Universal Resource Lifecycle Management

Marcos Báez, Fabio Casati, Maurizio Marchese

#Dipartimento di Ingegneria e Scienza dell'Informazione, University of Trento
Via Sommarive, 14 38100 Trento (Italy)

[1]baez, casati, marchese@disi.unitn.it

***Abstract*— This paper presents a model and a tool that allows Web users to define, execute, and manage lifecycles for any artifact available on the Web. We show the need for lifecycle management of Web artifacts, and we show in particular why it is important that non-programmers are also able to do this. We then discuss why current models do not allow this simplicity, and we present a model and a system implementation that achieves lifecycle management for any URI-identifiable and accessible object. The most challenging parts of the work lie in the definition of a simple but universal model and system and in the ability to hide from the lifecycle modeler the complexity intrinsic in having to access and manage a variety of resources, which differ in nature, in the operations that are allowed on them, and in the protocols and data formats required to access them.**

I. Introduction

This work introduces concepts, methods, and a system for universal resource lifecycle management. Nearly every artifact, from web pages, documents, wikis, code, to non-software resources (houses in construction, purchase orders, etc.) goes through a lifecycle. In a few cases, the lifecycle of these artifacts is supported by a tool that allows their modeling, automation, monitoring, and management. This typically happens when the lifecycle is formalized and strictly followed. For example, the process of approving purchase orders and procuring the goods is, in some large companies, supported by a workflow management system. In these cases, a system can interpret a formal definition of the lifecycle and execute/enforce it.

In the majority of cases however, the lifecycle is informally defined, and is executed, monitored, and managed "by hand", if at all. This is because generic process management tools are too complex and too rigid for this purpose, and are tightly coupled with the artifact they manage. For example, consider the execution of a software project, which includes the development and delivery of documents and code. The code is usually managed through a source control system, while the documents can be developed collaboratively online via the likes of Google docs[1] or Zoho[2]. For each type of artifact, the team often defines a "quality plan" along with the lifecycle that the artifacts should follow. For example, design documents should be first reviewed and discussed by the development team, then reviewed by and discussed with the chief architect, and then signed off by the project manager. A unit manager, architect, or project manager, would like to know at a glance which documents are in a given status, which are late, and which have issues that need special attention. A team member/developer, would like to visualize the lifecycle of the documents he is in charge of, so that he knows what he is supposed to do with the document, and to automate the process of making it available to the team, sending it for review, collecting the reviews, sending it to the chief architect after revision, getting it signed off, and so on.

Today these types of lifecycles are modeled informally (sometimes even verbally) and they are mainly executed by hand typically by sending emails and editing access/visibility rights. The status is typically tracked by updating a MS project document or some spreadsheet.

Process and lifecycle management in these cases, using tools such as workflow managers, is unfeasible. First, the team would have to learn yet another tool, characterized by models (e.g., workflow models) typically fairly complex and anyways, despite marketing claims, targeted at programmers, not at users such as project managers. Second, the majority of everyday lifecycles are unstructured and flexible, and traditional workflow systems are not good at this (we will discuss this in detail in the related work section). Third, the progression through the lifecycle is often controlled by a human based on his /her judgment, not by an engine based on pre-defined rules. It is the developer, team leader, or project manager, who decides when the artifact can move to the next step of the lifecycle and which is this next step. Fourth, the decision of what to do at a given step in the lifecycle may itself change over time rather than being predetermined. For example, I may want to send the document to two rather than three reviewers, and decide who the reviewers are on the fly, or I may decide to post it and allow (i.e., set access rights so that) all my team to enter review comments. Fifth, in real projects typically there are a set of different kinds of artifacts (code, web pages, documents, etc) managed with different tools (CVSs, Web text editors, etc), distributed across the organization and managed by different owners. Using different lifecycle management tool for each of these would be practically unthinkable.

This paper proposes an abstractions framework and a supporting environment that overcome these limitations and enable universal resource lifecycle management. We use the terms "universal" and "resource" as we want the system to manage whatever can be identified by an URI, regardless of its nature, managing application, owner, or location. We realize that such universality can often be at odds with ease of use, and indeed this is one of the challenges we face and address. The main characteristics of the proposed approach, also corresponding to the main contributions of the paper, are the following:

[1] http://docs.google.com
[2] http://www.zoho.com

- The system is targeted at advanced web users (e.g., users comfortable with writing on wikis), not only programmers. The lifecycle model is very simple, essentially based on state machines. There are no complex features such as path conditions, transactions or exceptions.
- There is no need for modeling the resource being managed and its properties. The resource can be a "black box" from the lifecycle perspective. This is key both to universality and to keep the model simple from the perspective of the lifecycle designer who does not need to worry about the specifics of each resource.
- We support automation of operations on the resources (e.g., changing access rights, submitting for reviews, etc.), achieved by *actions* that can be associated to *phases* (states) and executed upon entering a phase. Actions are where both the complexity and the resource type-specific behavior reside (e.g., sending a Google doc for review also requires setting access rights, and the way this is done is Google docs-specific). They are written by programmers, who populate a library of useful actions.
- The model is targeted at unstructured lifecycles, where there is a high potential variability and the need to place the human in the driver's seat. For example, the lifecycle owner can determine when the resource transitions to the next phase or which is the next phase among the possible ones.
- The lifecycle management tool is hosted and available as a service, together with the lifecycle design interface and the *monitoring interface*, i.e., the interface a project manager would use to visualize status and history of the resources under her responsibility.

In the following we describe both the model and the prototypal system (named *Gelee*) in detail, together with the reasoning behind the various choices. We do this by starting from a concrete example (which is also the reason why we started developing this system), and extracting and abstracting requirements from it. Then, after discussing and comparing with the state of the art, we detail the model, the Geele system architecture, implementation, and validation. We then discuss possible extensions and how these can be applied.

II. MOTIVATING SCENARIO

A. EU Projects

At the heart of our interest in this problem was the participation in several European Union (EU) projects and in particular one in which we act as coordinators, called *LiquidPub*[3]. So, we use this as a case study. EU projects involve people from different organizations working collaboratively (a project consortium) to achieve a project goal. EU projects are typically organized in *work packages*, each including tasks, deliverables, and milestones. Each of these has owners and collaborators (usually expressed as consortium partners, not people), and deadlines. In this motivating scenario we focus on deliverables. A project has a number of deliverables ranging from 20 to 40 or more, depending on the size. In Liquidpub we have 35.

EU project coordinators typically define "quality plans" for deliverables, outlining essentially a desired lifecycle for them. This adds to the rules (and hence parts of the lifecycle) defined by the EU itself. For every deliverable there are one or more responsible parties playing different roles, with different levels of visibility or access rights. Moreover, each deliverable has its own lifecycle, which is comprised of different steps involving different activities and people.

For example, consider a typical scenario involving the production of a "State of the Art" deliverable. In the early phase of its elaboration, there is a small group of people sharing a document (maybe using Google Docs or a Wiki) in which they define the document structure and collaborate on specific sections, providing access rights as needed. Then, at some point (informally or formally defined as part of the quality plan) the document is shared with a wider group of people (specific reviewers, or the project team at large) to get feedbacks. The iteration of the elaboration and review phases continues until reviewers are satisfied. At this point the draft is transformed in the appropriate format, sent to the funding agency (EU in or case) for evaluation before a specified deadline, and possibly published on the project web site (either immediately or after EU approval). Very often, the work on the document continues, for example to prepare a survey paper for a journal.

The above represents an ideal scenario. Internal deadlines can be missed, reviewers can be changed, phases can be shortened or skipped to make it in time, different deliverables can be merged into one or vice versa, etc.

B. Problem and Requirements Abstraction

From the above scenario we generalize requirements and desiderata for two classes of people involved in the project: *project managers* (who define the lifecycle, e.g., the project coordinator in our example) and *artifact owners* (who are responsible for driving the execution on an artifact, e.g., the responsible of a deliverable). If we take the perspective of project managers – people responsible for managing a relatively large set of artifacts - we would like to:

1. Define the lifecycle of the different artifacts (we use the terms *artifact* and *resource* interchangeably). For example, define the quality plan that describe what every deliverable should go through (see e.g., Fig. 1),
2. Associate the lifecycle to resources, possibly customizing it as needed for the resource (some deliverable may require specific treatment, for example our *state of the art* deliverable that was developed by integrating pieces done by the various project partners).
3. Avoid – as much as possible - the concerns of resource-specific details. We don't want to define different models based on whether the deliverable is done with Google Docs, or latex over Subversion.
4. Monitor lifecycles. We (as project managers) would like to be able to have a picture of the status of the lifecycle

[3] http://project.liquidpub.org

for each artifact at any given point in time, with particular attention to delays.

5. Simplicity. The user is the average scientist doing research, not a programmer. These kinds of users should be able to define, execute, and monitor the lifecycles.
6. Flexibility and robustness. The web has taught us that things that work well are not only those that are simple but also those that are robust to failures or imprecision. Ideally it should be possible for the lifecycle to be partially specified and still be usable and useful for managing the artifacts' evolution.

If we take instead the perspective of the artifact owner, we identify the following requirements:

1. The owner should be able to go through the lifecycle, advancing from a phase to the next, and while doing so, (automatically) initiating and executing the necessary actions.
2. The execution should be independent on the specifics of the resource. For all lifecycles, owners "simply" have to decide when they are ready to progress to the next phase.
3. The abstraction and interfaces should be simple and integrated with the tool managing the resource, to simplify usage.
4. The owner should have the possibility to deviate from the prescribed lifecycle. Changes (such as skipping a formal internal review due to delays) are the norm and imposing a fixed model would make the tool and abstractions useless. Furthermore, some parts of the lifecycle may be left to be decided by the owner or may have been unknown/undecided at lifecycle definition time. This means that the lifecycle for each object is only loosely defined beforehand.

Today, resource lifecycles in contexts like project executions are in the vast majority of cases managed by one tool: Microsoft Project. The reason is simple: MS Project is intuitive and imposes little or no unnecessary overhead. The challenge that is laid out for us therefore is to provide a way to facilitate the definition and execution of lifecycles and the management of the various artifacts and their progress while achieving to the possible extent a level of simplicity, flexibility, and intuitiveness similar to that of MS Project.

III. Related Work

A. Workflow Management Systems

Workflow systems allow the definition, execution, and management of workflows. In general, workflow systems describe a business process as a set of tasks, to be executed in the order defined by the model. They are related to our work since they describe a flow model and actions to be executed on objects. They are however different since i) they do not focus on lifecycle management (they do not focus on the evolution of an object, but rather they model arbitrary actions to be executed by human or automated resources), ii) they are fairly rigid and prescriptive (they work well for structured, repeatable processes), iii) they are targeted to programmers and often designed for mission-critical applications (in fact they are not significantly less complex than Java for example), and iv) the corresponding software platform is large and complex to operate and maintain.

Interesting lessons can however be learned by looking both at research in workflow evolution and adaptive workflow and at research on semi-structured workflow models, including in particular scientific workflows that are targeted at scientists. In the area of adaptive workflows, several approaches have been proposed to provide dynamic process management [1][2][3], mostly focusing on managing migration of instances when the corresponding model is changed. In this paper we approach the problem by decoupling (or as we define later, *light-coupling)* instances and models, and automated migration is not required – also because the progression of the flow is always done by humans.

A similar approach to the flexibility we offer in the lifecycle management is provided by the PROSYT system [4]. PROSYT takes the artifact-based approach in which operations and conditions for these operations can be defined over the concept of *artifact type*. Nonetheless, each artifact type defines just one possible lifecycle, and runtime lifecycle model changes are not allowed. In contrast, our approach provides independence from the resource being managed (universality), late binding of phases, actions, and resources, and we focus on simplicity in the model and system due to the nature of our target users.

Finally, *scientific workflows* were developed for environments in which experiments need to be conducted. Experiments can be considered as sets of actions operating on large datasets [5]. Due to the nature of the environment, it is not often possible to anticipate a scientific workflow, so model-changes and user intervention at runtime are necessary to provide flexibility. Other requirements like reproducibility, detailed documenting and analysis are also of concern. The main difference is that scientific workflows focus on workflows more than lifecycles of artifacts and that require a level of expertise which is well beyond the level of complexity of activity diagrams, MS project, and in general the target complexity level we have in Gelee.

B. Document Management

The approach introduced in this work has roots also in the document engineering community. In this area, models and tools are developed around the concept of documents, which are particular types of resources. In [6] the notion of document-centered collaboration is introduced. There, the activities of collaboration and coordination are considered aspects of the artifact rather than workflows. For this, they attach computation to documents (i.e., a word processor), whose actions define the workflow. The approach is focused on decoupling documents from workflows rather than providing a workflow or lifecycle modeling approach. In essence, this idea of separating the artifact from the workflow is aligned with our idea of decoupling artifacts from lifecycles, but in addition, we build a flexible, reusable and simple lifecycle management model on top.

An interesting framework for document-driven workflows was proposed in [7], which requires no explicit control flow. In this approach, the boundary of the flexibility is defined by the dependency among documents (in terms of inputs and outputs). Nevertheless, as workflow operations are associated to changes in the documents, these changes must be done under the control of the workflow. In our approach, the lifecycle operations are associated to transitions, not to changes in the document. Thus, artifact processing (i.e., editing a Google Doc document) is separated from the model.

In [8], the processing of artifacts, from the creation to completion and archiving, is captured by lifecycles. Nonetheless, the flexibility offered is more focused on the artifact representation rather than lifecycle evolution and execution. Our model aims instead at providing flexibility in the lifecycle modeling and execution, and decoupling among lifecycle models and instances.

C. Lifecycle Modeling Notations

At present, there are a variety of models, notations, and languages for describing lifecycles. The most popular class of models is UML, and within UML the most common approach is to model lifecycles using state machines, that have exactly the purpose of modeling the state and evolution of an object, and the events that cause state transition [9]. State machines have been extended in a variety of ways, e.g., by allowing guards to be placed on transitions, to associate actions to transitions (statecharts [9]), and the like.

We essentially reuse finite state machines as the base for the lifecycle model we propose. The contributions of Gelee are not so much in the basic model, but rather in the instantiation and execution model, in the light-binding (described next) between models and instances and in how we cope with the heterogeneity of the possible resources to be managed and correspondingly with the different kinds of actions they support.

Other notations have been used to model lifecycles and processes. The most common ones are Petri nets and activity diagrams and their variations and extensions (which include also workflows and service composition notations such as BPMN [10]). We did not base our implementation on these notations as we find them more appropriate for describing workflows and procedures (generic sets of actions to be executed according to some ordering constraints) more than lifecycles (evolution of the state through which a resource goes through, and allowed actions in each state). In any case the essence of the differences of URLM would still lie in the aspects mentioned above, not so much in the base notation.

IV. Concepts, Models, and Languages

In the following we first describe the lifecycle model at a high level, then discuss its execution semantics in terms of overall lifecycle executions and action executions.

A. Lifecycle model: basics

In essence, a resource lifecycle is a set of phases and phase transitions. In this, it is analogous to state machines and state charts. The phase describes the stage in life in which the resource is, while transitions denote possible evolutions. At any given moment, a resource is in one and only one phase. Fig. 1 illustrates all the elements of the lifecycle with our example of Section 2.

At the lifecycle level, all the model needs to know of the resource is its URI and its *type*, a string whose main purpose is to denote which is the managing application. For example, resource types can be Wiki page, Google doc, Zoho project, SVN repository, etc. If the resource is password-protected, the model will also need login information. No other information is needed for the lifecycle to be able to manage the resource.

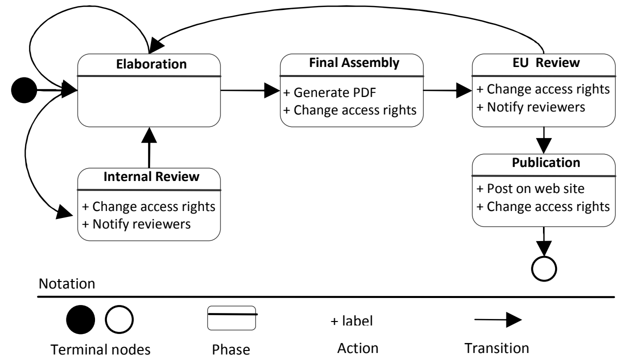

Fig. 1–EU Project deliverable lifecycle

Phases can have associated *actions*. Actions are operations that are executed on the resource as the phase is entered. Examples of actions are: changing access rights, notifying reviewers, etc (see Fig. 1). Actions have parameters which are typically instantiated as a lifecycle begins. For example, “notify reviewers” could have as parameter the “reviewers list”, which is an information we could have or not beforehand. At the lifecycle model, neither the lifecycle composer (the one designing the lifecycle) nor the resource lifecycle owner (the person(s) in charge of advancing the lifecycle on a specific resource) needs to be concerned with how they are implemented.

All actions associated to a phase are executed in parallel and anyway in a non-deterministic order. Any sequencing must be imposed by splitting the phases. They are not guaranteed to succeed and there is no transactional semantic imposed by the model (nothing prevents the action itself, inside its implementation, of having a transactional behavior). The *expected* behavior is that when the actions complete, the lifecycle owner may advance the lifecycle to the next phase (details on how this occurs are provided below).

While the lifecycle model is designed by a non-programmer, actions are meant to be developed by programmers. Actions have a type that is resource independent (e.g., change access rights), and an implementation that is resource-specific (change the access rights to a wiki page or to a flickr stream). Notice that in this way the actions hide the specificities of each resource type, and this allows the same model to be applied to resources of different types.

When designing a lifecycle model, lifecycle composers can select the actions from a library. Since not all action types

make sense for all kinds of resources, and since not all action types may have an implementation for a given resource (e.g., we may not have implemented the *changeAccessRight* action for google docs), the actions they select may impact the resource types to which the lifecycle can be applied. Thus, models referring to resource-specific actions will have more limited applicability.

Executing actions is not the only purpose for having phases in a model. It is perfectly reasonable (and indeed useful, as we experienced) to have "empty" phases considering that one of the main purposes of lifecycles is also monitoring. For example, if the "Elaboration" phase in Fig. 1 involves editing a document in Google Docs we may still want to show that the current phase is "Elaboration", even if there is no action executed from the lifecycle. The model includes several other features not discussed in detail here, such as deadlines and time constraints as well as annotations. Annotations are in particular used to explain why a lifecycle owner does not follow the standard flow, as discussed below.

B. *Lifecycle Execution*

A lifecycle instance is a particular execution of a lifecycle on a given resource. When the lifecycle instance begins, the lifecycle is associated to a specific resource and actions can be configured if necessary (e.g., reviewers email addresses are inserted). The lifecycle remains active until an end phase is reached. End phases are phases with no associated actions, and their purpose is only to denote that the lifecycle instance is complete in a certain final state.

In Gelee there is no analogous of a workflow engine. The engine is the lifecycle owner, who executes the lifecycle instances (i.e., moves the tokens from phase to phase) and, while doing so, initiates the execution of actions. This allows the human to be in control, while keeping the infrastructure very simple and lightweight.

Another important aspect is that the model is *descriptive* rather than *prescriptive*. Its purpose is to describe a desired lifecycle (and the associated actions), not to impose it. In fact, the lifecycle owner can at any time move the token to any phase (even if this would not be allowed by following the state machine). One can argue that the model could include mandatory transitions or actions, but this is one of the many instances where we had to veto our desire to add features for the sake of keeping the model as lightweight as possible for lifecycle owners and designers.

Finally, lifecycle owners can change the lifecycle followed by a resource, in other words they can change the model associated to a lifecycle instance.

The above denotes a *light-coupling*[4] between models and instances. Owners *can* change the lifecycle of a resource without changing the model, and designers *can* change the model without affecting running instances if they so desire. If designers change a lifecycle model, they can request to propagate the change to running lifecycles. Upon receiving the request, lifecycle owners can accept or reject the change, and if they accept, they can indicate in which phase the lifecycle instance should end up in the modified model. Therefore, even in the presence of change, the problem of instance migrations is here reduced to state migration. In terms of the lifecycle definition, the light-coupling between model and instance means that the XML that describes the lifecycle definition is self-contained.

A similar light-coupling exists between lifecycles and resources: nothing prevents several lifecycle to be defined on the same URI, and nothing prevents several lifecycle instances on the same URI to be running. Overall this makes for a very flexible model, with the intent of giving a flexibility close to that of MS Project kind of tools while allowing lifecycle modeling and automation of actions.

Taking our example of Fig. 1, in Table I we give an example of a lifecycle model definition interpreted by Gelee.

TABLE I
EXAMPLE OF A XML DEFINITION OF THE LIFECYCLE

```
<process uri="">
<name>EU Project deliverable lifecycle</name>
<!—Information about the version-->
<version_info>
 <version_number>1.0</version_number>
 <created_by>lpAdmin</created_by>
 <creation_date>08/07/2008</creation_date>
</version_info>
<!—List of suggested resource_types-->
<resource>
 <resource_type>MediaWiki page</resource_type>
</resource>
<!—Definition of the phases-->
<phases_list>
 <phase id="elaboration">
  <name>Elaboration</name>
 </phase>
 <phase id="internalreview">
  <name>Internal review</name>
  <!—Actions to be executed -->
  <action_call>
   <action>
    <name>Change access rights</name>
    <uri>changeAccessRights</uri>
     <parameters>
     <!—Parameters to be specified at design-->
      <param id="paramID"> value </param>
     </parameters>
   </action>
    ...
  </action_call>
 </phase>
 <phase id="finalassembly">
  <name>Final assembly</name>
  ...
 </phase>
 ...
</phases_list>
<!—The list of suggested transitions-->
<transition_list>
 <transition>
  <from> BEGIN </from><to>elaboration</to>
 </transition>
  ...
</transition_list>
</process>
```

[4] We use the term *light-coupling* instead of *loose coupling* to emphasize the difference with respect to the traditional usage of the loose coupling term in the web service context, which refer to the client not being hardcoded to interact with a specific service.

C. *Action Execution*

The same compromise between definition and runtime flexibility that exists in the lifecycle model is provided to actions. The actions' parameter can be fixed at definition time, instantiated at lifecycle instantiation time, or as the corresponding phase is entered. At execution time, the action is invoked by calling a URI that identifies a web service (either REST or SOAP) of the Gelee plugin component that implements the action (details are provided in the next section), passing as parameters a link to the *object* and a *callback* URI.

Upon completion, or periodically during execution, the action can then call the callback URI and update on its status. The status messages are arbitrary, except two defined by the model, corresponding to *failure* and *successful* completion. The status messages have only information purposes. Their interpretation or follow-up actions are left to the owner.

The attentive reader will have noticed that there is no analogous of workflow data, neither following the blackboard approach nor the data flow approach [11]. The owner inserts all parameters "by hand". Any additional desired behavior must be part of the action implementation.

We mention here that while in this paper we focus on actions that are relevant from a lifecycle perspective, the same approach can be followed by other, perhaps domain-specific applications. For example, one of the goal of LiquidPub is to develop services to extract knowledge from scientific documents stored in various formats and managed by various repositories.

D. *Roles and Access Rights*

Several users, playing different roles, interact with Gelee and the resources. These roles define the set of operations users can perform over the lifecycle. In particular there are three main roles: the lifecycle manager, the lifecycle instance owner and the token owner. The *lifecycle manager* is the person in charge of administrating a lifecycle, and thus, this role allows the user to design and modify the lifecycle. The *lifecycle instance owner*, however, is assigned to the person who instantiates the lifecycle on the resource. This role allows the user to drive execution and modify the model followed by the lifecycle instance. Finally, the *token owner* role belongs to the user in charge of performing a transition at a given phase. Unlike the instance owner, its responsibilities are limited to follow the allowed transitions, and typically to specific transitions only (e.g., it can be ok to allow a team member to determine that the deliverable is ready for internal review, but not to determine that it is ready for EU review).

From the point of view of the resource we have also the *resource owner*, as the person who has full access rights over a given resource, and who can assign permissions for it. Thus, instance and resource owners can assign permissions or visibility rules over the instance and resources respectively.

V. Gelee At Work

This section describes the basic elements that allow the *Gelee* prototype to support lifecycle management. We first define the overall architecture, and then discuss two among the most important aspects: action-resource interaction and lifecycle widgets. Although we present them "embedded" in the architecture section of the paper we draw the reader's attention to it as they correspond to key decisions in terms of keeping the model simple, and in terms of usability and reusability.

A. *Overall Architecture*

The Gelee architecture is simple, especially due to the fact that there is no analogous of a workflow engine that advances the flow from step to step. In essence, the system supports design and monitoring as well as invocation of actions that, from the core system perspective, are black boxes and are embedded into resource type-specific plug-ins that can be added as needed. As the primary goal of Gelee is to manage online resources and to have a system that is simple and usable, it was natural to provide lifecycle management as a service, and therefore hosted. Fig. 2 depicts the high-level architecture, composed essentially of three layers: the data tier, the kernel and the user interface.

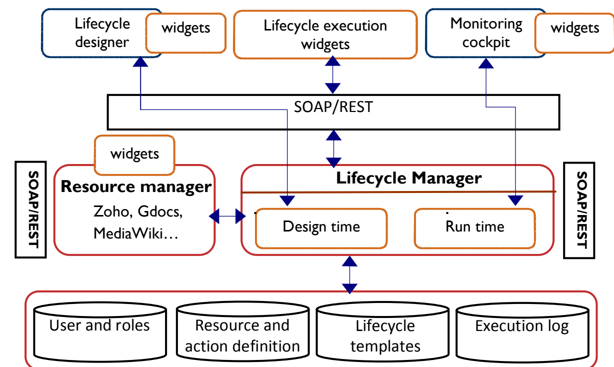

Fig. 2 - Gelee high-level architecture

At the bottom of the figure we have the data tier, which includes the repositories for users and roles, resources and actions definitions, templates, as well as execution logs (including model evolution). The lifecycle manager is the hearth of the system, and it has a design time and a runtime module. The design time interacts with a lifecycle designer GUI (discussed next) via a SOAP and REST interface and receives definitions and modifications to a lifecycles. The runtime module receives lifecycle instance events (progression from phase to phase as dictated by the instance owner), sent by the *lifecycle execution widgets*, and action execution results, sent by resource plug-ins and discussed next. The interaction also in this case occurs via SOAP or REST messages. As a consequence of instance progression events, the lifecycle manager looks up the action list for the new phase reached by the lifecycle and contacts the resource type-specific plug-in to execute them.

B. *Resources and Actions*

Different resources are in general managed by different applications (Wiki, Flickr, etc..). In many cases, although the managing application is different, the kinds of actions that can

be executed on the resource are similar. For example, in both Wiki and Google-Docs I can have the possibility of sending a document for review, or generating a PDF. Some of these actions are semantically equivalent but may require different parameters (i.e., the “signature” details are different). The implementation instead is certainly different and depends on the managing application.

This separation between *action types* and *action implementations* is another way in which Gelee supports light-coupling. Designers can define lifecycles (including definition of actions) that can be made applicable to different resource types. When a lifecycle is instantiated on a specific URI (and therefore on a specific resource of a specific type), actions types are resolved to specific action signatures and implementations.

The interfacing between the Gelee platform and a specific resource occurs through plug-in adapters. Developers can create adapters for any kind of resource, and implement actions that support a given functionality. The action implementation may correspond to an existing action type defined earlier for other resource types (e.g., send for review) or it can be a new action type that does not exist in Gelee. In both cases, the adapter needs to *register* the new action implementation with Gelee, to make Gelee aware that there is an action implementation for a specific resource type has been added, or that a completely new action type is introduced. The registration also includes information that Gelee needs for invoking the action.

The action definition is standard and includes information about i) the action type, which defines how to access the action; ii) parameters, and the time at which their values have to be associated (bound); and iii) descriptive metadata. This definition allows Gelee platform to handle all actions in a standardized fashion. Table II shows the XML of an action definition.

TABLE II
EXAMPLE OF AN ACTION TYPE DEFINITION XML

```
<action_type uri=”changeAccessRights”>
 <name>Change Access Rights</name>
 <!—Information about the version-->
 <version_info>
  <version_number>1.0</version_number>
  <created_by>lpAdmin</created_by>
  <creation_date>08/07/2008</creation_date>
 </version_info>
<!—Action parameters -->
<parameters>
 <param bindingTime=”[def|inst|call|any]”
        required=”[yes|no]”>
  <name></name>
  <value></value>
 </param>
</parameters>
</action_type>
```

Once defined, actions are available for use in lifecycle definition. When defining lifecycles, users can browse through all actions as there is not yet, in general, a binding to a resource type (unless the user restricts a lifecycle to a type or a set of types). For modifications at runtime, only actions for which there is an implementation for the resource being managed are shown. An example is shown in Fig. 3. Action execution occurs as discussed in the previous section by invoking the resource adapter as specified by the action definition.

In Gelee we currently defined adapter for the examples provided in this paper (Google docs and wiki), but they can be added at any time and since they are invoked via REST, they can be completely separate from the system. Anybody can just develop an adapter, make it available on the Web, and begin defining actions that use this new adapter, without changing anything in the Gelee installation. Using and modifying a recent buzzword, this can be considered a sort of *cloud adaptation*.

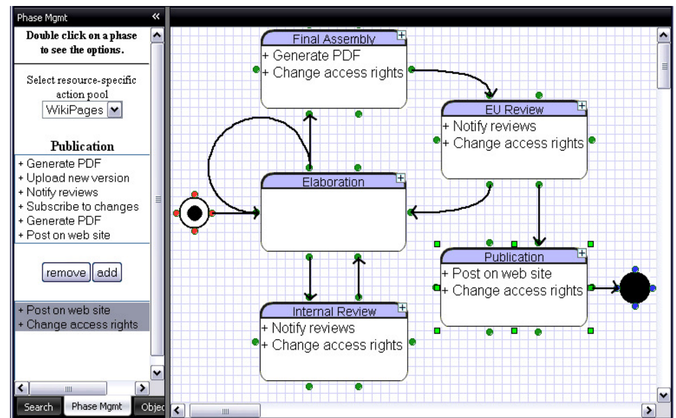

Fig. 3 - Gelee lifecycle designer

C. *Lifecycle Widgets*

Gelee offers UI-level integration through resource type-specific widgets and web interfaces. Web UI are provided for the lifecycle designers (Fig. 3) and for the monitors. Both offer an AJAX-style interface, easy and immediate to use.

Execution of lifecycles is instead different, as ideally it is integrated with the resource it manages, for example, it is shown in the same web page, or as a browser plug-in. For this aspect we use widgets. Widgets are components ready to be integrated with web applications (or even desktops, even if we did focus only on browsers till know). Through widgets, users see the lifecycle and the resource they manage side by side, as shown in Fig. 4.

A widget interacts with the lifecycle management platform, which then interacts with the resource. In other words, there is no direct interaction between the widget and the resource.

Widgets are also subject to visibility attributes. Attributes like access rules are automatically auto-discovered from the lifecycle definition. Thus, a user interacting with a widget could be requested for authentication or not based on the visibility attributes, and also, different users could have different views of the same lifecycle (i.e., managers, resource owners, and stakeholders in general).

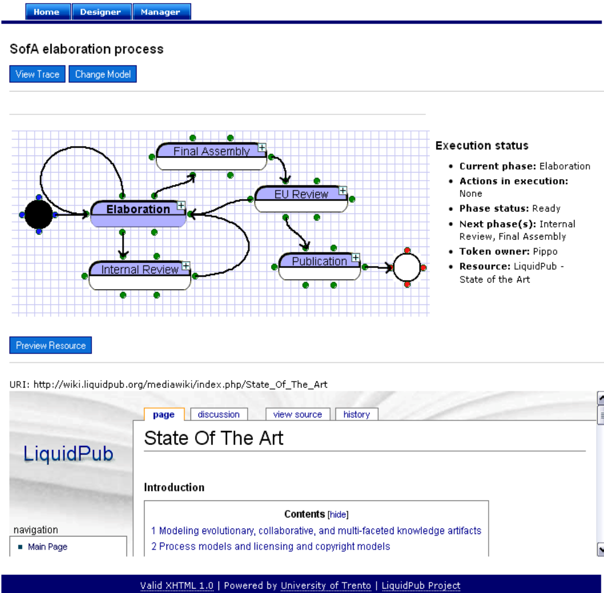

Fig. 4. Integrated Resource Lifecycle Management widget

VI. Conclusion and Future Work

In this paper we have described a universal resource lifecycle management model and the Gelee prototypal system[5]. The current status of the framework is that components have been implemented (but not integrated) except the monitoring interface that has been only designed. Hence the source is available but the integrated platform is not yet available. Resource plug-ins currently include Google Docs and MediaWiki.

We tried to design the Gelee platform based on a very concrete case (i.e. European Projects) and based on what we and the people in the projects would like and would feel comfortable. In this kind of design and developments, we have the unique advantage that we ourselves ("we" writing the paper, "we" members of the project, but also "we" as researchers in general) are the users of the work and therefore it is easier to define users' requirements users are comfortable with, especially in terms of resisting the temptation to make the approach feature-rich but then inflexible or complex. In this sense, the hardest parts of the work were in identifying the level of complexity of the model and the light-binding approach. The philosophy behind the design choice is to seek simplicity whenever we can and tackle complexity only if and when needed. Users who need simple things need not be bothered with complexity.

Other innovative aspects of our framework are (1) the action-resource model, which we believe provides a useful abstraction from the composition perspective; (2) extensibility and breadth of resource access and functionality. This is a significant departure for example from workflow models or even from service composition models.

The approach we have followed here is to put the complexity in the implementation of the actions, while keeping the model both general and simple, from the action perspective, to the composition designer. Indeed the lifecycle model can be described in about a page and learned in a matter of minutes, literally. And it can be used to control any resource for which there is a plug-in.

The approach is also kept clean and extensible by leveraging plug-ins for resources, which can be externally managed and for which we only need a URI of the manager and an action interface for which we define the format, and that is very extensible.

In terms of future work, besides completing the monitoring aspect, interesting aspects include the integration with engines for those cases where engines are actually needed, and the challenge here lies in doing so keeping the same level of simplicity and flexibility. Another aspect we think it is interesting to explore is to link the lifecycle to complex resource types, and specifically to composed resources. This is a need we also have in the project, as sometimes the artifact (which in liquidpub are called scientific knowledge objects) are structured, for example the state of the art is composed of the main documents, the references, presentations, etc… and managing a complex resource with components and with potentially independent but somehow interacting lifecycles is something that is part of our future explorations.

[5] Gelee is available for free and is developed in open source at *https://dev.liquidpub.org/svn/holms/trunk/*.

References

[1] M. Reichert and P. Dadam. "ADEPTflex: Supporting Dynamic Changes of Workflow without Loosing Control". *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

[2] Peter Dadam, Manfred Reichert, Stefanie Rinderle, Martin Jurisch, Hilmar Acker, Kevin Göser, Ulrich Kreher, Markus Lauer: "Towards Truly Flexible and Adaptive Process-Aware Information Systems". *UNISCON 2008*: 72-83.

[3] W.M.P. van der Aalst, Mathias Weske, and Dolf Grünbauer. "Case handling: A newparadigm for business process support", Data & Knowledge Engineering, 53(2):129-162, 2005.

[4] Cugola, G. 1998b. Tolerating deviations in process support systems via flexible enactment of process models. *IEEE Transactions of Software Engineering* , 24, (11), November.

[5] M. Weske, G. Vossen, C.B. Medeiros. *Scientific Workflow Management: WASA Architecture and Applications*. Fachbericht Angewandte Mathematik und Informatik 03/96-1, Universitat Muster, 1996.

[6] LaMarca, A., Edwards, K., Dourish, P., Lamping, J., Smith, I., and Thornton, J. 1999. Taking the work out of workflow: Mechanisms for document-centric collaboration. In *Proceedings of the 6th European Conference on Computer-Supported Cooperative Work* (ECSCW '99, Copenhagen, Denmark, Sept. 12–16), Kluwer Academic, Dordrecht, Netherlands.

[7] J. Wang and A. Kumar. "A framework for document-driven workflow systems", In *Business Process Management*, pages 285–301, 2005.

[8] A. Nigam and N. S. Caswell, ''Business Artifacts: An Approach to Operational Specification,'' IBM Systems Journal 42, No. 3, 428–445 (2003).

[9] D. Wodtke and G. Weikum, "A Formal Foundation for Distributed Workflow Execution Based on State Charts". *Proc. Int'l Conf. on Database Theory*, Delphi, Greece, January 1997.

[10] Van der Aalst, W. M. P., ter Hofstede, A. H. M., Weske: Business Process Management: A Survey. *Business Process Management 2003*.

[11] Alonso, G., Casati, F., Kuno, H., & Machiraju, V. (2003). *Web services*. Springer.