Sara Migliorini[1,†]
Mauro Gambini[1,†]
Marcello La Rosa[2]
Arthur H.M. ter Hofstede[2,3]

Pattern-Based Evaluation of Scientific Workflow Management Systems

[1] University of Verona, Italy
[2] Queensland University of Technology, Australia
[3] Eindhoven University of Technology, The Netherlands
[†] part of this work was conducted while visiting Queensland University of Technology, Australia

February 2011

Contents

1 Introduction 1

2 Overview of Scientific WfMSs 3
- 2.1 Scientific vs. Business WfMSs 3
- 2.2 Advantages and Disadvantages 5

3 Introduction to Kepler, Taverna and Triana 7
- 3.1 Kepler 7
  - 3.1.1 Modeling Paradigm 8
  - 3.1.2 Kepler Routing Constructs 12
- 3.2 Taverna 19
  - 3.2.1 Modeling Paradigm 20
  - 3.2.2 Taverna Routing Constructs 21
- 3.3 Triana 21
  - 3.3.1 Modeling Paradigm 22
  - 3.3.2 Triana Routing Constructs 22

4 Pattern-Based Evaluation of Scientific WfMSs 27
- 4.1 Workflow Control-Flow Patterns 27
  - 4.1.1 Basic Control-Flow Patterns 27
  - 4.1.2 Advanced Branching and Synchronization Patterns 34
  - 4.1.3 Multiple Instance Patterns 47
  - 4.1.4 State-based Patterns 50
  - 4.1.5 Cancellation and Force Completion Patterns 54
  - 4.1.6 Iteration Patterns 54
  - 4.1.7 Termination Patterns 57
  - 4.1.8 Trigger Patterns 58
  - 4.1.9 Results 59
- 4.2 Workflow Data Patterns 62
  - 4.2.1 Data Visibility Patterns 62
  - 4.2.2 Data Interaction Patterns 64

4.2.3 Data Transfer Patterns ... 67
4.2.4 Data-based Routing ... 69
4.2.5 Results ... 71
4.3 Workflow Resource Patterns ... 73
4.3.1 Creation Patterns ... 73
4.3.2 Push Patterns ... 74
4.3.3 Pull Patterns ... 74
4.3.4 Detour Patterns ... 74
4.3.5 Auto-Start Patterns ... 75
4.3.6 Visibility Patterns ... 75
4.3.7 Multiple Resource Patterns ... 75

5 Scientific Workflow Patterns ... 77

6 WfMS Design Recommendations ... 85

7 Related Work ... 87

8 Conclusion ... 91

References ... 92
References ... 92

Chapter 1
Introduction

Scientific Workflow Management Systems (WfMSs) are software systems developed for automating scientific experiments that need to deal with huge amounts of data. The main goal of these systems is to facilitate the reuse and integration of domain specific functions and tools through a graphical environment. Scientific WfMSs can be used to automate repetitive error-prone activities, such as data access, integration, transformation, analysis and visualization, allowing scientists to focus on the domain specific aspects of their work. At the same time, they optimize workflow execution and resource consumption in a transparent way, scheduling the work on distributed architectures, such as a cluster of computers or a Grid environment.

The effectiveness of a scientific WfMS mainly depends on the available functions for a particular scientific domain and the integration with domain-specific tools. The diverse nature of tasks being performed across different scientific domains (e.g. biology, physics) has led each scientific community to develop and adopt a new workflow solution best suited for their requirements, rather than extending an existing solution [1]. As a result, to date there is a variety of offerings and still no accepted standard scientific WfMS.

In light of this, in this paper we aim to provide a comparative analysis of different scientific WfMSs. The purpose is to evaluate their suitability independently from the available domain-specific functions. In the Business Process Management (BPM) community, the Workflow Patterns Initiative [2] provides a framework to compare the suitability of WfMSs for business process design based on a set of recurring features (so-called *patterns*), as they are provided by various WfMSs. Inspired by this initiative, we present a pattern-based evaluation of three well-known open-source scientific WfMSs: Kepler [3], Taverna [4] and Triana [5]. In doing so, we also compare them with traditional WfMSs. In particular, we discuss why some groups of patterns are directly supported by business WfMSs but not by scientific WfMSs. For example, we observed that in some cases a pattern is not directly supported because it is deemed not to be relevant for that application domain; in other cases this is a reflection of the immaturity of the system. In order to conduct

this pattern-based evaluation of scientific WfMSs, we first unambiguously defined the behavior of the routing constructs offered by the three systems in question in terms of Colored Petri Nets (CPNs). Moreover, we identified a set of new workflow patterns emerging from the analysis of the functionality offered by the systems investigated. These patterns are not available in business WfMSs and mainly relate to various ways in which input data can be prepared and combined before being passed to a particular task. We also discussed the possibility to support these new patterns in a business WfMS, taking the YAWL environment [6] as a representative system.

The remainder of this paper is organized as follows. Chapter 2 discusses the main characteristics of scientific WfMSs and their differences with respect to business WfMSs. Chapter 3 introduces the three considered scientific WfMSs: Kepler, Taverna and Triana and formalizes their routing constructs. Chapter 4 presents the pattern-based evaluation of these systems. Chapter 5 formalizes four new workflow patterns that emerged from the analysis of these systems, while Chapter 6 presents some design recommendations for WfMSs derived from the strengths of both business and scientific WfMSs. Finally, Chapter 7 summarizes related work and Chapter 8 draws conclusions.

Chapter 2
Overview of Scientific Workflow Management Systems

At a glance business WfMSs and scientific WfMSs seem very similar. They both rely on a graphical representation of a workflow in order to enhance comprehensibility and ease the modeling activity. This is justified by the fact that both classes of systems deal with concurrent entities and a graphical representation can be helpful [7]. In particular, business WfMSs tend to capture the interaction among different concurrently operating human resources, while scientific WfMSs are often intended as an integrated problem solving environment to chain specialized applications. Despite this, business and scientific WfMSs are built upon two substantially distinct execution paradigms which yields fundamental differences between them. Section 2.1 presents the main differences between scientific and business WfMSs, whereas Section 2.2 discusses the advantages and disadvantages of scientific over business WfMSs.

2.1 Scientific vs. Business WfMSs

The major difference between scientific and business WfMSs is in their execution paradigm. Scientific WfMSs are typically data-flow oriented, i.e. the enablement of workflow tasks is determined by data availability. Business WfMSs are control-flow oriented, this means that task enablement is determined by the relative temporal ordering imposed on tasks.

More precisely, scientific WfMSs rely on the *Process Networks* computational model [8], in which tasks can run in parallel exchanging data through ideally unbounded channels[1]. A *data-flow dependency* $A \dashrightarrow B$ between two tasks A and B means that B *may* need some data produced by A to execute. However, A and B can potentially execute in parallel, because A can produce its output even before its completion, or the data produced by A may not

[1] Kepler represents an exception, since it also provides other computational models, see Chapter 3.

be needed for a particular execution instance of B. In this kind of system data are exchanged among tasks through ideally unbounded channels inside tokens, capturing both the data and the fact that data are available. In the following we refer to this kind of token as *data token*.

Business WfMSs on the other hand are control-flow oriented, i.e. they focus on the definition of the temporal and logical relationships among tasks [9], which determine a partial order for task enablement. The semantics of a business workflow can be described in terms of *Petri Nets* [10] and classic control-flow structures based on task termination relationships. A *control-flow dependency* $A \rightarrow B$ between two tasks A and B means that an instance of B shall only run after the completion of an instance of A. In business WfMSs the concept of *token* coincides with the concept of thread of control.

The execution paradigm adopted by business WfMSs makes it simpler to understand workflow execution, because the control-flow is made explicit and does not need to be derived from data dependencies. In scientific workflows the control-flow can be hard to deduce, as it is defined in terms of fine-grained data exchanges. Modeling control-flow intensive tasks by only using data-flow constructs often leads to overcomplicated models [11, 12]. For this reason the majority of scientific WfMSs provide some specialized modeling constructs, here called *routing constructs*, to explicitly describe control-flow dependencies such as loops or conditional branches. The adoption of a data-flow computational model in place of a control-flow one, determines three fundamental differences between scientific and business WfMSs:

1. *Data tokens vs. shared variables.*
   In business WfMSs data are usually stored inside variables which are shared among tasks inside a predefined scope. In scientific WfMSs on the other hand, there are no shared variables and tokens represent both the availability of data and the data values themselves. In this way each task receives its own copy of the data and its execution generally does not produce side effects for the other tasks. To clarify this aspect, let us consider a task A which needs an input value that may be produced by both a task B and a task C. Suppose that this input value is stored into a shared variable x and B writes a value b into x. If C writes another value c before A has read the value b, this value will be lost. Conversely, in scientific WfMSs the data produced by B and C are queued into a channel and retained until A consumes both of them.
2. *Self-concurrent vs. self-sequential behavior.*
   In scientific workflows each task can safely execute concurrently with itself. If a data-flow dependency is defined between two tasks A and B, as soon as an instance of A produces the necessary data, a new instance of B can start executing concurrently with other instances of B. This characteristic of scientific WfMSs reflects the need in a scientific domain to run the same process multiple times with different data sets. In business WfMSs the concurrent execution of the same task needs to be modeled explicitly, e.g. through a multiple instance task. Such constructs also

require to take into account synchronization issues that may arise from the execution of multiple instances of a given task. For instance, specific rules need be defined to merge the values produced by all instances of a task in a single shared output variable (e.g. an array).

3. *Individual semantics vs. collective semantics.*
   In scientific workflows tokens contain data and are consumed in the order of their arrival: they are not mutually interchangeable, unless explicitly stated by some special constructs. Conversely, in business workflows tokens represent threads of control that do not carry data information and usually are not distinguished from each other.

Finally, the purpose of scientific and business WfMSs is different. Business WfMSs are usually human-centric, their main goal being the coordination of work performed by various human resources within or across organizations. In contrast, scientific WfMSs focus on the automation and optimization of computations performed by one or just a few users, in which human intervention is considered marginal [13, 12]. Moreover, the nature of these interventions is usually different: no work is assigned to users (i.e. all tasks are automated), and users may only be involved in taking routing decisions. This latter paradigm bears close resemblance to *system WfMSs* used for enterprise application integration, as opposed to *human-centric WfMSs* [14].

2.2 Advantages and Disadvantages

The main advantage of scientific WfMSs is the ability to naturally support parallel computations. Data are contained inside tokens and there are no shared variables, thus multiple instances of the same activity can safely execute in parallel with other instances of the same task or of different tasks. Parallel computations in business WfMSs on the other hand have to be carefully designed, because data aspects are mostly implicit and synchronization operations must be explicitly managed, as data are usually contained inside shared variables.

Scientific WfMSs have been developed for supporting long-running intensive computations on huge amounts of data. For this reason, most offerings support the efficient execution of workflows exploiting Grid technologies.

Scientific WfMSs are more scalable than business ones, because in the former systems composite tasks can accept the necessary input also when the computation is started, and they can produce the computed output even before their completion. Two composite tasks A and B, between which a mutual relation $A \dashrightarrow B$ and $B \dashrightarrow A$ is defined, can start running in parallel and when the data needed from one task becomes strictly necessary, that task can suspend itself waiting for this input from the other task. Business WfMSs can only overcome this limitation by implementing these tasks as communicating processes, not as simple functions.

A potential problem determined by the adoption of a data-flow paradigm in scientific WfMSs is the difficulty to follow the workflow execution steps, because control-flow relations are implicit and based on fine-grained data exchanges. As a result, it is difficult to keep track of all concurrent task instances at a time, given that these instances may belong to different parts of a model which are apparently unrelated to each other.

Another drawback comes from the assumption that scientific workflows are executed only by one person at a time and human interactions are limited to perform a choice or provide a missing input. Therefore, little or no support is provided for human activities inside a process. One may argue that human interaction is not needed in scientific workflows, due to the nature of the performed experiments. However, this undoubtedly limits the applicability of these systems to experiments that largely depend on human interaction and in which domain expert knowledge plays a major role.

Finally, some systems provide very poor support for the representation of control-flow relations. Thus, capturing simple routing constructs such as loops or conditional executions may thus not be possible unless by hard-coding these constructs in a programming language [12].

Chapter 3
Introduction to Kepler, Taverna and Triana

This chapter introduces the three open-source scientific WfMSs considered in this work: Kepler, Taverna and Triana. We chose these systems because they are among the most mature and used open-source scientific WfMSs [1].

For each system we present the core features and the adopted terminology, and we formally describe its main routing constructs. A *routing construct* is a construct used for capturing workflow logic, such as the routing of tokens, and not a specific domain function provided by the system. The behavior of such constructs is formalized in terms of Colored Petri Nets (CPNs) [15]. The CPN representations presented in this chapter make use of advanced arc expressions whenever possible for reducing the complexity of the construction.

3.1 Kepler

Kepler [16] is an open-source scientific WfMS developed by the members of the Science Environment for Ecological Knowledge (SEEK) project and the Scientific Data Management (SDM) project. It extends Ptolemy II [17], a software system for modeling, simulating, and designing concurrent, real-time systems, developed at UC Berkeley.

Kepler inherits from Ptolemy II the support for multiple heterogeneous models of computations (MoCs), captured by the notion of *directors*, that allow the representation of different kinds of systems. The distinctive characteristic of Kepler is the separation between the adopted MoC from the structure of the workflow, which is built combining a set of polymorphic components, called *actors*. An actor implements a functionality of interest for a particular domain, and its behavior can change on the basis of the adopted MoC. The main idea is that a complex model can be built hierarchically combining different heterogenous models with different MoCs.

Table 3.1 Summary of Kepler characteristics.

Developers	NSF-funded Kepler/CORE UC Davis, UC Santa Barbara, and UC San Diego.
Parent project	Ptolemy II
Evaluated Release	1.0.0
Platforms	Windows, Linux, Mac OS X
Development Language	Java
Workflow Language	MoML (XML-based)
License	BSD License
Website	`http://kepler-project.org/`
Domain of application	Physics, Ecosystems, Bioinformatics

3.1.1 Modeling Paradigm

Kepler relies on an *actor-oriented modeling paradigm*: a workflow model is a composition of independent components called *actors* that represent operations or data sources. Actors communicate through interfaces called *ports*. Ports can be of input, output or mixed type and they are connected through *channels* that are directed from the output port of an actor to the input port of another actor. Each channel can transport a single stream of data. In addition to ports, actors can have a set of *parameters*, which configure and customize their behavior. Parameters can be statically specified during the workflow design, or dynamically determined through parameter ports.

Given the same actors configuration, different execution semantics can be specified through the choice of a particular *director*. A director defines how actors are executed and how they communicate with each other. The communication between actors is mediated by an object called *receiver*, which is provided by the director and determines if the communication is buffered or synchronous. The behavior of an actor adapts to the execution and communication semantics provided by the director: this feature is called *behavioral polymorphism*.

3.1.1.1 Director Responsibilities

The main activities performed by a director are:

1. It invokes the *pre-initialize* methods of all actors just before starting the workflow execution. This method is invoked only once per each execution (even if there are multiple runs), and prior to all other activities.
2. It *type-checks* all connections and ports.

3. It passes all the necessary information to the next actor by invoking its *initialize* method. This method is invoked at each actor run.
4. An actor run usually includes multiple *iterations*, each of which involves a call to *pre-fire, fire* and *post-fire* methods. The main actor functionalities are implemented in the fire method.

The sequence of activities performed by a director can be summarized as [16]:

$$\textit{pre-initialize} \rightarrow \textit{type-check} \rightarrow (\textit{initialize}, (\textit{pre-fire}, \textit{fire}^*, \textit{post-fire})^*)^*$$

where the asterisk denotes activities that may be executed zero or more times during each workflow execution.

3.1.1.2 Default execution semantics

The actor execution is centered on the notion of *data token* that represents a chunk of data. When an actor receives the necessary input tokens, it can be executed by the director the required number of times and it produces new tokens on its output port(s). As long as an actor receives tokens, it usually continues to be fired and to start executing.

3.1.1.3 Synchronous Data-Flow (SDF) Director

The Synchronous Data Flow (SDF) director is designed for *sequential* and *simple* workflows in which the order of invocation of the actors can be statically determined from the model before its execution. Components cannot change the routing of tokens during the execution. In order to use the SDF director some conditions have to be met:

1. The data consumption and production rate of each actor have to be constant and declared. If an actor reads one piece of data and computes a single result in output, it shall always read and output a single data token. This data rate cannot change during workflow execution.

    a. Workflows that require a dynamic control structure, such as the `BooleanSwitch` actor that sends an output on only one of its two output ports depending on a control value, cannot be used with an SDF director because the number of tokens produced in each output port changes at each execution.
    b. The SDF director assumes that each actor consumes and produces exactly one token per channel on each firing. Actors that do not follow this convention have to declare the number of tokens they produce or consume via appropriate parameters.

2. By default the SDF director requires that all actors inside the workflow are connected, otherwise it cannot determine the concurrency relations between disconnected parts.

Before starting workflow execution, the SDF director pre-calculates the order in which actors execute and how many times each actor needs to be fired to complete a single workflow iteration. The SDF director controls the number of times a workflow is iterated through the *iterations* parameter. By default, this parameter is set to 0, which means that the workflow will iterate forever. Values greater than zero specify the actual number of times the director should execute the workflow.

3.1.1.4 Dynamic Data-Flow (DDF) Director

The Dynamic Data-Flow (DDF) director, as the SDF director, executes a workflow in a *single execution thread*, meaning that tasks cannot be performed in parallel. However, unlike the SDF director, the DDF director does not pre-schedule workflow execution: it determines how to fire actors at run-time, and data production and consumption rates can change during workflow execution. The *iterations* parameter of the DDF director is used to specify the number of times the workflow is iterated. As for the SDF director, the default value is 0, which means that the workflow will iterate forever.

3.1.1.5 Process Network (PN)

The Process Network (PN) director is designed for managing workflows that require *parallel processing* on *distributed computer systems*. In a PN workflow each actor has an independent Java thread and the workflow is driven by data availability: actors can execute as soon as the necessary input tokens are available in their input ports. Produced tokens are passed to the connected actors through channels of unbounded capacity. These channels are persistent: they retain the received tokens until the actor is able to consume them. A PN workflow terminates when no other components can execute due to the lack of input data, unlike the previous two directors for which the number of desired workflow iterations has to be specified. If tokens are always generated and available to downstream actors, a workflow may not terminate. For instance, the workflow in Fig. 3.1 may not terminate, because the `Constant` actors `input_01`, `input_02` and `input_03` by default always produce an output when "asked" by the director. In order to obtain a finite execution of the workflow, the `firingCountLimit` parameter of each `Constant` actor need to be set to the number of values to be produced.

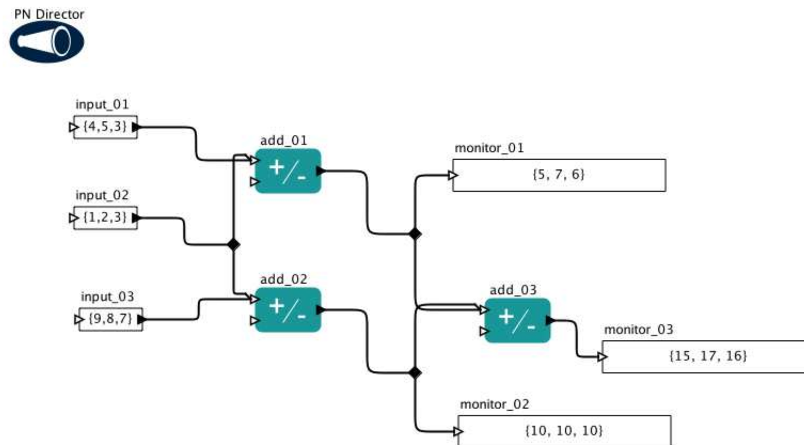

Fig. 3.1 Example of use of the PN director.

3.1.1.6 Continuous Time (CT)

The Continuous Time (CT) director introduces a notion of time for modeling workflows able to predict how a system evolves over time. Data tokens passed through the system have a timestamp that the director uses to determine the step and the stop condition. For instance a CT director can be used to define a system that predict the population growth over time. The prediction function is calculated within an interval which is determined by the step condition (e.g. every hour), until the stop condition is reached (e.g. for a duration of 10 years). The system is usually described in terms of start conditions and several equations, which are used to predict the state of the system at some specified time in the future.

3.1.1.7 Discrete Event (DE)

The Discrete Event (DE) director works with timestamps as the CT does. However, timestamps are not used to approximate functions and schedule executions, but to measure average wait times and occurrence rates. Actors send *event tokens*, which consist of tokens containing data and a timestamp; the director reads these tokens and places them on a global workflow timeline.

3.1.1.8 Choosing a Director

The choice of a particular director depends on the characteristics of the process to be modelled. Table 3.2 summarizes the criteria useful to perform such

a choice. The first parameter is the *dependence on time*, the SDF, DDF and PN directors do not need to schedule the actors actions at specific times. The CT director in the other hand is designed to describe dynamic systems that depend upon a continuously varying time parameter, or workflows that are used to perform numerical integration; while the DE director is designed to describe workflows in which events are executed at specified times or for scheduling simulations (i.e. a queuing system). The second criterion regards the ability to perform *distributed executions*: only the PN director provides this possibility. The final parameter is the need for a *constant data rate* of the consumed and produced tokens: only the SDF director has this characteristic.

Table 3.2 Criteria for choosing a particular director.

Director	Dependence on time	Distributed execution	Constant data rate
SDF	No	No	Yes
DDF	No	No	No
PN	No	Yes	No
CT	Yes	-	-
DF	Yes	-	-

Kepler supports hierarchical embedding of a workflow into another workflow, depending on the compatibility of their directors. The allowed combinations are determined by two factors: i) the requirements of the director relative to the actors under its control, e.g. the SDF director requires a constant and declared data consumption and production rate); ii) the semantics exported by the director to the actor within which it is placed, e.g. a `Nondeterministic Merge` actor cannot be used in the presence of an SDF director, as this director does not allow non-determinism during workflow execution.

3.1.2 Kepler Routing Constructs

In this section we describe the behavior of some Kepler routing actors in terms of CPN models. In these models, places are managed as queues, because in scientific WfMSs tokens are consumed in the order of their arrival. One way to obtain this behavior with standard CPNs is to use a single control token for each place that contains a list of data tokens: when a transition fires it consumes the data token at the head of the list in each place of the preset, and adds a data token to the tail of the list in each place of the postset. In graphical representations the head of the list is shown on the left side, while the tail of the list is shown on the the right side.

The **Relation** Operator

The `Relation` operator with one incoming channel and several outgoing channels, duplicates the data token received from the incoming channel to all the connected outgoing channels. The CPN in Fig. 3.2 shows the behaviour of this operator with two outgoing channels: all the values contained in *input* are duplicated in both out_1 and out_2. For instance, if *input* initially contains the list $[a, b, c]$, then at the end both out_1 and out_2 will contain the same list.

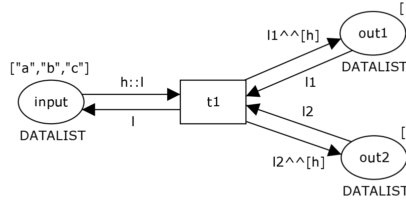

Fig. 3.2 A CPN representation of the Kepler `Relation` operator with one incoming channel containing the list $[a, b, c]$ and two empty outgoing channels.

The `Relation` operator with several incoming channels and one outgoing channel routes the tokens received by any of the incoming channels to the unique outgoing channel. This type of `Relation` operator can be used only with the SDF and DDF directors (i.e. the sequential ones). In this case, the operator first analyzes the content of the first connected channel and outputs all the values contained in it, then it proceeds with the second channel and so on. The behavior of the operator with two incoming channels is represented by the CPN in Fig. 3.3. At the beginning only transition t_1 is enabled, it transfers the values contained in in_1 to *out*. When no other tokens are available in in_1, a control token is placed in *second*, enabling transition t_2 which transfers the values contained in in_2 to *out*. For instance, if in_1 initially contains the list $[a, b]$, and out_2 contains the list $[x, y]$, at the end *out* will contain the list $[a, b, x, y]$. If at the beginning no tokens are available in in_1, transition t_1 will immediately put a control token in *second* enabling transition t_2.

The **Nondeterministic Merge** Actor

The `Relation` operator with several incoming channels and one outgoing channel cannot be used with the PN Director. The merge of multiple channels with this director can be obtained using the `Nondeterministic Merge` actor which reads the tokens contained in the incoming channels and concatenates them in an arbitrary order. Given the same content of the incoming channels, each execution of this actor potentially produces a different output.

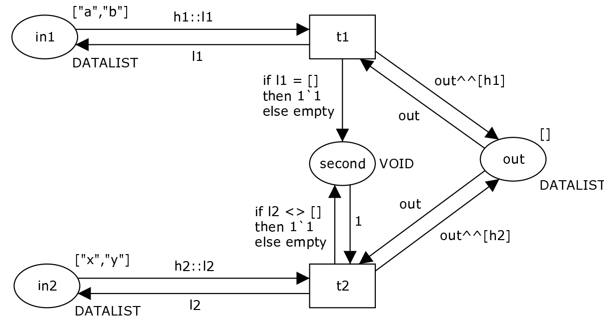

Fig. 3.3 A CPN representation of the Kepler `Relation` operator two incoming channels and one outgoing channel.

The behavior of a `Nondeterministic Merge` with two incoming channels is represented by the CPN in Fig. 3.4. Initially both transitions t_1 and t_2 are enabled. Supposing that in_1 contains the list $[a]$ and in_2 contains the list $[b]$, depending on which transition is executed first, the final output in place *out* can be $[a, b]$ or $[b, a]$.

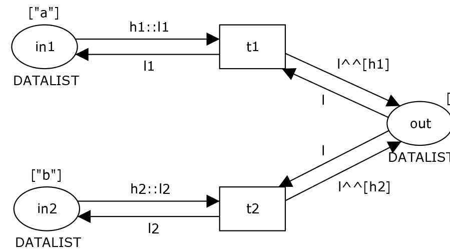

Fig. 3.4 A CPN representation of the Kepler `Nondeterministic Merge` actor.

The `Repeat` Actor

The `Repeat` actor reads a sequence of data tokens and outputs it a specific number of times. The size of the block can be specified through the `blockSize` parameter, while the number of copies is determined by the `numberOfTimes` parameter. These two parameters can be fixed at design-time or determined at run-time using a parameter port. The CPN in Fig. 3.5 shows the behavior of the actor with `blockSize` equals to one (the *repeat* transition reads one value at a time from *in*) and the number of repetitions is fixed at design-time to 3 (the value read by *repeat* is concatenated three

times in output). If in initially contains the list $[a, b]$, place out will ultimately contain the list $[a, a, a, b, b, b]$ and in will then be empty.

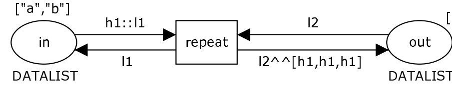

Fig. 3.5 A CPN representation of the `Repeat` actor with parameters `blockSize` = 1 and `numberOfCopies` = 3 fixed at design-time.

The CPN in Fig. 3.6 shows the behavior of the actor when `blockSize` equals one and the number of repetitions is determined at run-time. Place in_2 contains the number of repetitions to be performed and is intended as a place of capacity one, not as a queue: a transition can change the contained value only by first consuming the existing token and then inserting another token with the new value. Place in_1 contains the data values to be duplicated, while places p_3 and p_4 contain control tokens that determine the enablement of transitions t_1 and t_2, respectively. Transition t_1 reads the next data value to be duplicated from in_1 and puts that value in p_1. Similarly transition t_2 reads the number of copies to be performed from in_2 and puts that value in p_2. Transition t_4 executes n times, at each execution it places the data value held in p_1 into the out place and decrements the value n contained in p_2. After n executions of t_4, the value in p_2 equals zero and transition t_3 is enabled. Transition t_3 removes the data value from p_1 and enables again t_1 and t_2 by placing a control token in p_3 and p_4, respectively. If in_1 initially contains the lists $[a, b]$ and in_2 the value 3, at the end out will contain the list $[a, a, a, b, b, b]$.

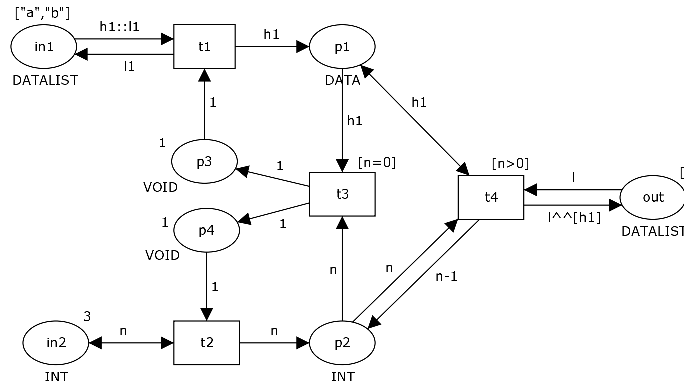

Fig. 3.6 A CPN representation of the `Repeat` actor with the parameter `numberOfTimes` determined at run-time by the value in place in_2.

The `Switch` Actor

The `Switch` actor has two input ports, called `input` and `control`, and one output multiport, called `output`. This actor routes the data received through the `input` port to one of the several channels connected to the output multiport, on the basis of the value received on the `control` port. The `control` port is used to select the desired output channel. One or more channels can be connected to the output multiport, if the value received on the `control` port is greater than the number of connected channels, the data value received in `input` will be lost.

The behavior of the `Switch` actor with two output ports is represented by the CPN in Fig. 3.7: the data value read from place *input* will be transferred to place out_1 if the control value is 1, to place out_2 if the control value is 2, or lost if the control value is different from 1 or 2 (i.e. inserted into place *lost*). If place *control* initially contains the list $[1, 2, 1, 3]$ and place *input* contains the list $[a, b, c, d]$, at the end out_1 will contain the list $[a, c]$, out_2 will contain the list $[b]$, while the value d will be lost. The generalization of this structure to the case of a generic number n of output channels is trivial.

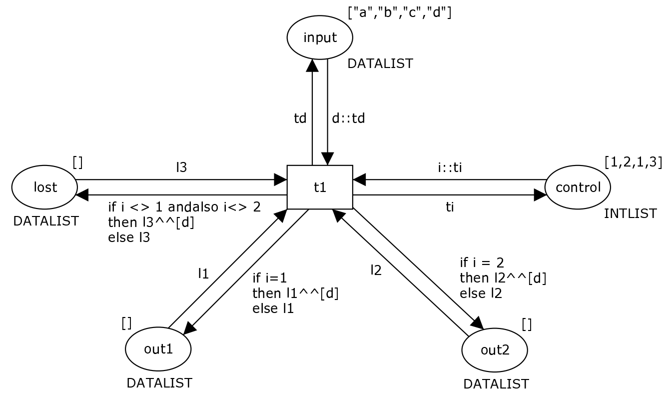

Fig. 3.7 A CPN representation of a `Switch` actor with two output channels.

The `Boolean Switch` actor has a very similar behavior, except that it consumes a boolean control value and has exactly two output ports, called `trueOutput` and `falseOutput`. The CPN implementation of this actor can be obtained from the one in Fig. 3.7 by replacing the arc conditions $i = 1$ and $i = 2$ with $i =$ *true* and $i =$ *false*, respectively, and dropping the *lost* place.

The `Select` Actor

The `Select` actor has an input multiport called `input`, an input port called `control`, and an output port called `output`. It selects a piece of data from one of the incoming channels connected to the `input` port, on the basis of the value received on the `control` port. The `control` port is used to select the desired input channel.

The behavior of a `Select` actor with two input channels can be captured by the CPN in Fig. 3.8. If the value read from place *control* is 1, transition t_1 puts a control token in p_1, enabling transition t_2 that transfers the value in $input_1$ to place *out*. If on the other hand the control value is 2, transition t_1 puts a control token in p_2 and transition t_3 is enabled, determining the transfer of the value in $input_2$ to *out*. Place *start* constraints the execution of transition t_1 to the completion of t_2 or t_3: initially *start* contains a token and t_1 can fire consuming that token, then it can execute again only when t_2 or t_3 produces another token in the same place. For instance, if *control* contains the list $[1, 1, 2, 3]$, $input_1$ contains the list $[a, b]$ and $input_2$ contains the list $[x, y]$, at the end *out* will contain the list $[a, b, x]$ and the control value 3 has no effect. The generalization of this structure to a generic number n of input channels is straightforward.

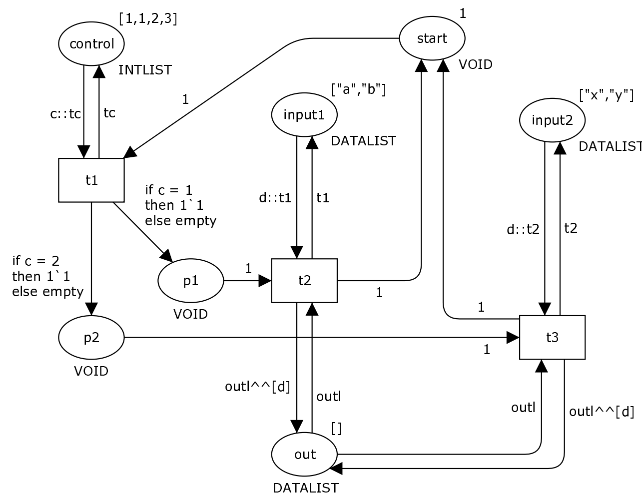

Fig. 3.8 A CPN representation of the `Select` actor with two input channels.

The `DDF Boolean Select` actor has a very similar behavior, except that it has exactly two input ports for data and a boolean control value. Its CPN implementation can be obtained from the one in Fig. 3.8 by substituting the condition $c = 1$ and $c = 2$, with $c = true$ and $c = false$, respectively.

The **SyncOnTerminator** Actor

The `SyncOnTerminator` actor receives as input a stream of data. As soon as a new piece of data is received, it is passed to the output port except for a particular piece of data which is called *terminator*, that is produced as output only when it has been received a certain number of times. Once the terminator has been produced as output, no other data will be outputted.

The CPN in Fig. 3.9 shows the behavior of a `SyncOnTerminator` actor with a terminator value equal to "term" and the number of required terminators equal to 2. If the data value consumed by t_1 contains a value different from "term", this value is directly copied in *out*, otherwise the counter contained in p_1 is incremented. The value in p_1 maintains the number of times the terminator value has been encountered. When this number equals the predefined limit, the value "term" is produced in the output. Once the value "term" has been outputted, no other data will be consumed. If *source* initially contains the list $[a, b, term, c, term, f]$, at the end *out* will contain the list $[a, b, c, term]$.

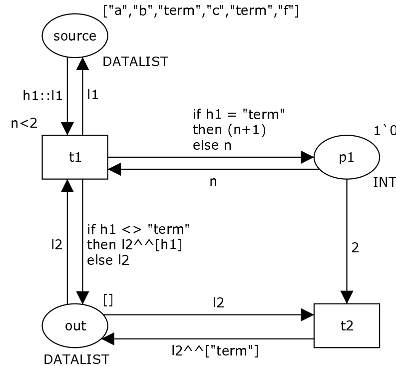

Fig. 3.9 CPN representation of the `SyncOnTerminator` actor.

The **Ramp** Actor

The `Ramp` actor works like a for loop: it allows one to execute a task a specified number of times. The number of iterations is controlled through the parameters `firingCountLimit`, the number of times the actor should iterate, `init`, the initial value of the counter, and `step`, the increment to apply to the counter at each iteration.

The behavior of the `Ramp` actor is exemplified by the CPN in Fig. 3.10: places *limit*, *index* and *step* contain the values of the parameters `firingCo-untLimit`, `index` and `step`, respectively. If the counter value read from

index is smaller than the limit value, transition t_1 puts the counter value in p_1 and another iteration of the transition *fun* is performed, then t_2 increments the counter value with the value of *step* and places this updated value in *index*. Otherwise, if the counter is greater than or equal to the limit, a control token is produced in *out*.

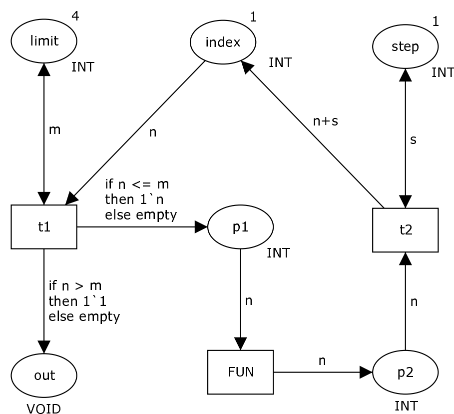

Fig. 3.10 A CPN representation of the `Ramp` actor.

3.2 Taverna

Taverna [18] is a free software tool for designing and executing scientific workflows, created by the ^{my}Grid project. It was originally developed in the biological domain and its primary aim was allowing the construction of a scientific workflow from numerous remote web services. Therefore, a significant effort was put towards collecting and organizing these web services into a reusable set of components. Nowadays, Taverna is used in many domains, such as bioinformatics, chemioinformatics, astronomy, social sciences and music. Moreover, it is directly integrated with the ^{my}Experiment initiative[1] whose aim is to collect, find, use and share scientific workflows.

A workflow in Taverna is specified through the Scufl (Simplified Conceptual Unified Flow Language) language. Scufl is essentially a data-flow language that allows the definition of graphs of data interactions among different local and remote services provided by external applications.

[1] `http://myexperiment.org`

Table 3.3 Summary of Taverna characteristics

Developers	^{my}Grid Team University of Manchester, UK
Parent project	^{my}Grid
Evaluated Release	2.1
Platforms	Windows, Linux, Mac OS X
Development Language	Java
Workflow Language	Scufl
License	LGPL
Website	`http://www.taverna.org.uk`
Application domains	Biology, Bioinformatics, Chemioinformatics Astronomy, Social Sciences and Music

3.2.1 Modeling Paradigm

A workflow in Taverna contains a set of individual steps called *processors*. Each processor receives data on its input ports, performs some operations on these data and produces data on its output ports. Processors are connected with each other through *data links* that join the output port of one processor to the input port of another.

Beside data links, some additional ordering constraints among processors that do not require a flow of data can be defined. These links are called *coordination links*: they specify precedence conditions among processors, such as a processor can execute only when another one has completed.

A workflow can have zero or more formal inputs that are represented as *sources*. Similarly, the global outputs of a workflow are represented as *sinks*. The execution of a workflow starts from the sources and finishes when all sinks have either produced their output or failed.

Contrary to Kepler, Taverna provides only one model of computation. Each processor in Taverna is considered as a function and two processors connected with a data link represent a function composition, if processor A corresponds to function f_A and processor B to function f_B, then the result of B will be $f_B(f_A(x))$ where x is the input provided to A.

Taverna provides a configurable mechanism for fault management: if a service fails, for instance because the underlying machine is down, it will be initially invoked again a certain number of times, after which an alternative service will be invoked. Users can explicitly specify a list of alternatives for each processor. A Scufl processor definition includes: the implementing services, the number of retries, the time between retries, and optionally an alternative service to be used if the service of first choice fails.

3.2.2 Taverna Routing Constructs

Taverna offers several computational processors in many scientific fields, however it does not provide routing constructs, except for a merge operator. This choice perhaps comes from the simplicity with which routing constructs (e.g. if-then-else, or switch-case statements) can be defined directly in the workflow editor using custom Beanshell scripts. In this section we describe the behavior of the merge construct using CPNs. The same assumptions made in Sec. 3.1.2 about the management of places as queues are also used here.

The `Merge` Operator

The `Merge` operator routes into a unique channel the data values received from its connected incoming channels. Its CPN representation is the same as that of the Kepler `Relation` operator with multiple incoming channels and one outgoing channel. An example of this operator with two incoming channels is depicted in Fig. 3.3.

3.3 Triana

Triana [19] was originally developed as a visual workflow-based problem-solving environment for the gravitation wave detection project GEO600 and used as a rapid analysis tool for wave data. Triana workflows were initially built from Java tools and executed on local machines or remote ones using Java RMI. More recently, Triana components have evolved into flexible proxies that can represent a number of local and distributed primitives. For instance, they can represent a Java object, a legacy system, a web-service, a web-service resource framework, a Grid job, a local file or a remote file.

Table 3.4 Summary of Triana characteristics

Developers	Cardiff University
Parent project	–
Evaluated Release	4.0
Platforms	Windows, Linux, Mac OS X
Development Language	Java
License	Apache open source license version 2
Website	`http://www.trianacode.org/`
Application domains	Bioinformatics

3.3.1 Modeling Paradigm

Triana workflows are composed of components, which accept, process and output data. A *component* is a unit of execution with a defined port interface. It has several properties, such as an identifier, input and output ports, a number of optional parameters, and a proxy/reference to the part of the component that will actually do the work. The component specification is encoded in XML with a format similar to WSDL, while the source code of each component can be viewed, modified and recompiled directly within the environment. A component may be implemented as a Java method call on a local object or as an interface to a *distributed component* which can be:

- a *grid-oriented component*: an application that is executed on the Grid via a Grid resource manager (e.g. GRAM, GRMS, Condor/G).
- a *service-oriented component*: an application that can be invoked via a network interface, such as a web service or a JXTA service.

Grid-oriented and service-oriented components can be used together inside the same workflow.

As in the other scientific WfMSs, in Triana dependencies among tasks are mainly data dependencies. However, Triana also supports the representation of control-flow dependencies through special trigger messages.

3.3.2 Triana Routing Constructs

As done for the other two systems, in this section we describe the behavior of the Triana routing constructs using CPNs. The same assumptions made in Sec. 3.1.2 about the management of places are also valid here.

The `Loop` Component

The `Loop` component allows one to iterate the execution of a task or a sequence of tasks. It initially waits for data in its first input port (pre-loop state). As a piece of data is received on this port the defined exit condition of the loop is evaluated. If this condition is met, the received data value is redirected to the first output port. Otherwise, the received data value is redirected to the second output port and the loop enters an in-loop state. An in-loop state is the same as the pre-loop state, except that data are awaited for on the second input port. During an in-loop state, when a data value is received in the second input port, the exit condition is evaluated and if it is met, the received data value is redirected to the first output port, otherwise it is redirected to the second input port. On the basis of the defined exit condition, two kinds of loops can be defined:

1. *Count Loop*: which iterates a specified number of times (similar to a `for` statement in an imperative programming language).
2. *While Loop*: which iterates as long as a particular condition is met.

The behavior of the `Loop` component with a while loop condition is exemplified by the CPN in Fig. 3.11. The *in* place contains the data values, while the *par* place contains the parameter for the loop test. If the current data value is smaller than the value in *par*, a loop iteration is performed and the new computed value will replace the old one in *in*. Otherwise, if the data value is equal to or greater than the *par* value, it is redirected to place *out*. If place *in* initially contains the list $[8, 3, 5]$ and *par* is equal to 20, at the end place *out* will contain the list $[64, 81, 25]$, as transition *fun* computes the double of the received value, which is stored back in *in*.

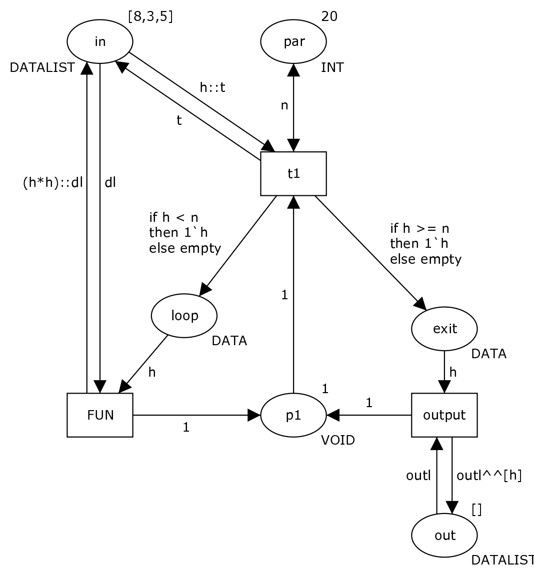

Fig. 3.11 The CPN representation of the `Loop` component of Triana.

The `If` Component

The `If` component has two input ports: the integer value received from the first port is used for evaluating a condition, based on which the data value received from the second port is redirected to one of the two output ports. In particular, if the value received from the first input port is less than the specified `If` parameter, the data value received from the second input port is produced on the first output port, otherwise the data value is produced on

the second output port. The `If` condition can only be a comparison between integer values.

The behavior of the `If` component is captured in the CPN of Fig. 3.12: place *in* contains the integer values used to evaluate the condition, while *par* contains the component parameter used in the condition, and *data* contains the data values. Transition t_1 compares the current *in* value with the *par* one. If the *in* value is less than the parameter, the data value is outputted in out_1, otherwise it is outputted in out_2. If *in* contains the list $[9, 3, 7]$, *data* contains the list $[a, b, c]$ and *par* contains the value 7, at the end place out_1 will contain the list $[b]$, while out_2 will contain the list $[a, c]$.

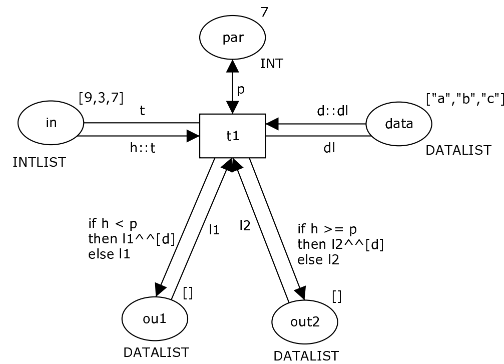

Fig. 3.12 The CPN representation of the `If` component of Triana.

The `Duplicator` Component

The `Duplicator` component duplicates the value received from its input port to each of its output ports. The CPN representation of this component is the same as that of the Kepler `Relation` operator with one incoming channel and multiple outgoing channels depicted in Fig. 3.2.

The `Merge` Component

The `Merge` component waits to receive a data value from at least one of its input ports; the received value is immediately redirected to its output port. A parameter can be specified to determine the order in which the received values are produced for the output when an input is available in more than one input port at the same time. The CPN representation of this component when the order parameter is not specified, is the same as that of the Kepler `Nondeterministic Merge` actor depicted in Fig. 4.13. The CPN repre-

sentation of this component when an order parameter is specified, is the same as that of the Kepler `Relation` operator with multiple incoming channels and one outgoing channel in Fig. 3.3.

Chapter 4
Pattern-Based Evaluation of Scientific Workflow Management Systems

In this chapter we analyze the three scientific WfMSs introduced in Sec. 3 in terms of workflow patterns support. For the evaluation we consider only the standard constructs provided by the default distribution and we do not refer to any ad-hoc extension or third-party additional library. In the following the term *task* is used to uniformly denote a Kepler actor, a Taverna processor or a Triana component.

4.1 Workflow Control-Flow Patterns

Workflow Control-Flow Patterns (WCPs) [20] describe a set of recurring features that are commonly offered by WfMSs for defining the flow of control among various tasks.

In scientific WfMSs dependencies among tasks are data dependencies: a relation $A \dashrightarrow B$ between A and B means that B may need some data produced by A to complete its execution. A control-flow dependency $A \rightarrow B$ on the other hand is concerned with the execution termination of tasks: a relation between A and B means that B can start to execute only when A terminates. A control-flow dependency can always be represented in terms of a data dependency by assuming that A produces an output only just before completing; while the opposite does not hold. During the following analysis, we always assume that the output is produced only at task completion.

4.1.1 Basic Control-Flow Patterns

Basic Control-Flow patterns capture elementary aspects of process control.

WCP-01 **Sequence**

Description – *A task in a process is enabled after the completion of a preceding task in the same process [20].*

Realization – Given the assumption stated above about the possibility to mimic a control-flow dependency through a data-flow dependency, a sequence relation between two tasks A and B can be obtained in all the three systems by defining a data-flow dependency between A and B.

Fig. 4.1 depicts an example of the Sequence pattern in Kepler: the actor `monitor` can execute only when the previous actor `add` completes and produces an output value.

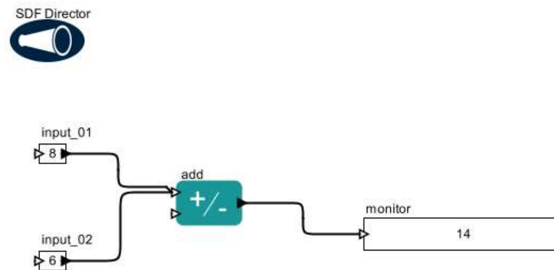

Fig. 4.1 Example of WCP-01 Sequence in Kepler.

Taverna allows one to define two kinds of dependencies between tasks: data links, which represents data dependencies, and coordination links, which represents control-flow dependencies of kind *run-after*. The example in Fig. 4.2 depicts a set of tasks among which data links (black arrows) and coordination links (gray lines) are defined: even if, the various `Create_Lots_Of_Strings` processors can be executed in parallel, the coordination links defined among them ensure that they are sequentially executed.

Finally, Fig. 4.3 depicts an example of the Sequence pattern in Triana: the `Sqrt` component can execute only when `Random` has completed; in a similar way `DoubleView` can start only when `Sqrt` completes and outputs the computed value.

WCP-02 **Parallel Split**

Description – *The divergence of a branch into two or more parallel branches each of which execute concurrently [20].*

Realization – The Kepler `Relation` operator can be used to broadcast the incoming data to different channels and thus to activate different subsequent

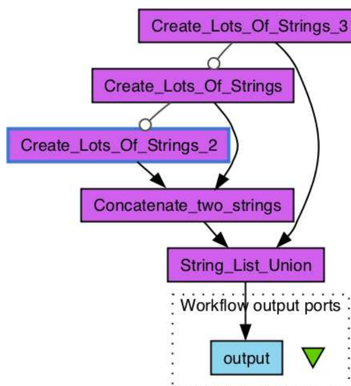

Fig. 4.2 Example of WCP-01 Sequence in Taverna with some data and control-flow dependencies defined among tasks.

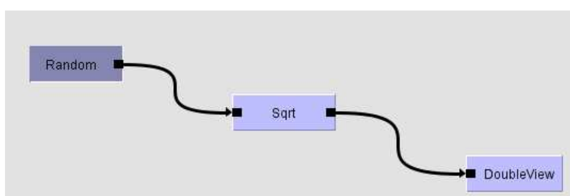

Fig. 4.3 Example of WCP-01 Sequence in Triana.

actors. In the example of Fig. 4.4 the value contained in `input` is passed to all the subsequent tasks `task_01`, `task_02` and `task_03`, and these may execute in parallel. In this case data tokens are used also as a control token.

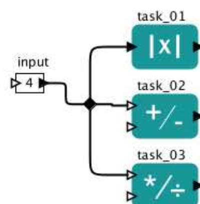

Fig. 4.4 Example of WCP-02 Parallel Split implemented in Kepler with the *Relation* operator. The constant value contained in *input* is passed to all the subsequent tasks.

In Taverna the output port of a processor A can be connected with the input port of more than one processor $B_1, \ldots, B_n$, in this way after the completion of A, all the other processors $B_1, \ldots, B_n$ can start to execute in parallel. In the example of Fig. 4.5 the output produced by

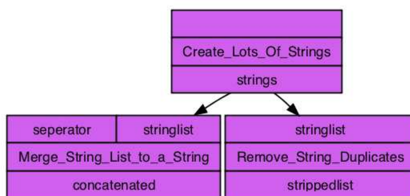

Fig. 4.5 Example of WCP-02 Parallel Split implemented in Taverna by connecting the same output port of a processor to the input ports of several other processors.

`Create_Lots_Of_Strings` is redirected to all the subsequent activities that can start executing in parallel.

In Triana a Parallel Split can be obtained by simply increasing the number of output ports of a component. In this way a copy of the produced data is generated in each output port of the task. Fig. 4.6 depicts an example of a Parallel Split realized by augmenting the number of output ports of *ConstGen*: the produced data value is redirected to all connected components that are enabled in parallel. Moreover, in Triana a Parallel Split can also

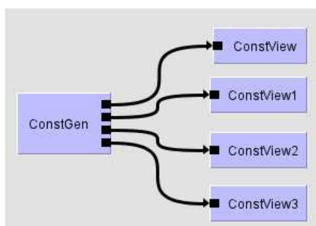

Fig. 4.6 Example of WCP-02 Parallel Split implemented in Triana augmenting the number of output ports of a component.

be obtained through the `Duplicator` component which redirects the data value received in input to each of its output ports. In Fig. 4.7 the data value received in the input port of `Duplicator` is redirected to all its output ports (there can be more than two) enabling all the subsequent `ConstView` components in parallel.

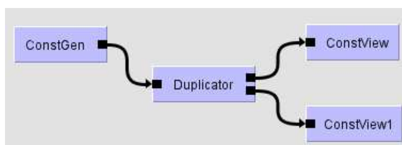

Fig. 4.7 Example of WCP-02 Parallel Split implemented in Triana using the `Duplicator` component.

WCP-03 **Synchronization**

Description – *The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled [20].*

Realization – In the three considered systems, a task can have one or more input ports and can execute only when a data token is available in each channel connected to its input ports. Therefore, any task having more than one input port can be used for synchronizing the execution of the previous tasks.

WCP-04 **Exclusive Choice**

Description – *The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to precisely one of the outgoing branches based on a mechanism that can select one of the outgoing branches [20].*

Realization – The Kepler `Switch` actor introduced in Sec. 3.1.2, routes the data received on its `input` port to only one of the connected output channels, on the basis of the control value received in the `control` input port. The channel number contained in the control value can be determined on the basis of a logical expression. Fig. 4.8 depicts an example of an Exclusive Choice implemented with the `Switch` actor. The only drawback is the possibility to

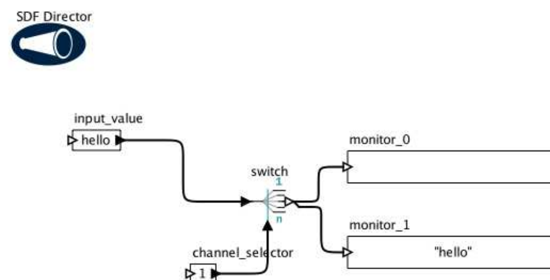

Fig. 4.8 Example of WCP-04 Exclusive Choice implemented in Kepler with the `Switch` actor.

lose a data token when the control value is out of range, but we can assume that the conditions are properly implemented to produce a valid output for the selection. An Exclusive Choice with only two options can also be modeled in Kepler through the `Boolean Switch` actor. As explained in Sec. 3.1.2, this actor outputs the received data value on one of its two output ports on

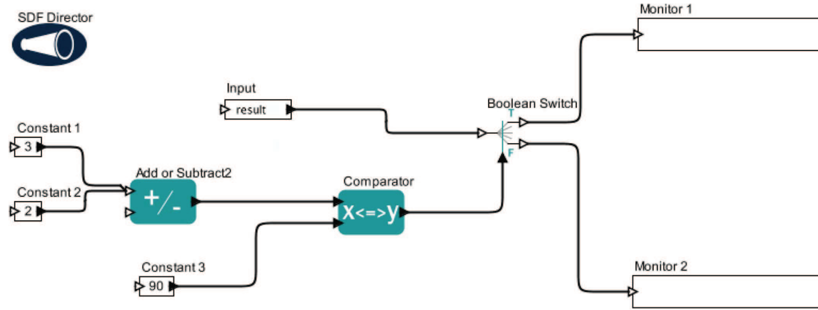

Fig. 4.9 Example of WCP-04 Exclusive Choice implemented in Kepler using the `Boolean Switch` actor.

the basis of a boolean control value. Fig. 4.9 depicts an example of Exclusive Choice implemented with the `Boolean Switch` actor, the data received from `Input` is redirected to `Monitor 1` or `Monitor 2` on the basis of the control value produced by `Comparator`.

As regards to Taverna, none of the processors available in the default catalogue is able to redirect the thread of control to only one of the subsequent branches on the basis of the evaluation of a particular condition. However, Taverna allows one to directly define and execute custom Beanshell scripts inside the design environment. In the example of Fig. 4.10 the

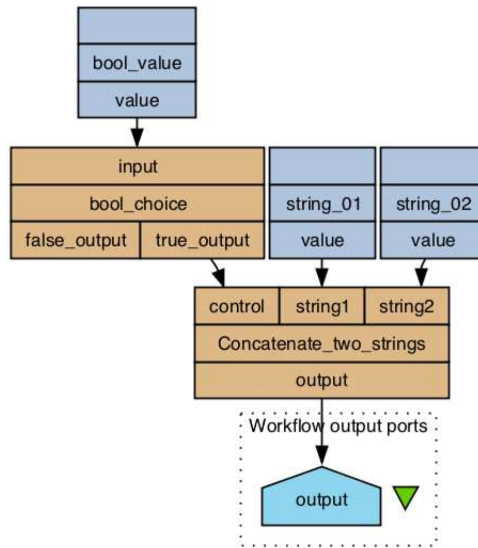

Fig. 4.10 Example of WCP-04 Exclusive Choice implemented in Taverna creating an ad-hoc processor `bool_choice`.

bool_choice processor is a custom defined Beanshell script: it reads a value through its input port and based on that value produces an output either in the true_output or the false_output port. Moreover, an additional input port (control) has been added to the subsequent processor for accepting the control value. This value is used only for synchronization purposes and is not considered during the computation. Notice that this additional port is necessary, as we cannot create a control-flow dependency between an output port (i.e. the true_output or false_output port of bool_choice) and a processor, but only between two processors. Anyway, the pattern cannot be considered supported in Taverna, because we have to create from scratch an additional ad-hoc processor.

In Triana an Exclusive Choice can be implemented using the If component which has been explained in detail in Sec. 3.3.2. Fig. 4.11 depicts an example of an Exclusive Choice: the data value provided by ConstGen is redirected by the If component to only one of the subsequent Multiply components on the basis of a condition evaluated on the ConstGen1 value. An Exclusive Choice with more than two alternatives can be obtained by concatenating several If components (i.e. the false output port of the If component representing the first alternative is connected with another If component representing the second alternative, and so on).

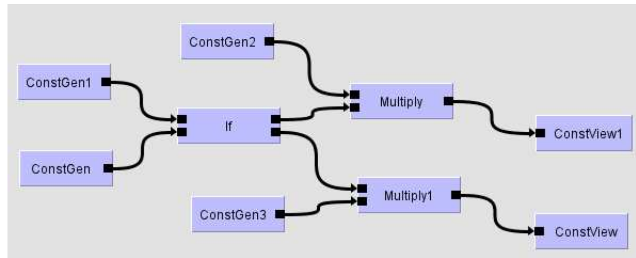

Fig. 4.11 Example of WCP-04 Exclusive Choice implemented in Triana using the If component.

WCP-05 **Simple Merge**

Description – *The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in the thread of control being passed to the subsequent branch [20].*

Realization – The Kepler Relation operator can be used with the SDF and DDF director to combine the output produced by several previous actors into a unique channel, as illustrated in the example of Fig. 4.12. Similarly, the Nondeterministic Merge actor can be used with the PN director

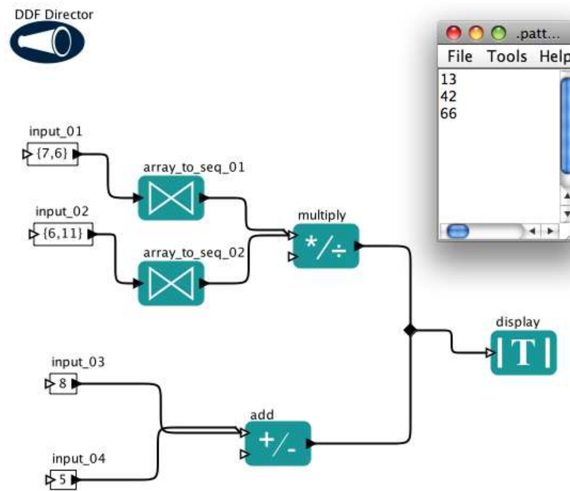

Fig. 4.12 Example of WCP-05 Simple Merge implemented in Kepler using the `Relation` operator.

to non-deterministically merge the output produced by various actors, as illustrated in the example of Fig. 4.13.

In Taverna the `Merge` operator puts into a unique channel the outputs produced by all the incoming processors. In the example of Fig. 4.14 the output produced by the two processors `Merge_String_List_to_a_String` and `Remove_String_Duplicates` are combined into a unique channel. Whenever a data value is produced by any of the two processors, a new instance of the subsequent `output` processor is generated.

In Triana this pattern can be implemented using the `Merge` component explained in Sec. 3.3.2. In the example of Fig. 4.15 the data values produced by the `If` and `StringGen1` components are redirected into a unique channel by the `Merge` component, so that whenever a new data value is produced by one of these two components, the `Merge` passes this value to `StringView`.

4.1.2 Advanced Branching and Synchronization Patterns

Advanced Branching and Synchronization patterns characterize more complex branching and merging concepts which arise in process modeling.

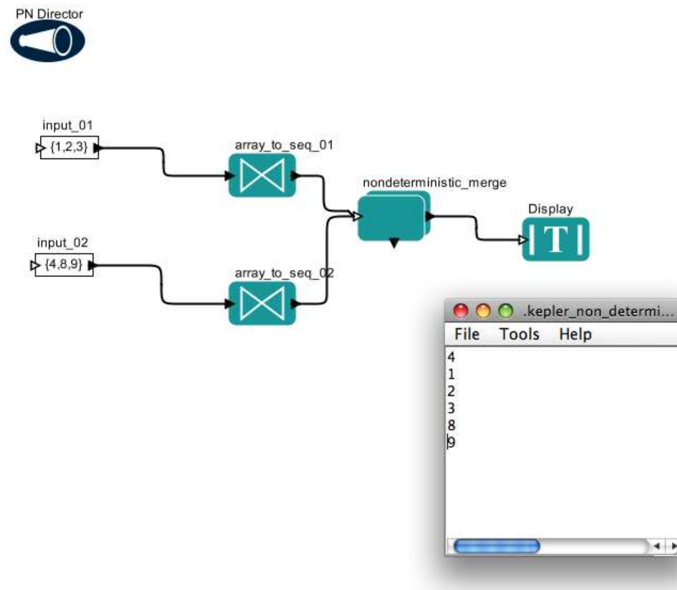

Fig. 4.13 Example of WCP-05 Simple Merge implemented in Kepler using the `Nondeterministic Merge` actor: different executions of the same workflow can produce different final outputs.

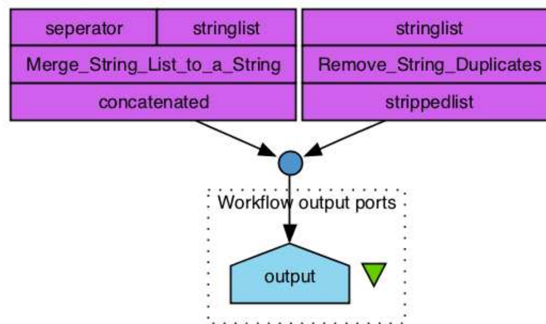

Fig. 4.14 Example of WCP-05 Simple Merge implemented in Taverna using the `Merge` operator which is depicted as a blue circle.

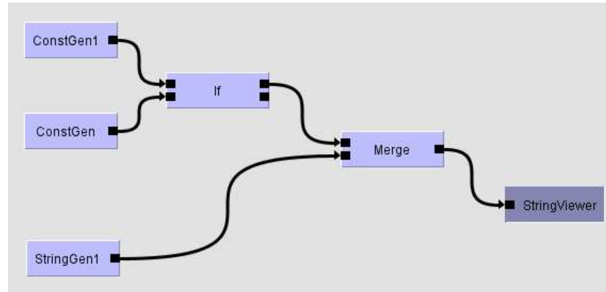

Fig. 4.15 Example of WCP-05 Simple Merge implemented in Triana using the `Merge` component.

WCP-06 Multi Choice

Description – *The divergence of a branch into two or more branches such that when the incoming branch is enabled, the thread of control is immediately passed to one or more of the outgoing branches based on a mechanism that selects one or more outgoing branches [20].*

Realization – In Kepler a Multi-Choice can be implemented using the `Relation` operator that broadcasts the incoming data to all the connected output channels and a `Boolean Switch` actor for each of these channels. The behavior of both components has been described in depth in Sec. 3.1.2. In

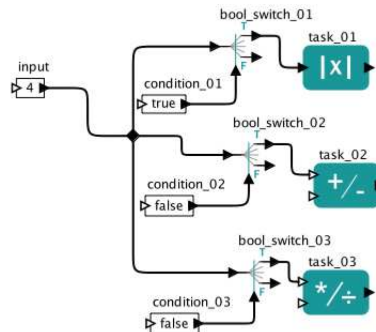

Fig. 4.16 An Example of WCP-06 Multi Choice implemented in Kepler using the `Relation` operator and several `Boolean Switch` actors.

the example of Fig. 4.16, one or more of the three tasks `task_01`, `task_02` and `task_03` can be enabled in parallel on the basis of the evaluation of the conditions `condition_01`, `condition_02` and `condition_03`, respectively. If a condition evaluates to true, the corresponding task is executed; otherwise a skip is performed (the incoming data is directly passed to the

subsequent activity through the `false` port). In the example of Fig. 4.16 the conditions are simply constant values, but they can be the result of any complex expression. In order to ensure the presence of a default branch, these expressions have to be defined so that one of them is valid when all the other expressions are false. Even if the model represented in Fig. 4.16 can be considered a solution for obtaining a Multi Choice in Kepler, in [20] the authors explicitly stated that this work-around is not considered to constitute direct support for the pattern.

In Taverna the implementation of this pattern requires the availability of the `bool_choice` Beanshell script described for the Exclusive Choice pattern (WCP-04). As a consequence, Multi Choice is not supported in Taverna.

In Triana a Multi Choice can be implemented by broadcasting a particular value (WCP-02 Parallel Split) to many `If` components. Fig. 4.17 depicts an example of a Multi Choice: one or more `Multiply` components can be enabled on the basis of the evaluation of the corresponding `If` condition. However, this solution is not considered as direct support for the pattern. Moreover, as for Kepler, the presence of a default branch has to be guaranteed by a proper definition of the conditions associated with the various `If` components.

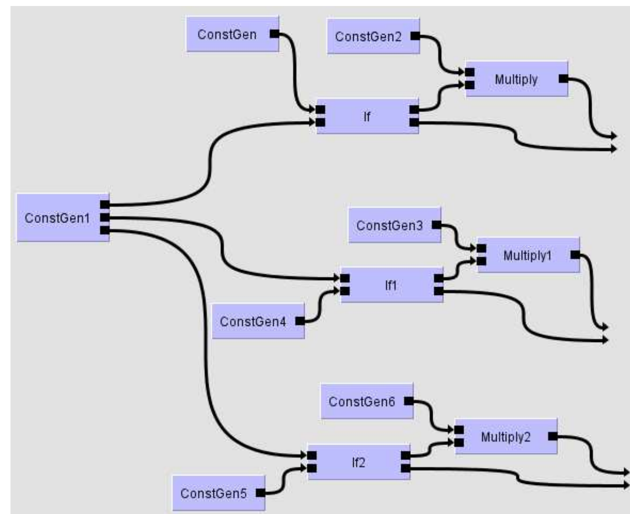

Fig. 4.17 An example of WCP-06 Multi Choice implemented in Triana augmenting the output ports of the `ConstGen1` component and using several `If` components.

WCP-07 **Structured Synchronizing Merge**

Description – *The convergence of two or more branches (which diverged earlier in the process at a uniquely identifiable point) into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. The Structured Synchronizing Merge occurs in a structured context, i.e. there must be a single Multi-Choice construct earlier in the process model with which the Structured Synchronizing Merge is associated and it must merge all of the branches emanating from the Multi-Choice. These branches must either flow from the Structured Synchronizing Merge without any splits or joins or they must be structured in form (i.e. balanced splits and joins) [20].*

Realization – A Structured Synchronizing Merge requires that the involved branches have diverged earlier in the process at a uniquely identifiable point. In particular, this point can be a Multi Choice (WCP-06). According to the Multi Choice implementation previously given, in Kepler a Structured Synchronizing Merge can be obtained by combining each pair of branches generated by the same `Boolean Switch` actor through a `Relation` operator (or `Nondeterministic Merge` actor), or a `DDF Boolean Select` with the same condition of the corresponding `Boolean Switch`. Then the output channels of these `Relation` operators, or `DDF Boolean Selector` actors, are synchronized through another actor with multiple input ports. Fig. 4.18 illustrates a possible implementation of this pattern: one, two or all three branches outgoing from the initial `Relation` operator can be active, depending on the evaluation of the condition associated with the various `Boolean Switch` actors. Each `DDF Boolean Select` actor is connected with the same condition as its corresponding `Boolean Switch` actor, in this way it awaits the completion of the branch if this is active, or executes immediately in the other case. The same behavior can be obtained by substituting the `DDF Boolean Select` actors with `Relation` operators or `Nondeterministic Merge` actors which do not require a connection with the enablement conditions.

The same implementation can be also obtained in Triana using a `Merge` component in place of each `Relation` operator. Fig. 4.19 depicts an example of the implementation of this pattern: the `Add` component is enabled only when the active `Multiply` components have completed.

Taverna does not support this pattern, because it has no processors able to selectively redirect a value to one channel.

WCP-08 **Multi Merge**

Description – *The convergence of two or more branches into a single subsequent branch such that each enablement of an incoming branch results in*

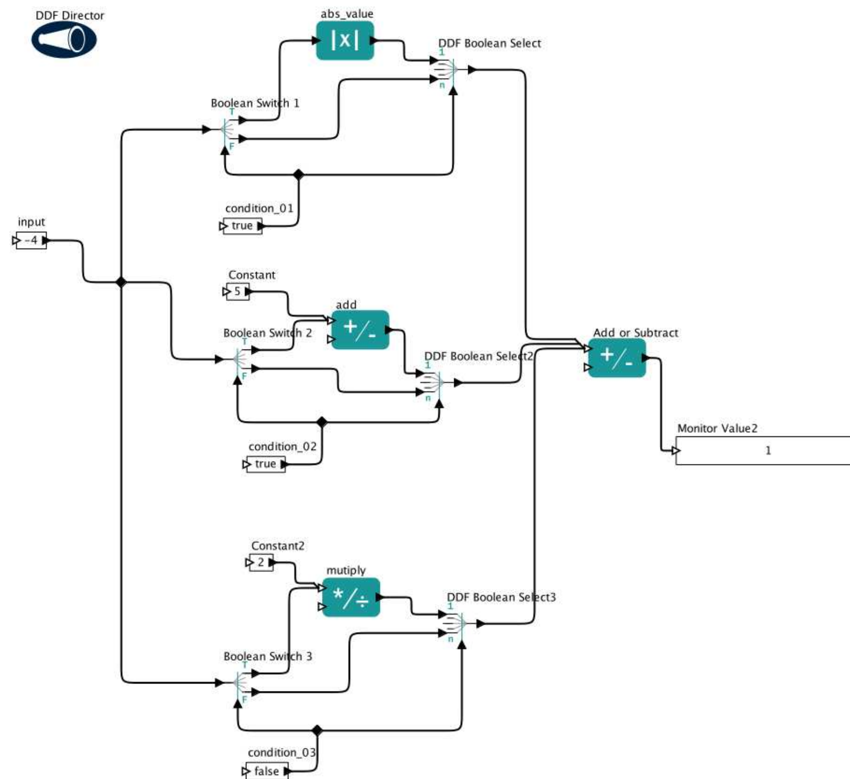

Fig. 4.18 An Example of WCP-07 Structured Synchronizing Merge in Kepler that synchronizes one, two or all three branches depending on which of them are active.

the thread of control being passed to the subsequent branch [20].

Realization – The difference between this pattern and its safe version WCP-05 simple Merge is that in the former it is possible for more than one incoming branch to be active simultaneously, while for the latter only one incoming branch can be active, i.e. considering its CPN implementation, the place p in which the merge is performed must be safe and cannot contain more than one token at a time. Therefore, the distinction between these two patterns concerns the context in which they are used. For all the three considered systems, the implementations given for WCP-05 can be also used in an unsafe context, producing the required behavior. Therefore, WCP-08 is supported by all of them.

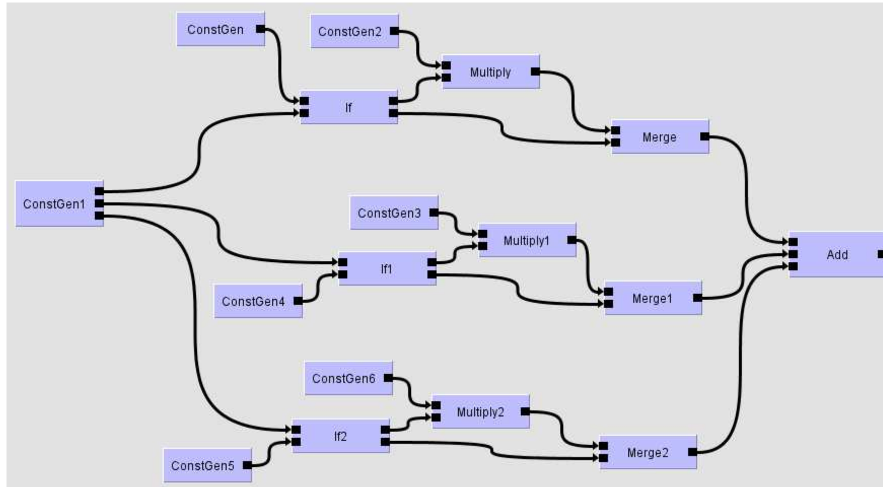

Fig. 4.19 A Example of WCP-07 Structured Synchronizing Merge in Triana that synchronizes one, two or all three branches depending on which of them are active.

WCP-09 **Structured Discriminator**

Description – *The convergence of two or more branches into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when the first incoming branch has been enabled. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The Structured Discriminator construct resets when all incoming branches have been enabled. The Structured Discriminator occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the Structured Discriminator is associated and it must merge all of the branches emanating from the Structured Discriminator. These branches must either flow from the Parallel Split to the Structured Discriminator without any splits or joins or they must be structured in form (i.e. balanced splits and joins) [20].*

Realization – The example in Fig. 4.20 illustrates a possible implementation of this pattern in Kepler where `task_03` represents the discriminator. This actor needs two inputs, the first from a `Relation` operator (or a `Nondeterministic Merge`) that merges into a unique channel the data produced by the branches to be synchronized, and the second from another distinct branch. The second branch (containing the actor `dummy`) produces only one value, therefore `task_03` is enabled only one time consuming the first data value produced by one of the branches to be synchronized and blocking all the other data tokens that subsequently arrive. Even if this solution is able to simulate the behavior of a blocking discriminator in an acyclic context, it is not able to reset the construct when exactly one piece of data is

received from each channel connected with the `Relation` operator, removing the unnecessary ones from the workflow. Moreover, a dummy constant value or a trigger port has to be used for ensuring a single enablement of the subsequent actor, this input acts as a control value and does not influence the computation. Therefore, the pattern cannot be considered supported.

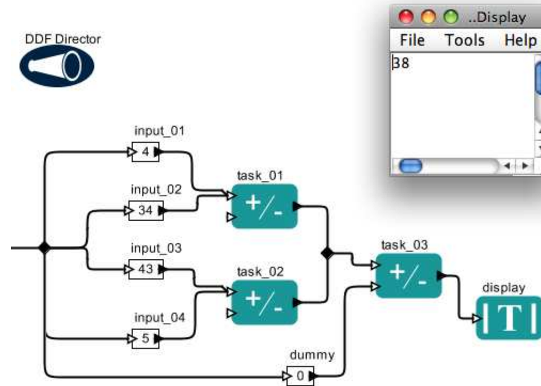

Fig. 4.20 An Example of WCP-09 Structured Discriminator implemented in Kepler.

This solution can be implemented also in Triana with the same considerations about the impossibility to reset the construction in a cyclic context, thus the pattern cannot be considered supported. Fig. 4.21 depicts an exam-

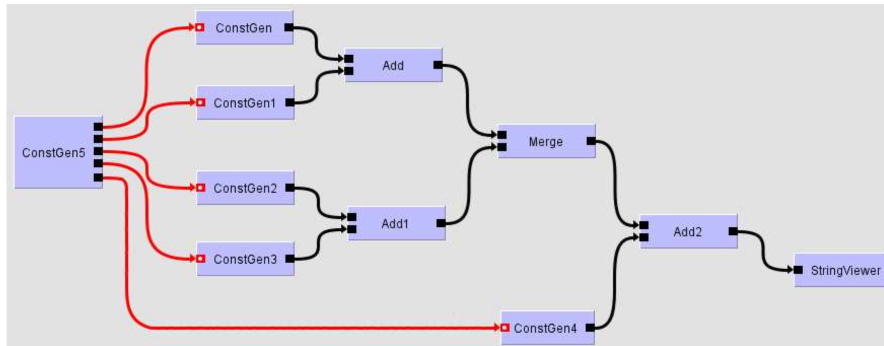

Fig. 4.21 An example of WCP-09 Structured Discriminator implemented in Triana. The red arrows denote trigger connections.

ple of an implementation of this pattern in Triana where the discriminator is represented by the `Add2` component. As for the actor `task_03` in Fig. 4.20, `Add2` needs two inputs, one from the `Merge` component that merges into a unique channel the output produced by the branches to be synchronized, and

the other one from the ConstGen4 component. Since the latter component produces one data token only, Add2 is enabled only once, exactly after the completion of one of the branches to be synchronized.

The implementation provided for Kepler and Triana cannot be adopted in Taverna, because the Constant processor used for generating the synchronization token, always provides a value any time it is invoked and there is no way to limit the number of firings, as in Kepler. Let us consider the example in Fig. 4.22, it depicts an attempt to implement a discriminator in Taverna: the outputs of the two processors String_constant_2 and String_constant_3 are put into a unique channel by the Merge operator. The processor String_constant should produce a unique value, so that Concatenate_two_strings will be enabled only one time as soon as one between String_constant_2 or String_constant_3 has completed, while the output of the other processor will be discarded. Unfortunately, String_constant produces a data value any time one of the other two processors terminates, thus the remaining activity is not blocked. In Taverna when a processor receives inputs from more than one processor, if one of them produces a constant value, as String_constant, a balancing of its tokens is applied by default. In this way, any time the other processors produce a value, a constant is also produced, regardless of how many times the constant processor has been executed so far. This happens also in Kepler, but it provides the possibility to limit the number of firings for a constant actor, while Taverna does not. Therefore, the pattern cannot be considered supported.

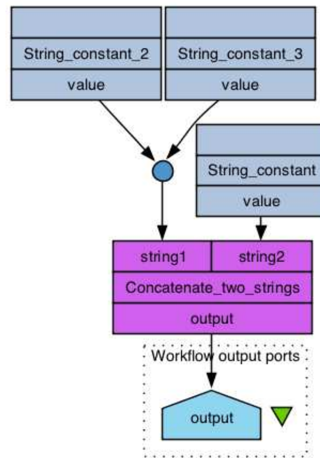

Fig. 4.22 An attempt to implement WCP-09 Structured Discriminator in Taverna.

WCP-28 **Blocking Discriminator**

Description – *The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. The Blocking Discriminator construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the Blocking Discriminator has reset [20].*

Realization – The same considerations presented for WCP-09 Structured Discriminator are valid for the three systems also when the branches to be synchronized come from different points of divergence. Fig. 4.23 depicts an example of Blocking Discriminator in Kepler which is similar to the structured one in Fig. 4.20; while Fig. 4.24 depicts an example of Blocking Discriminator in Triana which is similar to the one in Fig. 4.21. However, as for WCP-09 none of the three systems directly support the pattern.

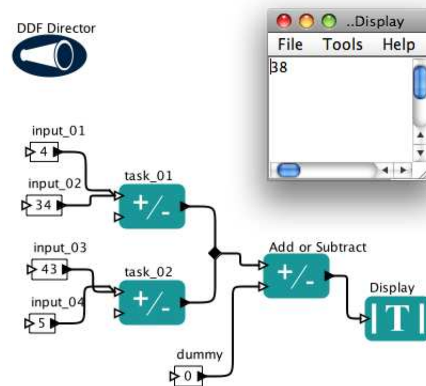

Fig. 4.23 An example of WCP-28 Blocking Discriminator implemented in Kepler.

WCP-29 **Cancelling Discriminator**

Description – *The convergence of two or more branches into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when the first active incoming branch has been enabled. Triggering the Cancelling Discriminator also cancels the execution of all of the other incoming branches and resets the construct [20].*

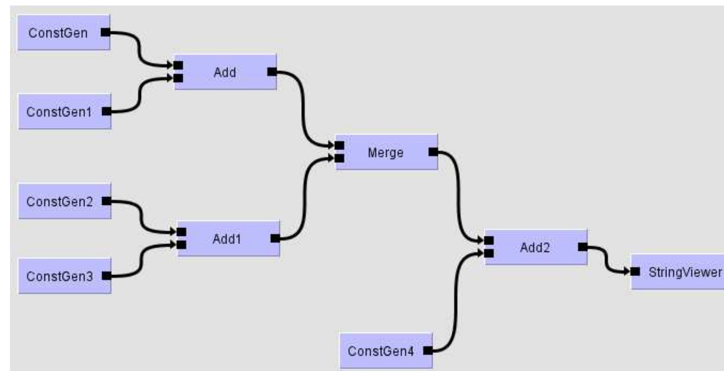

Fig. 4.24 An example of WCP-28 Blocking Discriminator implemented in Triana.

Realization – The Canceling Discriminator pattern cannot be implemented in any of the analyzed systems, because they are not able to withdraw an activity or eliminate data tokens previously produced.

WCP-30 **Structured Partial Join**

Description – *The convergence of two or more branches (say m) into a single subsequent branch following a corresponding divergence earlier in the process model such that the thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Subsequent enablements of incoming branches do not result in the thread of control being passed on. The join construct resets when all active incoming branches have been enabled. The join occurs in a structured context, i.e. there must be a single Parallel Split construct earlier in the process model with which the join is associated and it must merge all of the branches emanating from the Parallel Split. These branches must either flow from the Parallel Split to the join without any splits or joins or be structured in form (i.e. balanced splits and joins) [20].*

Realization – A Structured Partial Join can be implemented in Kepler similarly to the Structured Discriminator using a certain number of `dummy` tokens. In particular, as regards to the example in Fig. 4.20, the input `dummy` has to be a sequence of length k, where k is the number of tasks to be synchronized, or it can be a single constant value with `firingCountLimit` equals to k. In either case, there is no guarantee that exactly k different branches will be synchronized. Moreover, as for the Blocking Discriminator this construction cannot be reset. Therefore, the pattern is not supported in Kepler nor in Triana.

For Taverna the considerations stated for the Structured Discriminator hold also for this pattern, thus WCP-30 is not supported in this system either.

WCP-31 **Blocking Partial Join**

Description – *The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches has been enabled (where* $2 = n < m$*). The join construct resets when all active incoming branches have been enabled once for the same process instance. Subsequent enablements of incoming branches are blocked until the join has reset [20].*

Realization – The same considerations stated for Structured Partial Join (WCP-30) are also valid for the three systems in an unstructured context, thus this pattern is not supported either.

WCP-32 **Cancelling Partial Join**

Description – *The convergence of two or more branches (say m) into a single subsequent branch following one or more corresponding divergences earlier in the process model. The thread of control is passed to the subsequent branch when n of the incoming branches have been enabled where n is less than m. Triggering the join also cancels the execution of all of the other incoming branches and resets the construct [20].*

Realization – The Canceling Partial Join pattern cannot be implemented in any of the analyzed systems, because they are not able to withdraw an activity or eliminate data tokens previously produced.

WCP-33 **Generalised And-Join**

Description – *The convergence of two or more branches into a single subsequent branch such that the thread of control is passed to the subsequent branch when all input branches have been enabled. Additional triggers received on one or more branches between firings of the join persist and are retained for future firings. Over time, each of the incoming branches should deliver the same number of triggers to the AND-join construct (although obviously, the timing of these triggers may vary) [20].*

Realization – The same considerations presented for the synchronization pattern (Wcp-03) also hold in an unstructured context. Therefore, a generalized And-Join can be obtained by connecting the output ports of the tasks to be synchronized with the input ports of a synchronizing task.

Wcp-37 **Local Synchronizing Merge**

Description – *The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when each active incoming branch has been enabled. Determination of how many branches require synchronization is made on the basis on information locally available to the merge construct. This may be communicated directly to the merge by the preceding diverging construct or alternatively it can be determined on the basis of local data such as the threads of control arriving at the merge [20].*

Realization – This pattern is not supported by any of the three analyzed systems, because there is no way to determine the number of active branches.

Wcp-38 **General Synchronizing Merge**

Description – *The convergence of two or more branches which diverged earlier in the process into a single subsequent branch such that the thread of control is passed to the subsequent branch when either (1) each active incoming branch has been enabled or (2) it is not possible that any branch that has not yet been enabled will be enabled at any future time [20].*

Realization – This pattern is not supported by any of the three analyzed systems, because there is no way to determine the number of active branches, nor to determine if a branch that has not yet been enabled will be enabled at any future time.

Wcp-41 **Thread Merge**

Description – *At a given point in a process, a nominated number of execution threads in a single branch of the same process instance should be merged together into a single thread of execution [20].*

Realization – This pattern can be implemented only in Kepler using the `SyncOnTerminator` actor described in Sec. 3.1.2. In particular, we can obtain the pattern by ensuring that each incoming token contains the "ter-

minator” value: after a predefined number of termination tokens have been received, a single termination token is produced as output.

WCP-42 **Thread Split**

Description – *At a given point in a process, a nominated number of execution threads can be initiated in a single branch of the same process instance [20].*

Realization – This pattern can be implemented in Kepler only using the `Repeat` actor introduced in Sec. 3.1.2, as illustrated in Fig. 4.25.

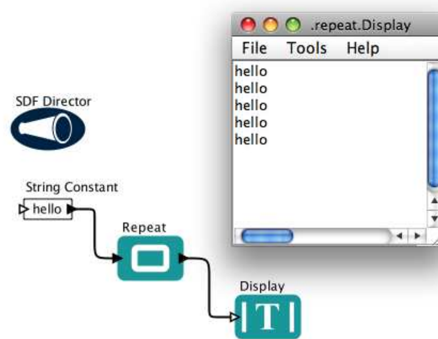

Fig. 4.25 An example of WCP-42 Thread Split in Kepler implemented with the `Repeat` actor.

4.1.3 Multiple Instance Patterns

Multiple Instance patterns describe situations where there are multiple threads of execution active in a process model related to the same activity.

WCP-12 **Multiple Instances without Synchronization**

Description – *Within a given process instance, multiple instances of a task can be created. These instances are independent of each other and run concurrently. There is no requirement to synchronize them upon completion. Each of the instances of the multiple instance task that are created must execute within the context of the process instance from which they were started (i.e. they must share the same case identifier and have access to the same data*

elements) and each of them must execute independently from and without reference to the task that started them [20].

Realization – In scientific WfMSs the creation of multiple instances is naturally supported by the data-flow paradigm: any time a task receives the necessary input data, a new instance of that task is created. By supplying a sequence of data tokens to a task, a new instance of the task is created for each element of the sequence. In particular, if a sequence of length n is provided to A, the system potentially creates n instances of A, which start its execution independently from each other until the end of the workflow is reached, so no synchronization of these instances is performed.

WCP-13 **Multiple Instances with a Priori Design-Time Knowledge**

Description – *Within a given process instance, multiple instances of a task can be created. The required number of instances is known at design time. These instances are independent of each other and run concurrently. It is necessary to synchronize the task instances at completion before any subsequent tasks can be triggered [20].*

Realization – None of the analyzed systems support this pattern. Even if scientific WfMSs support the generation of multiple instances of the same activity, these instances are not synchronized at completion, but run independently from each other until workflow termination.

WCP-14 **Multiple Instances with a Priori Run-Time Knowledge**

Description – *Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications, but is known before the task instances must be created. Once initiated, these instances are independent of each other and run concurrently. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered. [20].*

Realization – None of the analyzed systems support this pattern. Even if scientific WfMSs allow one to generate n instances of the same activity, where n is known at run-time, the considerations about synchronization issues made for WCP-13 also hold in this case.

WCP-15 **Multiple Instances without a Priori Run-Time Knowledge**

Description – *Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered [20].*

Realization – None of the analyzed systems support this pattern. Even if scientific WfMSs allow one to create n instances of the same activity, where n is dynamically determined at run-time before completion, the considerations about synchronization issues made for WCP-13 also hold here.

WCP-34 **Static Partial Join for Multiple Instances**

Description – *Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered. Subsequent completions of the remaining $m-n$ instances are inconsequential, however all instances must have completed in order for the join construct to reset and be subsequently re-enabled [20].*

Realization – None of the analyzed systems support this pattern. Two main problems can be identified: i) how to verify that k of the n instances have been completed; ii) how to synchronized these k instances. The second problem is the same as highlighted for the partial join of instances of different tasks (WCP-30, WCP-31). In [21] the authors assume the existence of a *Bundle* actor that is able to perform this synchronization. Unfortunately, this actor is not part of the standard Kepler distribution and cannot be obtained combining the available actors.

WCP-35 **Cancelling Partial Join for Multiple Instances**

Description – *Within a given process instance, multiple concurrent instances of a task (say m) can be created. The required number of instances is known when the first task instance commences. Once n of the task instances have completed (where n is less than m), the next task in the process is triggered and the remaining $m-n$ instances are cancelled [20].*

Realization – None of the analyzed systems support this pattern both for the impossibility to withdraw the remaining activities and for the synchronization issues described in the previous pattern (Wcp-34).

Wcp-36 **Dynamic Partial Join for Multiple Instances**

Description – *Within a given process instance, multiple concurrent instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. At any time, whilst instances are running, it is possible for additional instances to be initiated providing the ability to do so had not been disabled. A completion condition is specified which is evaluated each time an instance of the task completes. Once the completion condition evaluates to true, the next task in the process is triggered. Subsequent completions of the remaining task instances are inconsequential and no new instances can be created [20].*

Realization – None of the analyzed systems support this pattern for the same reasons stated for Static Partial Join for Multiple Instances (Wcp-34).

4.1.4 State-based Patterns

State-based patterns describe situations in which the execution depends on the state of a process instance.

Wcp-16 **Deferred Choice**

Description – *A point in a process where one of several branches is chosen based on interaction with the operating environment. Prior to the decision, all branches represent possible future courses of execution. The decision is made by initiating the first task in one of the branches i.e. there is no explicit choice but rather a race between different branches. After the decision is made, execution alternatives in branches other than the one selected are withdrawn [20].*

Realization – In Kepler the `BrowseUI` actor allows one to display a web page containing an HTML form and retrieve the inserted values as XML name/value pairs. This actor allows users to chose among different alternatives the next task to perform discarding the other ones. However, interactions with the external environment are not limited to user choices. Other kinds of

interaction can be the arrival of a message or the expiry of a timer. Therefore, the pattern cannot be considered directly supported.

In Taverna the `Choose` processor shows a window with several choices for the user, once the user has performed a selection, the processor returns the selected option. However, as for WCP-04 Exclusive Choice, there are no available processors able to produce a (control) value only on one of its output ports based on a data value received as input. Let us consider the example in Fig. 4.26, the output of the `Choose` processor (i.e. the user choice) is passed to the `process_choice` processor, which is a custom Beanshell script that produces an output on only one of its output ports based on the user choice. Moreover as for the Exclusive Choice, the additional input ports `input1` and `input2` are added to the subsequent processors `constant1`, respectively, `constant2`. In this way only one of them is enabled based on the user choice. However, even with this custom processor the pattern cannot be considered directly supported, because as explained for Kepler the interaction with the external environment cannot be limited to user choices.

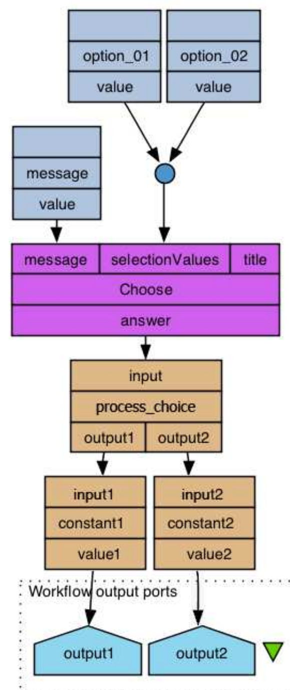

Fig. 4.26 An example of WCP-16 Deferred Choice implemented in Taverna using the `Choose` processor.

Triana does not support this pattern because there are no components able to collect inputs from users.

WCP-17 **Interleaved Parallel Routing**

Description – *A set of tasks has a partial ordering defining the requirements with respect to the order in which they must be executed. Each task in the set must be executed once and they can by completed in any order that accords with the partial order. However, as an additional requirement, no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time) [20].*

Realization – The three analyzed systems do not impose the existence of a complete order among tasks. In Taverna and Triana if no data-flow dependencies are defined between two tasks, they are always potentially executed in parallel, so WCP-17 is not supported. In Kepler on the other hand the SDF and DDF directors can execute only one task at a time and the scheduling is determined on the basis of the declared dependencies. Actors, among which no data dependencies are defined, can be executed in any order but only one at a time. Therefore, WCP-17 can be implemented in Kepler choosing one of these two directors.

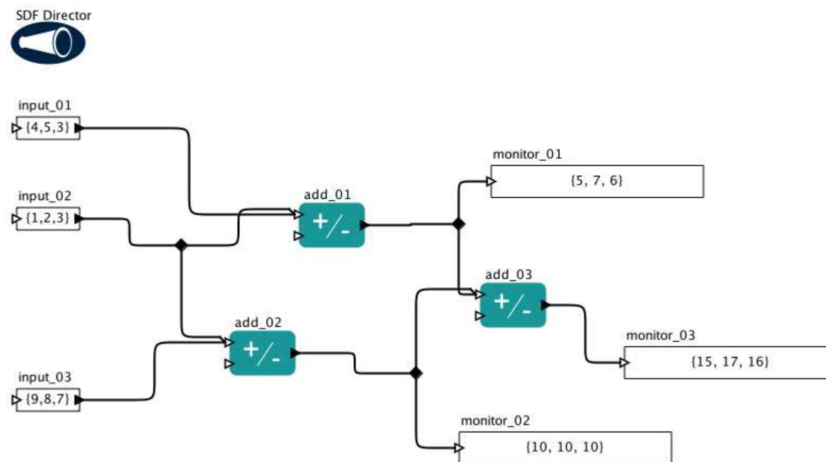

Fig. 4.27 An example of Interleaved Parallel Routing (WCP-17) in Kepler.

In the example of Fig. 4.27 a data-flow dependency is defined between tasks `add_01` and `add_03`, and between `add_02` and `add_03`. Between `add_01` and `add_02` on the other hand no dependencies are defined. However, the SDF director ensures that only one of them is executing at a time.

WCP-18 **Milestone**

Description – *A task is only enabled when the process instance (of which it is part) is in a specific state (typically a parallel branch). The state is assumed to be a specific execution point (also known as a milestone) in the process model. When this execution point is reached the nominated task can be enabled. If the process instance has progressed beyond this state, then the task cannot be enabled now or at any future time (i.e. the deadline has expired). Note that the execution does not influence the state itself, i.e. unlike normal control-flow dependencies it is a test rather than a trigger [20].*

Realization – None of the three analyzed systems support this pattern. The reaching of a milestone can be simulated by a message sent to the task when the process instance reaches a particular state. However, in the three systems channels are persistent and there is no way to delete a produced token held in a channel.

WCP-39 **Critical Section**

Description – *Two or more connected subgraphs of a process model are identified as "critical sections". At runtime for a given process instance, only tasks in one of these "critical sections" can be active at any given time. Once execution of the tasks in one "critical section" commences, it must complete before another "critical section" can commence [20].*

Realization – This pattern is not supported in any of the analyzed systems.

WCP-40 **Interleaved Routing**

Description – *Each member of a set of tasks must be executed once. They can be executed in any order but no two tasks can be executed at the same time (i.e. no two tasks can be active for the same process instance at the same time). Once all of the tasks have completed, the next task in the process can be initiated [20].*

Realization – Interleaved Routing can be considered a specialization of Interleaved Parallel Routing (WCP-17), where tasks are unordered and any interleaving among them is valid. As regards to the implementation of this pattern in the three systems, the same considerations made for WCP-17 still hold, thus only Kepler directly supports this pattern.

4.1.5 Cancellation and Force Completion Patterns

Description – Cancellation patterns involve the ability to withdrawn an activity (Wcp-19 Cancel Activity), or a set of task instances in the same case (Wcp-25 Cancel Region) or an entire process instance (Wcp-20 Cancel Case). Alternatively, they can involve the cancellation (Wcp-26 Cancel Multiple Instance Activity) or the forced completion (Wcp-27 Complete Multiple Instance Activity) of a multiple instance activity.

Realization – None of the analyzed systems supports any cancellation pattern: there is no way to cancel or force the completion of an activity that is executing or has to be executed in the future.

4.1.6 Iteration Patterns

Iteration patterns capture repetitive behaviour in a workflow.

Wcp-10 **Arbitrary Cycles**

Description – *The ability to represent cycles in a process model that have more than one entry or exit point. It must be possible for individual entry and exit points to be associated with distinct branches [20].*

Realization – Arbitrary Cycles (Wcp-10) can be implemented in Kepler by using the `Relation` operator or the `Nondeterministic Merge` actor for representing the various entry points, and the `Boolean Switch` or `Switch` actor for representing the various exit points. Fig. 4.28 depicts an example of an arbitrary cycle with two entry and two exit points. The same construction can also be obtained in Triana by substituting the `Relation` operators with `Merge` components and the `Boolean Switch` actors with `If` components.

This pattern cannot be represented in Taverna due to the lack of an processor for exclusively activating a single branch out of an number of possible branches on the basis of a particular condition (Exclusive Choice).

Wcp-21 **Structured Loop**

Description – *The ability to execute a task or sub-process repeatedly. The loop has either a pre-test or post-test condition associated with it that is either evaluated at the beginning or end of the loop to determine whether it should continue. The looping structure has a single entry and exit point [20].*

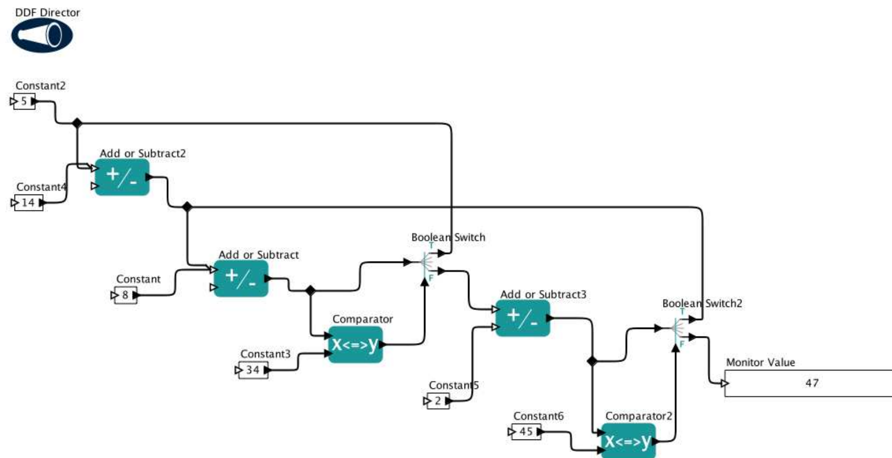

Fig. 4.28 An example of Arbitrary Cycles (WCP-10) in Kepler implemented with some `Relation`operators and `Boolean Switch` actors.

Realization – In Kepler a Structured Loop can be obtained using the `Ramp` actor described in Sec. 3.1.2, which works like a for-loop allowing the execution of one or more tasks a specified number of times. Fig. 4.29 depicts an example implementation of a Structured Loop using this actor: the `Ramp` actor represents the unique entry point of the loop, while the `Boolean Switch` actor represents the unique exit point. The `Ramp` is initially enabled after

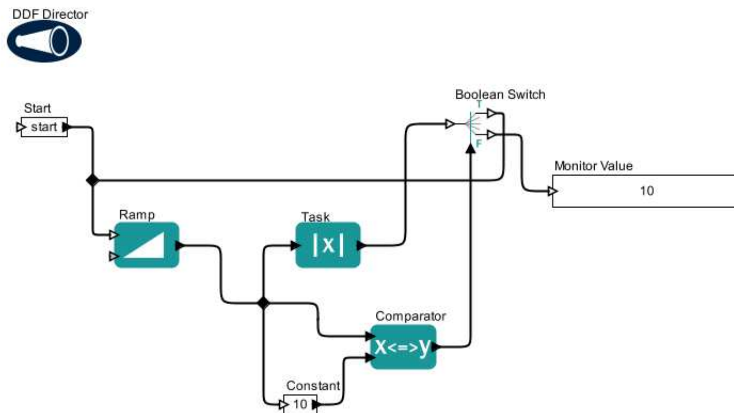

Fig. 4.29 An example of Structured Loop (WCP-21) in Kepler implemented with the `Ramp` actor.

the completion of the `Start` actor. It increments the value of the loop index on the basis of the step parameter and produces in output the incremented

value. This incremented value is used as a trigger message by the `Task` actor and is used by the `Boolean Switch`for evaluating the exit condition. If the incremented value is less than the predefined limit, a new instance of `Ramp` is executed, otherwise the loop terminates and the `Monitor Value` actor is executed. Alternatively, in order to obtain a loop based on a more generic condition than an index increment (i.e. a generic while-loop instead of a for-loop), the `Ramp` can be directly substituted with the involved tasks and the `Boolean Switch` can receive the control value from a task implementing the desired condition. as in Fig. 4.30. In this example the `Multiply` actor

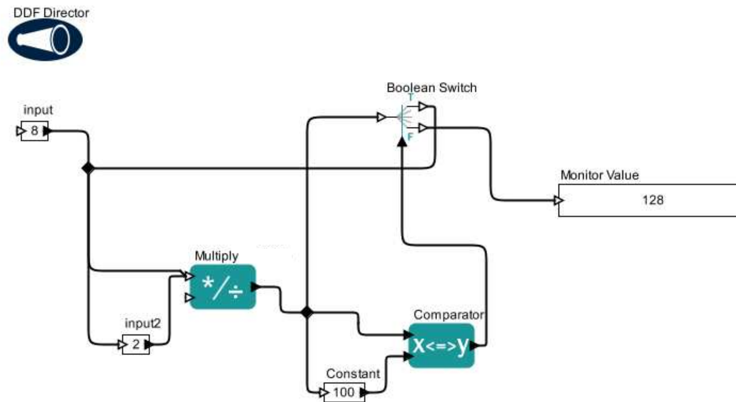

Fig. 4.30 An example of Structured Loop (WCP-21) in Kepler implemented with a `Relation`operator and a `Boolean Switch`actor.

repeatedly doubles the value received in input, until the produced value is less than 100. In particular, the value produced by this actor is used by the `Comparator` actor for evaluating the loop condition and by the `Boolean Switch` actor for providing the new input value to `Multiply`.

In Triana a Structured Loop can be obtained using the `Loop` component described in Sec. 3.3.2. Fig. 4.31 depicts an example of use of the `Loop` component: initially a data value is provided to `Loop` by `ConstGen`, the loop condition is evaluated, if this condition is false the received data value is provided to the `Incrementer` component (it can be any component to be iterated) and after its execution the loop condition is evaluated again; otherwise, the loop terminates and the data value is passed to `ConstView`. Besides the use of a `Loop`, Triana allows one to iterate the execution of several tasks by grouping them into a unique composite task and by defining a hidden control task over them. This control task, automatically added when a group of tasks is created, handles the execution of the group. In the example of Fig. 4.32 `Group` is a group of tasks to which a hidden control task has been attached. In other words, a condition is defined which determines the number of times the group is executed before passing control to the subse-

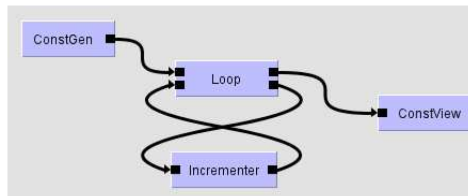

Fig. 4.31 An example of Structured Loop (WCP-21) implemented in Triana using the `Loop` component.

quent task. This is achieved by producing a value which the subsequent task can consume. Finally, a Structured Loop as the one in Fig. 4.30 can also be

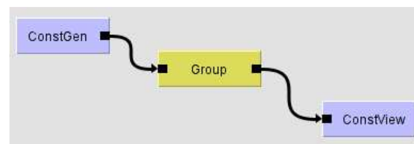

Fig. 4.32 An example of Structured Loop (WCP-21) implemented in Triana using a hidden control task attached to `Group`.

obtained in Triana by substituting the Kepler `Relation` operator with the Triana `Merge` component, and the Kepler `Boolean Switch` actor with the Triana `If` component.

Taverna does not support this pattern because it does not provide any processor for performing loops or for conditionally activating a branch on the basis of the evaluation of a particular condition.

WCP-22 **Recursion**

Description – *The ability of a task to invoke itself during its execution or an ancestor in terms of the overall decomposition structure with which it is associated [20].*

Realization – None of the analyzed systems supports this pattern.

4.1.7 Termination Patterns

Termination patterns deal with the various ways a workflow can complete.

WCP-11 **Implicit Termination**

Description – *A given process (or sub-process) instance should terminate when there are no remaining work items that are able to be done either now or at any time in the future and the process instance is not in deadlock. There is an objective means of determining that the process instance has successfully completed [20].*

Realization – Implicit Termination is the default behavior in scientific WfMSs: a workflow execution terminates when there are not sufficient data tokens left to execute any task.

WCP-43 **Explicit Termination**

Description – *A given process (or sub-process) instance should terminate when it reaches a nominated state. Typically this is denoted by a specific end node. When this end node is reached, any remaining work in the process instance is cancelled and the overall process instance is recorded as having completed successfully, regardless of whether there are any tasks in progress or remaining to be executed [20].*

Realization – This pattern is supported in Kepler only through the `Stop` actor, which can be used to terminate a running workflow.

4.1.8 Trigger Patterns

Trigger patterns deal with external signals that may be required to start certain tasks.

WCP-23 **Transient Trigger**

Description – *The ability for a task instance to be triggered by a signal from another part of the process or from the external environment. These triggers are transient in nature and are lost if not acted on immediately by the receiving task. A trigger can only be utilized if there is a task instance waiting for it at the time it is received [20].*

Realization – Transient Trigger (WCP-23) is not supported in any of the three systems. A trigger can be obtained in scientific WfMSs by adding an additional input port to a task. The input value received through this port is not used during the computation, but is useful only for synchronization

purposes. However, channels are persistent, tokens that flow through them are retained until the connected tasks become able to consume them.

WCP-24 **Persistent Trigger**

Description – *The ability for a task to be triggered by a signal from another part of the process or from the external environment. These triggers are persistent in form and are retained by the process until they can be acted on by the receiving task [20].*

Realization – As explained for WCP-23, a Persistent Trigger (WCP-24) can be obtained by adding an additional port whose values are not used during the computation, but are useful only for synchronization purposes. In Kepler some actors already have a `trigger` port that has no declared type (i.e. it can accept any data type). Connecting a trigger input port is optional, but if a `trigger` port is connected to a channel, the actor will also wait for an input on that port before firing.

4.1.9 Results

Table 4.1 summarizes the WCPs support of the three scientific WfMSs under consideration. Following the evaluation criteria established in [20], a + rating (direct support) or a ± rating (partial support) is assigned when the system provides a construct that completely, respectively, partially satisfies the description of the pattern when used in a context satisfying the context assumption. A − rating (no support) is assigned otherwise. Although workarounds are possible which achieve the desired behavior through the use of various constructs, such as task replications or loops, they are not considered as direct realizations for a pattern.

Kepler provides more control-flow constructs than the other two systems, and thus it supports more WCPs. Taverna is the system with the least number of control-flow constructs and it thus supports a limited number of WCPs. The choice of Taverna to provide only a limited number of routing constructs is compensated by the ease with which one can define a new processor through the use of Beanshell scripts, as it was e.g. done for the `bool_choice` processor in the Exclusive Choice (WCP-04) pattern. On the other hand, the definition of a new component in Kepler is not so trivial because it requires sophisticated programming skills, as actors are processes with their own state and they have to provide polymorphic behavior in order to be able to adapt to the chosen director.

Basic control-flow patterns are supported by all three systems, except for WCP-04 Exclusive Choice which is not supported in Taverna, as this system

Table 4.1 WCP support in the three scientific WfMSs.

Pattern	Kepler	Taverna	Triana
WCP-01 Sequence	+	+	+
WCP-02 Parallel Split	+	+	+
WCP-03 Synchronization	+	+	+
WCP-04 Exclusive Choice	+	-	+
WCP-05 Simple Merge	+	+	+
WCP-06 Multi Choice	-	-	-
WCP-07 Structured Synchronizing Merge	-	-	-
WCP-08 Multi Merge	+	+	+
WCP-09 Structured Discriminator	-	-	-
WCP-10 Arbitrary Cycles	+	-	+
WCP-11 Implicit Termination	+	+	+
WCP-12 M.I. without Synchronization	+	+	+
WCP-13 M.I. with a priori design-time knowledge	-	-	-
WCP-14 M.I. with a priori run-time knowledge	-	-	-
WCP-15 M.I. without a priori run-time knowledge	-	-	-
WCP-16 Deferred Choice	-	-	-
WCP-17 Interleaved Parallel Routing	+	-	-
WCP-18 Milestone	-	-	-
WCP-19 Cancel Activity	-	-	-
WCP-20 Cancel Case	-	-	-
WCP-21 Structured Loop	+	-	+
WCP-22 Recursion	-	-	-
WCP-23 Transient Trigger	-	-	-
WCP-24 Persistent Trigger	+	+	+
WCP-25 Cancel Region	-	-	-
WCP-26 Cancel Multiple Instance Activity	-	-	-
WCP-27 Complete Multiple Instance Activity	-	-	-
WCP-28 Blocking Discriminator	-	-	-
WCP-29 Canceling Discriminator	-	-	-
WCP-30 Structured Partial Join	-	-	-
WCP-31 Blocking Partial Join	-	-	-
WCP-32 Canceling Partial Join	-	-	-
WCP-33 Generalised And-Join	+	+	+
WCP-34 Static Partial Join for M.I.	-	-	-
WCP-35 Canceling Partial Join for M.I.	-	-	-
WCP-36 Dynamic Partial Join for M.I.	-	-	-
WCP-37 Local Synchronizing Merge	-	-	-
WCP-38 General Synchronizing Merge	-	-	-
WCP-39 Critical Section	-	-	-
WCP-40 Interleaved Routing	+	-	-
WCP-41 Thread Merge	+	-	-
WCP-42 Thread Split	+	-	-
WCP-43 Explicit Termination	+	-	-

does not have a construct for redirecting the produced output to only one specific output port, on the basis of the evaluation of a particular condition.

In terms of the advanced branching and synchronization patterns, the three systems lack any support for the partial synchronization of different tasks. Instances of different tasks that execute in parallel can run independently from each other or can all be synchronized (hence not selectively). Different tasks can be synchronized by connecting their output ports to the input ports of the same subsequent task. In this case that task will await values on all its input ports, which signal completion of all the preceding activities. This can be considered a real limitation of scientific WfMSs because during experiments it may be convenient to start a number of activities in parallel and wait the completion of only one (or a sub-set) of them before proceeding. For instance, to perform a complex operation, such as DNA matching, it is reasonable to apply multiple techniques in parallel: the first available result will be used by the subsequent activities, while the other ones are discarded when they arrive, or the activities producing them can be canceled altogether.

Similarly, a result common to all three systems is their support for the multiple instance patterns. The data-flow paradigm underlying scientific WfMSs provides a natural mechanism for generating multiple instances, but none of the three systems is able to synchronize them at completion: the created instances run independently from each other until the workflow completes. This choice also comes from the assumption that several instances of the same experiment can be performed in parallel with different inputs and their executions are mutually independent. When all the experiment instances are terminated, the scientist collects the obtained results and manually derives global outcomes. This also originates from the fact that scientific WfMSs have been developed for automating large-scale experiments, rather than for coordinating the work of a group of agents.

Another important limitation of scientific WfMSs is the absence of a mechanism for canceling running activities. The scientific domain is characterized by an high degree of uncertainty: if the result provided by an activity does not satisfy the expectations, the user has to be able to cancel the other running activities and change those that still have to be performed. This limitation probably comes from the immaturity of the considered systems and in the future scientific WfMSs may be expected to integrate several facilities for exception handling and for managing compensation activities.

As for the state-based patterns and in particular to Interleaved Parallel Routing (WCP-17) and Interleaved Routing (WCP-40), we can observe that these patterns are related to the ability to limit the number of running threads and thus the number of used resources. In scientific WfMSs available resources are automatically managed by the underlying environment (e.g. a Grid environment) in a transparent way for the user.

Arbitrary Cycles (WCP-10) and Structured Loop (WCP-21) are supported by Kepler and Triana, while Taverna has no specific constructs for performing loops and these cannot be built through the use of other constructs either due

to the lack of a choice construct. As highlighted in [22] Taverna only allows the representation of acyclic workflows models (i.e. Direct Acyclic Graphs, DAGs). All three systems support the hierarchical decomposition of tasks, however a composite task cannot contain itself in its decomposition, thus the Recursion pattern (WCP-22) is not supported; recursive definitions can be obtained only through use of the underlying programming language (Java).

A scientific workflow terminates when there are not sufficient data tokens for executing another task instance, thus WCP-11 Implicit Termination is the commonly supported termination pattern. Only Kepler has a specific construct to also explicitly terminate a workflow execution that reaches a specific point.

Finally, support for triggers can be realized by adding an additional input ports to tasks. However, channels are persistent: tokens that flow through them are retained until the connected tasks are able to consume them. Therefore, only Persistent Trigger (WCP-24) is supported.

4.2 Workflow Data Patterns

Workflow Data Patterns (WDPs) collect language features for describing and managing data resources during process execution. The three considered scientific WfMSs share a common computational model in which data are carried only by tokens and there are no shared variables. As a consequence, the supported data patterns are substantially the same for all the three systems.

4.2.1 Data Visibility Patterns

Data Visibility patterns regard the potential contexts in which a data construct can be defined and utilized [23]. In scientific WfMSs data are contained only inside tokens, variables are local to each task instance, there are no global variables and tasks can communicate only providing a value to their output port(s) and reading a value from their input port(s).

WDP-01 **Task Data**

Description – *Data elements can be defined by tasks which are accessible only within the context of individual execution instances of that task. [23].*

Realization – Variables defined inside tasks are accessible only within the context of their individual execution instances.

WDP-02 **Block Data**

Description – *Block tasks (i.e. tasks which can be described in terms of a corresponding subprocess) are able to define data elements which are accessible by each of the components of the corresponding subprocess [23].*

Realization – This pattern is not supported in the three systems, because the tasks inside a sub-workflow can communicate only through channels, variables are local to each atomic task and there is no way to define variables shared by all the tasks inside the same sub-workflow.

WDP-03 **Scope Data**

Description – *Data elements can be defined which are accessible by a subset of the tasks in a case [23].*

Realization – As observed in the previous pattern, in the three systems there is no way to define shared variables at any level.

WDP-04 **Multiple Instance Data**

Description – *Tasks which are able to execute multiple times within a single case can define data elements which are specific to an individual execution instance. [23].*

Realization – Each task of a scientific WfMS can be safely executed concurrently with itself multiple times, more specifically any time a new input is provided to a task, the system creates a new instance of that task. Variables used by that task are specific to each individual execution instance.

WDP-05 **Case Data**

Description – *Data elements are supported which are specific to a process instance or case. They can be accessed by all components of the process during the execution of the case [23].*

Realization – As observed in WDP-03, in the three systems there is no way to define shared variables at any level.

WDP-06 **Folder Data**

Description – *Data elements can be defined which are accessible by multiple cases on a selective basis. They are accessible to all components of the cases to which they are bound [23].*

Realization – As observed in WDP-03, in the three systems there is no way to define shared variables at any level.

WDP-07 **Workflow Data**

Description – *Data elements are supported which are accessible to all components in each and every case of the process and are within the context of the process itself [23].*

Realization – As observed in WDP-03, in the three systems there is no way to define shared variables at any level.

WDP-08 **Environment Data**

Description – *Data elements which exist in the external operating environment are able to be accessed by components of processes during execution [23].*

Realization – External resources, such as a database or a file, can be accessed by tasks in the workflow in various ways. For instance, in Kepler the `Open Database Connection` actor allows one to open a connection with a local or remote database, while other actors, such as `Database Writer` and `Database Query`, interact with a database in various ways. Moreover, other actors, such as the `Directory Listing`, allow one to access files in a remote file system. The other two scientific WfMSs provide similar components for accessing a local or remote database/file system.

4.2.2 Data Interaction Patterns

Data Interaction patterns examine the various ways in which data elements can be passed between components in a process and how the flow of data elements is determined by the characteristics of the individual components. In particular, we distinguish between Internal Data Interaction patterns, which regard the communication between components within the same process, and External Data Interaction patterns, which regard the interaction of a process element with the external environment [23].

WDP-09 **Task to Task**

Description – *The ability to communicate data elements between one task instance and another within the same case [23].*

Realization – In scientific WfMSs tasks can communicate only through channels that connect the output port of a task with the input port of another task. Moreover, as regards to the relationship between data perspective and control-flow perspective, the same channels are used to pass both control-flow and data tokens (*integrated control and data channels* [23]), as control-flow is directly specified in terms of data dependencies.

WDP-10 **Block Task to Sub-Workflow Decomposition**

Description – *The ability to pass data elements from a block task instance to the corresponding subprocess that defines its implementation [23].*

Realization – The input and output ports of a block task (a composite actor in Kepler, a nested workflow in Taverna, or a group in Triana) are connected to the input and output ports of some of its constituent sub-tasks. Therefore, data elements are passed from a block task to its components through channels (*explicit data passing via data channels* [23]).

WDP-11 **Sub-Workflow Decomposition to Block Task**

Description – *The ability to pass data elements from the underlying subprocess back to the corresponding block task. [23].*

Realization – As stated for the previous pattern, the input and output ports of some internal components are connected to the input and output ports of the corresponding block task, the final output is passed to the block task through channels (*explicit data passing via data channels* [23]).

WDP-12 **to Multiple Instance Task**

Description – *The ability to pass data elements from a preceding task instance to a subsequent task which is able to support multiple execution instances. This may involve passing the data elements to all instances of the multiple instance task or distributing them on a selective basis. The data passing occurs when the multiple instance task is enabled. [23].*

Realization – Scientific WfMSs transparently support the generation of multiple instances. As soon as the necessary input data is provided to a task a new instance for that task is created. Each task instance receives distinct input data through tokens and works on its own data without side effects for the other task instances in the process (*instance-specific data passed by value or by reference* [23]).

WDP-13 **from Multiple Instance Task**

Description – *The ability to pass data elements from a task which supports multiple execution instances to a subsequent task. The data passing occurs at the conclusion of the multiple instance task. It involves aggregating data elements from all instances of the task and passing them to a subsequent task [23].*

Realization – As stated for the previous pattern, in scientific WfMSs each time the necessary data is provided to a task, a new instance of that task can be created. The data produced by each task instance is redirected to its output port(s) inside individual tokens and passed to the sub-sequent activity as soon as they are produced. Therefore, the pattern cannot be considered directly supported, because it requires the aggregation of the data elements produced by the various instances before passing them to the subsequent task, which is not possible. This problem stems from the impossibility of synchronizing multiple instances of the same task at completion.

WDP-14 **Case to Case**

Description – *The passing of data elements from one case of a process during its execution to another case that is executing concurrently [23].*

Realization – This pattern is not supported in the three systems, as there is no way to communicate between two different instances of a workflow.

WDP-15 (WDP-19, WDP-23) **Task (Case/Workflow) to Environment – Push Oriented**

Description – *The ability of a task (case or process environment) to pass data elements to a resource or service in the operating environment [23].*

Realization – In the three analyzed systems each task can pass the computed output to the external environment after its completion, for instance

by writing the computed results into an external database, or on a local or remote file system.

WDP-16 (WDP-20, WDP-24) **Environment to Task (Case/Workflow) – Pull Oriented**

Description – *The ability of a task (case or process environment) to request data elements from resources or services in the operational environment. [23].*

Realization – In the three systems each task (or process or entire workflow) can receive the necessary input also from an external process or through accessing an external resource, for instance by reading the data from an external database, a local or remote file system.

WDP-17 (WDP-21, WDP-25) **Environment to Task (Case/Workflow) – Push Oriented**

Description – *The ability of a task (case or process environment) to receive data elements from resources or services in the operational environment. [23].*

Realization – These patterns are not supported as only tasks inside the workflow can start the connection with the external environment, for instance by reading data from an external database.

WDP-18 (WDP-22, WDP-26) **Task (Case/Workflow) to Environment – Pull Oriented**

Description – *The ability for a task (case or process environment) to receive and utilize data elements passed to it from services and resources in the operating environment on an unscheduled basis [23].*

Realization – These patterns are not supported as only tasks inside the workflow can start the connection with the external environment, for instance by writing data to an external database.

4.2.3 Data Transfer Patterns

Data Transfer patterns consider the manner in which the actual transfer of data elements occurs between one process component and another [23].

WDP-27, WDP-28 **Data Transfer by Value – Incoming, Outgoing**

Description – *The ability of a process component to receive incoming data elements by value [23].*

Realization – In scientific WfMSs data are passed to tasks by means of tokens on channels that connect output and input ports. Each task operates on a copy of the received values.

WDP-29 **Data Transfer - Copy In/Copy Out**

Description – *The ability of a process component to copy the values of a set of data elements from an external source (either within or outside the process environment) into its address space at the commencement of execution and to copy their final values back at completion [23].*

Realization – This pattern is supported by all three systems, the data value collected from an external resource (e.g. a remote database) can be copied to the local address space, similarly the produced output can be copied back to an external resource.

WDP-30 **Data Transfer by Reference - Unlocked**

Description – *The ability to communicate data elements between process components by utilizing a reference to the location of the data element in some mutually accessible location. No concurrency restrictions apply to the shared data element [23].*

Realization – The data passed from one task to another inside tokens can also be the name of a file or the address of an external (remote) resource, and generally no concurrency restrictions are applied to the shared data.

WDP-31 **Data Transfer by Reference - With Lock**

Description – *The ability to communicate data elements between process components by passing a reference to the location of the data element in some mutually accessible location. Concurrency restrictions are implied with the receiving component receiving the privilege of read-only or dedicated access to the data element. The required lock is declaratively specified as part of the data passing request [23].*

Realization – This pattern is not supported because there is no way to declaratively specify a lock on certain data elements transferred by reference.

WDP-32 **Data Transformation - Input**

Description – *The ability to apply a transformation function to a data element prior to it being passed to a process component. The transformation function has access to the same data elements as the receiving process component [23].*

Realization – This pattern is not supported.

WDP-33 **Data Transformation - Output**

Description – *The ability to apply a transformation function to a data element immediately prior to it being passed out of a process component. The transformation function has access to the same data elements as the process component that initiates it [23].*

Realization – This pattern is not supported.

4.2.4 Data-based Routing

Data-based Routing patterns capture the various ways in which data elements can interact with other perspectives and influence the overall operation of a process instance [23].

WDP-34 **Task Precondition - Data Existence**

Description – *Data-based preconditions can be specified for tasks based on the presence of data elements at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively [23].*

Realization – In scientific workflows dependencies among tasks are data dependencies, i.e a task can be activated only if the necessary input is available.

Therefore, this pattern is naturally supported by the adopted computational model.

Wdp-35 **Task Precondition - Data Value**

Description – *Data-based preconditions can be specified for tasks based on the value of specific parameters at the time of execution. The preconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated precondition evaluates positively [23].*

Realization – This pattern is not supported because in the three systems task activation depends on data availability, but it is not possible to impose any specific condition on their values.

Wdp-36 **Task Postcondition - Data Existence**

Description – *Data-based postconditions can be specified for tasks based on the existence of specific parameters at the time of task completion. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively [23].*

Realization – This pattern is not supported in the three systems, a data-dependency specifies only a condition for task activation and not for its completion.

Wdp-37 **Task Postcondition - Data Value**

Description – *Data-based postconditions can be specified for tasks based on the value of specific parameters at the time of execution. The postconditions can utilize any data elements available to the task with which they are associated. A task can only proceed if the associated postcondition evaluates positively [23].*

Realization – This pattern is not supported in the three systems, a data-dependency captures data availability only, not specific constraints on data values, and influence task activation only and not completion.

WDP-38 **Event-based Task Trigger**

Description – *The ability for an external event to initiate a task and to pass data elements to it [23].*

Realization – This pattern is not supported because communication with the external environment can be initiated only by tasks inside the workflow and not by the environment.

WDP-39 **Data-based Task Trigger**

Description – *Data-based task triggers provide the ability to trigger a specific task when an expression based on data elements in the process instance evaluates to true. Any data element accessible within a process instance can be used as part of a data-based trigger expression [23].*

Realization – This pattern is not supported.

WDP-40 **Data-based Routing**

Description – *Data-based Routing provides the ability to alter the control-flow within a case based on the evaluation of data-based expressions. A Data-based Routing expression is associated with each outgoing arc of an OR-split or XOR-split. It can be composed of any data-values, expressions and functions available in the process environment providing it can be evaluated at the time the split construct with which it is associated completes. Depending on whether the construct is an XOR-split or OR-split, a mechanism is available to select one or several outgoing arcs to which the thread of control should be passed based on the evaluation of the expressions associated with the arcs [23].*

Realization – This pattern is supported in Kepler and Triana with the same mechanisms described for the Exclusive Choice (WCP-04) and the Multi Choice (WCP-06). It is not supported in Taverna on the other hand for the same reasons explained for WCP-04 and WCP-06, namely the absence of constructs for routing the thread of control only to one of the subsequent branches, based on the evaluation of a particular condition.

4.2.5 Results

Table 4.2 summarizes the WDP support of the three systems. The same criteria used for rating the control-flow patterns in Sec. 4.1.9, are also applied

Table 4.2 WDP support in the three scientific WfMSs.

Pattern	Kepler	Taverna	Triana
WDP-01 Task Data	+	+	+
WDP-02 Block Data	-	-	-
WDP-03 Scope Data	-	-	-
WDP-04 Multiple Instance Data	+	+	+
WDP-05 Case Data	-	-	-
WDP-06 Folder Data	-	-	-
WDP-07 Workflow Data	-	-	-
WDP-08 Environment Data	+	+	+
WDP-09 Task to Task	+	+	+
WDP-10 Block Task to Sub-Workflow Decomposition	+	+	+
WDP-11 Sub-Workflow Decomposition to Block Task	+	+	+
WDP-12 to Multiple Instance Task	+	+	+
WDP-13 from Multiple Instance Task	-	-	-
WDP-14 Case to Case	-	-	-
WDP-15 Task to Environment – Push-Oriented	+	+	+
WDP-16 Environment to Task – Pull-Oriented	+	+	+
WDP-17 Environment to Task – Push-Oriented	-	-	-
WDP-18 Task to Environment – Pull-Oriented	-	-	-
WDP-19 Case to Environment – Push-Oriented	+	+	+
WDP-20 Environment to Case – Pull-Oriented	+	+	+
WDP-21 Environment to Case – Push-Oriented	-	-	-
WDP-22 Case to Environment – Pull-Oriented	-	-	-
WDP-23 Workflow to Environment – Push-Oriented	+	+	+
WDP-24 Environment to Workflow – Pull-Oriented	+	+	+
WDP-25 Environment to Workflow – Push-Oriented	-	-	-
WDP-26 Workflow to Environment – Pull-Oriented	-	-	-
WDP-27 Data Transfer by Value - Incoming	+	+	+
WDP-28 Data Transfer by Value - Outgoing	+	+	+
WDP-29 Data Transfer - Copy In/Copy Out	+	+	+
WDP-30 Data Transfer by Reference - Unlocked	+	+	+
WDP-31 Data Transfer by Reference - With Lock	-	-	-
WDP-32 Data Transformation - Input	-	-	-
WDP-33 Data Transformation - Output	-	-	-
WDP-34 Task Precondition - Data Existence	+	+	+
WDP-35 Task Precondition - Data Value	-	-	-
WDP-36 Task Postcondition - Data Existence	-	-	-
WDP-37 Task Postcondition - Data Value	-	-	-
WDP-38 Event-based Task Trigger	-	-	-
WDP-39 Data-based Task Trigger	-	-	-
WDP-40 Data-based Routing	+	-	+

here. The data-flow modeling paradigm adopted by scientific WfMSs makes the ways in which data are managed and passed by these systems essentially uniform (the only difference is the inability of Taverna to support Data-based Routing).

As regards to the Data Visibility patterns, we can observe that variables are local to each task instance and there is no way to define variables shared by a subset of tasks or the entire workflow.

As regards the Data Interaction patterns, tasks can communicate only using data tokens exchanged through channels, and this holds also for block tasks and their sub-workflow decompositions. The interaction with the external environment, the request for the provision of an external data element can be initialized only by a task, case or workflow.

For the Data Transfer patterns, we note that the data contained inside a token can be a value or a reference to an external resource. In the latter case, a task can work directly on the external resource or make a local copy of it, writing back the result at completion.

Finally, as concerns the Data-based Routing, data availability drives the computation, but data dependencies influence only task activation, not their completion. Moreover, they only capture that certain data need to be present, not that data need to take on certain values.

4.3 Workflow Resource Patterns

Workflow Resource Patterns capture the various ways in which resources are represented and utilized in workflows [24]. Scientific WfMSs consider processes that are usually enacted by only one user at a time, thus they do not have to manage different agents or different roles with related authorization and authentication issues [13]. Moreover, little or no user interaction is needed to perform an activity, no work is assigned to human agents and the human intervention is usually limited to perform run-time decisions. As a result, only few resource patterns are supported, as discussed in the following.

4.3.1 Creation Patterns

Creation patterns correspond to limitations on the manner in which a work item may be executed. They are specified at design time, usually in relation to a task, and serve to restrict the range of resources that can undertake work items that correspond to the task [24].

As stated above, the three considered scientific WfMSs do not provide a mechanism for identifying and distinguishing resources, in particular human agents. Work items are automatically executed as soon as they have

the necessary input without the need to be explicitly allocated to a particular resource. Thus, as far as the creation patterns are concerned, only the Automatic Execution (Wrp-11) pattern is supported in the three systems.

4.3.2 Push Patterns

Push patterns characterize situations where newly created work items are proactively offered or allocated to resources by the system. These may occur indirectly by advertising work items to selected resources via a shared work list or directly with work items being allocated to specific resources [24].

In the three considered scientific WfMSs, work items are directly allocated by the system and executed as soon as they are enabled by the availability of the necessary input. Therefore, the only supported push pattern is the Distribution on Enablement (Wrp-19) pattern.

4.3.3 Pull Patterns

Pull patterns correspond to the situation where individual resources are made aware of specific work items, that require execution, either via a direct offer from the system or indirectly through a shared work list. The commitment to undertake a specific task is initiated by the resource itself rather than the system [24].

In the three considered scientific WfMSs it is the system that schedules the execution of work items and these automatically start to execute when the necessary inputs are available. In particular, in Kepler it is the director that orchestrates workflow execution. If follows that none of the pull patterns is supported by the three systems.

4.3.4 Detour Patterns

Detour patterns refer to situations where work item distributions that have been made for resources are interrupted either by the system or at the instigation of the resource. As a consequence of this event, the normal sequence of state transitions for a work item is varied [24].

These patterns are not supported because work items are automatically executed and they cannot be intentionally suspended, re-routed, re-allocated or canceled.

4.3.5 Auto-Start Patterns

Auto-start patterns relate to situations where execution of work items is triggered by specific events in the lifecycle of the work item or the related process definition. Such events may include the creation or allocation of the work item, completion of another instance of the same work item or a work item that immediately precedes the one in question [24].

The Commencement on Creation (Wrp-36) pattern is directly supported by the analyzed systems. Wrp-36 refers to the ability of a resource to commence execution on a work item as soon as this is created. In the three systems a work item is executed immediately after its enablement.

:and Chained Execution (Wrp-39).

4.3.6 Visibility Patterns

Visibility patterns classify the various scopes in which work item availability and commitment are able to be viewed by resources [24].

These patterns are not supported: none of the three systems provides a facility to visualize the list of available or committed work items.

4.3.7 Multiple Resource Patterns

Multiple Resource patterns relax the one-to-one correspondence between resources and work items that have assumed in previous patterns, allowing that a resource can work on different work items simultaneously and that multiple resources can work on the same work item [24].

As far as this group of patterns is concerned, in each of the three systems the same resource can work on different work items simultaneously, thus the Simultaneous Execution (Wrp-42) pattern is supported.

Chapter 5
Scientific Workflow Patterns

In this chapter we formalize four new patterns that have emerged from the analysis of the scientific WfMSs under consideration. These patterns concern various ways in which the input values provided to a task can be combined before the task execution and thus they can be classified as *routing patterns.* The behavior prescribed by these patterns is formally described in terms of CPNs. The CPN representations make use of advanced arc expressions whenever possible for reducing the complexity of the construction.

Swp-01 **Dynamic Input Size**

Description – *The ability to consume* n *data elements from the same input channel, where* n *is determined at run-time on the basis of the value received from another input channel.*

Example – In the geographical domain a segmented property is a property that changes its value along the path of a road. Examples are the road width, the speed limits, the number of lanes, etc. Suppose we have a task A that given a road network r and a particular segmented property p produces a set of homogeneous segments, i.e. the portion of r characterized by the same value of p. The number of segments produced by A depends on the characteristics of the road network and the property considered; we can assume that at completion A also generates another output containing the number of identified segments. Given two instances of A, denoted as A_1 and A_2, which work on the same road network but consider two different properties: e.g. the number of lanes and the speed limits, a subsequent tasks B that operates on the segments produced by A_1 and A_2, in order to compare the two properties, it has to consume from A_1 and A_2 a variable number of tokens which depends on the number of identified segments.

Motivation – This pattern provides a means for determining, at run-time, the number of tokens required from a task A in order to execute a task B. This pattern can be considered as the dynamic version of WCP-41 Thread Merge.

Overview – This pattern is exemplified by the CPN in Fig. 5.1. Place *size* contains the value of n, while *data* contains the actual data values to consume. Transition t_2 generates a list of size n of data values from *data*, while transition t_3 adds this list to the output place. Place *start* ensures that t_1 can execute again only when an output for the previous value of n has been produced.

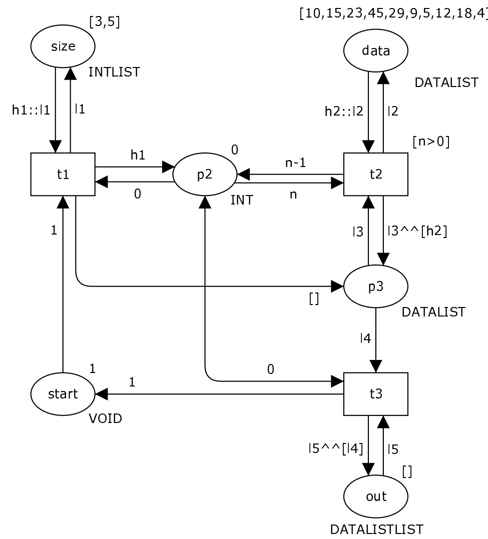

Fig. 5.1 CPN representation of the SWP-01 Dynamic Input Size pattern.

Assuming that *size* initially contains the list $[3, 5]$ and *data* contains the list $[10, 15, 23, 45, 29, 9, 5, 12, 18, 4]$, at the end *out* will contain the list $[[10, 15, 23], [45, 29, 9, 5, 12]]$, *data* will contain the list $[18, 4]$ and *size* will contain the empty list.

Implementation – This pattern is supported in Kepler through the `SyncOnTerminator` actor where the parameter `numberOfOccurences` is provided using a parameter port.

SWP-02 **Dynamic Token Replication**

Description – *The ability to generate n copies of a data element d received in input, where the number n is determined at run-time on the basis of the value received from another input channel.*

Example – Let us consider a task A that determines the correlation between two pieces of information x and y, for instance the temperature and the pressure, or the temperature and the humidity. Suppose we want to determine the correlation between the temperature and several other factors, whose values are not known at design-time, because depend on the availability of different instruments. Multiple copies of the same temperature information have to be generated and passed to A, which combines each copy with another piece of information. The number of data tokens to be created is determined by another task B which tests the availability of the other measuring instruments.

Motivation – This pattern allows one to dynamically determine the number of copies of a data token to produce. As each produced data token generates a new instance of the subsequent task that consumes it, this pattern can be considered the dynamic version of WCP-42 Thread Split.

Overview – This pattern is exemplified by the CPN in Fig. 5.2. Transition *rep* consumes two inputs: the number n of copies to produce from in_1, and the value d to replicate from in_2, and produces n tokens with value d in p_3. Place in_1 is intended to have capacity one: only one token can be held inside it at time. Transition t_2 is executed n times for adding the produced copies of d to the output list. If t_1 produces another value for n before *rep* starts again, the next execution of *rep* will use this new value, otherwise *rep* uses the previous value for n.

Let us assume that p_1 initially contains the list $[2, 4]$, in_1 contains the value zero and in_2 contains the list $[10, 20, 30]$. In this situation only transition t_1 is enabled, because the value in in_1 is zero. t_1 transfers the first value in p_1 to in_1, enabling transition *rep*. Transition t_1 can fire without the constraint that the previous value produced for place p_1 is actually used or has been used exactly one time. For instance, if *rep* is executed for all data values in in_2 before another execution of t_1 is performed, the final output in *out* will be $[10, 10, 20, 20, 30, 30]$. Otherwise, supposing that *rep* is executed only once before t_1 is executed again, the final output in *out* will be $[10, 10, 20, 20, 20, 20, 30, 30, 30, 30]$, and so on.

Implementation – This pattern is directly supported by Kepler through the `Repeat` actor, when the parameter `numberOfTimes` is passed to the actor using a parameter port instead of being fixed at design time.

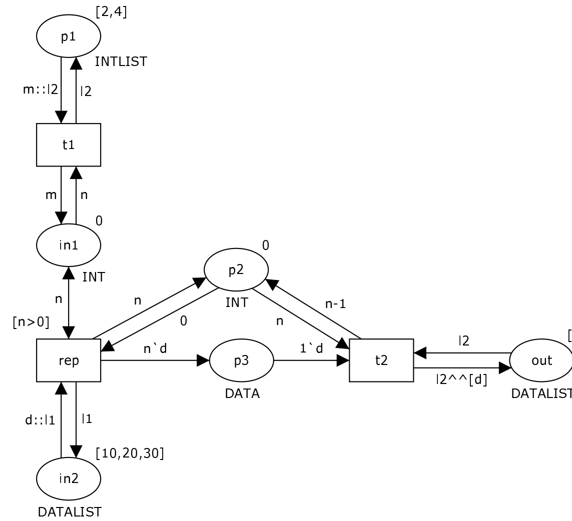

Fig. 5.2 CPN representation of the SWP-02 Dynamic Token Replication pattern.

SWP-03 **Dynamic Balancing of Input Tokens**

Description – *The ability to replicate a data element received from an input channel in order to balance the number of tokens received from another input channel.*

Example – Let us consider a task A that receives as input the values of the temperature and pressure produced by two different instruments. Suppose that the two instruments provide new measurements with a different rate, for instance the temperature is measured every hour, while a new pressure value is detected every two hours. Task A can be executed any time a new temperature is available by reusing the previous pressure value, if a new pressure value is not available at that time.

Motivation – This pattern provides a means for balancing the data produced by two or more tasks with different production rates.

Overview – The behavior of this pattern is formalized by the CPN in Fig. 5.3. The last values read from in_1 and in_2 are stored in mem_1 and mem_2, respectively. If a data value is available in in_1 but not in in_2, the value stored in mem_2 is used. Symmetrically, if a data value is available in in_2 but not in in_1, the value stored in mem_1 is used. Transition t_2 combines the two data elements and puts the obtained pair in *out*. Place *start* ensures that a new iteration is performed only when the previous pair has been outputted.

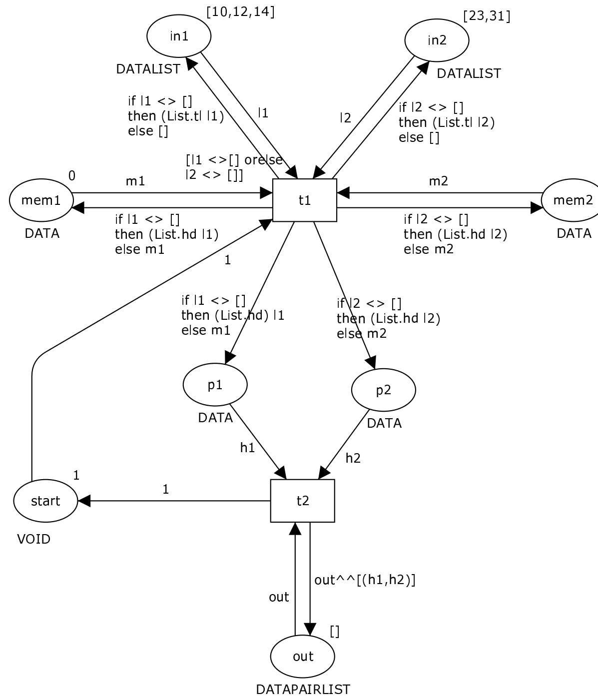

Fig. 5.3 CPN representation of the SWP-03 Dynamic Input Balancing pattern.

Assuming that in_1 initially contains the list $[10, 12, 14]$ and in_2 contains the list $[23, 31]$, at the end *out* will contain the list $[(10, 23), (12, 31), (14, 31)]$.

Implementation – This pattern is directly supported by Kepler and Taverna when the considered tasks are constant generators; in this case a new constant is provided any time a value is produced by one of the other involved tasks. This behavior can be changed in Kepler by fixing the number of firings for a constant generator actor.

SWP-04 **Cartesian Product of Input Tokens**

Description – *T*he ability to compute the cartesian product of the data values contained into two or more channels connected to the same task, so that this task can be executed on each possible combination of inputs.

Example – Let us consider a task A that receives as input a set of letters L and a dictionary D, and produces as output the words in D that are only composed of letters in L. Suppose there are two instances of A, called A_1

and A_2, that work on two different sets of letters L_1 and L_2 and two distinct dictionaries D_1 and D_2. A task B that has to find the common words produced by A_1 and A_2, can receive as input the cartesian product of the outputs produced by A_1 and A_2, and return those words that are paired with themselves. For instance, if A_1 produces the words $\{w_1, w_2, w_3, \ldots\}$ and A_2 produces the words $\{u_1, u_2, u_3, \ldots\}$, task B will receive as input the pairs $\{(w_1, u_1), (w_1, u_2), (w_1, u_3), (w_2, u_1), \ldots\}$.

Motivation – This pattern provides a means for combining the data produced by different tasks before passing these data to another task.

Overview – The behavior of this pattern is exemplified in Fig. 5.4. Places in_1 and in_2 contain the input values. Transition t_1 reads the first value in in_1 and combines it with all the values in in_2, adding the produced pair to p_4. When all values in in_2 have been combined with the current value of in_1, transition t_3 is enabled: it copies back in in_2 its original values and consumes the value in in_1 which has been completely combined. If all values in in_1 have been combined, t_3 consumes only the value in in_1 and does not copy back the values in in_2.

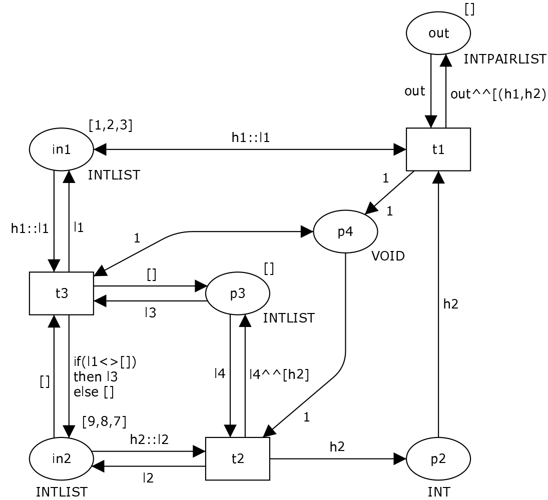

Fig. 5.4 CPN representation of the SWP-04 Cartesian Product of Input Tokens pattern.

Assuming that in_1 initially contains the list $[1, 2, 3]$ and in_2 the list $[9, 8, 7]$, at the end *out* will contain the list $[(1, 9), (1, 8), (1, 7), (2, 9), (2, 8), (2, 7), (3, 9), (3, 8), (3, 7)]$.

Implementation – This pattern is directly supported by Taverna which provides the possibility to specify for each task when to work on the dot product or cross product of the received inputs. In the first case the data received in each input port is combined in tuples as soon as these data tokens arrive and they are used only once. In the second case the tuples corresponding to the cross product of all the received data tokens are computed and passed to the task.

The support for the above patterns is summarized in Table 5.1.

Table 5.1 SWP support in the three scientific WfMSs.

Pattern	Kepler	Taverna	Triana
SWP-01 Dynamic Input Size	+	-	-
SWP-02 Dynamic Token Duplication	+	-	-
SWP-03 Dynamic Input Balancing	-	±	±
SWP-04 Cartesian Product of Input Tokens	-	+	-

Chapter 6
Workflow Management Systems Design Recommendations

In this chapter we would like to discuss the maturity of business WfMSs with respect to their application to the domain of scientific workflows, and in particular we try to answer a simple question: what can business WfMSs learn from scientific WfMSs?

The differences discussed in Section 2.1 mainly depend on the different execution paradigms adopted by the two classes of WfMSs. Similarly, the four new patterns defined in Chapter 5 are made possible by the adoption of an execution semantics grounded in Process Networks, where data tokens are retained inside unbounded channels until a task can consume them. Therefore, these new patterns cannot easily be realized in systems based on shared variables. A variable only retains the most recent obtained data value and earlier values cannot be accessed any longer. Moreover, not only a produced value has to be retained until it is used, but it also has to be removed when consumed by a task. The choice of avoiding shared variables and adopting a model based on communication channels, allows one to overcome many synchronization problems and potential errors deriving from concurrent computation. In this respect, business WfMSs are less suitable to perform sophisticated parallel computations, as data parallelism is not naturally supported and task synchronizations have to be carefully defined.

Moreover, some patterns require that a task is able to access the value contained in a channel in advance, in order to determine the input needed from another channel, and eventually to suspend itself and wait for the necessary data tokens. For instance, this is the case in the dynamic input size pattern (Swp-01), where a task A dynamically determines the number of data tokens to consume from the value received in another input channel. If the number of data tokens currently available is not sufficient, task A is suspended until new data are available. This pattern can be implemented in systems such as Kepler, where a task is not a simple function, but a process with its own state. The ability of a task to suspend itself in order to wait for another input dramatically improves the scalability of a system. To our knowledge, the majority of business WfMSs do not have the ability to

suspend a task instance due to the unavailability of a given mandatory data item. Exceptions are for example FLOWer and COSA.

Despite the possibility of emulating the patterns described in Chapter 5 with complex structures in business WfMSs, the main limitation of these systems is that not all data-flow dependencies can be mimicked via control-flow dependencies, while the reverse is always possible.

Chapter 7
Related Work

In [16] Ludäscher et al. discuss the main characteristics of Kepler and compare these with business WfMSs. From this comparison, the authors draw a set of requirements for scientific WfMSs which have to be integrated in the near future. For instance, they mention the support for user interaction and the separation between the workflow engine and the designer, which are now provided as a single application.

Similarly, in [13] the authors present the distinctive characteristics of scientific WfMSs with respect to business WfMSs, discussing the differences between data-flow and control-flow dependencies, which have been further analyzed in Chapter 2.

In [25] the authors identify seven basic requirements for scientific workflows and define a first reference architecture for scientific WfMSs based on four layers: operational, task management, workflow management and presentation layer. The operational layer consists of a wide range of heterogeneous and distributed data sources, software, tools and their operational environment. The task management layer abstracts from the underlying data sources into data products, and the various software tools into tasks. It aims to provide efficient data management and task execution and it deals with data provenance (i.e. tracking data origins). The workflow management layer is responsible for workflow execution and monitoring. Finally, the presentation layer deals with workflow design, as well as with presentation and visualization of the workflow execution.

A formal definition of the computational model adopted by Taverna is given by Sroka et al. in [26, 27, 28]. The aim of their work is to formally describe the behavior of Taverna workflows in terms of trace semantics and define a notion of observational equivalence among models based on trace equivalence. In particular, in [28] the authors argue that trace equivalence is an acceptable equivalence relation for Taverna, since Taverna workflows are directed acyclic graphs without choices. Most effort is put on the definition of a type system and on the verification of the correct composition of the various modeling components, considering the type (port) interface provided

by the involved components. In this paper we adopted a different approach. We formally described the behavior of the main routing constructs of the analyzed systems using standard Colored Petri Nets (CPNs).The same approach is adopted also for defining the new routing patterns. The difference of approaches comes from the difference of purposes: the aim of the authors was to define a notion of equivalence among Taverna models and formally identify the set of valid Taverna models. Our aim is to provide a formal definition of the behavior of some routing constructs, in a way that they can be easily compared with the constructs provided by traditional business WfMSs and the behavior prescribed by the workflow patterns.

Another attempt to formalize the concept of scientific workflows can be found in [29]. The authors take as starting point the actor-oriented design model adopted by Kepler and provide a formal definition for it. Then, they extend the proposed framework to describe the type system associated with the actors. Based on this formal model, the authors define the *primitive* concepts of a scientific workflow: actor, port, data-flow connection, data-type, actor aggregation and actor refinement. The proposed framework can be used to validate a workflow in terms of type consistency between connected actors, and to find type-conformant actors or sub-workflows for replacement.

In [22] the authors evaluate scientific WfMSs in terms of their Grid support. For this purpose they propose a taxonomy which considers four aspects: (a) workflow design, (b) workflow scheduling, (c) fault tolerance, and (d) data movement. The first aspect is concerned with the ability to represent only DAGs or more general non-DAG models, to define an abstract model that does not refer to the specific Grid resources, and to automatically compose a workflow from the specification of high-level requirements. The workflow scheduling aspect deals with centralized or decentralized scheduling, local or global decision making, and static or dynamic planning . Fault tolerance can be at task level or at workflow level. Finally, data movement is concerned with how remote resources are managed during computation.

In this paper, we provided a comprehensive evaluation of scientific WfMSs based on the workflow control-flow patterns [20], data patterns [23] and resource patterns [24] . The control-flow patterns aim to document fundamental requirements that arise during business process modeling. They describe a set of recurring features that are commonly found in WfMSs for defining the flow of control among various tasks. Similarly, the workflow data patterns describe language features for defining and managing data resources during business process execution. Finally, the workflow resource patterns, characterize the way in which work is distributed to the available resources associated with a process and managed through to completion. In the BPM field, these patterns have been extensively applied to evaluate the suitability of business process modeling languages and their underlying execution environments [30, 31, 32, 33].

In [21], Ustun et al. performed a pattern-based evaluation of Kepler for two types of WCPs: multiple instance and repetition patterns. They conclude

that Kepler is more powerful than business WfMSs in the representation of these patterns, due to the adopted data-flow paradigm. However, they assume the presence of an actor, called `Bundle`, that is not part of the standard Kepler library and cannot be obtained combining existing actors. As highlighted in our pattern-based evaluation, Kepler, and more generally, scientific WfMSs, naturally support the generation of multiple instances of the same task, but do not commonly provide mechanisms to synchronize these multiple instances. Thus, the majority of multiple instance patterns (e.g. WCP-13, WCP-14, WCP-15) are not actually supported by the scientific WfMSs examined here.

A first attempt to define a set of new workflow patterns for scientific workflows is due to Uston et al. In [34] the authors define a set of scientific workflow patterns based on the distinction between control and data tokens and the relationships among them. These patterns differ from the ones identified here, because they rely on the relations between control and data dependencies, while ours describe different ways to combine and prepare the input data of a task. Actually, in scientific WfMSs there is no clear distinction between control and data tokens, because they are all treated in the same way, independently from the fact that the contained values are used for the computation or only for synchronization purposes. Therefore, we do not consider this distinction and concentrate on the features offered by the analyzed systems.

Chapter 8
Conclusion

The contribution of this paper is threefold. First, it provides a precise characterization of the execution semantics of scientific WfMSs through the analysis of three of the most widely-used offerings: Kepler, Taverna and Triana. Second, it uses this characterization to conduct a comprehensive evaluation of the three offerings in question. This is achieved through a pattern-based analysis of the control-flow, data and resource perspectives of these scientific WfMSs using the well-known collection of patterns from the Workflow Patterns Initiatives. Third, it provides a comparative analysis of scientific WfMSs and business WfMSs, and it highlights some difficulties that arise in using business WfMSs for designing scientific workflows. This analysis resulted in the identification of four patterns that deal with advanced data dependencies, which are not part of the Workflow Patterns collection and are not supported by business WfMSs.

The choice of these offerings was motivated by the fact that they are among the most mature and used open-source scientific WfMSs [1]. As such, we believe our findings may also have applicability to other scientific WfMSs.

A possible avenue for future work is to study how the missing control-flow patterns can be integrated in the three considered systems, particularly support for some of the multiple instance and advanced synchronization patterns. A careful consideration though is required as to the extent to which such support is really required in the application domain. Similarly, extended support for the resource patterns could be studied so as to enable a more seamless integration of human agents into scientific workflows. Another opportunity for future work is to investigate to what degree a single system can cater for both the needs of business WfMSs and scientific WfMSs.

References

1. Vasa Curcin and Moustafa Ghanem. Scientific workflow systems - can one size fit all? In *Proceedings of the 4th Cairo International Biomedical Engineering Conference (CIBEC 2008).*, pages 1–9, Cairo, 2008. IEEE Computer Society.
2. Workflow Patterns Initiative. `http://workflowpatterns.com`.
3. The Kepler Project. `http://kepler-project.org/`.
4. Taverna Workbench. `http://www.taverna.org.uk/`.
5. Triana Problem Solving Environment. `http://www.trianacode.org/`.
6. Arthur H.M. ter Hofstede, Wil M.P. van der Aalst, Michel Adams, and Nick Russell. *Modern Business Process Automation: YAWL and its Support Environment.* Springer-Verlag, November 2009.
7. W. M. P. van der Aalst. Three Good Reasons for Using a Petri-net-based Workflow Management System. In S. Navathe and T. Wakayama, editors, *Proceedings of the International Working Conference on Information and Process Integration in Enterprises (IPIC'96),*, pages 179–201, Massachusetts, 1996. Cambridge.
8. Edward A. Lee and Thomas M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83:773–801, 1995.
9. J. Becker, M. Kugeler, and M. Rosemann, editors. *Process Management: A Guide for the Design of Business Processes.* Springer, 2003.
10. Wil M. P. van der Aalst. The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems, and Computers*, 8:21–66, 1998.
11. Shawn Bowers, Bertram Ludäscher, Anne H.H. Ngu, and Terence Critchlow. Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In *Proceedings of the ICDE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow'06)*, Atlanta, GA, 2006. IEEE Computer Society.
12. Ingo Wassink, Han Rauwerda, Paul Vet, Timo Breit, and Anton Nijholt. E-BioFlow: Different Perspectives on Scientific Workflows. In Mourad Elloumi, Josef Küng, Michal Linial, Robert F. Murphy, Kristan Schneider, and Cristian Toma, editors, *Proceedings of the 2th International Conference on Bioinformatics Research and Development (BIRD 2008)*, volume 13 of *Communications in Computer and Information Science*, pages 243–257, Vienna, Austria, 2008. Springer Berlin Heidelberg.
13. Bertram Ludäscher, Mathias Weske, Timothy McPhillips, and Shawn Bowers. Scientific Workflows: Business as Usual? In *Business Process Management*, volume 5701/2009 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2009.
14. Mathias Weske. *Business Process Management: Concepts, Languages, Architectures.* Springer-Verlag, Secaucus, NJ, USA, 1 edition, November 2007.
15. Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems.* Springer Berlin Heidelberg, June 2009.
16. Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 18:1039–1065, 2006.
17. Edward A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001.
18. Thomas Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20:3045–3054, 2004.

19. Shalil Majithia, Matthew Shields, Ian Taylor, and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. pages 514–521, San Diego, California, 2004. IEEE Computer Society.
20. Nick Russell, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and Nataliya Mulyar. Workflow Control-Flow Patterns: A Revised View. Technical Report BPM-06-22, BPM Center Report, 2006. `http://www.bpmcenter.org`.
21. Ustun Yildiz, Adnene Guabtni, and Anne H. H. Ngu. Business versus Scientific Workflows: A Comparative Study. In *Proceedings of the 2009 Congress on Services - I (SERVICES'09)*, pages 340–343, Washington, DC, USA, 2009. IEEE Computer Society.
22. Jia Yu and Rajkumar Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Record*, 34:44–49, 2005.
23. Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In L. Delcambre, C. Kop, H.C. Mayr, J. Mylopoulos, and O. Pastor, editors, *Proceedings of the 24th International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 353–368, Klagenfurt, Austria, 2005. Springer.
24. Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support, 2005.
25. Cui Lin, Shiyong Lu, Xubo Fei, Artem Chebotko, Darshan Pai, Zhaoqiang Lai, Farshad Fotouhi, and Jing Hua. A Reference Architecture for Scientific Workflow Management Systems and the VIEW SOA Solution. *IEEE Transactions on Services Computing*, 2:79–92, 2009.
26. Jacek Sroka and Jan Hidders. Towards a Formal Semantics for the Process Model of the Taverna Workbench. Part I. *Fundamenta Informaticae*, 92:279–299, 2009.
27. Jacek Sroka and Jan Hidders. Towards a Formal Semantics for the Process Model of the Taverna Workbench. Part II. *Fundamenta Informaticae*, 92:373–396, 2009.
28. Jacek Sroka, Jan Hidders, Paolo Missier, and Carole Goble. A formal semantics for the Taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490–508, 2010.
29. Shawn Bowers and Bertram Ludäscher. Actor-Oriented Design of Scientific Workflows. In Lois Delcambre, Christian Kop, Heinrich Mayr, John Mylopoulos, and Oscar Pastor, editors, *Proceedings of the 24st International Conference on Conceptual Modeling (ER 2005)*, volume 3716 of *Lecture Notes in Computer Science*, pages 369–384, Klagenfurt, Austria, 2005. Springer.
30. Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and Petia Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In Markus Stumptner, Sven Hartmann, and Yasushi Kiyoki, editors, *Proceedings of the 3rd Asia-Pacific Conference on Conceptual modelling (APCCM'06)*, volume 53 of *CRPIT*, pages 95–104, Hobart, Australia, 2006. Australian Computer Society, Inc.
31. Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, and Arthur H. M. ter Hofstede. Analysis of Web Services Composition Languages: The Case of BPEL4WS. In Il-Yeol Song, Stephen W. Liddle, Tok Wang Ling, and Peter Scheuermann, editors, *Proceedings of the 22nd International Conference on Conceptual Modeling (ER 2003)*, volume 2813 of *Lecture Notes in Computer Science*, pages 200–215, Chicago, IL, USA, 2003. Springer.
32. Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. On the Suitability of BPMN for Business Process Modelling. In Schahram Dustdar, José Luiz Fiadeiro, and Amit P. Sheth, editors, *Proceedings of the 4th International Conference Business Process Management (BPM 2006)*, volume 4102 of *Lecture Notes in Computer Science*, pages 161–176, Vienna, Austria, 2006. Springer.

33. Petia Wohed, Nick Russell, Arthur H. M. ter Hofstede, Birger Andersson, and Wil M. P. van der Aalst. Patterns-based evaluation of open source Bpm systems: The cases of jBPM, OpenWFE, and Enhydra Shark. *Information & Software Technology*, 51:1187–1216, 2009.
34. Ustun Yildiz, Adnene Guabtni, and Anne H. H. Ngu. Towards Scientific Workflow Patterns. In Ewa Deelman and Ian Taylor, editors, *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science (WORKS'09)*, pages 1–10, Portland, Oregon, 2009. ACM.