Chapter 3

Workflow Design Space

3.1 Introduction

When designing a workflow it is important to know what the design space is in which one is operating. In this chapter we first look at design approaches in general and then investigate the limits of the design space. We abstract from a lot of detail. One could see the workflow design problem as the task of constructing a complex workflow that is equivalent to a certain predefined computable function out of a set of standard components. As we will see this problem is in general unsolvable due to Rice’s theorem. Any formalism that tries to model automatic workflow composition in any generality is bound to fail. The work in this chapter is based on two previous publications by the author[120, 121] of this thesis. Suitable formalisms for reasoning about specific workflows is the subject of the next chapter. At the end of this chapter there is a discussion on the practical implications of the limits on the design space and what will be needed in a formal approach to specific workflows.

3.2 Related Work

A workflow is in essence a concurrent computational process. A number of formal approaches to the study of concurrent systems exist, amongst others: process algebras[74], guarded command languages[59] and actor theory[75]. Since most of these approaches commit themselves to certain philosophical preconceptions about the modeling of concurrent processes, they lack the flexibility to express the design issues we want to study. A well known dichotomy in this context is the division between shared memory communication and message passing communication. A choice for actor theory for instance would imply an undesirable restriction to message passing systems. Much practical work on design methods and in particular the use of different types of execution control has been done within the Ptolemy[79] project.
Although this project is aimed at embedded systems research, the fact that
the Kepler[31] scientific workflow management system is based on Ptolemy
shows its applicability to workflow design.

Workflows and related tools such as workflow engines which coordinate
the execution of a workflow are certainly not limited to e-Science. Within the
Business Process community much engineering as well as research has been
done [24]. In fact the SOA paradigm[64] which has been widely adopted
within the e-Science field, originated in the Business Process field. This can
be seen in the WSDL language being adopted for describing services, as well
as the use of the SOAP protocol for communicating with webservices. Although
the gridservice architecture defined in [64] leaves open the possibility
of another implementation, no complete alternative to SOAP in gridservices
has been implemented. Another area where the Business Process community
is influencing e-Science is workflow description languages. The most
widely used workflow description language in the Business Process field is
the Business Process Execution Language (BPEL), in [61] it is shown that
the BPEL for Web Services (BPEL4WS) [35] is suited for use in e-Science
Applications. BPEL4WS is a very expressive language, as can be seen in
the comparison[24] between different workflow definition languages from the
Business Process domain. While within the BP field being able to express
every form of business process and being able to reliably execute workflows
are the main priorities, e-Science has additional priorities which sometimes
take precedent over the BP priorities. First of all e-Science workflows have
to deal with massive amounts of data and massive (parallel) computation.
Furthermore sharing knowledge is important within e-Science, which places
extra emphasis on knowledge transfer associated with workflows. Due to
the knowledge transfer task of workflows in e-Science, proper representation
plays a more important part in the design of these workflows.

3.3 Workflow Design

Within e-Science different approaches exist to compose a workflow:

- **Concrete**: the manual combination of a set of elementary workflow
  components into a workflow.

- **Abstract to Concrete Design**: given an abstract high level descrip-
  tion of a computational task and a number of elementary workflow
  components, design a workflow that is equivalent.

- **Abstract to Concrete Construction**: given an abstract high level
  description of a computational task and a number of elementary work-
  flow components let an automatic design process generate a workflow
  that is equivalent.
Abstract workflows can be a mechanism to share not only workflow components, but also common e-Science workflow patterns such as running processes in parallel. They can also be used to share knowledge on the design pattern associated with a very particular technique, for instance data assimilation which is used as a case study in chapter 7. Another technique employed in workflow construction is the composite workflow. In a composite workflow one workflow element can be an interface to another complete sub workflow. This helps in keeping workflows understandable. Formally this implies workflow components need to be compositional. The composite workflow should be equivalent to a workflow where hierarchy is removed as illustrated in figure 3.1. Most workflow systems only allow for computational workflow elements, however some\[27\] allow human activity to be represented in a workflow as well.

### 3.3.1 Workflow design using abstraction

In section 3.2 it was already mentioned that formal methods are applied most often in workflow research for reasoning about expressivity. They are used to determine:

- What control flow constructs are needed.
- The execution model and workflow language that enable these control flow constructs.

We study reasoning about representation as a hierarchical problem. In abstract to concrete design methods one has to be able to represent a workflow at different levels of abstraction. If one wants to reason about this in a formal way it puts different demands on a formalism then when one is considering
CHAPTER 3. WORKFLOW DESIGN SPACE

Figure 3.2: Workflow design problem

the expressivity of workflows in a concrete workflow design method. In a hierarchical representation abstract descriptions should be computationally equivalent to detailed low level descriptions of workflows.

Now we will define a number of important concepts in workflow design using abstraction.

**Problem definition:** This is the most abstract representation of a workflow. It consists of a definition of desired input, as well as the desired output. These can either be defined in terms of data or processes. In e-science context it can be viewed as the research question associated with an experiment.

**Atomic workflow component:** Represents a computational process for which there is a direct mapping to an actual implementation. This can be a deterministic process like adding two variables, or it can be a non-deterministic process: a user entering a value based on the input he sees. In practice this usually is a web service, but it can also be an action performed by a human. For the rest of this chapter we assume an atomic workflow component to be computational. Furthermore as mentioned in the previous chapter, we assume it to be consistent. This means a workflow component must always be able to reach a final finished state and the content of its output must always remain consistent with the associated data definitions.

**Control Flow:** In a workflow data moves from one workflow component to another. The rules by which this data movement happens is called control flow. An example of such a rule is allowing or disallowing loops in a workflow.

**Execution model:** Not only does data need to be moved but execution of workflow components has to be started and stopped. The execution model takes care of orchestrating this execution and ensures only allowed
3.3. WORKFLOW DESIGN

Control flow constructs are used. Furthermore the execution model can be used to ensure workflow components are only connected when both components use the same data type.

**Compositionality:** is a property which can hold for workflow components, data connectors and a combination of both. Two or more atomic workflow components can be composed together to form a single composite workflow component. The behavior of this composite component can be explained in terms of its parts, by composing the parts together no new behavior emerges. Through composition internal actions are hidden from direct observation, the composite component presents itself to the outside only in terms of its inputs and outputs. Through compositionality complicated control flow patterns can be represented as one composite data connector. An entire workflow can also be represented as one composite workflow component.

**Workflow composition:** Workflow composition is achieved by connecting output and input of the workflow components through data connectors. These data connectors can be atomic communication channels, composite channels representing control flow patterns such as split or merge or in its most abstract form all interaction of a workflow can be one composite object with links to all workflow components. The composition of a workflow representation starts with a problem definition as well as a set of available atomic workflow components. The representation of a workflow needs to strike a balance between generality and specificness, resulting in an ideal workflow which is neither too abstract nor too detailed. This idea is illustrated in figure 3.2. Workflow design can be bottom up, top down, automatic or done by hand. To efficiently find a grounded design which satisfies the problem definition, the design space needs to be constrained. By grounded design is meant: a design that only consists of existing implemented workflow components, when represented in its most atomic form. In bottom up design the problem definition constrains the atomic workflow components that can be used in the first and last step of the workflow due to the fact that the inputs of the first step and outputs of the last step must match those of the problem definition. The outputs of the first step and inputs of the last step then form the constraints for the steps which can be used in between.

Within such a design process one should know what the design space looks like and what its limits are. We will look at this design space formally. For this we will use formal definitions of process and data set, of which workflow component and data connectors are respective instances. We show how a design space can be set up using composition of both processes and data sets. Note that this form of composition is more specific than that which happens in current workflow systems, where a combination of workflow components and connectors is composed together. In the design space that will be represented here both connectors and components can be composed separately.
CHAPTER 3. WORKFLOW DESIGN SPACE

3.4 Theoretical limits of Workflow design

In this section we give an informal treatment of the workflow design problem. We abstract from a lot of detail. As mentioned in the previous chapter when designing a workflow the goal is to answer a research question. In our treatment of the workflow design problem the research question is equivalent to a predefined computable function. The end result of the design process is a complex workflow, constructed from standard components, that is computationally equivalent to this function. As we will see this problem is in general unsolvable due to Rice’s theorem. Any formalism that tries to model automatic workflow composition in any generality is bound to fail. Yet the following definitions give a feel for the issues that are at stake.

3.4.1 Building blocks

We conceptualize a complex workflow as a collection of Turing complete processes which share data sets. The shared data sets may be thought of as memory locations, databases or variables:

Definition 3.4.1 (The class of data types) A class of recursive (effectively decidable) types $\tau$ defined on the class of binary strings $\Sigma = \{0, 1\}^*$ and closed under boolean operations.

Definition 3.4.2 (The class of data sets) A countable class $\Delta$ of typed data sets $d^t(t \in \tau)$ defined as memory locations with unlimited storage capacity.

Definition 3.4.3 A process is a deterministic computational function that has been defined in a Turing complete computational system. A process uses at least one dataset, to read its inputs and write its outputs.

As mentioned in definition and illustrated in figure 3.3 a process has at least one dataset on which to operate. One process is however allowed to manipulate more than one distinct dataset. This principle is illustrated in figure 3.4. This is needed because datasets will have to function as data connections in a workflow. Thus one process should be able to connect
3.4. THEORETICAL LIMITS OF WORKFLOW DESIGN

Figure 3.4: Illustration of a single process with multiple datasets

Figure 3.5: Illustration of an elementary workflow component

to multiple other processes. The connections in workflows are typed as is defined in the definition of datasets. A dataset can potentially accept multiple types. These complex data types are not trivial to implement, however using prefix-free or fixed length data types this is possible within the given definition of datasets. The possibility of using complex data types is needed later on, when the merging of multiple datasets into one abstract representation will be explored.

3.4.2 Workflow construction

Now that the basic building blocks have been defined we move to the construction of workflows starting with workflow components.

**Definition 3.4.4 (The class of elementary workflow components)**

An elementary workflow component $E = (I, O, D, P)$ consists of a computable function $P$ that takes the data set $I^i$ as input (read) and writes the result of the computation to the output data set $O^j$, where $i, j \in \tau$. We have $D = (I^i \cup O^j \cup D_{Int})$ where $D_{Int}$ are the internal data sets used by $P$.

Such an elementary workflow component is our formal equivalent to real world workflow components such as Kepler[31] actors, units in Triana[96] and processors in Taverna[104].

These workflow components are combined to form workflows. The way in which workflow components are combined has to be restricted in order to reflect the properties of actual workflow systems. That is why for our exploration of workflow design we are only interested in the class of consistent workflows, which is defined as follows:

**Definition 3.4.5 (The class of consistent workflows)** A workflow is a set of elementary workflow components $W$. A set of elementary workflow components $W$ is consistent if for each $E = (I, O, D, P_i) \in W$:
Figure 3.6: Illustration of example 3.4.7 an elementary workflow consisting of two elements connected through a shared dataset

- I and O are datasets of \( P_i \) contained in \( D \)
- \( P_i \) has an accepting computation for each possible variable assignment to members of I
- Each computation ends in an assignment of values to members of O that is consistent with their data-types.

As an example we show how the simplest possible elementary workflow component forms the identity workflow. The most basic workflow possible.

**Example 3.4.6 (identity workflow)** We take workflow component \( E_i = (IO, IO, IOP_i) \) where dataset \( IO \) is the only dataset. It is used for both input and output. Process \( P_i \) has just one state which is both its start state and accepting state. It does nothing with the data, resulting in a workflow that passes its input data unchanged to its output.

It should be clear that this simplest of workflows is consistent. For every possible allowed input there is an accepting state and output that is consistent with its data type. Workflows usually consist of more than one element, so next we show how one can sequence two elementary workflow components in a consistent workflow.

**Example 3.4.7 [sequencing]** Suppose we have three different datasets \( d_1, d_2, d_3 \) and two processes \( p_1, p_2 \). With these we can construct two elementary workflow components \( e_1 = (d_1, d_2, \{d_1, d_2\}, p_1), e_2 = (d_2, d_3, \{d_2, d_3\}, p_2) \). The output of \( e_1 \) is the input of \( e_2 \). If we define the datasets and processes to perform in the same way as example 3.4.6 we have a consistent workflow consisting of a sequence of two elementary workflow components.

### 3.4.3 Complex workflow construction

In the same way as the simple example of sequencing, more complex data connections can be set-up. It is not the purpose of this formalization to look at this type of expressivity as it has been shown extensively using other methods [23]. However we do want to look at the implications of employing more complex workflow patterns. The collection of all basic building blocks for the design of both simple and complex workflows is defined as a workflow repository:
3.4. THEORETICAL LIMITS OF WORKFLOW DESIGN

Figure 3.7: Illustration of encapsulation of processes, an elementary workflow can be simulated on a computationally equivalent single process with multiple datasets

**Definition 3.4.8 (Workflow repository)** A workflow repository $R$ is a finite set of elementary workflow components $E_i = (I_i, O_i, D_i, P_i)$ with disjunct data sets $D_i$.

The construction of complex workflows out of such a workflow repository involves two highly non-trivial operations: merging of data sets and encapsulation of processes. We briefly discuss both.

**Encapsulation of processes**

Suppose we have two workflows $E_i = (I_i, O_i, D_i, P_i)$ and $E_j = (I_j, O_j, D_j, P_j)$. We define a new encapsulated workflow that is the result of a merge of the two workflows and that is intended to be computationally equivalent to the combination of the two original workflows:

$$E_{i||j} = (I_i \cup I_j, O_i \cup O_j, D_i \cup D_j, P_i||P_j)$$

Two types of computational flow are possible, consecutive where one component in $E_{i||j}$ starts execution after the previous one has reached an accepting state, or parallel where more than one component is executing at the same time. Consecutive execution of two or more $p \in P_i, j$ can be simulated by $P_i||P_j$ through appending the transition function of $p \in P_i||P_j$. Parallel execution can be simulated on $P_i||P_j$ through dovetailing transition functions of elements of $P_i$ and $P_j$ involved. The created transition function for $P_i||P_j$ emulates the transitions in the set of processes $P_i \cup P_j$ using the datasets $D_i, j$.

This operation is non trivial. It might be impossible to construct adequate scheduling for the merged routines and to prove the equivalence and in relation to the merge operation on data sets some of the data sets in $O_i, j$ and $I_i, j$ might have to be added to $D_i, j$. 

CHAPTER 3. WORKFLOW DESIGN SPACE

Figure 3.8: Illustration of merging data sets, an elementary workflow can be simulated on a computationally equivalent process using one dataset

Merging of data sets

We can define a merge operation for data sets with the following redefinition of the signature:

$$d^k_i \otimes d^i_l = D_{\{k\} \cup \{l\}}^{(i\vee j)}$$

This means we take the conjunction of the tests for the types and the union of the indices. A related merging operation can be defined for classes of data sets.

The merging of data sets will not result in an unacceptable rise in the computational complexity $O$ of workflow components:

**Theorem 3.4.9** The time taken by a workflow component consisting of one dataset and one process to simulate $n$ steps of a workflow component consisting of multiple datasets and one process is $O(n^2)$.

For a proof, see [77] where the complexity of emulating $n$ steps of a multi-tape Turing machine on a single-tape Turing machine is shown to be $O(n^2)$.

**Lemma 3.4.10** A consistent elementary workflow can always be emulated on a workflow component consisting of one process and one dataset, where $n$ steps of the elementary workflow take no more than $O(n^2)$ steps of this workflow component.

This is a consequence of definition 3.4.3 and theorem 3.4.9.

Due to the non trivial nature of the encapsulation operation employed the workflow component in lemma 3.4.10 is not necessarily consistent. The merge operation is also non trivial. By means of merging of data sets we create links between elementary work flows. Even if the types of the data sets are the same, timing issues might cause the data to be corrupt or overwritten.

Execution model

Generally speaking in order to construct a complex workflow on the basis of a workflow repository a number of issues have to be dealt with:
• The creation of a workflow topology by means of merging input and/or output data sets (possibly of different signature) and the encapsulation of processes.

• The selection of an execution model for the topology.

The execution model deals with the timing and scheduling issues mentioned in the previous sections on encapsulation and merging operations. In general it is the equivalent of a director in the Kepler workflow system or the data flow mechanism which is implicit in most other workflow management systems. In the rest of this chapter we will concentrate on topologies as they are the part a workflow designer manipulates. Execution models are also important as they deal with the timing and scheduling issues mentioned previously. The workflow designer has to select a suitable execution model, however the construction of an execution model is not his domain. Therefore, it is assumed that for the class of consistent workflows at least one consistent execution model exists for each workflow. With this assumption in place additional limitations to workflow design still remain.

3.4.4 Workflow design limits

Any concurrent computational process that is associated with a workflow can exist in at least three extreme guises:

• A consistent elementary workflow $W$ based on a multitude of processes combined in a complex topology.

• A workflow component consisting of one process and multiple datasets

• A workflow component consisting of one process and one dataset.

This analysis gives us the coordinates of a design space for workflow components. Although it must be noted that this domain does not allow the construction of composite workflows, it allows us to study a number of design issues in a formal context. The workflows can be ordered along two dimensions: data-complexity and process-complexity. The four extreme corners of these dimensions are:

• Multi Process, Multi Data (MPMD) workflow: the consistent elementary workflow $W$ based on a multitude of processes using a multitude of data sets.

• Single Process, Multi Data (SPMD) workflow: A multi dataset process. This is often the start of an abstract-to-concrete design process: the application is conceived as a single complex process working on a multitude of data sets.
CHAPTER 3. WORKFLOW DESIGN SPACE

Figure 3.9: Lattice of all possible representations for a computationally invariant workflow

- Single Process, Single Data (SPSD) workflow: A single dataset process. Conceptually this might be interpreted as a classical compiled application working on a dedicated database.

- Multi Process, Single Data (MPSD) workflow: this is a degenerate case in which a multitude of processes use one dataset. One might think of a collection of agents managing a single database.

This domain describes implementation variants for routines that are computationally equivalent. Both dimensions generate a lattice-like structure. If one starts with a MPMD workflow one can gradually combine either processes or datasets to create a SPSD workflow in the limit. This is illustrated in figure 3.9 where each arrow indicates either a combination (or split) of a dataset or process. The combination of maximum generality and simplicity is to be found within an SPSD workflow. All different workflows are computationally invariant, i.e. they compute exactly the same function and are of comparable complexity (according to theorem 3.4.10). The only reason to favor one version in the domain over another is a matter of desirable design qualities.

It must be noted that this domain is far too complex to be characterized by any finite analysis. Thus instead of a precise analysis of all design issues, we concentrate on some important issues.

We can study the influence of parameterization in the context of this domain. Each combination of two processes into a single more abstract process representation leads to extra parameters being created in this more
abstract representation. Each time multiple datasets are replaced by a single dataset, this single dataset has to accept additional types. This leads to more complex data types. In the ultimate SPSD form we end up with something that can take the description of any process with any data and execute it. In other words a universal Turing machine. Such a universal Turing machine $T_U$ is the ultimate parameterized routine.

That there are limits to generalization by means of parameterization is clear on the basis of the Halting set. There is no recursive routine that in general will decide whether a Turing machine will stop on a certain input.

This is bad news for workflow designers since the universal Turing machine is the ultimate SPSD workflow. It gives rise to the following claim:

**Claim 3.4.11** In workflow design simplicity, generality and consistency are mutually exclusive.

In other words a workflow that is both general and simple will be inconsistent. Another way of picturing the situation is the following: suppose one starts the design of a workflow system on the basis of an MPMD representation. By way of merging datasets (thus creating the necessity for complex data types) and merging processes (by means of adding parameters to the data types) one can create more general workflows using less datasets and less processes. The price one has to pay is computational complexity and data set complexity: the ultimate SPSD workflow is undecidable and thus inconsistent. In other words: simplicity and generality imply complexity of both the data and the computational processes.

We can now define the **general abstract to concrete workflow design problem**: given the description of a computational process and a number of elementary workflow components can we construct a workflow with a topology that is computationally equivalent? In other words: given a SPMD workflow, can we automatically construct an equivalent MPMD workflow? Formally:

**Definition 3.4.12 (General workflow design problem)** Given a workflow component consisting of a single process and multiple datasets $E_i = (I, O, D, p_i)$, where $D$ is a collection of multiple datasets and given a repository $R$ of elementary workflow components employing a set of processes $P$ using $D$ as datasets can we construct a workflow $W = (I, O, D, P)$ that is computationally equivalent to $p_i$?

Due to Rice’s theorem[111] this problem is undecidable. Rice’s theorem states that for every non-trivial property of any partial function it is undecidable whether any other partial function computes this property. The computational equivalence of two workflow representations is non-trivial in almost any case, the behavior of a workflow does not hold for all partial functions nor for none. Thus for the general case of workflow representation
this means automatic workflow composition is not possible. Only when serious constraints are placed on the properties of workflow components and data connectors does this become possible.

3.5 Discussion

Automatic workflow composition in the general case is not possible. Under certain restrictions however, there can be a role for automatic workflow composition. From a software engineering standpoint this seems attractive (it can drastically reduce design time). On the other hand its role will always be a limited one, since it cannot guarantee the correctness of all aspects of a workflow. From a methodological point of view automatic composition is not always desirable. Within e-Science it is still the responsibility of the scientist performing an experiment to make sure that it is scientifically sound. Ensuring user transparency becomes increasingly important when automatic composition is employed. Thus in practice it is not always desirable to have an abstract workflow which is specified to such a degree that a consistent concrete workflow can be automatically derived from it. Letting the user interactively add information to the abstract workflow specification allows for a more general applicability of an abstract workflow and also ensures that the user knows what is happening. This would be especially true when the abstract workflow takes the form of a design pattern such as parameter sweep or data assimilation which can be employed in many different domains.

To formally verify whether a workflow satisfies the problem definition and behaves correctly, all atomic workflow components and their connectors have to be taken into account. This is to be expected when no part of such a workflow has been constructed before. However, when there is the possibility of reusing previous designs, formally verifying the reused parts, which were already verified previously, is inefficient. The workflow design lattice introduced earlier in this section offers a solution. By using the property of compositionality previously created workflows and connectors can be abstracted to single components. Compositionality can be used as the mechanism to abstract both data and process in the workflow design lattice. If these abstracted components have been verified, their internal actions will not have to be verified again when they are reused in a new design. They have been shown to satisfy their problem definition, thus this can be used for defining the composite component, leaving out all the details of what happens internally. The big problem however, as already identified for automatic workflow composition, is that the correctness of an entire workflow is undecidable for the general case. In the workflow design lattice proving one representation is computationally equivalent to another is impossible due to Rice’s theorem in the same way that it is impossible to prove a specific
3.5. DISCUSSION

Figure 3.10: checking whether sub workflow with different execution model implements sub problem description

workflow satisfies the requirements when performing automatic workflow composition. Thus when using any formal approach to workflow verification, a careful approach has to be taken in which constraints are placed on the expressiveness of the formal model of a workflow. In the next chapter we shall see that different formalisms take different approaches in representing workflows. Each with benefits and limitations when trying to verify different aspects of a workflow design.

The issue of execution models was skipped in the formal approach represented here. For setting up the lattice representing the workflow design space it was not necessary to go into much detail concerning execution models. However any scientist who actually needs to formally check the validity of his workflow but also wants to use the hierarchical abstract to concrete design method presented, will need a formalism which can handle execution models in a practical way. A very good example where both abstract to concrete design and execution models come into play is the use of multiple execution models. When creating a representation of a workflow with multiple models of execution (illustrated in Figure 3.10) one can first create the workflow using execution model A and leave the sub problem, which can only be solved under execution model B, as a problem definition. In other words the sub problem is left as a stub to be resolved in a different
workflow system with a different model of execution. One often used model of execution is data-flow which allows for high data throughput, but puts limits on expressivity. Sometimes both expressivity and high throughput are needed in one experiment. As an example: let A be a data-flow based system and B a less constraint execution model which allows a user to steer execution. By allowing this, system B possibly violates the constraints of system A. If this is the case then the composition of WFA and WFB is not valid. This is clearly an undesirable situation, for which the simple solution is not using system B within A. In practice however it can often be the case that execution model B could violate the constraints, but that the actual sub workflow that was created under B does not. It can therefore be useful to test for this case.

The definitions of process and dataset presented in this chapter are not the most practical way of formally reasoning about actual instances of workflows as used by an e-Scientist. That is why in the next chapter there is a review of formalisms which can be used in the workflow design process itself. Such a formalism needs to have certain properties. First of all it needs to be able to abstract both details of process and communication through composition. It also needs to be expressive enough to express all possible workflow patterns, but also all possible processes which can be workflow components. Practical tools such as simulators, model checkers and theorem provers have to be available for such a formalism. Preferably these tools are already tailored towards workflows.

3.6 Conclusions

In this chapter we presented a number of evaluation criteria for workflow components and we constructed a domain of computationally equivalent workflows with different process and data complexities. We have shown that computationally equivalent workflows can be evaluated in terms of two dimensions: data complexity and process complexity. Using this formal framework we have proved that maximal simplicity, generality and consistency are mutually exclusive. We have defined a formal version of the General Workflow Design Problem and have shown that this problem is undecidable in the general case thus putting limitations on formal verification of workflows. We have shown that this limits the applicability of automatic workflow composition, but also calls for careful selection of formalisms used in a abstract to concrete design methodology.