-%20A%20System%20Of%20Patterns.pdf

Eulero Library:

https://onesearch.unifi.it/primo-explore/openurl?sid=google&auinit=L&aulast=Carnevali&atitle=Eulero:%20A%20Tool%20for%20Quantitative%20Modeling%20and%20Evaluation%20of%20Complex%20Workflows&id=doi:10.1007%2F978-3-031-16336-4_13&vid=39UFI_V1&institution=39UFI&url_ctx_val=&url_ctx_fmt=null&isSerivcesPage=true

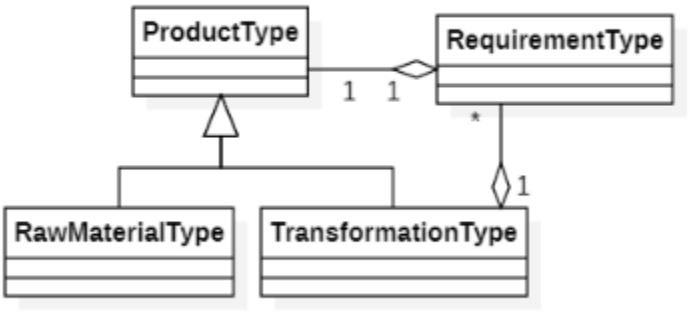
Github project: https://github.com/oris-tool/eulero

ATTENTION! Note that the Eulero Library is currently at version 2.0, which is however not publicly available. Thus, the provided paper link and github project should be used to understand how to model

and analyze a structure tree. Subsequently, version 2.0 will be presented in a live session.

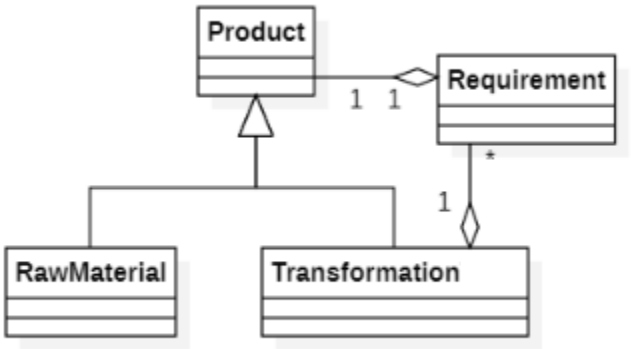
Workflow Definition:

To define a workflow, it is necessary to create a composite structure that can handle all node types of the DAG:

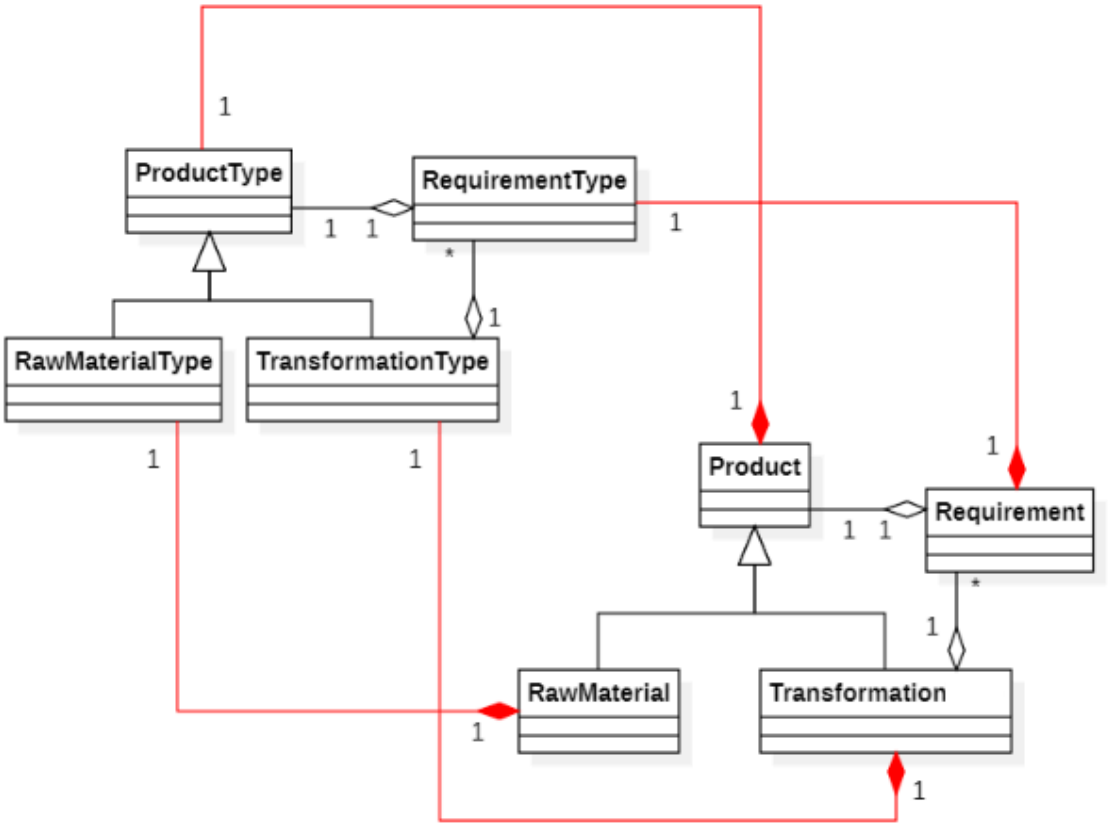


All the possible DAGs of this composition represent the Knowledge Level of the Reflection architectural pattern, and they stand as the “templates” from which real, operative DAGs are created.

The operational level composition can be represented in a very similar way:

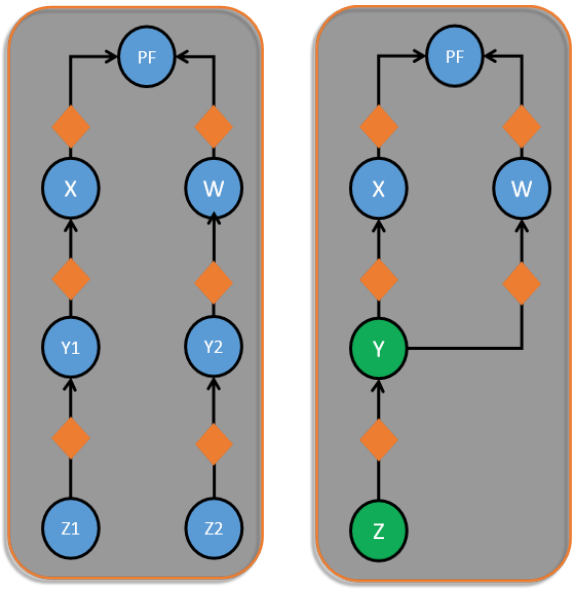


These two compositions are logically identical, but their semantic and use is completely different: a Knowledge DAG represents a “Type”, and is associated with many Operative DAGs (called Concrete DAGs) in a one -to-many fashion, as shown below.



Note: The composite pattern designed in this way can create two kinds of DAGs: Unshared, which represent workflows considering separate, parallel productions whenever a new branch is created; and Shared, which represent workflows with joined productions (of the same type), regardless of the branch or the level where the production takes place. Each has pros and cons, and completely changes the behavior of the Operative level and their DAG counterparts.

Again, it is possible to notice that Shared DAGs are a generalization of Unshared DAGs: the latters can be extracted from the former.



Unshared DAG Shared DAG

Automated Workflow Generation :

The next step, once the workflow has been defined and developed, is to find a way to automate their generation, in order to provide the next steps with enough different samples of workflows to properly experiment on.

The automated generation should **NOT** be completely random, but should pivot around a few hyper-parameters, such as DAG height, width, branching (the number of new branches leaving a node) etc.

Of great importance is the concept of workflow "parallelism", which represents how much of the DAG processes can be run in parallel.

An objective for the assignment is also to find a proper measure of the degree of parallelism in a DAG, and integrate it as a hyperparameter to generate workflows.

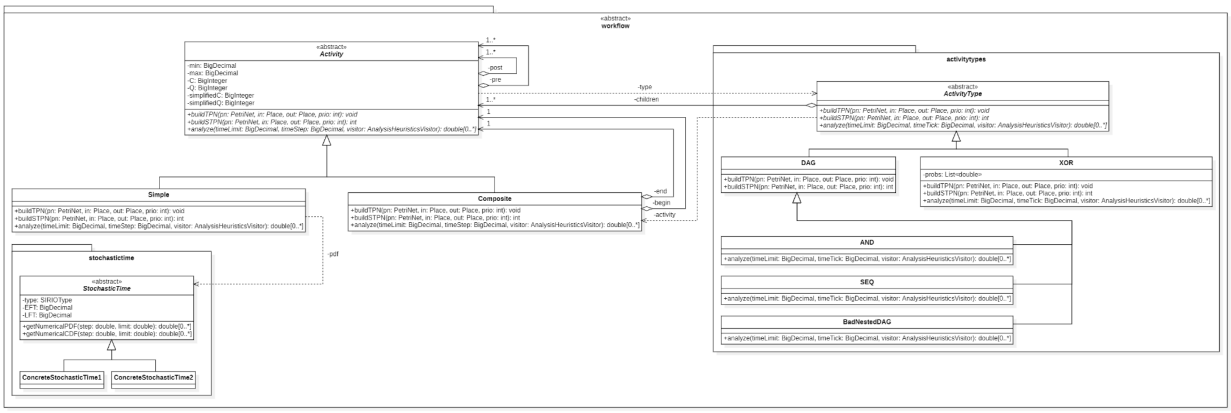
Structured Tree Transposition :

To enable quantitative analysis of the workflow's end-to-end processing times, the workflows as represented in the domain model must be characterized with PDFs associated with each production, and then translate the resulting workflow into Structured Tree form.

This will be fundamental to perform the analysis through the Java library Eulero/ROSPO to conduct the analysis of the completion time of given workflows. It is therefore essential to study the Eulero library to evaluate:

- The PDFs that are usable within the library, without needing to redefine them.
- The composition of structured Trees, a formalism in which workflows need to be translated for subsequent analysis.
- Identifying the functions that enable the analysis of the generated structured trees.

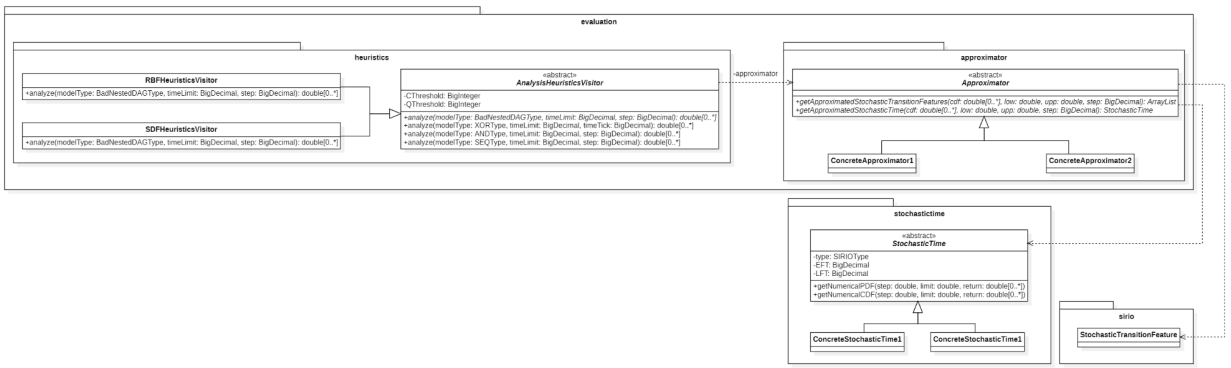
Here, we provide the UML class Diagram of the Eulero 2.0 Library for the package responsible for the modelling of a workflow. As mentioned earlier, the UML diagram will be discussed in a live session, after fundamentals and preliminaries of the library will be explored in the provided paper.



Quantitative Analysis :

The structure tree formalism is designed to be explored through the Visitor Pattern. In particular we provide two Visitor implementations that perform the quantitative analysis of the instanced model.

Here, we provide the UML class Diagram of the evaluation package of the Eulero 2.0 library:



As the analysis performed by Eulero is an approximate analysis, the Visitor leverage on an approximator package that include the logic and classes that perform approximations.

As mentioned above, details will be discussed in a live session.

Once the analysis functions have been identified, it is necessary to define functions that enable the analysis through Eulero, providing Directed Acyclic Graphs (DAGs) among those generated and translated into structured trees with associated PDFs.
Subsequently, carry out an experimentation with the generation of non-trivial DAG and analysis of the completion time in various cases.

(Optional) Warehousing experimentation :

Furthermore, it would be a great achievement to experiment on the resulting model by adding an additional functionality, regarding the use of a Warehouse module, which should enable tracking of available Products, both in terms of current **and** future storage.

The **current storage** of products would be immediately available to be used at the Operative level of the domain model, by replacing the usual processing times' distribution with an immediate availability.

The **future storage** would instead track productions that are currently running, and that will provide extra products to be stored in the warehouse (essentially, a soon-to-be current storage).

Experimentation should cover cases where products are immediately available, are available **just-in-time** (when the process that actually needs the availability is ready to go), or available **faster-than-processing** (the products are going to be available after the process should have begun, but are faster than the normal processing).

Mandatory Testing :

The developed Java domain model should be thoroughly tested through the use of testing libraries or suites. You can use whatever is best for you, but some of the most useful are:

- Junit
- Mockito
- AssertJ
- any equivalent …

For monolithic applications, it is highly recommended to separate tests from the actual domain model code, and use a Unit/Integration testing approach.

For any questions :

Leonardo Paroli, Giovanni Fontani

- leonardo.paroli@unifi.it
- giovanni.fontan1@unifi.it
- Usually at STLab, Santa Marta, on Monday, Tuesday, Friday (send us a notification to check if we're available)

Riccardo Reali

- riccardo.reali@unifi.it
- Usually at STLab, Santa Marta, on Tuesday and Thursday (send a notification to check availability)