VIP ANR-09-COSI-03
Virtual Imaging Platform

M1.2.4: Workflow designer architecture.

Nadia Cerezo MODALIS (I3S) cerezo@i3s.unice.fr
Johan Montagnat MODALIS (I3S) johan@i3s.unice.fr

Abstract

To assist VIP end-users, non specialized in high-performance distributed computing systems, in creating new simulation workflows for exploiting the platform, a dedicated workflow designer will be implemented. In the project proposal, a notion of workflow template was considered to provide an accessible yet extensible framework for new simulation workflow instances. However, the detailed analysis of the VIP use cases revealed the real complexity of this workflow design phase. The concept of workflow template was extended in Conceptual Workflows. The VIP workflow designer will be developed with the objective of editing Conceptual Workflows and translate them into executable artifacts that can be handled by the MOTEUR workflow engine.

Contents

1 Introduction

The SHIWA workflow designer aims at simplifying the workflow design process for end-users who are expert in their application domain, but not in the distributed computing environment exploited for achieving high performance in the SHIWA platform. The objectives of the designer are actually twofolds:

- to facilitate the design of the 4 VIP simulator workflows within the platform; and
- to facilitate the integration of new simulators in the future.

The problem of complexity of workflow design for distributed computing infrastructures is well known from the Scientific Workflows community [5]. There is a gap between the low level description of computing processes usually available when targeting high-performance distributed infrastructures and the high-level view of domain experts. Workflows architected are cluttered by numerous technical non-functional concerns that increase their design complexity and severely impair their readability. In the VIP project proposal, the notion of workflow templates was considered to provide end-users with pre-formatted workflows which could be instantiated for a particular simulator with minimal changes (see VIP Milestone 1.2.3 [2]). However, the study of VIP use-cases revealed the complexity of the object preparation and simulation workflows that lead to substantially different workflows when operationalized.

Consequently, a study was conducted to broaden the concept of workflow templates and address the problem of scientific workflows design more globally, through the notion of **Conceptual Workflow** as defined in section 2. Section 3 then shows how Conceptual Workflows can be applied to the VIP use cases and how the VIP workflow designer will assist VIP end-users in editing workflows and creating new use cases.

2 Conceptual Workflows

In terms of Scientific Workflows (SWs) abstraction levels, the distinction has long been made between **Abstract** and **Concrete** [6], on the Infrastructure side of Scientific Workflow Frameworks (SWFs). **Concrete Workflows** are meant for direct execution by an Enactor over a Distributed Computing Infrastructure (DCI) and, as such, they bind each activity to a specific resource. The extreme rigidity of such models make them impractical to store SWs, which is why most SWFs also handle **Abstract Workflows**, whose activities are not necessarily bound to any resource. As a rule, the conversion from Abstract to Concrete is done automatically at runtime, using the most recent available information about the infrastructure. We make a similar distinction on the User side of SWFs (see figure 1).

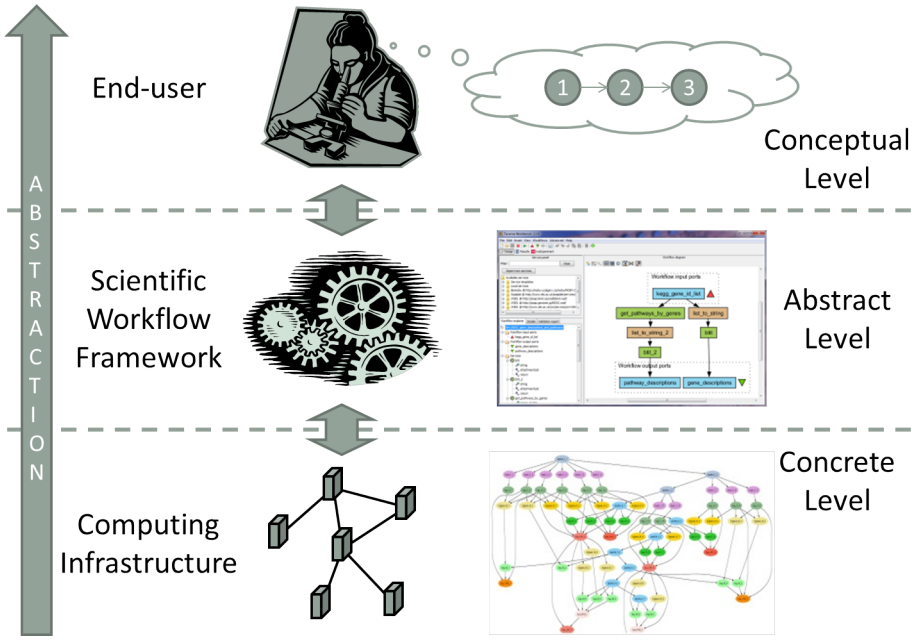

Figure 1: Scientific Workflow Abstraction Levels

When scientists design their in-silico experiments, they start with more or less precise ideas about goals (i.e. what the simulation should do) and methods (i.e. how to achieve the goals). To go from that informal concept of SW to an actual Abstract Workflow implies not only determining precise goals and methods, (both pertaining to the end-user's **domain**), but also catering to various concerns (roughly divided between **technical** and **Quality of Service (QoS)**) which soon clutter the SW to the point where the original simulation is barely recognizable.

Sharing the simulations themselves is useful for peer validation as well as collaborations. But technical and QoS know-how and workarounds might also be valuable assets to a community. Unfortunately, when all three types of concerns (i.e. **domain**, **technical** and **QoS**) are tangled indiscriminately, the way they are in Abstract and Concrete SWs, sharing and reuse remain almost as tedious as with simple scripts. The legibility of SWs is itself hindered by the mix of types of concerns and users generally rely on external and informal descriptions (e.g. keywords, descriptions in natural language, informal diagrams) to find and decrypt SWs.

To the best of our knowledge, there is not yet a formal way to describe simulations at the high level of abstraction of the end users' scientific domain. Note that portals are indeed at that level of abstraction, but they are inherently specialized and opaque (i.e. they hide the underlying SW), which makes them impractical for sharing and reuse.

We thus propose the formal definition of a third level of abstraction, closer to end-users and their domains. A **Conceptual Workflow** might be purely high-level, pertaining only to a given scientific domain, and thus exist only as a reference protocol, or be tied to all due technical and QoS concerns, but in a way that the distinction between those things remains clear, in order to ease sharing and reuse.

2.1 High-level elements

Definitions. A **Conceptual Workflow** is a Directed Graph (DG):

- whose vertices are either

  Conceptual Inputs

  Conceptual Outputs

  or **Conceptual Workflows** themselves

- and whose edges are **Conceptual Links**.

Conceptual Workflows are nested so as to provide encapsulation: parts of the workflow that are deemed too detailed or that are reused can be kept inside a sub-workflow and thus at a lower level of abstraction from the viewpoint of the parent workflow.

Each **Conceptual Workflow** can be associated to any number of **Functions** which are domain concepts such as *AlignImage* or *GeneratePhotons*.

Conceptual Inputs represent the starting materials of the in-silico experiment, whereas **Conceptual Outputs** represent its results. Both can be associated to any number of **Datasets** which are domain concepts such as *BrainModel* or *SyntheticSonogram*.

A **Conceptual Link** is a purposely loose concept, so as to capture many types of interactions that are possible between **Conceptual Workflows**.

Figure 2 is a representation in UML of the classes we just described and their relationships.

Additionnal Remarks. Because in-silico experiments are inherently very data-driven, **conceptual links** will most often translate to a set of data transfers and seldom to anything else (e.g. precedence ordering). Even when it translates purely to data transfers, a **Conceptual Link** should not be confused with a **Data Link**, since it restrains neither number nor types of data transferred.

Let H be the set of **Conceptual Workflows** and R be the binary relation of **Conceptual Link**, the relation R is transitive: $\forall x, y, z \in H, xRy \wedge yRz \Rightarrow xRz$

Graphical convention.

- **Conceptual Inputs** (resp. **Conceptual Outputs**) are represented by trapezoids with the smaller edge below (resp. above).
- Children **Conceptual Workflows** are represented by rectangles.
- **Conceptual Links** are represented by dotted arrows.

Figure 3 sums up those conventions.

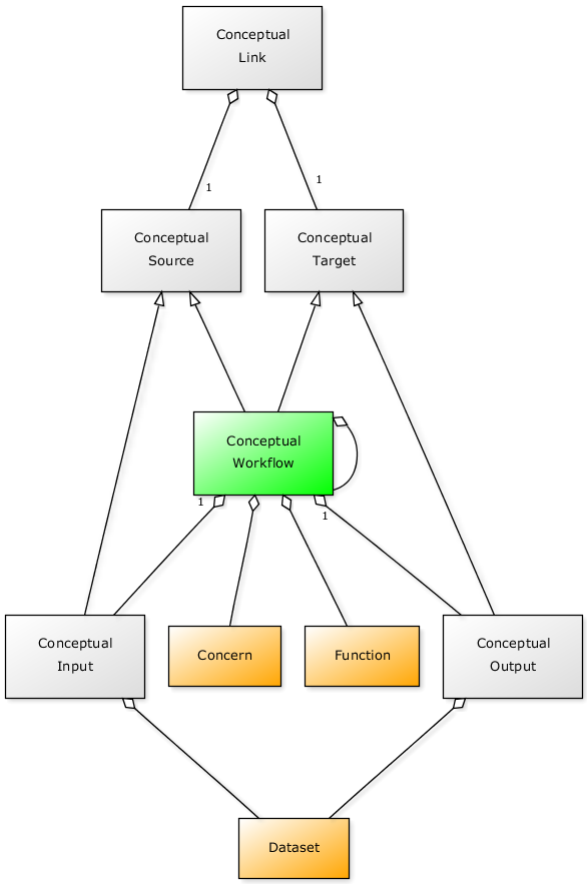

Figure 2: Meta-model - High-level elements
Diagram generated with yuml.me
Default cardinality is * (many)

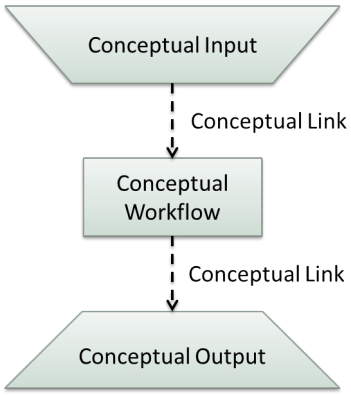

Figure 3: Graphical Convention - High-level elements

2.2 Lower-level elements

The ultimate purpose of a **Conceptual Workflow** is either to be converted to an abstract workflow or to ease the sharing and reuse of an abstract workflow by increasing its legibility (potentially both).

It is thus essential to describe lower-level elements, that pertain to abstract workflows, if not in a fully detailed and readily executable way, at least in a precise enough manner that the link between conceptual and abstract levels remains clear.

Traditional elements. As of now, we have found necessary to consider the following elements that are commonly found in abstract workflows:

- **Activities**: processing units such as web services, grid jobs, tasks, scripts or sub-processes.
- **Input** (resp. **Output**) **Ports**: arguments (resp. products) of activities.
- **Inputs** (resp. **Outputs**): arguments (resp. products) of the workflow itself.
- **Data Links**: data transfers between activities.
- **Order Links**: precedence control links that ensure the target activity doesn't start until the source activity is finished.

Figure 4 is a representation in UML of the classes we just described and their relationships.

Additional Remarks. Occasionally, data transfers between activities are done implicitly. If an activity A produces data at a specific location where the activity B retrieves it, or if the SWF takes care of the transfer transparently (so that each activity will process local data), it is an implicit data transfer in the sense that it won't appear on the abstract workflow.

For such a transfer to take place successfully, knowledge of it must be present at some level. For instance, maybe the user has to know that all activities must be fed the same path as input for the entire process to work, or the service descriptors contain information about the implicit data that can be interpreted by the enactor transparently for the user (as is the case with jGASW descriptors).

In both cases, the knowledge is required but not explicit on the workflow itself and thus it hinders both reading and sharing it.

To alleviate this problem, we introduce the concept of implicit port that denotes the production or consumption of implicit data by an activity.

Graphical convention.

- **Activities** are represented by rounded rectangles,
- **Ports** by small rectangles,
- **Implicit Ports** by small rhombuses,
- **Inputs** (resp. **Outputs**) by trapezoids with the smaller edge below (resp. above),
- **Data Links** by arrows,
- and **Order Links** by arrows with a circle at the end.

Figure 5 sums up those conventions.

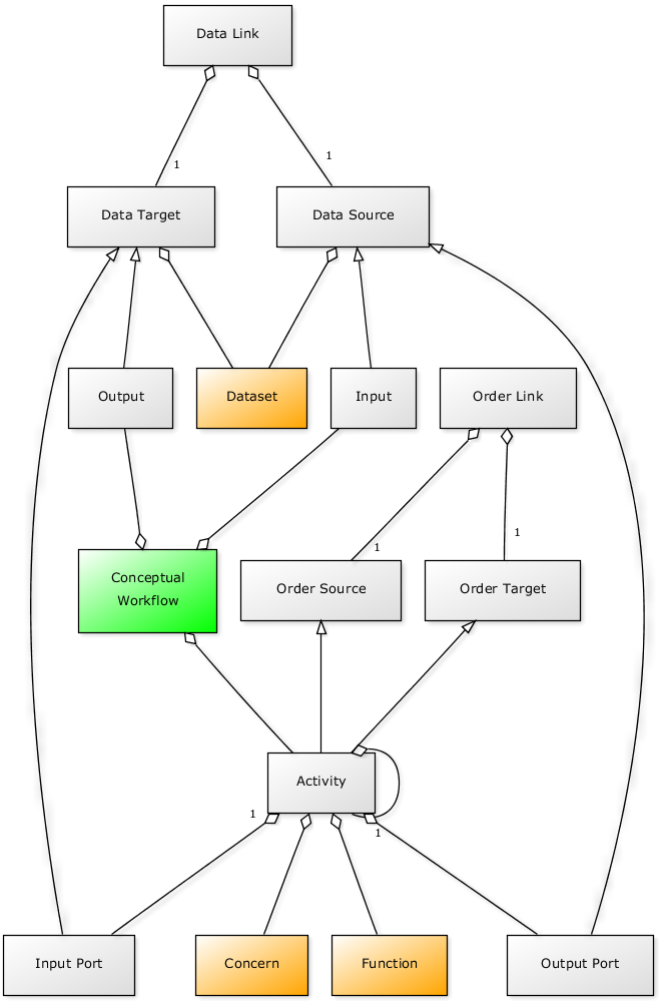

Figure 4: Meta-model - Lower-level elements
Diagram generated with yuml.me
Default cardinality is * (many)

2.3 Patterns

Definitions. Encapsulation on its own (through nesting of Conceptual Workflows and Activities) does not provide enough flexibility to untangle concerns and ease sharing and reuse.

Taking inspiration from Aspect-Oriented Programming (AOP), we propose the notion of **Patterns**: reusable fragments that can be woven into a base **Conceptual Workflow**. They are themselves **Conceptual Workflows**, special in that they feature **Join Points**.

Join Points are placeholders in a **Pattern** that must be replaced with elements of the base process when the **Pattern** is plugged in.

Pattern application. A **Pattern** is applied to a **Conceptual Workflow**. The application process depends on the abstraction level and structure of the given **Pattern**: each **In Join Point** must be replaced by a **Source** in the base process, whereas **Out Join Points** must be replaced by **Targets**.

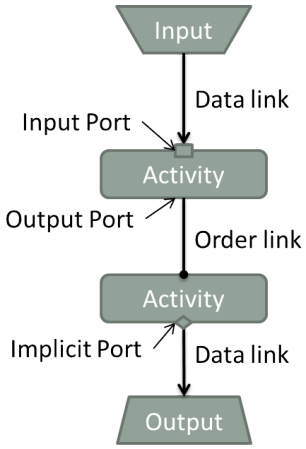

Figure 5: Graphical Convention - Lower-level elements

The compatibility between link types and source/target types must be preserved. For instance, if a given **In Join Point** is connected to the rest of the **Pattern** by a **Data Link**, then it can only be replaced by a **Data Source**; **Conceptual Sources** and **Order Sources** will not do.

Table 1: Compatibility between Links and Sources/Targets

Abstraction Level	**Link type**	**Source type**	**Target type**
High-level	Conceptual Link	Conceptual Source	Conceptual Target
Lower-level	Data Link	Data Source	Data Target
	Order Link	Order Source	Order Target

Table 1 features all compatibility cases and Figure 6 is a representation in UML of the classes we just described and their relationships.

Graphical convention.

- **Patterns** are represented by hexagons,
- pattern application by a dotted line,
- and **Join Points** by crossed circles.

Figure 7 sums up those conventions and emphasizes the difference between high-level and lower-level patterns.

3 VIP workflow designer

Conceptual Workflows will be of little use if there is no way to translate them into regular abstract workflows that can be delegated to existing SWFs. While the conversion *Abstract* → *Concrete* takes place between a SWF and a DCI and thus has to be entirely automated, the conversion *Conceptual* → *Abstract* has to rely on much user knowledge and decision-making and thus can at best be computer-aided.

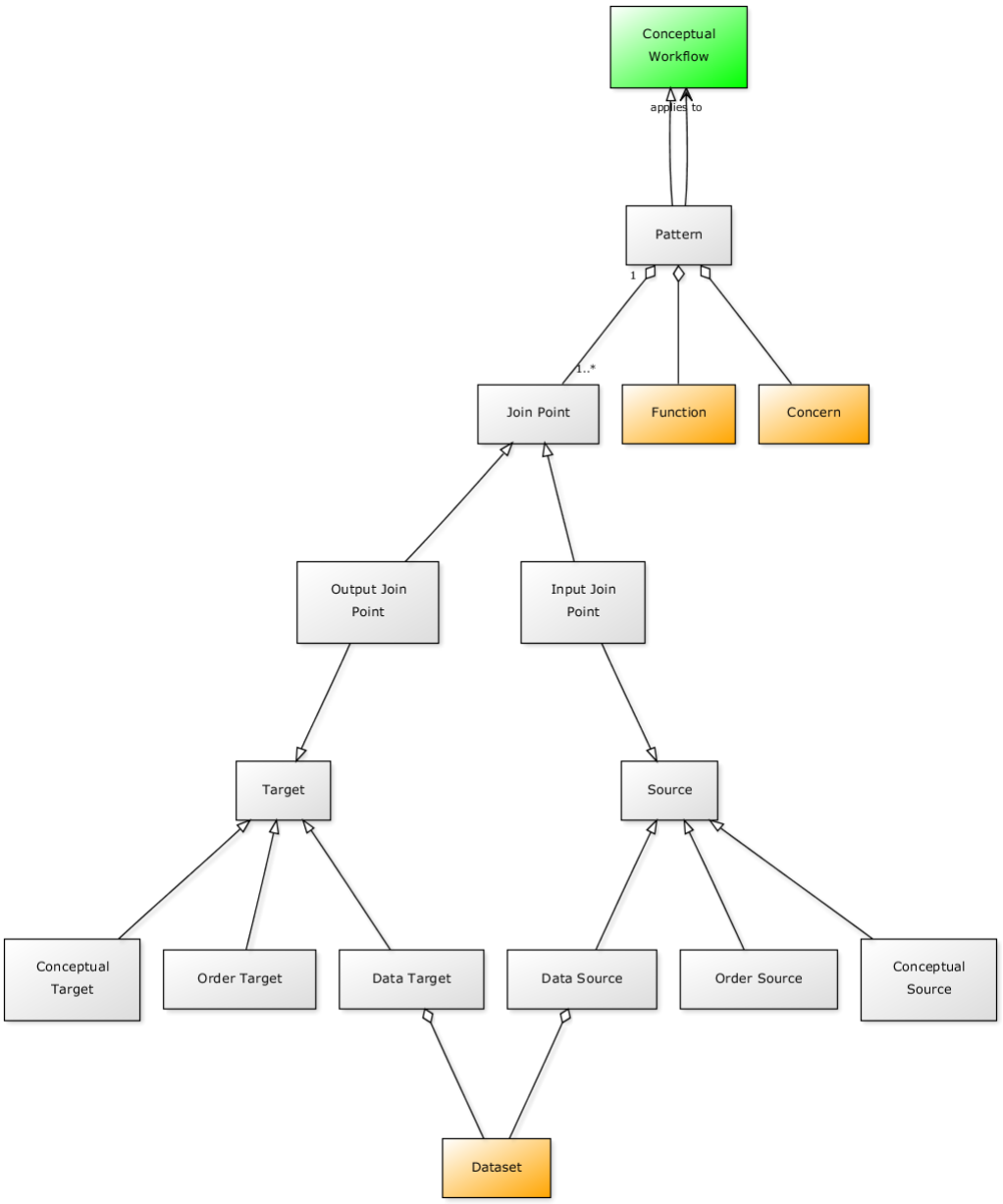

Figure 6: Meta-model - Patterns
Diagram generated with yuml.me
Default cardinality is * (many)

The VIP workflow designer will integrate a Conceptual Workflow editor to manipulate such workflows and a user-driven converter to produce GWENDIA Abstract Workflows that can be enacted through the MOTEUR workflow engine. Section 3.1 describes the conversion process and section 3.2 illustrates how this process will be applied to the 4 VIP simulator workflows.

3.1 Conversion Steps

As of now, we contemplate a conversion process in 3 steps (see figure 8):

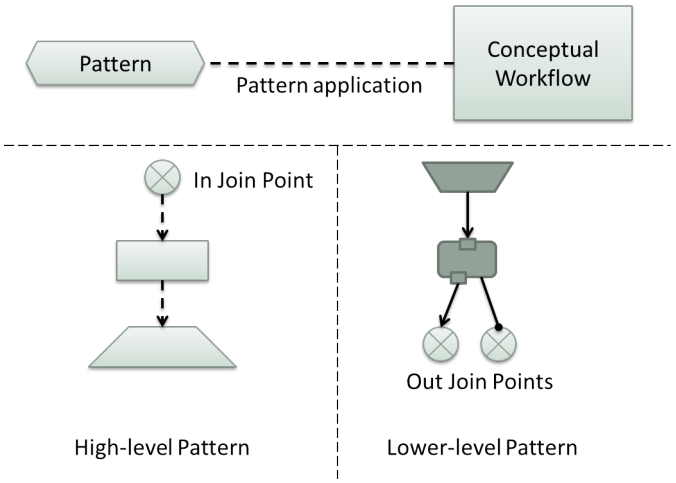

Figure 7: Graphical Convention - Patterns

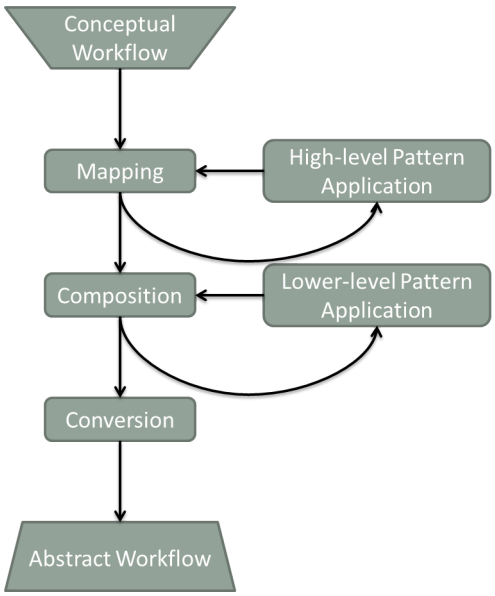

Figure 8: From Conceptual to Abstract SW - Conversion Steps

1. **Mapping** from **Conceptual Workflows** to **Activities** that fulfill the **Functions** they're associated to, if any. Every high-level **Pattern** is applied and another phase of **Mapping** is done after each **Pattern** application.
2. **Composition** of the **Activities** mapped at the previous step, until every **Port** is attached. Every lower-level **Pattern** is applied and another phase of **Composition** is done after each **Pattern** application.
3. **Conversion** of the result of the previous step into an abstract workflow that can be delegated to an existing SWF.

When mapping Conceptual Workflows to Activities, there are three cases:

- **1 to 1 mapping**, when there is an **Activity** that fulfills the associated **Functions**

exactly. Those matches should be easy to detect automatically and suggest to the user.

- **1 to n mapping**, when the associated **Functions** are not fulfilled by a single **Activity**, but by many composed in a sub-workflow. Assisting the user through 1 to n mapping will likely prove much more complex than in the 1 to 1 case. Obviously, if entire workflows are treated as meta-activities and semantically annotated like regular activities, then the system can do the same in both cases of mapping, but there currently is not a wealth of such annotated workflows. Another possibility would be to suggest partial matches (i.e. **Activities** that fulfill part of the associated **Functions**, according to their annotations) and maybe further assist the user in composing them by limiting the suggestions to **Activities** that are compatible with those the user has already chosen.

- **n to 1 mapping**, when a given activity encompasses **Functions** associated to more than one **Conceptual Workflow**. This case poses two main problems: (1) how to merge **Conceptual Workflows**, yet preserve legibility and flexibility? (2) what depth of merging should the system consider when looking for matches?

3.2 Use cases

The simulation cores of all four simulators integrated from start in the platform (i.e. SIMRI, FIELD-II, Sindbad and Sorteo) conform to the same generic template:

1. An object model (e.g. a heart model, a brain model) and simulation parameters (e.g. scanner specifications) are given to the simulator as inputs.

2. The simulator runs in a highly parallelized fashion, to leverage grid resources and achieve acceptable performance.

3. A synthetic medical image is produced as output.

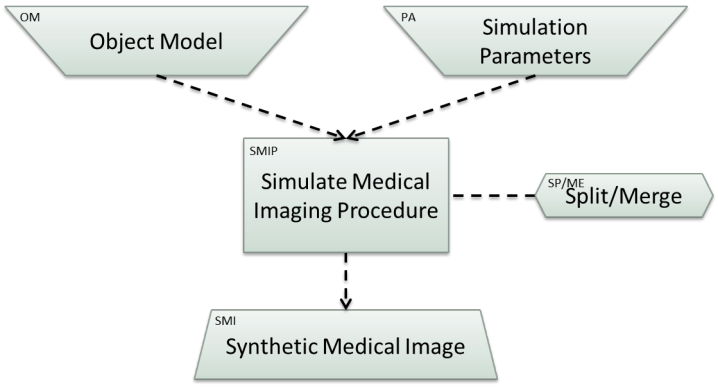

Figure 9: VIP simulation core template

Figure 9 shows a graphical representation of that template as a Conceptual Workflow.

Before we delve into each simulator's specifics, we shall take a closer look at the way those simulators are parallelized.

In the case of VIP simulators, as of now, only data parallelism is used: huge volumes of data are split into independent chunks, processed separately (as simultaneously as resources allow) and merged to produce the final output. Hence the name "Split/Merge".

The Split/Merge pattern is shared (though implemented in different ways) by all four VIP use cases.

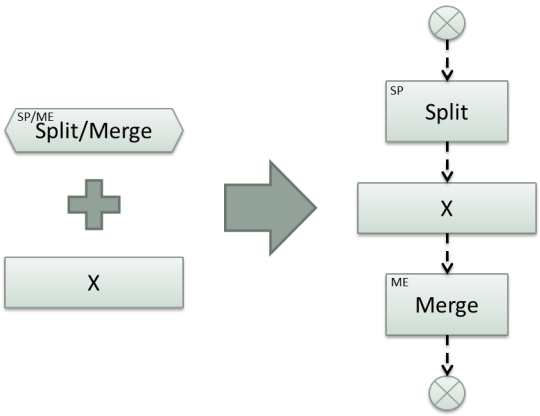

Figure 10: Split/Merge pattern

At the purely conceptual level, applying a Split/Merge pattern to a conceptual workflow implies weaving two additional steps into the process (as shown on figure 10): a pre-processing step to split input and a post-processing step to merge output. Both steps are unlikely to be generated automatically, as they rely on domain knowledge about the data as well as user preferences in terms of data splitting.

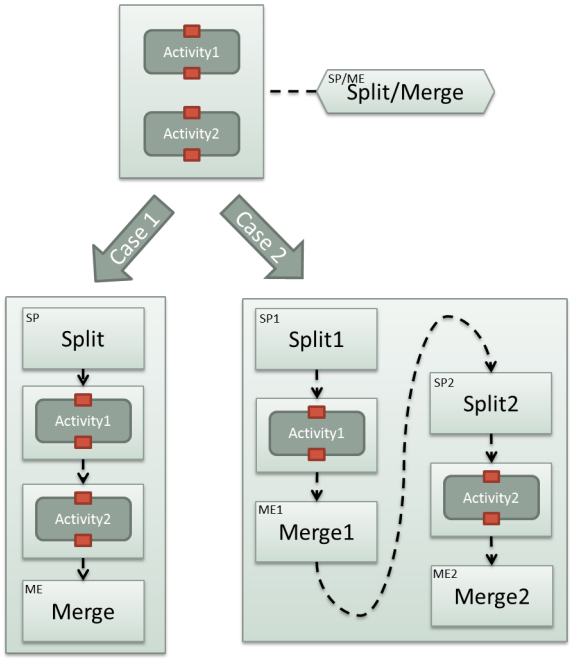

Figure 11: Split/Merge cases

When a conceptual workflow has been mapped to two or more activities, weaving becomes less straightforward. There are two main cases (shown on figure 11):

- **Case 1**: data independence is valid for the entire processing chain and thus data is split and merged only once at the very beginning and at the very end of the chain respectively.
- **Case 2**: data independence is only valid at the scale of each activity and thus data must be merged after each processing step in the chain.

Any other case is a composite of those two and suggests that the conceptual workflow should itself be split or detailed through encapsulated sub-workflows.

3.2.1 SIMRI

Mapping. Because SIMRI was meant all along for deployment on the grid, it was parallelized using MPI [1]. As a result, the main activity *simri_calcul* already implements a Split/Merge strategy, thus the pattern is already fulfilled. The Object Model is contained in a file stored on the target infrastructure at the provided url $fileObjLfn$. The results are stored and their url $outputFileLfn$ is returned. There are no inputs corresponding to simulation parameters.

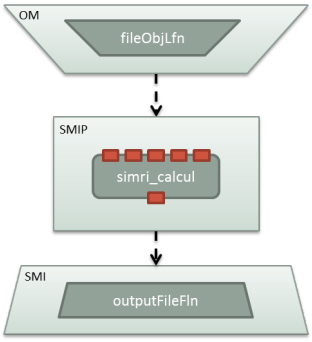

Figure 12: SIMRI Mapping

Figure 12 shows the result of the mapping phase, which, in the case of SIMRI, is fairly straightforward.

Composing. The composition between the activities mapped in the previous step is trivial and can be automated, provided the activities and inputs/outputs are correctly annotated: one of the 5 input ports of *simri_calcul* takes the url of the input file and its output port produces the desired result.

This composition step leaves 4 out of the 6 ports of *simri_calcul* unattached, as shown on figure 13.

One of the input ports of *simri_calcul* expects a filename that can be computed automatically from the url of the input file, through the script $detFileOut$ (which requires an additional input $fileoutName$). In order to properly run on the target infrastructure, *simri_calcul* also needs the name of the MPI version to use $cstMpiVersion$ as well as the url of a package of libraries and executables $simriReleaseLfn$.

This composition step leaves only one port unattached, as shown on figure 14.

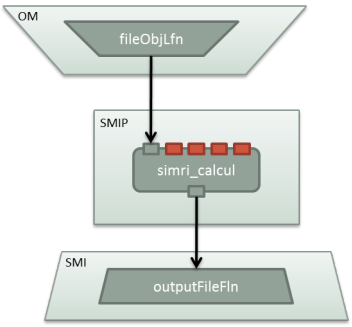

Figure 13: SIMRI Composition (step 1)

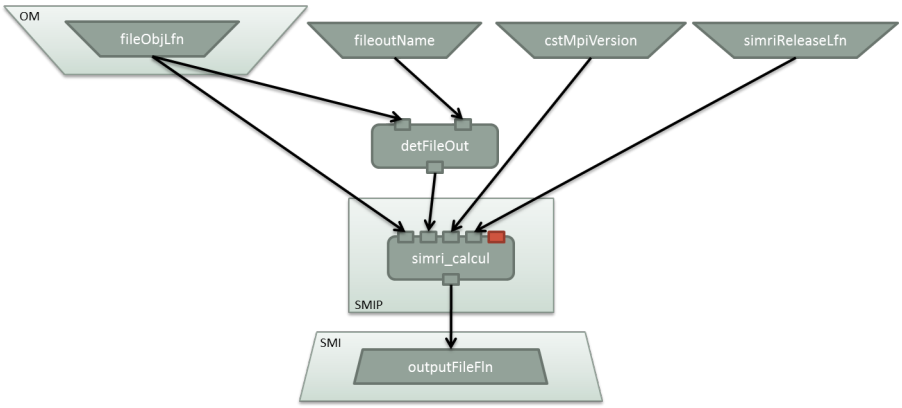

Figure 14: SIMRI Composition (step 2)

Generator Pattern. The last unattached port of *simri_calcul* expects the name of the directory where the simulation is performed. All simulators on the VIP platform follow the following naming convention: the directory name is the concatenation of a name given to the simulation itself and the exact date at which it is performed. Like Split/Merge, this directory naming sub-process is common to all four VIP use cases and, as such, should be captured inside a pattern to be reused in every use case. Let us introduce the pattern template *Generator* (shown on figure 15).

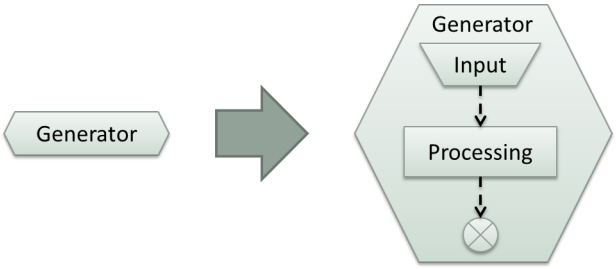

Figure 15: Generator Pattern

Any pattern that consists entirely of inputs pre-processed before they are fed to the base process is an instance of the *Generator*. The case of the VIP directory naming convention fits nicely into the *Generator* template, so much so that it has exactly one input (*SimulationName*) and one processing activity (*appendDate*) and thus mirrors the

conceptual template perfectly, as shown on figure 16.

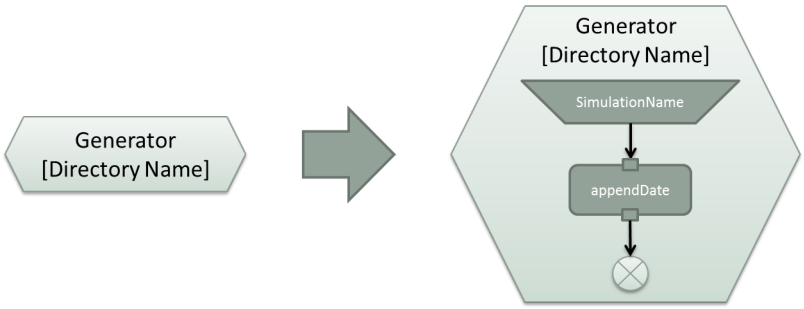

Figure 16: Generator Instance for Directory Name

Once we plug the [*DirectoryName*] instance of *Generator* we just defined into the only unattached port of *simri_calcul* left (see figure 17), we are almost done.

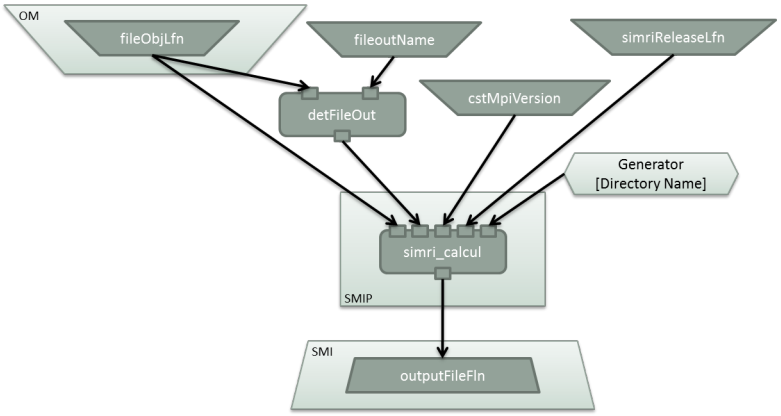

Figure 17: SIMRI Composing (final)

Conversion. The last conversion step to an actual abstract scientific workflow is, in the case of SIMRI, only a matter of format, from our system's internal model to a scientific workflow language such as GWENDIA. The final result is the GWENDIA workflow screen-captured in the MOTEUR2 interface on figure 18.

3.2.2 FIELD-II

Mapping. Data parallelism is present in FIELD-II in that different simulated radiofrequency lines are independent, but the number of lines itself is computed as part of the FIELD-II simulation process, which makes its parallelism a bit less straightforward than with the other three simulators. The main activity of the FIELD-II workflow is *SimulateRFLine*, which produces a partial sonogram, along a given radiofrequency line. The model is a matlab matrix called *TissueParameters_mat* and the ultrasound probe parameters are in a Matlab matrix called *ProbeParameters_mat*. The final output is another Matlab matrix, called *image_mat*. Figure 19 shows that preliminary mapping.

Before composing anything, we have to apply the *Split/Merge* pattern to the *SMIP* (Simulate Medical Imaging Procedure) conceptual workflow. Since it is mapped to only

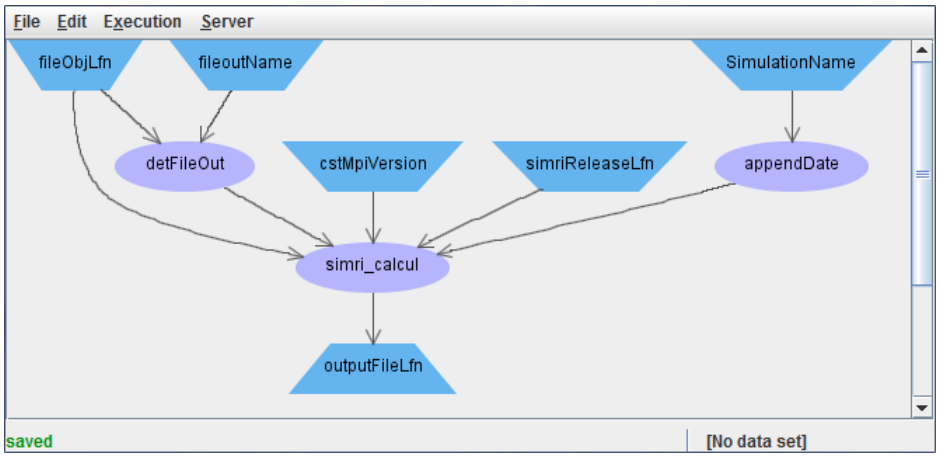

Figure 18: SIMRI Abstract Workflow (GWENDIA/MOTEUR2)

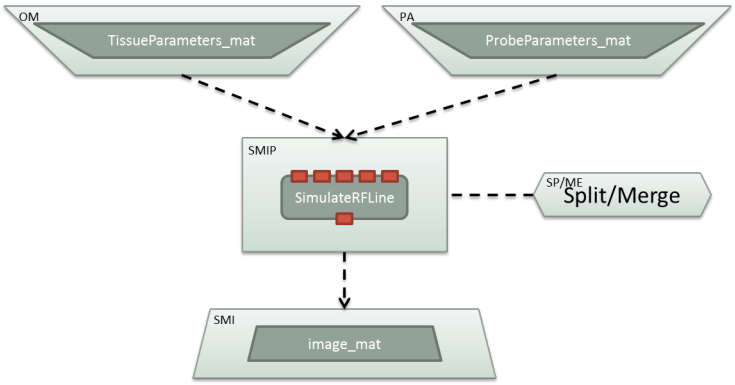

Figure 19: FIELD-II Mapping (step 1)

one activity, it is a trivial weaving, with the *Split* and *Merge* steps surrounding the activity, as shown on figure 20.

The newly inserted *Split* and *Merge* functions must be mapped in turn. To split data, the radiofrequency lines are generated by the *getRFLines* activity, which takes the number of lines *no_lines* as input. The partial sonograms are merged by the appropriately named activity *Merge*. The end result of the mapping phase is shown on figure 21.

Composing. All activities can be trivially composed, but for $SimulateRFLine \rightarrow Merge$ which is a type mismatch, shown in orange on figure 22: each call to *SimulateRFLine* produces a file, but *Merge* expects the path to the directory where all those individual files are stored. That path is extracted by the script *getDirToMerge*. To improve the flexibility of the simulator, the Matlab codes for the *Probe* (executed inside *SimulateRFLine*) and the *Merge* are provided at runtime as inputs: *ProbeCode_tgz* and *MergeCode_tgz* respectively.

The result of this composition step is shown on figure 23.

Consumer Pattern. There is an additional step for both FIELD-II and SINDBAD that is not captured on the VIP template (see figure 9), because it is irrelevant to the other two simulators (SIMRI and Sorteo): the final output is not only returned as a regular

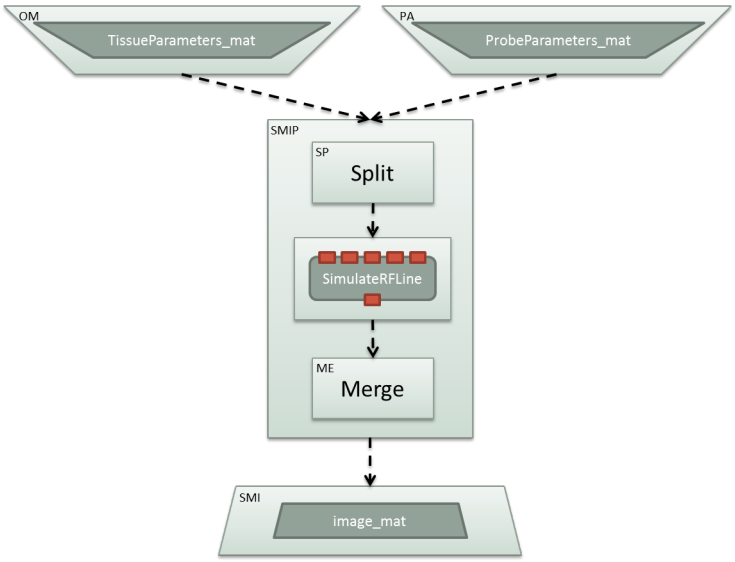

Figure 20: FIELD-II Weaving Split/Merge

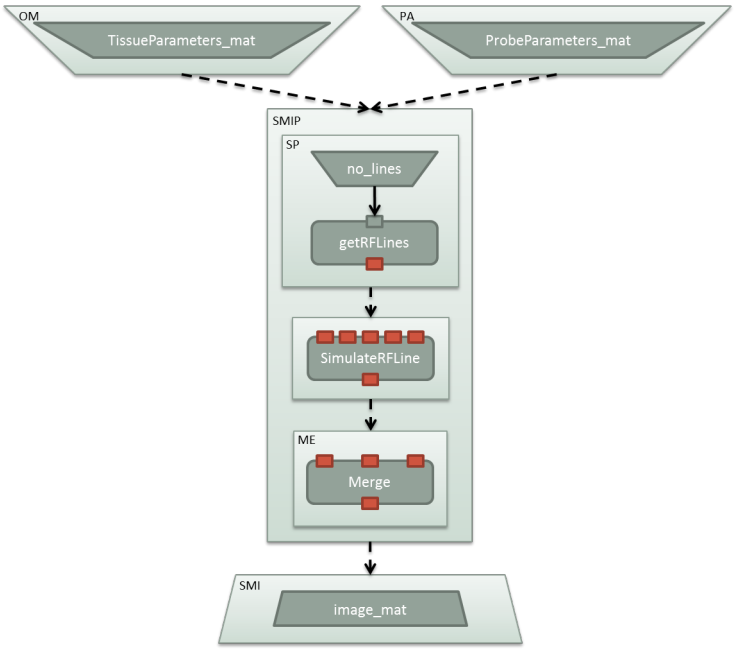

Figure 21: FIELD-II Mapping (step 2)

workflow output, but also stored on a web server. Let us introduce the pattern template *Consumer* (shown on figure 24).

Any pattern that consists entirely of post-processing activities and outputs, plugged on intermediary or final results of the base process is an instance of the Consumer. The case of the result upload, common to FIELD-II and Sindbad, fits into the *Consumer* template with its post-processing activity *put on web server* and its output *result URL*, despite its additional inputs *dataDir* and *pendingFile*, because those do not impact either its structure or the way it is woven into the base process. The [*ResultUpload*] instance of the

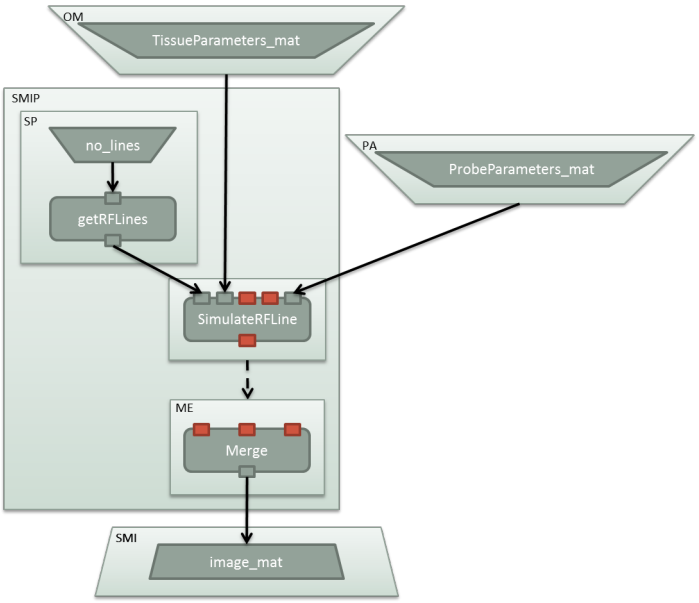

Figure 22: FIELD-II Composing (step 1)

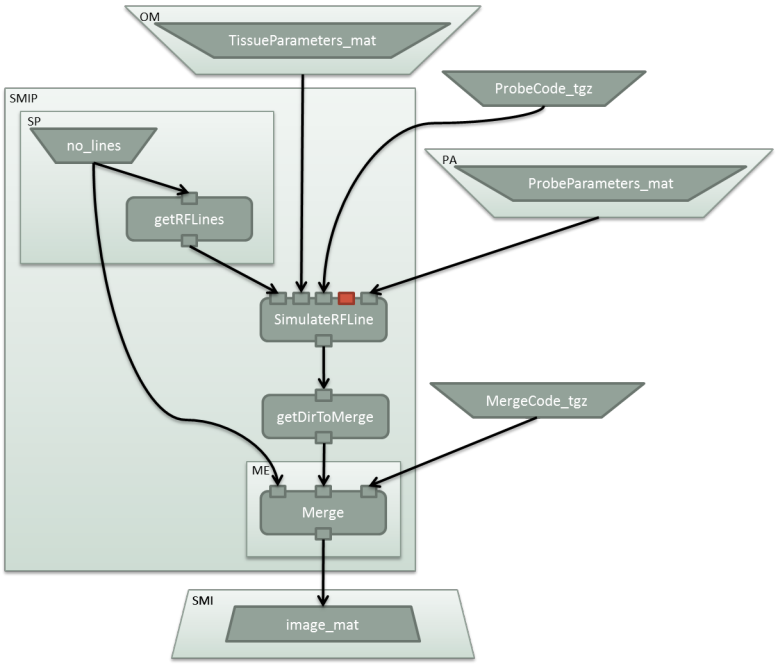

Figure 23: FIELD-II Composing (step 2)

Consumer pattern template is shown on figure 25.

The final composition step consists of plugging the [*DirectoryName*] instance of *Generator* (introduced on page 14) on the last unattached input port of *SimulateRFLine* and the [*ResultUpload*] instance of *Consumer* we just defined on the output port of *Merge*, to capture the very final result of the simulation. The result is shown on figure 26.

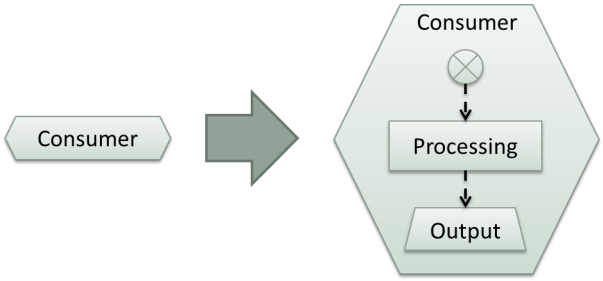

Figure 24: Consumer Pattern

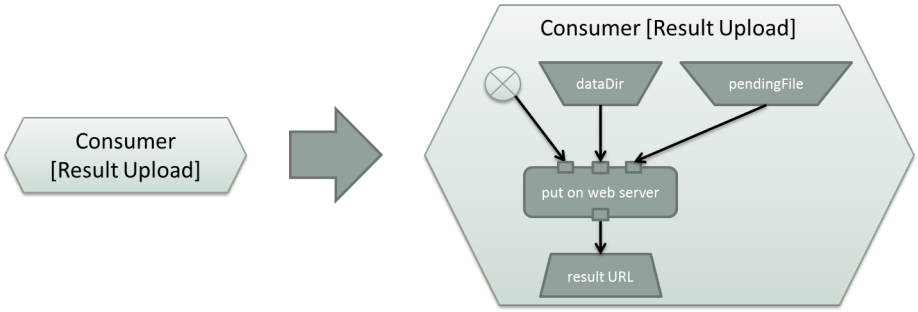

Figure 25: Consumer Instance for Result Upload

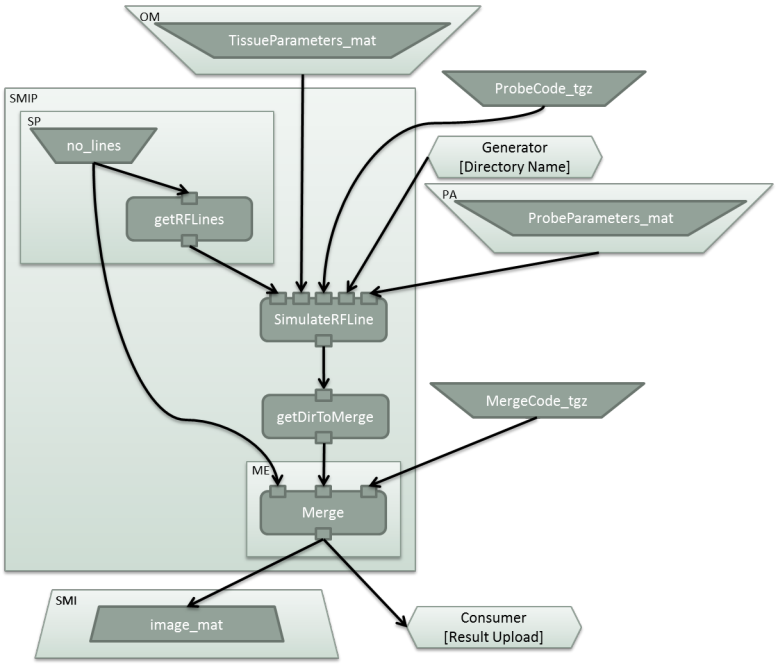

Figure 26: FIELD-II Composing (final)

Conversion. The conversion to a GWENDIA abstract scientific workflow (shown on figure 27) requires additional translation logic to infer the necessity for a control link between *SimulateRFLine* and *Merge* to produce the desired behavior (i.e. the unique call to *Merge* must be done after all the calls to *SimulateRFLine* are over).

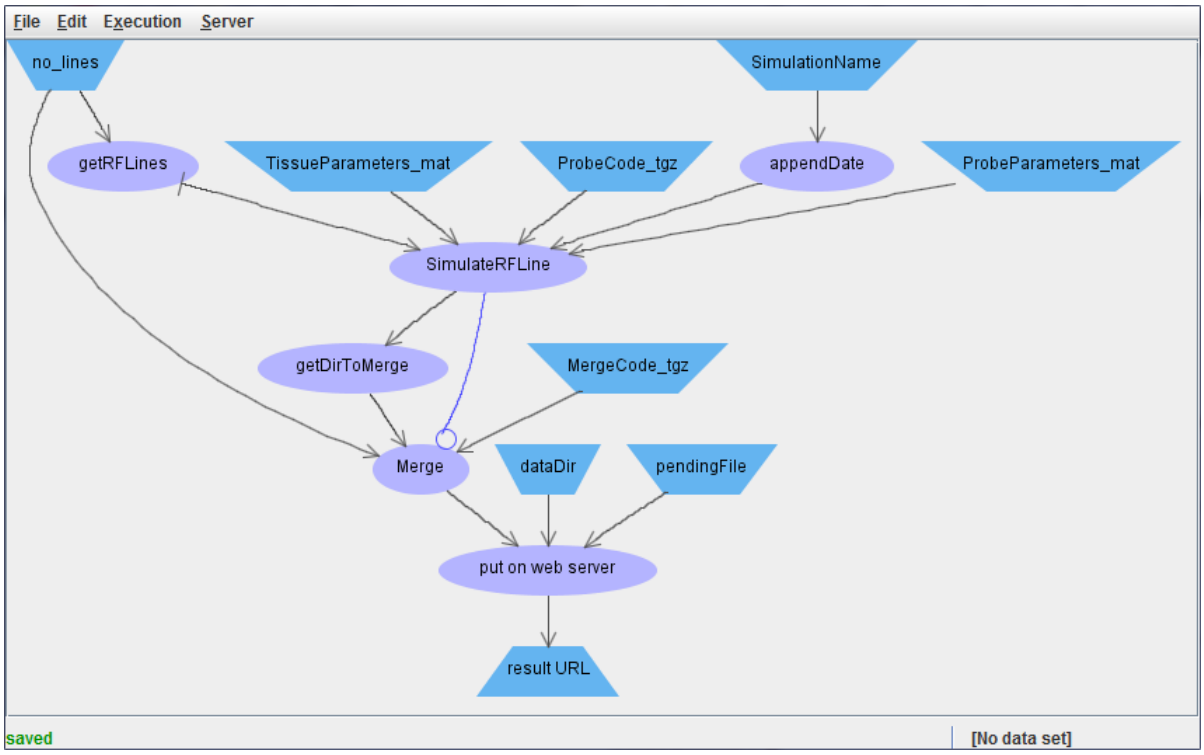

Figure 27: FIELD-II Abstract Workflow (GWENDIA/MOTEUR2)

3.2.3 Sindbad

Mapping. There are two input files for the model (*phantom* and *pegs4dat*) and three for the simulation parameters (*scan_file*, *chain* and *geo*). The main activity is called *sindbad*. The final synthetic CT scan is called *ResultFile*.

A Sindbad simulation comprises a large number of projections, each of them resulting from the merging of an analysis and a Monte-Carlo computation. There are three levels of inherent data parallelism:

- Projections (i.e. the individual 2D images) are independent.
- The analysis and the computation (Monte-Carlo algorithm) of each projection can be done in parallel.
- Random samplings of each Monte-Carlo algorithm can be computed simultaneously.

Like with FIELD-II, the main conceptual workflow is mapped to only one activity and therefore weaving the *Split*/*Merge* pattern is trivial. On the *Split* side, the number of data chunks is computed through *projection_count* and then *split_count* activities. On the *Merge* side, the merge activity is already composed with the activity *count_in_dir* that watches the directory where projections are stored to launch the merge at the appropriate time.

The result of this mapping and weaving of the *Split*/*Merge* pattern is shown on figure 28.

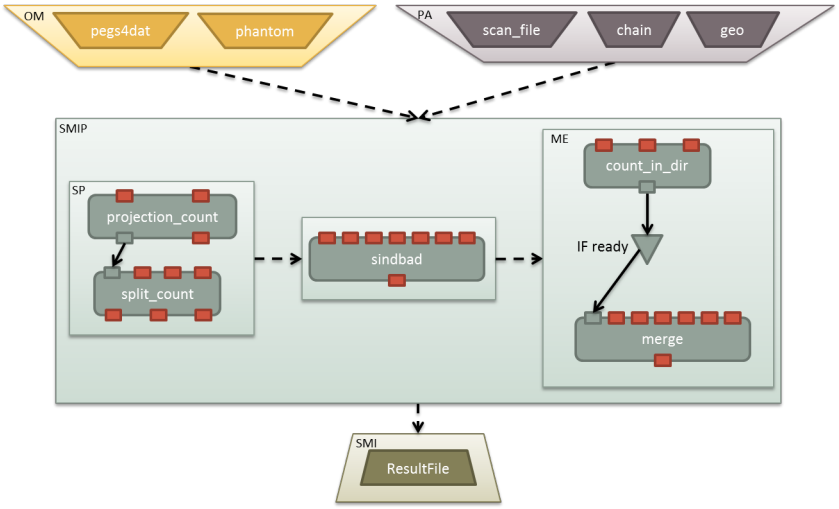

Figure 28: Sindbad Mapping

Composing. Though most of them are trivial, the Sindbad workflow features more than enough composition links to make it difficult to draw and read. To ease both processes a little, we used color codes instead of rigid boxes to show to which high-level input/output each input/output belongs to and we duplicated most of them. It is only a graphical duplication: for instance, the exact same input *geo* is plugged into *sindbad* and *merge*, but duplicating its graphical representation prevents convoluted arrows in the graph.

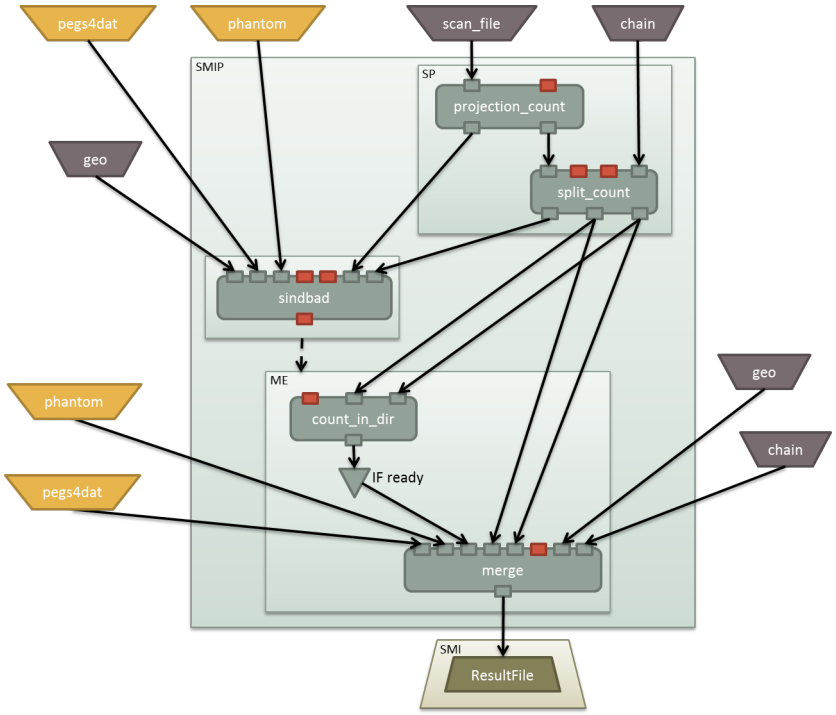

Figure 29: Sindbad Composing (intermediary)

The result of the most straightforward and, if annotations allow, automated composi-

tions is shown on figure 29.

The link between *sindbad* and *count_in_dir* is not straightforward because in some cases (which can only be determined dynamically by the *split_count* activity) data is not split and thus the *Merge* step must be bypassed entirely. That behavior is achieved by the addition of another conditional construct, like the one featured between *count_in_dir* and *merge*.

At that point, there are 6 ports left to attach:

- the additional inputs

  nb_jobs_max for $projection\_count$ and $default\_split$

  $nb\_particules\_par\_job\_max$ for $split\_count$

  $sindbad\_release$ for *sindbad* and *merge*

- the $[DirectoryName]$ instance of *Generator*, abbreviated in $G[DN]$, plugged in *sindbad*.

In both cases (with and without *merge* step), the *ResultFile* should also be uploaded on a web server, through the $[ResultUpload]$ instance of *Consumer* introduced on page 19.

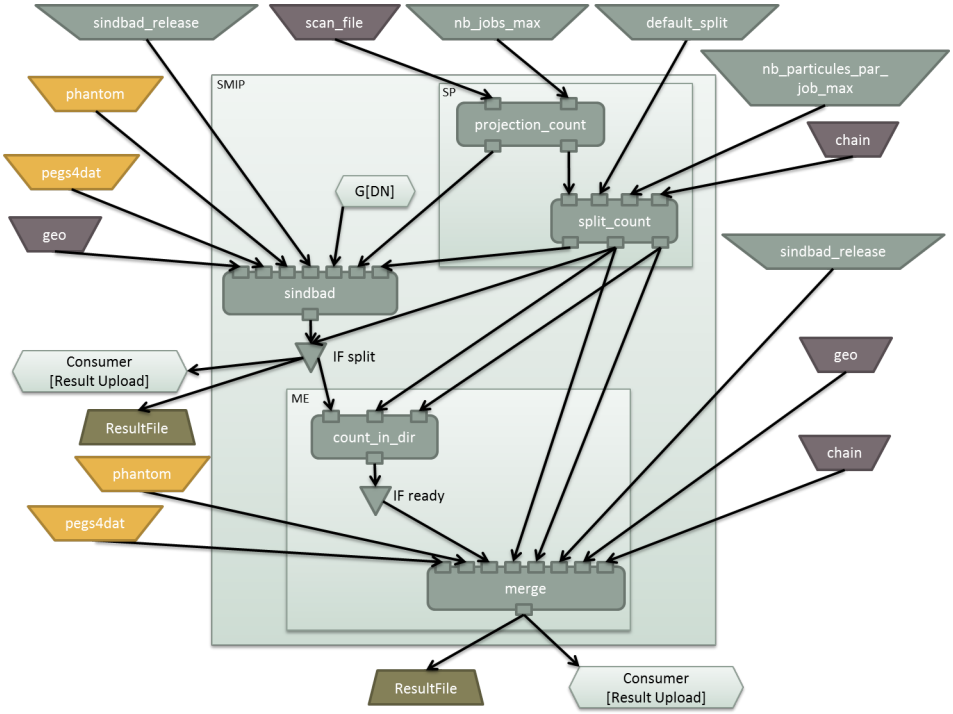

Figure 30: Sindbad Composing (final)

The final composition result is shown on figure 30.

Conversion. The translation to an actual GWENDIA abstract scientific workflow (shown on figure 31) is again, as with SIMRI, only a matter of format, except for a simplification we took the liberty of using to keep our diagrams legible for the purpose of the present

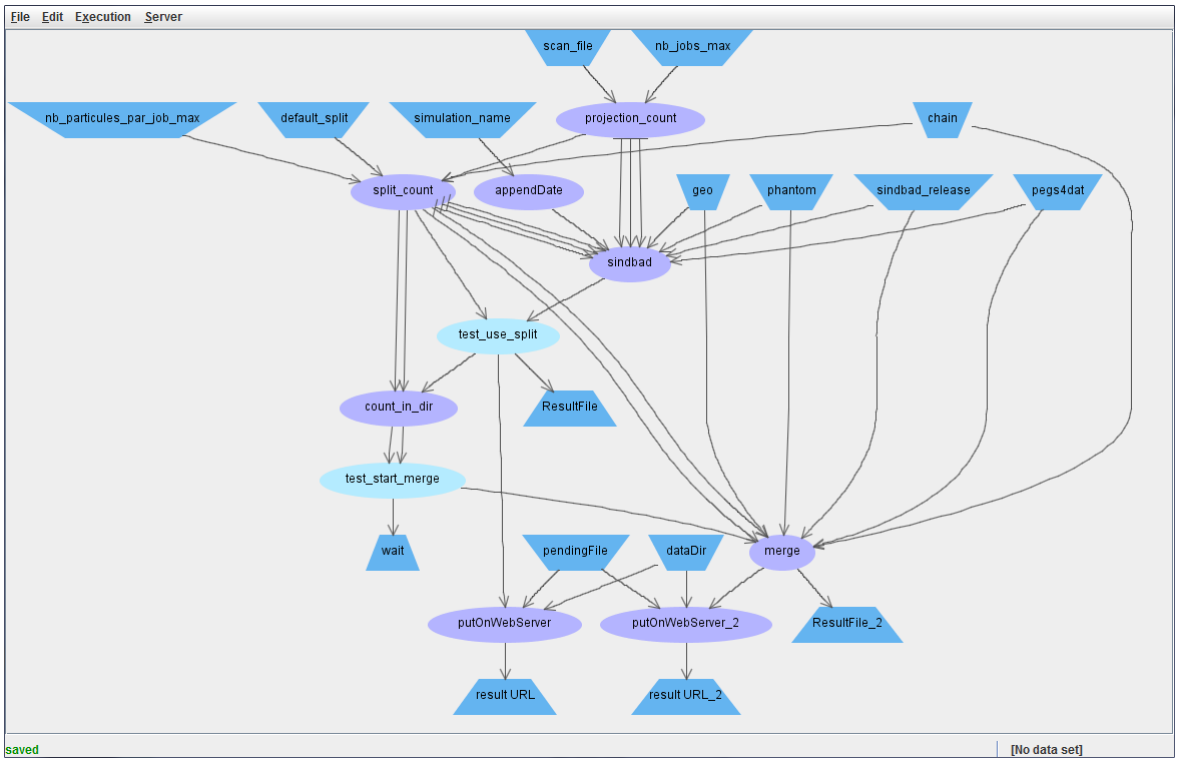

Figure 31: Sindbad Abstract Workflow (GWENDIA/MOTEUR2)

report: multiple data transfers between two activities were merged into just one graphically.

This does raise the issue of legibility with activities that have as many ports as Sindbad. On an actual platform dedicated to conceptual workflows, one way to alleviate this problem would be to group ports and arrows in much the same way we did here, but with the possibility for the user to expand such nests and look into the detailed ports and data links of any activity.

3.2.4 Sorteo

Mapping. Sorteo simulates a PET procedure through a Monte Carlo algorithm [4]. It is done in two separate steps: first singles (short for single photons) are generated with *sorteo_singles*, then emissions are computed from them with *sorteo_emission*. The object model is contained in a file called $fantome\_v$, the simulation parameters are in a file called *text_protocol* and the final output is a file called *sinogram*.

The data parallelism present in Sorteo can be found in any Monte Carlo algorithms: they are based on multiple random samplings that are inherently independent and can be computed in parallel. But weaving the *Split/Merge* pattern here is less trivial than in the other three simulators: since there are two activities, we have to determine in which of the two cases shown on figure 11 we are: can the data be split only once for both processing steps? Or must data be merged between the two activities? In the case of Sorteo, the second processing step (i.e. *sorteo_emission*) requires all generated photons, i.e. the merged outputs of all *sorteo_singles* calls. Therefore, we find ourselves in case 2:

we have to apply a separate *Split/Merge* pattern to each activity.

As it happens, both Sorteo simulation activities already implement the *Split* part of the *Split/Merge* pattern: they only process the part of the data indicated by one of their inputs. The *Merge* steps are achieved with activities *sorteo_single_end* for the first part and *sorteo_emission_end* for the second.

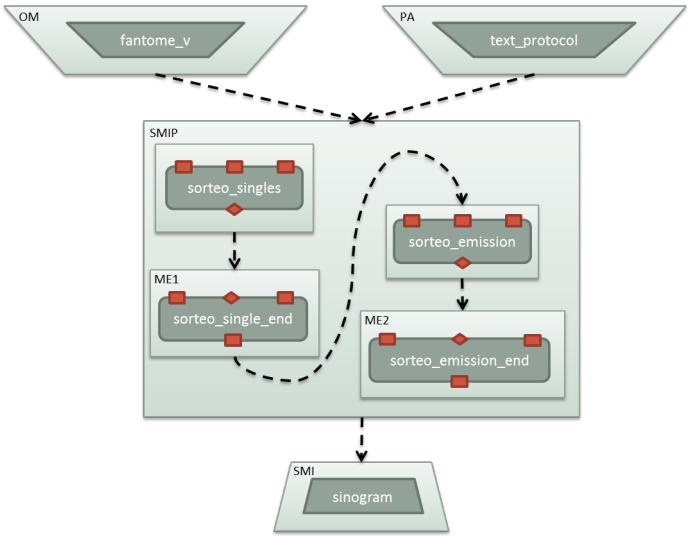

Figure 32: Sorteo Mapping

The result of mapping is shown on figure 32.

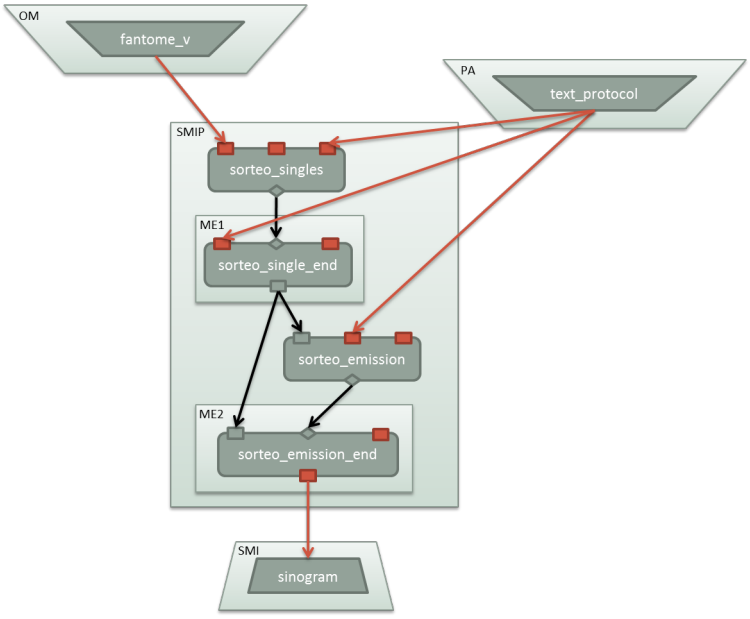

Figure 33: Sorteo Composing (step 1)

Composing. When trying to compose the activities mapped at the previous step, we encounter many type mismatches, shown in orange on figure 33:

- as its name suggests, *text_protocol* is not in binary format, as expected by *sorteo_singles*, the activity *CompileProtocol* will remedy that,
- both *sorteo_singles* and *sorteo_emission* expect a number indicating which data to process, which is produced by the script *generateJobs*,
- *sorteo_single_end* and *generateJobs* need the total number of jobs, an information that is contained in *text_protocol* but must be extracted through the script *parse_text_protocol*,
- and the final output is supposed to be of type raw SINO, but *sorteo_emission_end* produces an LMF file, the conversion of which is done by the converter activity $LMF2RAWSINO$ (which in turn also requires the binary protocol produced by $CompileProtocol$).

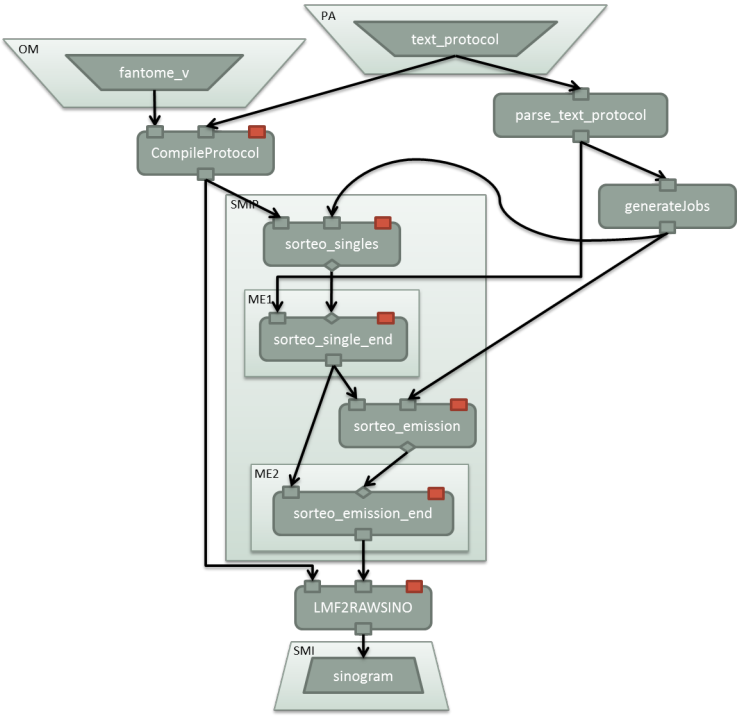

Figure 34: Sorteo Composing (step 2)

With all those type mismatches resolved, we are left with 6 input ports unattached, as shown on figure 34. All of those expect the name of the directory, as produced by the $[DirectoryName]$ instance of $Generator$, abbreviated in $G[DN]$, used in all other use cases as well.

The final result of the composition is shown on figure 35.

Conversion. When translating to an actual GWENDIA abstract scientific workflow (as shown on figure 36), we need not only make sure that all activities are fed the same directory name, but also that merge activities (*sorteo_single_end* and *sorteo_emission_end*) are appropriately synchronized. In practice, the implicit data links (i.e. the links between implicit ports) are translated into control links in GWENDIA, to ensure that merge activities wait until all processing tasks are done.

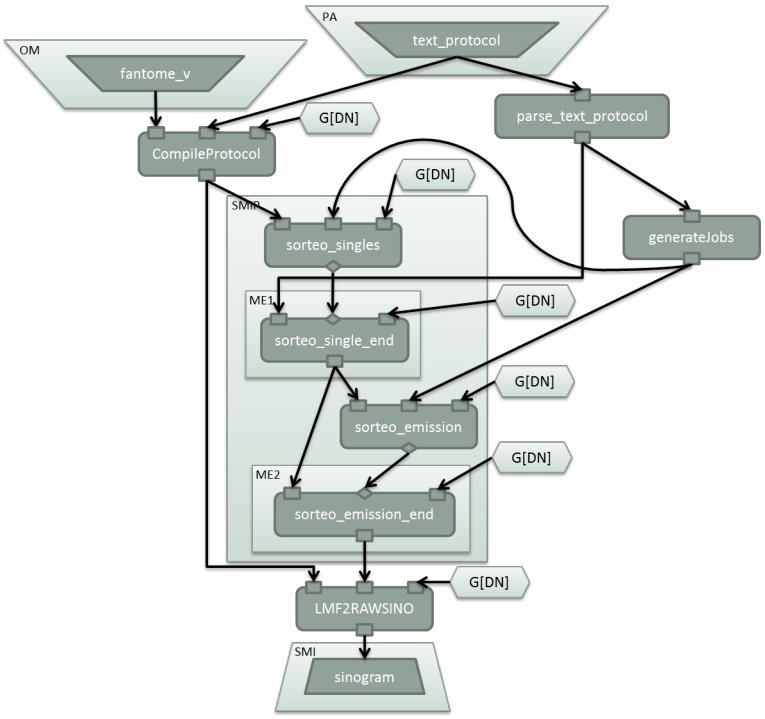

Figure 35: Sorteo Composing (final)

4 Conclusion

The VIP workflow designer will be presented to the end-user through a workflow editing graphical interface. It will include a Conceptual Workflow editing tool and a user-assisted workflow conversion tool. To efficiently assist the end user, the workflow designer will rely on extensive knowledge on data processing tools captured as semantic annotations and manipulated through inference engines well tested in the area of semantic Web technologies. The instrumentation of data processing tool services using the VIP project's ontology and their querying through a semantic reasoner is already considered in VIP Deliverable 1.2.1 "Data and tools repositories with basic query interface" [3]. The VIP workflow designer will interface with this tools repository and implement specific reasoning processes to fulfill its goals.

References

[1] Hugues Benoit-Cattin, Fabrice Bellet, Johan Montagnat, and Christophe Odet. Magnetic Resonance Imaging (MRI) Simulation on a Grid Computing Architecture. In *Biogrid'03, proceedings of the IEEE CCGrid03(Biogrid'03)*, , pages 582–587, Tokyo, Japan, May 2003.

[2] D. Friboulet, A. Gaignard, B. Gibaud, T. Glatard, and J. Montagnat. Simulation use-cases joint study with users. Technical Report M1.2.3, VIP project, April 2011.

[3] A. Gaignard, J. Montagnat, T. Glatard, and R. Ferreira Da Silva. Data and tools repositories with basic query interface. Technical Report D1.2.1, VIP project, September 2011.

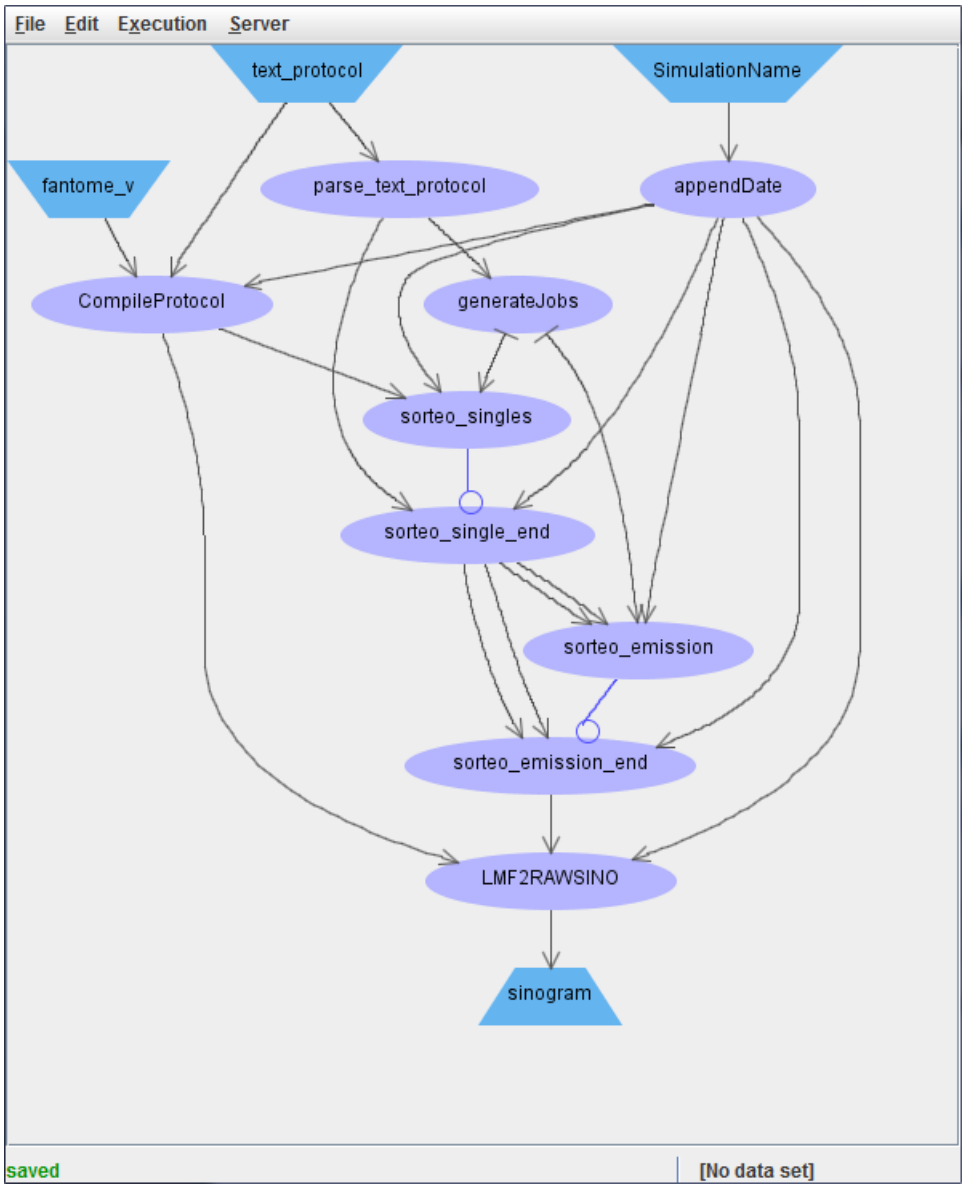

Figure 36: Sorteo Abstract Workflow (GWENDIA/MOTEUR2)

[4] Anthonin Reilhac, Carole Lartizien, Nicolas Costes, Sylvain Sans, Claude Comtat, Roger N. Gunn, and Alan C. Evans. PET-SORTEO: a Monte Carlo-based Simulator with high count rate capabilities. *IEEE Transactions on Nuclear Science (TNS)*, 51(1):46–52, February 2004.

[5] Ian Taylor, Ewa Deelman, Dennis Gannon, and Matthew Shields. *Workflows for e-Science*. Springer-Verlag, 2007.

[6] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD records (SIGMOD)*, 34(3):44–49, September 2005.