Autonomic Workflow Activities: The AWARD Framework

Luis Assuncao
ISEL, CITI/FCT-UNL

Carlos Goncalves
ISEL, CITI/FCT-UNL

Jose C. Cunha
CITI / Dep° Informática FCT-UNL

This article is an extended version of the paper “Autonomic Activities in the Execution of Scientific Workflows: Evaluation of the AWARD Framework” published in Proceedings of the 9th IEEE International Conference on Autonomic and Trusted Computing (ATC 2012), with the following distinct contributions: i) A discussion about the autonomic characteristics of the AWARD model; ii) The support for dynamic reconfigurations illustrated with useful reconfiguration scenarios.

Acknowledgements. Thanks are due to PEst-OE/EEI/UI0527/2011, CITI/FCT/UNL-2011-2012.
Abstract

Workflows have been successfully applied to express the decomposition of complex scientific applications. This has motivated many initiatives that have been developing scientific workflow tools. However, the existing tools still lack adequate support to important aspects namely, decoupling the enactment engine from workflow tasks specification, decentralizing the control of workflow activities, and allowing their tasks to run autonomous in distributed infrastructures, for instance on Clouds. Furthermore, many workflow tools only support the execution of Direct Acyclic Graphs (DAG) without the concept of iterations, where activities are executed millions of iterations during long periods of time and supporting dynamic workflow reconfigurations after certain iteration.

We present the AWARD (Autonomic Workflow Activities Reconfigurable and Dynamic) model of computation, based on the Process Networks model, where the workflow activities (AWA) are autonomic processes with independent control that can run in parallel on distributed infrastructures, e.g. on Clouds. Each AWA executes a task developed as a Java class that implements a generic interface allowing end-users to code their applications without concerns for low-level details. The data-driven coordination of AWA interactions is based on a shared tuple space that also enables support to dynamic workflow reconfiguration and monitoring of the execution of workflows. For evaluation we describe experimental results of AWARD workflow executions in several application scenarios, mapped to the Amazon (Elastic Computing EC2) Cloud.

**Keywords**: scientific workflows; autonomic computing; tuple space; parallel and distributed processing; cloud;
The emergence of e-Science initiatives (Foster & Kesselman, 2003), (Hine, 2006), (NeSC, 2012) in the past decade has shown an increased need to improve the software tools and environments that assist the scientists toward conducting more productive experimentation related to the modeling of physical and virtual phenomena in application domains. Such experimentation involves complex computational processes for simulation, visualization, with access to large data sets, and increasingly requires high degrees of user interaction.

The workflow paradigm was early adopted by business enterprises for partially or totally automating the manufacturing and office processes (Hollingsworth, 1995).

Workflow-based environments provide a useful approach to support adequate abstraction levels for scientists and engineers, and also allow flexibility in the mappings between the application abstractions and the lower layers of the underlying distributed computing infrastructures based on cluster, Grid, and Cloud platforms.

Workflows have been used for the development of scientific applications in a diversity of domains (Taylor et al., 2007), (Yu & Buyya, 2005), (Deelman et al., 2005) and (Chin et al., 2011). Such efforts have been supported by multiple workflow tools (Lazweski, 2011), such as Triana (Triana, 2011), Taverna (Taverna, 2011) and Kepler (Kepler, 2011). Due to the increasing complexity of the applications and the diversity of the execution infrastructures there is still a need for improving the support provided by the workflow tools. In our previous work on a geostatistics application (Assuncao et al., 2009), as well as in other works, such as (Deelman, 2007), (Deelman et al., 2009) and (Chin et al., 2011), some difficulties were identified regarding existing tools:
i) Concerning the application development there is a need for a more clear separation of the specification of the application logic from the details of the workflow enactment. Although there are multiple proposals addressing this issue (Chin et al., 2011), the application developer is often faced with the need to understand and implement details at the level of workflow engine. For example in Kepler, in order to develop a new Actor, the programmer needs to implement methods that are difficult to understand and are dependent on the enactment engine, e.g. `preinitialize(); initialize(); prefire(); fire(); postfire()` and `wrapUp()` (Kepler, 2011);

ii) Concerning the workflow computation, several models allow parallel execution, e.g. by following the Process Networks (PN) model (Kahn, 1974), with a well known semantics (Parks, 1995) as in Kepler (PN Director) (Kepler, 2011) and in (Agun & Bowers, 2011). However, in such systems activities are usually executed as threads within the same process, which then acts as a monolithic workflow engine. Furthermore, a unique central entity, as the Kepler Director, handles the global coordination. There is a clear need for a more decentralized control and coordination of the workflow activities, each designed as an autonomic component with separate control and behavior. This enables a more flexible workflow management, and eases the mappings of the workflow activities across Clusters, Grids, and Clouds;

iii) Concerning flexibility in the workflow management, which is an issue addressed in business workflows (Dadam & Reichert, 2009), (Halliday et al., 2001), (Burkhart & Loos, 2010), it has also been a trend towards enabling more complex scientific experiments. For example, allowing separate parts of a workflow to be launched or controlled individually by different users, and dynamic reconfiguration of the workflows,
in response to changes in the application logic and behavior, e.g., as required in computational steering experiments, or in interactive workflows. Although several approaches have been proposed (Ngu et al., 2008), (Yu & Buyya, 2010), there are still insufficient reports on experimentation illustrating the feasibility of the above dimensions in real applications. We also note that there is still a lack of available, operational working prototypes of easy-to-use workflow tools supporting the above experiments.

In order to address the above issues, we present the AWARD (Autonomic Workflow Activities, Reconfigurable and Dynamic) model. Our goal is to provide a framework used to support the practical evaluation of solutions to the above concerns, in distinct application scenarios. The article illustrates the characteristics of the AWARD framework and the implementation of a working prototype, which may run stand-alone in a single machine, or be launched in a distributed Cloud infrastructure. We describe results of experiments on workflows execution, which are used to evaluate the following AWARD benefits:

i) AWARD follows the Process Networks (PN) model, extended with autonomic processes for modeling the workflow activities (AWA), with decentralized control and allowing flexible mappings to distributed infrastructures such as the Cloud (Amazon EC2);

ii) The AWARD shared space is used for data-driven communication and coordination of AWA activities as well as for enabling dynamic reconfigurations;

iii) End-users do not need to know and implement details related to an enactment engine for developing the application algorithms (Tasks). These tasks are developed as Java classes with a generic interface that encapsulates calls to Web services, or/and run local or
remote processes, allowing legacy applications developed in other languages, e.g., C or Fortran.

This article is organized in the following way. In the next section “The AWARD Model” is described including a working example that illustrates how workflows are specified. In section “AWARD implementation” we describe the design and implementation of the AWARD framework, and the developed associated tools to manage workflow execution. In section “Related Work”, AWARD is compared to related work. In section “AWARD Evaluation” we present results from the experimentation with AWARD and its evaluation in different scenarios. Finally the last section presents conclusions and future work.
The AWARD Model

A workflow in AWARD is defined as a graph of interconnected nodes, where each node is a configurable software component, AWA (Autonomic Workflow Activity) that encapsulates a workflow activity for executing a specific task.

The model overview

Figure 1 illustrates how a workflow graph is mapped to the AWARD model. The workflow flows are transparently supported by a logical shared space (the AWARD Space) that follows the Linda model (Carriero & Gelernter, 1989) where tuples represent communication tokens (data or control) passed between AWA activities.

The flows from the outputs to the inputs of the AWA activities are represented by tuples stored in a global shared space (AWARD Space) that indirectly supports all interactions between the workflow activities. The model assumes that the AWA nodes and their inputs and outputs have unique logical names (string IDs), as keys that are used to transparently index and retrieve tuples from the AWARD Space.

Figure 1. A workflow graph and the corresponding AWARD workflow
A Task is any software component that encapsulates an algorithm to solve a problem. It implements a generic interface that specifies the Task execution entry point,

\[ \text{EntryPoint}(\text{Parameters}, \text{Arguments}) \rightarrow \text{Results}, \]

where Parameters is a list of initial parameters, Arguments is a list of items from the activity inputs and Results is a list of items to be mapped to activity output tokens. Any AWA Task implementing that interface can be executed as a local thread, or as an operating system process in the local host computer, or as a job submitted to a remote cluster, or as an invocation of a Web Service. Furthermore, one entire workflow can itself be encapsulated inside an AWA Task, supporting workflow hierarchies. An important characteristic of the AWARD model is the transparency of Task development concerning the details of interactions involving the AWARD Space. The programmer can develop AWA tasks without any knowledge related to tuple representation and management in the AWARD Space.

As a simple example, the Java class in Figure 2 illustrates a task that adds two integer numbers received as inputs of the corresponding AWA.

```java
public class TaskIntegerAdd implements IGenericTask {
    public Object[] EntryPoint(Object[] Parameters, Object[] Arguments) {
        Integer x = (Integer) Arguments[0]; // from input 1
        Integer y = (Integer) Arguments[1]; // from input 2

        Integer result = new Integer(x.intValue() + y.intValue());
        Object[] Results = new Object[1];
        Results[0] = result; // return a single object
        return Results;
    }
}
```

*Figure 2. An example of a simple AWA task.*

Besides the Task element, a given configuration of an AWA includes the specification of the following elements:
i) A set of input and output ports, where each port has an associated data type, and a state that can be enabled or disabled. For each output port, a list of input ports to where the tokens will be sent;

ii) A configurable input mapping, defining how the Arguments that should be passed to Task invocation, are obtained from the input ports;

iii) A configurable output mapping, defining how the Results from the Task execution are forwarded to the output ports;

iv) A fundamental component of the AWARD model, an Autonomic controller, which controls the life-cycle of each workflow activity, including the control of iterations and dynamic reconfigurations.

Figure 3. Model of an AWA and interactions through the AWARD Space

As shown in Figure 3, firstly, the AWA Autonomic controller gets the matching tuples from the AWARD Space for each enabled input port and passes them to the Task Arguments. Secondly, the software component with the Task implementation is dynamically loaded and its EntryPoint is called. Thirdly, the Results from Task execution are mapped to the enabled output ports of the
AWA activity, and the corresponding output tuples are generated and put into the AWARD Space.

The workflow activities (AWA) communicate asynchronously through the AWARD Space where data or control tokens produced by an activity are stored until the destination activity consumes them. Specific token processing orderings, for example FIFO, can be insured by the identification of each token through a sequential iteration number. Therefore the AWARD workflows are data-driven, allowing each activity to start, run and terminate separately without any centralized enactment engine, and supporting mappings to distributed architectures, for instance Cloud (Amazon AWS).

The Autonomic controller of an AWA activity

Figure 4 depicts the internals of the Autonomic controller: i) A state machine controls the execution of the AWA controller; ii) A rules engine with a knowledge base (a set of facts and rules) evaluates rules that produce new facts; and iii) Event handlers, which handle the requests for dynamic reconfiguration and for information on the current context of AWA activities, such requests being injected as tuples into the AWARD Space.

The rules engine allows to specify conditions to trigger the configuration of the Autonomic controller, encompassing the following functionalities: i) Flexible state machines to control the life-cycle of the AWA autonomic controllers including special states for handling dynamic reconfigurations; ii) Configurable input and output mappings; and iii) Support for multiple iterations of the workflow activities, ensuring that input and output tuples for each iteration are handled in the AWARD Space as distinct.
Dynamic reconfigurations

The AWARD model supports dynamic reconfiguration in several dimensions: i) Each AWA is autonomous, with the workflow activities being launched individually; ii) Each AWA has an independent execution control, allowing its reconfiguration while the others continue running; iii) Each AWA is configurable in terms of the number and state of its ports and their links to other activities; iv) The data flow or control flow interactions between AWAs allow different rates and data granularities.

Dynamic reconfiguration in the AWARD model are supported by a collection of operators available through a Dynamic API (Figure 4) allowing any application tool to perform workflow reconfigurations, which is useful in many application scenarios where the workflow structure or behavior must be changed during execution.

In the following we would like to highlight some real and useful basic dynamic reconfiguration scenarios, which are identified and presented in Figure 5 to Figure 10.
Figure 5. A new activity (f) is added to filter/transform data-flow between activities A and B.

Figure 6. A new activity (f) is added to support monitoring and recording of historical data.

Figure 7. A new activity (f) is added to support feedback dependencies.

Figure 8. A new activity (B’) is added to allow load balancing of activity B.

Figure 9. Change the behavior of B activity by changing the task parameters.

Figure 10. The behavior of B activity is changed with a new Task.
In order to support these workflow scenarios in AWARD, we designed and implemented an API with a set of low level operators as presented in Table I, Table II and Table III. A more detailed discussion of these scenarios and operators is not included, due to space limitations.

**Table I**

*Composition of operators to support a workflow dynamic reconfiguration*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginConfiguration</td>
<td>Specify a starting point of a configuration sequence invoking multiple operators that can involve multiple AWA activities. Returns a configuration identifier (CID)</td>
</tr>
<tr>
<td>EndConfiguration</td>
<td>Ends a configuration sequence by configuration identifier (CID)</td>
</tr>
<tr>
<td>BeginAwaConfig</td>
<td>Begins the configuration sequence of a specific AWA. Takes as an argument the iteration K from which the reconfiguration takes place.</td>
</tr>
<tr>
<td>EndAwaConfig</td>
<td>Ends the configuration sequence of a specific AWA activity.</td>
</tr>
</tbody>
</table>

**Table II**

*Operators to support structural and behavioral AWA reconfiguration*

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateOutput</td>
<td>Create a new Output in a specific AWA activity.</td>
</tr>
<tr>
<td>ChangeOutputLink</td>
<td>Define or change the destination of an Output.</td>
</tr>
<tr>
<td>CreateInput</td>
<td>Create a new Input for a specific AWA activity.</td>
</tr>
<tr>
<td>ChangeParameters</td>
<td>Change the Task parameters of a specific AWA activity.</td>
</tr>
<tr>
<td>ChangeTask</td>
<td>Change the Task of an AWA activity allowing the activity to execute a new algorithm.</td>
</tr>
<tr>
<td>ChangeOutputStrategy</td>
<td>Change the strategy for sending data to an Output. Two strategies are currently allowed: ALL – the output value is sent to all connected destination Inputs; or RoundRobin – The output value is alternately sent to the connected destination Inputs;</td>
</tr>
<tr>
<td>ChangeInputOrder</td>
<td>Change the order to consume tokens from an Input. Data tokens available in an Input can be processed in different orders according to the strategy for sending data from other predecessor activities: Iteration – The FIFO order is based on an iteration number; or Sequence – The FIFO order is based on a sequence number generated by predecessor activities; or Any – Tokens are consumed in any order.</td>
</tr>
<tr>
<td>EnableDisableOutput</td>
<td>Enable or Disable (toggle) an Output.</td>
</tr>
<tr>
<td>EnableDisableInput</td>
<td>Enable or Disable (toggle) an Input</td>
</tr>
<tr>
<td>ChangeMappingInputs</td>
<td>Allows changing the sequence how data Inputs of an AWA activity are mapped to Task Arguments. By default, data from Inputs are mapped to a sequence of objects to be passed as arguments to task invocation, according to the order as the Inputs are defined in the activity.</td>
</tr>
<tr>
<td>ChangeMappingOutputs</td>
<td>According to the generic task interface a task returns a sequence of objects, by default, mapped according to the order how Outputs are defined, allowing changing the order how the sequence of Task Results are mapped to the Outputs.</td>
</tr>
</tbody>
</table>
Table III

Operators to control the AWA life-cycle

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LaunchActivity</td>
<td>Start the execution of a new activity.</td>
</tr>
<tr>
<td>Suspend/Resume</td>
<td>Suspend/Resume the execution of an AWA activity</td>
</tr>
<tr>
<td>Terminate</td>
<td>Terminate the execution of an AWA activity.</td>
</tr>
</tbody>
</table>

In section “AWARD Evaluation” we present an example of a dynamic reconfiguration scenario for performance improvement, involving the operator ChangeTask (Table II) in order to replace a slow algorithm with another one.

The State Machine of the Autonomic controller

Figure 11 describes the main states of the state machine in the AWA autonomic controller, and their main transitions (T) and conditions (C), especially those involved in dynamic reconfigurations.

![Figure 11. The states of the AWA autonomic control](image-url)
When the execution starts, each AWA Autonomic controller subscribes to an asynchronous notification service mediated through the AWARD Space in order to receive the special reconfiguration tuples through the Dynamic Reconfiguration Handler. These tuples carry information to insert facts and rules into the rules engine of the AWA Autonomic controller. The evaluations of those rules force the state machine into the reconfiguration state where the corresponding actions are handled. Each AWA dynamic reconfiguration sequence is composed of a series of operators enclosed by special BeginAwaConfig and EndAwaConfig delimiters, whose execution ensures the atomicity of each reconfiguration sequence. As shown in Figure 11, when the handler receives the BeginAwaConfig operator, it updates the iteration number from which the reconfiguration takes place (UpdateIterationToConfig). The EndAwaConfig operator enables the condition (Cconfig) forcing the state machine to go to the Config state when the specified configuration iteration is reached, corresponding to enabling condition (CiterationToConfig).

All activities start in the Init state and the activity proceeds with normal execution (T1-Cnormal). However, when new activities are dynamically added in some reconfiguration scenarios, an immediate configuration is needed eventually. In such a case, the activity goes to the Wait state (T2-CwaitConfig), waiting for an immediate configuration (T3-Cconfig). This is enabled by condition (Cconfig).

The regular sequence of states per iteration is as follows:

**Idle** – If the transition (T4 - Cconfig AND CiterationToConfig) is enabled, the state machine enters into the configuration state (Config). The condition CiterationToConfig is enabled when the iteration number to apply the reconfiguration is reached. The transition (T5-Csuspend) is
enabled by the *Suspend* operator and leads the state machine to the *Suspend* state. When suspended, an AWA hangs until the *Resume* operator is invoked, which then leads the state machine to *Config* state. The transition (\(T6-C\text{terminate OR CmaxIterations}\)) leads the state machine to the *Terminate* state, when the *Terminate* operator or the maximum number of iterations is reached. The normal transition (\(T7-C\text{normal}\)) leads the state machine to the *Input* state.

*Input* – This state defines the point where the activity waits to get the data from the AWARD space into its inputs, and then maps these data to the arguments that will be passed to *Task* invocation.

*Invoke* – This state creates an instance of the *Task* (dynamic binding) according to the type (class) defined in AWA specification, and invokes the *EntryPoint()* method, by passing the arguments prepared in the *Input* state as well as eventual static parameters defined for the activity.

*Output* – In this state the result returned from *Task* invocation is mapped to the activity Outputs and stored in AWARD space. After this state, the state machine goes to the *Idle* state in order to start the next iteration (\(T8-\text{NextIteration}\)).

*Config* – In this state, the sequences of operators describing requests for dynamic reconfigurations are processed (\(T10-\text{Process Operators}\)), for instance, change the task, create Inputs and Outputs etc.. When the operator *EndAwaConfig* is processed the condition *Cconfig* is disabled and the state machine goes back to *Idle* state (\(T11-\text{CnotConfig}\)).
The autonomic characteristics of the AWARD model

According to the autonomic computing vision (Kephart & Chess, 2003), an autonomic system has self-management characteristics (self-configuration; self-optimization; self-healing and self-protection) allowing the system to make decisions on its own, based on high-level policies and/or it dynamically check its status to automatically adapt itself to changing conditions. An autonomic computing system consists of several autonomic elements, each of them managing its internal behavior and relationships to other autonomic elements based in policies specified by users or induced by the execution environment.

Any autonomic component needs an associated manager (Kephart & Chess, 2003), as presented in Figure 12, and described succinctly as follows. Through a set of sensors the manager constantly monitors and collects events originated internally or in the surrounding environment of the managed element. These events are analyzed and a changing plan enforces the current policies by provoking the execution of self-changes in the autonomic component.

![Figure 12. The architecture of an autonomic component](image-url)
We claim that any AWA activity in AWARD acts as an autonomic component because the AWARD Autonomic controller supports the main capabilities of an autonomic manager as follows:

i) The facts and rules stored in the knowledge base of the rules engine define the states in the life-cycle of an AWA activity;

ii) The asynchronous event handlers act as sensors to monitor special tuples in the AWARD Space, for example to support dynamic self-reconfigurations;

iii) Sequences of these tuples allow preparing a plan to change the structure or behavior of the AWA activity. This plan consists of a set of new facts and rules in the knowledge base of the rules engine.

iv) In the Config state the state machine executes the change plans.

Thus, AWARD workflows have autonomic attributes driven by the self behaviors of each AWA activity, or driven by external tools that inject the adequate tuples into the AWARD Space to produce workflow changes.

For many useful scenarios we insure the consistency of dynamic reconfiguration by relying on the transactional model associated with the specification of the composition operators (see Table I, Table II and Table III).

Nowadays a big challenge is how to unify a large number of distributed autonomic elements into a global autonomic system (Kephart, 2005). In AWARD work is ongoing to face this challenge when dynamic reconfigurations affect a large number of distributed AWA activities.
**AWARD workflow specification**

In order to specify an AWARD workflow, the information that must be provided corresponds to the 2\textsuperscript{nd} column of the table shown in Figure 13. As indicated in 3\textsuperscript{rd} column several items have default definitions.

<table>
<thead>
<tr>
<th>AWA</th>
<th>What user needs to specify</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inputs</td>
<td>Name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State (enable or disable)</td>
<td>Enable</td>
</tr>
<tr>
<td></td>
<td>Token Data Type</td>
<td></td>
</tr>
<tr>
<td>Outputs</td>
<td>Name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>State (enable or disable)</td>
<td>Enable</td>
</tr>
<tr>
<td></td>
<td>Token Data Type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>List of Input names to send tokens</td>
<td></td>
</tr>
<tr>
<td>Inputs Mappings</td>
<td>Function to map input tokens to Task Arguments</td>
<td>Array of tokens</td>
</tr>
<tr>
<td>Output Mappings</td>
<td>Function to map Task Results to output tokens</td>
<td>All Outputs receive results</td>
</tr>
<tr>
<td>Autonomic controller</td>
<td>Facts and Rules to the state machine</td>
<td>Basic AWA rules</td>
</tr>
<tr>
<td></td>
<td>Number maximum of Iterations</td>
<td>1</td>
</tr>
<tr>
<td>Task</td>
<td>Any Java class implementing the generic interface</td>
<td>Utility Task library</td>
</tr>
</tbody>
</table>

*Figure 13. The specification of an AWA*

**Example of a real workflow**

In order to illustrate the specification of a real AWARD workflow we implemented the workflow presented in Figure 14, which invokes the *Protein Identifier Cross-Reference Service* (PICR), described in (Cote et al., 2007) and available in (EBI, 2012). This web service allows access to a mapping algorithm that uses over 70 distinct databases organized as *UniProt Archive* (UniParc) as data source. For each iteration, the AWA PICR invokes the Web Service to map a protein identifier received from its input, producing two outputs: a *UniParc Protein Identifier* (UPI) and the protein sequence. The AWA specification includes a list of initial parameters.
including the URL of the PICR web service, a list of data base names and a taxonomy ID to limit the mappings to Homo Sapiens species.

Figure 14. Workflow that invokes PICR web service to map Protein sequences

Figure 15 illustrates a view of the workflow specification XML file with details of the AWA that invokes the PICR Web service and the corresponding definition of the input and output ports.
Figure 15. Workflow specification detailing the AWA that invokes the PICR web Service
We illustrate the relation between the AWA specification and the programming of AWARD tasks in Figure 16 by presenting the pseudo-code of the task that invokes the Protein Identifier Cross-Reference Service (PICR) Web Service. Note the simplicity in the programming specification to get the arguments and parameters, and the possibility to return many objects that will be mapped to AWA outputs.

```java
public class TaskInvokeProteinsWS implements IGenericTask {
    public Object[] EntryPoint(Object[] Parameters, Object[] Arguments) {
        ProteinID = Arguments[0]; // Protein ID is received from input Pws-IN
        URL = Parameters[0]; // PICR url: http://www.ebi.ac.uk/Tools/picr/service
        ListDBs = Parameters[1]; // list of databases used to map proteins
        TaxonomyID = Parameters[2]; // Limits mappings to Homo Sapiens species
        // Invoke the Web service. Returns null or a UniParc entry
        UPentry = InvokePICRwebService(ProteinID, URL, ListDBs, TaxonomyID);
        Object[] Results = new Object[2]; // Task returns 2 objects
        if (UPentry == null) {
            Results[0] = "UPI-No mappings to protein";
            Results[1] = "Sequence-No mappings to protein";
        } else {
            Results[0] = UPentry.getUPI(); // gets UniParc Protein Identifier (UPI)
            Results[1] = UPentry.getSequence(); // gets Protein Sequence
        }
        return Results;
    }
}
```

*Figure 16. Pseudo-code to AWA task that invokes the PICR Web Service*
AWARD Implementation

The AWARD model is currently supported by a working prototype that is being used to enable experimentation with real applications. It is developed in Java and uses JESS (Friedman-Hill, 2010) as the rule engine. It allows AWA activities to spread over several computers in a distributed environment. The AWARD Space was implemented as a standalone server application based on IBM TSpaces API (Lehman et al., 2001) that can be executed in any computer accessible from other computers where the AWA activities are running. The AWARD Space also supports associated tools to monitor workflow execution and to manage workflow reconfigurations at execution time. The prototype offers a set of tools to launch one or more AWA per computer, to browse and edit workflows definition XML files, and to manage the AWARD space for debugging and monitoring workflow execution, as well as a library of useful AWARD tasks, for instance a simple graphical text viewer to display AWA output results.

As an example, Figure 17 shows these tools in action for executing the workflow presented in the previous section, to invoke the Web Service Protein Identifier Cross-Reference Service (PICR) with 9 iterations. On top of Figure 17, the AWARD workflows tool allows to configure the AWARD execution environment: Directories to load executables and libraries; AWARD space TCP/IP address and port; enable or disable logging and execution times information; Monitoring the tuple spaces to help debugging actions and; Start the workflow execution. Furthermore this tool also supports the execution of some basic dynamic reconfigurations. The AWARD Workflow Browser allows inspecting the workflow specification using a friendly viewing. The two Output windows show the result of the execution of the workflow for 9 iterations with 9 different Proteins IDs.
Figure 17. AWARD tools in action to execute the workflow that invokes the PICR Web Service

For workflow execution in distributed environments, for instance Clouds, the AWARD framework provides a set of associated tools allowing: i) To launch a complete workflow with all AWAs in the same machine; ii) To launch one or more AWA per machine; iii) To monitor the AWARD Space; iv) To dump information from the AWARD Space, for instance concerning AWA execution time and tuples useful for debugging.
Related Work

Both AWARD and Kepler (PN Director) (Kepler, 2011), follow the semantics of Process Networks (PN) (Kahn, 1974). There are important differences between Kepler and AWARD. In Kepler the parallelism is based on the execution of each Actor by threads within the same monolithic process, and actors communicate using first-in-first-out memory buffers. The order of the execution (actors firing) is controlled by a centralized PN Director that can be very inefficient, as it must keep looking for actors with sufficient data to fire: If one actor fires at a much higher rate than another, the actors’ memory buffers may overflow, causing workflow execution to fail (Kepler, 2011). In addition the PN Director does not manage iterations, possibly leading to non-determinism and undefined termination of the execution. For instance in composite actors with workflow hierarchies, if two actors have computation threads, it is ambiguous which actor should be allowed to perform computation (Goderis et al., 2009).

Instead, AWARD activities are encapsulated in parallel processes (AWA) and can execute in distributed environments without a centralized control. Each AWA activity has autonomic control and communicates through a shared tuple space by producing/consuming tokens at different rates without overflow problems. AWARD supports the notion of iterations allowing determinism and well defined termination of the AWA activities. Using tuples for communication improves flexibility, e.g. supporting different granularities of complex data types and dynamic workflow changes are easily enabled by injecting reconfiguration tuples into the shared space. Although aiming at different objectives other works also rely on Tuple Spaces. For example the Workflow Enactment Engine (WFEE) (Yu & Buyya, 2004) uses a tuple space to provide an event-based notification for just-in-time scheduling. In (Heinis et al., 2005) tuple spaces are mainly used to coordinate the requests and notifications events between Navigators.
(workflow engine) and the Dispatchers (Task executors). However none of these works use tuple spaces to support Process Networks (PN) model.

Despite the advantages of the tuple spaces model it also has some disadvantages related to scalability and bottleneck problems (Obreiter & Graf, 2002). To address these problems some works rely in distributed tuple spaces, using caching and or replication techniques. For instance, Comet, a decentralized tuple space (Li & Parashar, 2005), Rudder (Li & Parashar, 2006), (Li & Parashar, 2007) provides a software agents framework for dynamic discovery of services, enactment and management of workflows, where an interaction space is used to coordinate task scheduling among a set of workers. Similar to AWARD space the task tuples contain data items among workflow nodes.

As the AWARD model is orthogonal to tuple space implementation, we argue that it is possible to map the AWARD Space onto the Comet space.

Closer to AWARD in (Agun & Bowers, 2011) a framework is based on persistent queues to support flow between activities, but the workflow engine is monolithic. Like in Kepler tasks are executed as threads, not allowing the execution of workflow activities on distributed infrastructures. Additionally, authors in (Agun & Bowers, 2011) claim that persistent queues can easily support provenance storage. This claim also applies to AWARD, as far as tuple spaces, including IBM TSpaces, also support persistence. In (Fernandez et al., 2011) a workflow system is presented with a decentralized architecture with an external storage (Multiset) as a shared space between workflow activities encapsulating a Chemical engine. Unlike AWARD Space, the Multiset contains coordination information and the workflow definition. Although the goal to support flexibility in business workflows has been a concern since the nineties (Dadam & Reichert, 2009), (Halliday et al., 2001), many issues still remain open, regarding the support for
dynamic changes (Burkhart & Loos, 2010). Such issues are also important in scientific workflows, and some are supported by AWARD. Namely, supporting dynamic behavioral changes, e.g. changing the execution Task and its parameters at runtime is important in scientific experiments where the behavior of algorithms and their parameters are not known in advance. A Kepler workflow (Kepler, 2011) is static and must be completely specified before starting the execution, not allowing changes during run-time. However a prototype implementation based on Kepler (Ngu et al., 2008), proposes a frame abstraction as a placeholder for actors to be instantiated at runtime (dynamic embedding), according to rules defined at design time. This approach can be compared to our proposal to change dynamically the algorithm of an activity. However AWARD is more flexible because we do not need to specify the alternative algorithms at design time as we allow to dynamically changing them by injecting tuples with the new Tasks. In the GridBus workflow engine (Yu & Buyya, 2010) the user can either specify the location of a particular service at design time, or leave it open until the enactment engine identifies service providers at run-time. This is easily supported in AWARD: If the location of the service (URL) is a parameter we can dynamically reconfigure the parameter by changing the service provider, or we can inject rules into the control unit of an AWA activity in order to search for service providers in any service directory.
AWARD Evaluation

AWARD allows executing workflows on heterogeneous environments. For instance, AWARD workflows have been executed in local area networks involving Windows and Linux computers. We describe our experiments on Amazon’s EC2 infrastructure (Amazon.com, 2012). The mapping of AWARD components (AWA and AWARD Space) onto the Amazon EC2 infrastructure is presented in Figure 18.

There is one dedicated EC2 instance to host the AWARD Space server, accessed from other EC2 instances through the TCP protocol. The AWARD Space server can be monitored and controlled from anywhere outside the Amazon cloud using the HTTP protocol and using Web interfaces to allow an end-user to follow the execution of the workflows.

End-users access the EC2 instances using any SSH client (we used PuTTY) and all data files (xml workflow definition files, input/output data files) are transferred to the shared storage using any ftp client (we used WinSCP). The shared storage is an EC2 volume mounted in all EC2 instances.

Figure 18. Mappings to execute AWARD workflows on Amazon EC2
All EC2 instances used have the following characteristics: Amazon Linux AMI 64 bit, 4 CPU (2 virtual cores with 2 EC2 Compute Units each), 7.5GB memory, 8 GB root device, and a 8 GB shared volume. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor (Amazon.com, 2012). To evaluate AWARD we defined a set of scenarios in order to answer the following questions: 1) What are the overheads for executing an AWARD workflow comparing with the execution of a similar workflow in Kepler, including dynamic reconfigurations to enable optimizations during the execution of long-term workflows (hundred or thousand iterations); and 2) What is the flexibility of AWARD: i) Concerning parallelism in workflows with a large number of activities to be launched and controlled by different scientists without a centralized control in distributed environments e.g. as a Cloud; ii) Concerning easily changing the mappings from the workflow activities to the virtual machines without any modification to the workflow specification.

**First scenario:** This involves the execution of a similar workflow in AWARD and Kepler as presented in Figure 19. The two Ramp activities generate a sequence of numbers in parallel, until a maximum number of iterations. These numbers are added in parallel and the results are multiplied. The Output activity writes the results to a file. The AWARD Add1 and Add2 activities have a Task with a sleep time of 1 second to simulate a long execution time. To achieve the same behavior in Kepler we customized the necessary Actors, in order to have the same sleep time, to manage the workflow iterations, and to report the execution time per iteration in a similar way as AWARD. In AWARD we log the execution times into the AWARD Space and in Kepler we log them into a file.

In Kepler we used a single AWS EC2 instance, and considered three cases for AWARD: In case 1 we ran the workflow using a single EC2 instance, including the AWARD Space server.
In case 2, two EC2 instances, one to host the AWARD Space server and the other to execute the workflow; in case 3, four EC2 instances, one to host the AWARD Space server and three EC2 instances to distribute the activities according to the workflow partitions p1, p2, p3 as shown in Figure 19a). In case 4, we used four nodes of a small dedicated cluster with a configuration similar to case 3, one node to host the AWARD Space and three nodes to launch the partitions p1, p2, p3 as shown in Figure 19a).

![Diagram](image1)

**Figure 19. An equivalent workflow in AWARD and Kepler**

The average execution time per iteration is shown in Figure 20 for executions with 1, 10, 50, 100, 500 and 1000 iterations. These results confirm that AWARD has some overheads compared to Kepler, which uses a dedicated thread for each Actor inside a monolithic process, whereas AWARD uses an independent process for each AWA. However for long-term workflows (with thousands of iterations) as shown in Figure 20, AWARD has small overheads compared to Kepler allowing us to conclude that AWARD becomes adequate to execute this class of workflows.
One should note that the execution time gets smaller for executions with multiple workflow partitions (case 3 and case 4), even for lower numbers of iterations, although the execution time reduction is not so strong when the iterations increase up to the thousands. This is explained by the cost due to tuple persistence and tuple matching when the size of the tuple space grows. In case 3 with multiple EC2 instances we also observed that Amazon EC2 instances have a high variability and fluctuation in performance as also reported in (Schad et al., 2010).

This simple workflow also allows us to illustrate the dynamic reconfiguration capabilities related to performance optimizations. As shown in Figure 20 the iteration execution time is always greater than 1 second due to the delay forced in the two Add activities. Unlike Kepler, in long-term workflows AWARD allows to change parameters or the Task (algorithm) of any activity. As an example we can change the Task of Add1-1sec and Add2-1sec activities to new algorithms with optimized performance.
Figure 21. Dynamic reconfiguration to reduce execution times.

Figure 21, illustrates the result of workflow execution when we dynamically change the two Add tasks with a new algorithm that is optimized for a delay of only 0.5 seconds after the 50th iteration. As shown, the average iteration execution time decreases after the 50th iteration leading to a reduction of the overall execution time. This is easily achieved by a simple command that injects the operator ChangeTask as a tuple in the AWARD Space. This tuple is handled by the Dynamic Reconfiguration Handler of the autonomic controller of the Add AWAs. Although this example is minimal and very simple it clearly demonstrates the AWARD capabilities to perform dynamic reconfigurations.

Second scenario: This uses a simulation of a Montage workflow as in (Deelman et al., 2005) and (Juve et al., 2010). For our experiments we used the workflow in Figure 22 with 24 activities and the execution time of each activity based in (Deelman et al., 2005). This shows how AWARD supports scenarios where multiple users can execute a large workflow interactively executing their specialized activities.
We experimented with several ways to launch this workflow: Launch all activities in the same EC2 instance or launching different partitions by different users on multiple EC2 instances. Figure 22 shows three possible partitions (p1, p2, p3) launched by 3 users in 3 different EC2 instances concluding on the AWARD flexibility to execute workflows with large number of activities that can be separately launched.

The number and structure of partitions is a complex issue and very application dependent because the dependencies on the workflow structure and related to data flows between workflow activities. Besides, the assignment of workflow partitions to computing nodes is another complex issue related to resource scheduling management, which is a NP-complete problem (Fernandez-Baca, 1989) , (Mohammadi Fard et al., 2012). We consider that these issues are
orthogonal to our work, leaving the decision for application developers to develop heuristic techniques for finding the more adequate solutions for these issues.

**Furthermore in the second scenario:** We experimented with a Text Mining application to extract relevant expressions (RE) from text documents. Such expressions, called *n*-grams, where *n* is a number of words, are sequences of words with strong semantic meaning, such as "parallel processing", "global finance crisis". The *LocalMaxs* algorithm supports a language independent approach for extracting RE (Silva et al., 1999). Apart its good precision, it has a low performance in terms of execution time, when executed sequentially, so a parallel implementation of this application has been developed using distributed tasks. For a problem instance with *n*-grams for *n*=6 the algorithm is modeled as the workflow presented in Figure 23. The activity *Start* defines a starting point to define the initial execution time and enabling the execution of the activities of phase 1 (1-1 to 1-6). The phase 1 of the algorithm counts all the occurrences of the *n*-grams that exist in a given corpus. The phase 2 of the algorithm calculates the glues of each *n*-gram that exist in a given corpus. The glue of a *n*-gram can be viewed as a cohesion metric, sticking the words together within the *n*-gram. Different *n*-grams usually have different cohesion values. One can intuitively accept that there is a strong cohesion within the 2-gram “European Union” or within the 5-gram “European Court of Human Rights”. The phase 3 of the algorithm finds all *n*-grams that can be considered relevant expressions. To decide if a *n*-gram is a relevant expression (RE) we need the value of the glue for all contiguous (*n*-1)-grams contained in the *n*-gram and the glue for all contiguous (*n+1*)-grams that contain the *n*-gram.
The activities RE3, RE4 and RE5 are used to store the relevant expressions that were found. Finally the activity End is used to record the completion time of workflow execution.

The partitions (p6-1…p6-6) shown in Figure 23, illustrate one possible set of parts of the workflow that can be executed separately and are defined by the Text mining application developer.

Note that the algorithm tasks that are encapsulated in the AWA activities were developed by other users previously in the C++ language, without any concern to the AWARD framework, and their integration into this workflow was performed by someone who is not a AWARD developer. As part of ongoing work we are developing a large scale workflow for this text mining case study using AWARD. This example is only briefly presented here to demonstrate the AWARD capabilities for real application scenarios.
Figure 24a) presents the execution times of this workflow using different approaches, when considering a single EC2 instance: i) using a shell script to sequentially start the workflow activities as processes; ii) using Kepler, respectively, under SDF and PN Directors; iii) using AWARD.

Figure 24b) shows the execution times of the AWARD workflow using 2, 3 and 6 partitions with one EC2 instance for each partition. Note that this scenario is not possible when we use Kepler or shell scripts.

The lessons learned from the above experiments are threefold: i) AWARD is clearly better than the sequential executions (shell script and Kepler SDF Director); ii) AWARD reveals a not significant overhead comparing to the Kepler PN Director; iii) The advantages of AWARD to execute workflow partitions in parallel using multiple computing nodes are clearly illustrated.
Conclusions and Future Work

This article presents the AWARD framework and an evaluation focusing on its benefits: i) Autonomic workflow activities (AWA) with decentralized control that can run in distributed architectures; ii) A shared Tuple Space for data-driven communication and coordination; iii) Development of application algorithms, easily integrated as workflow tasks, without requiring detailed knowledge of any enactment engine; iv) Dynamic reconfigurations of long-term workflows with multiple iterations; v) AWARD is supported by a working prototype that can be deployed on a single machine as well as on a distributed infrastructure, as the Amazon EC2 Cloud. Regarding i) we conclude that AWARD eases the experimentation with different partitions of the workflow activities and their mappings to the cloud environment. In addition, workflows with large number of iterations do not incur significant overheads. Regarding ii) we conclude that the AWARD Space is flexible to support data-driven coordination between activities. Furthermore its benefits include the possibility to logging runtime information that allows monitoring and managing workflow intermediary results. Regarding iii) we conclude, from our experience with the Text Mining workflow, that it is easy for other users to specify AWARD workflows and develop Tasks, involving the use of legacy code, without any knowledge of AWA internal functionality. Regarding iv) we present set of useful scenarios to illustrate the AWARD advantages to support dynamic reconfigurations. Regarding v) we confirm that all AWARD associated tools are running in EC2 instances without requiring any modification.

In ongoing work we are using AWARD to experiment with workflow scenarios that can benefit from using AWARD operators to dynamically change the workflow structure and behavior applied to real applications. Recently the AWARD framework was used to implement a
flexible *MapReduce* workflow supporting the *MapReduce* programming model, and its evaluation with a real text mining application (Goncalves et al., 2012).
References


