A chemical model for dynamic workflow coordination

Manuel Caeiro∗, Zsolt Németh† and Thierry Priol‡
∗E.T.S.E. Telecomunicación
University of Vigo, Campus Universitario, E-36310 Vigo
Email: Manuel.Caeiro@det.uvigo.es
†MTA SZTAKI Computer and Automation Research Institute
P.O. Box 63., H-1518 Hungary
Email: zsnemeth@sztaki.hu
‡IRISA, Campus Universitaire de Beaulieu
35042 Rennes, Cedex, France
Email: Thierry.Priol@irisa.fr

Abstract—This paper investigates a chemical workflow enactment model that is intended to coordinate workflows of large set of activities on a large number of resources in a self-evolving nature, based on a chemical analogy. The concept of chemical workflow engine is introduced for a concurrent, self-coordinating enactment exploiting as much parallelism as inherently present. The concept is aimed at supporting a generalized workflow language without any restrictions, modeling most workflow patterns, separating data and control flow, supporting dynamic changes by multiple versions and instances. The paper presents the notion and model of the chemical based coordination.

Index Terms—nature inspired computing, formal model, dynamic workflow

I. INTRODUCTION

The notion of workflows was borrowed from the business world and has emerged as a paradigm for executing complex distributed computations; in recent years became a dominating model for many scientific applications. Therefore, their various aspects have been well studied and some of their characteristic and partly unsolved challenges are summarized as: they are dynamic where decisions are made based on the latest available information; a workflow may be dynamically altered according to the results at a certain point in progress; the basic structure or the semantics of the workflow may change to some external events; workflows may breakdown and fail; workflows may have an evolution by refinements. Both the workflow and the infrastructure are in continuous change [1].

We focus our work on the enactment of scientific (i.e. computation intensive) workflows albeit the same problems appear in different settings like service compositions, for instance. In fact, workflow is just a proper example to study the potential of a non-conventional, chemical modeling approach. The research presented in this paper is driven by the observation that enactment of scientific workflows in an ever-changing (with respect to performance and availability) and fault prone distributed heterogeneous computing environment requires a complex coordination. Potentially, available information may be partial or not up to date. Furthermore, the presence of advanced workflow patterns [2] also may introduce some degree of uncertainty. We assume that such coordination requires frequent and abrupt changes where no predefined or static approaches are feasible.

Coordinating a large-scale system is far beyond the capabilities of humans even more, the complexity of such control is increasing so that even machines cannot cope with all possible or potential cases based on predefined conditions and algorithms. With the advent of the notion of autonomous computing [3] it became evident that such systems must exhibit some degree of autonomy, able to adapt to changes and they should provide some self-* properties. Most algorithms nowadays are sequential, expressed in imperative programming languages making the description of such algorithms complex and potentially incomplete. Nature-inspired formalisms may offer a well defined mechanism to find a more adequate solution. The current stage of our work is not aimed at establishing autonomous behavior based on a nature inspired computational model rather, to establish a nature inspired formal framework where such models can be realized.

Nature models have been applied to various problems, especially scheduling and optimization in the context of scientific workflows. Particle swarm based workflow scheduling and optimization is proposed by Abraham et al. [4] and Pandey et al. [5]. Sun et al. [6] take a neuroendocrine-immune system as a model and create a decentralized, evolutionary, scalable, and adaptive system for E-services workflow. Prodan et al. [7] apply genetic algorithm for static scheduling of DAG type workflows. Genetic algorithms appear in a particular aspect of workflow scheduling by Spooner et al. [8]. Simulated annealing was considered by Young et al. [9] for scheduling grid workflows.

In contrast, instead of solving some particular problems of workflows, our approach focuses on modeling and tries to put the entire workflow enactment into an abstract chemical framework, with a special emphasis on its dynamic behavior. Artificial chemistries are man-made systems that are similar to a real chemical system [10]. Dittrich et al. classify three main
axes of applying artificial chemistry: modeling, information processing and optimization. In this paper we explore the *modeling* aspect and the work is similar to [11], [12] where Banâtre et al. use a chemical model to express various issues related to self-organizing services and service invocations and also related to [13] where earlier studies with modeling static workflows using $\gamma$-calculus were done.

Workflows may change during execution due to any condition unforeseen at planning, user intervention, changes in the environment or for adaptation reasons. Changes may involve structural, functional, parametric changes; there may appear several versions and instances of the same workflow definition. The aim is to create a model that is flexible and allow establishing a fully dynamic enactment of workflow that can lead later to various adaptation strategies. We imagine a system that is able to react to any changes in its own behavior or the embedding conditions. A metaphor we took is chemistry: just like chemical reactions are self-governing, self-balancing, self-evolving processes, an “idealistic” distributed workflow enactment should possess many of these features.

The contribution of this paper is a methodology by developing an abstract model for the Chemical Workflow Engine (CWE). The work is based on thorough requirement analysis and is in partial alignment with the major challenges of scientific workflow [1]. The main considerations are: (i) support for a generalized workflow language without any restrictions; (ii) a clear separation and detailed model of data and control flow; (iii) modeling events and connectors to generalize control in the chemical model; (iv) support for dynamic structural and semantic changes by multiple workflows / multiple versions / multiple instances; (iv) detailed analysis of some fundamental constructs modeled as chemical entities and reactions.

II. THE ORIGINS OF THE CHEMICAL MODELING

This work applies a certain chemical modeling approach hence, its background, the family of the Gamma language (Gamma, the $\gamma$-calculus and the Higher Order Chemical language) are summarized here briefly. Note, this way of chemical modeling is different from computational (artificial) chemistries: while they model and simulate real chemical processes, our chemical model takes chemistry as a modeling metaphor. The first notion of the chemical model has been introduced as the *Gamma* formalism [14] to capture the intuition of computation as the global evolution of a collection of interacting elements (e.g., molecules). *Gamma* can be introduced intuitively through the chemical reaction metaphor. Programs are conceived as chemical solutions involving a (multi)set of molecules that can react in accordance with certain reaction conditions and actions. Computation proceeds by capturing the molecules satisfying certain reaction conditions and replacing them by the molecules specified in the actions; there is no concept of any centralized control, ordering, serialization rather, the computation is carried out in an indeterministic way according to local conditions. The result of a *Gamma* computation is obtained when a stable state is reached, namely, when no more reactions can take place.

It has been shown in [14] that some fundamental problems of computer science (sorting, prime testing, string processing, graph algorithms, etc.) can be expressed in Gamma in a more natural way than applying any other sequential semantics.

Based on the notion of the *Gamma* formalism, a higher-order model called $\gamma$-calculus [15] was conceived. In this calculus every element is considered as a molecule. Programs are made of Active Molecules (AMs), embedding reaction rules and actions, and Passive Molecules (PMs) representing data. AMs can capture AMs and/or PMs and produce new AMs and/or PMs in accordance with their reaction rules and actions. Also, AMs do vanish in reactions. In contrast, Gamma represented data entities as molecules and reaction rules were outside of the solution (no notion of molecules), they remained intact after a reaction.

The Higher Order Chemical Language (HOCL) [16] is a language based on the $\gamma$-calculus extended with expressions, types, pairs, empty solutions and names. HOCL uses the self-explanatory replace... by... if... construct to express active molecules.

\[
\text{replace } P \text{ by } M \text{ if } C
\]

formally corresponds to

\[
\gamma(P)[C].M
\]

with a major difference: while $\gamma$-abstractions are destroyed by the reactions, HOCL rules remain in the solution. replace... by... if... is followed by in (...) that specifies the solution the active molecule floats in. Notable extensions of HOCL are: types that can be added to patterns for matching; pairs in form of $A_1 : A_2$ where $A_1$ and $A_2$ are atoms; and naming that allows to identify and match active molecules. For example,

\[
\text{let } inc = \text{ replace } x \text{ by } x + 1 \text{ in } (1, 2, 3, inc)
\]

specifies an active molecule named inc which captures an integer and replaces it with its successor, floating in a solution together with integers 1,2,3.

III. THE GENERALIZED NOTION OF WORKFLOW REPRESENTATION

Due to the lack of standardized workflow languages we assume a general representation, not committed to any existing workflow language in order to demonstrate the capabilities of our model.

- **Function** represents an activity of the workflow, a task to be realized, such as the execution of a certain program. **Functions** usually require some input **Data Elements** that have to be processed in order to produce some output **Data Elements**. In addition, **Functions** need to be assigned to appropriate **Resources**

- **Connectors** support the description of control- and data-flow dependencies. There are different types of **Connectors** related to the control flow, such as splits and joins, or related to the data flow, such as the break down of a large data set into several smaller chunks of data. **Connectors** may involve the management and processing of **Data Elements**, as input and output items.
• **Events** represent a state before and/or after a **Function** or **Connector** is traversed. This element is strongly related to the control flow issues of workflows.

• **Data Element** represents a container of data used during the workflow execution. **Data Elements** are taken as input by **Functions** and **Connectors**.

• **Resource** is an entity necessary to realize the activity represented by **Functions**. Each **resource** includes a description of its capabilities and features, enabling the matching of appropriate **Resources** to **Functions**.

Control-flow is expressed by explicit connections between **Functions** and **Connectors** through **Events**. Any **Function** must be preceded by just one **Event** and succeeded by another **Event**. Similarly, a **Connector** must be preceded by one or several **Events** and succeeded by one or several **Events**, depending on the **Connector** type. Further constraints are: (i) it is not possible to connect two **Events** one to another and (ii) there is at least one start **Event** and at least one final **Event** representing the start and finish of the workflow, respectively. Data-flow is expressed implicitly by identifiers assigned to **data elements** and input and output parameters in **Functions** and **Connectors**.

The expressiveness and generality of the experimental notation comes from the connector types ranging from typical control-flow connectors to more specific ones used in scientific workflows: data loops [17] or the one-to-one and all-to-all connectors from SCULF [18]. Control-flow connectors are typically divided into four categories: sequential, parallel, conditional and iterative constructs. We considered the parallel and conditional connectors similarly to workflow languages such as Event-driven Process Chains (EPCs) [19] and YAWL [20]: AND-split/AND-join, OR-split/OR-join, and XOR-split/XOR-join connectors. In addition, our generalized concept also includes loops and sub-process calls.

### IV. The abstract model of the chemical workflow engine

**A. Molecules of the chemical workflow engine**

The coordination of workflow enactment is envisioned as an analogy of chemical reactions where **Active Molecules** (AMs) and **Passive Molecules** (PMs) are dissolved forming a chemical solution (or solutions) and they engage in reactions. Reactions are realizing the ‘computational step’; by a series of reactions the chemical matter evolves so that every reaction is driven by actual and local conditions. The execution goes on as long as there are molecules to react. If there are no more reactive molecules, i.e. the solution is inert, the execution stops. It is possible that some external events (e.g. user intervention) insert molecules that perturb the inertia. Exactly this manner makes it possible to realize dynamic adaptation, the execution can be controlled. The whole computation is not a sequence of instructions rather, a dynamically changing set of molecules that evolves according to some rules in an inherently concurrent and autonomic way.

The notion of the Chemical Workflow Engine (CWE) tries to realize this metaphor. The coordination is based on the interaction of specific AMs and PMs that are representing the elements of the workflow. Particularly, **Functions** (set $FUNCTION = \{f_1, f_2, \ldots, f_k\}$) and **Connectors** (CONNECTOR = \{c_1, c_2, \ldots, c_l\}) are AMs (because they involve some kind of processing or control, while **Events** (EVENT = \{e_1, e_2, \ldots, e_m\}), **Data Elements** (DATA = \{d_1, d_2, \ldots, d_n\}) and **Resources** (RESOURCE = \{r_1, r_2, \ldots, r_p\}) are represented as PMs. All these molecules are put into a solution called **main** that represents the coordination space:

\[
\text{main} = \langle FUNCTION, CONNECTOR, EVENT, DATA, RESOURCE \rangle
\]

It is important to indicate that molecules are abstract entities: **Data Elements** and **Resources** contain references to the real data elements and resources, but they are not the actual items. Similarly, **Functions** do not contain the programs (actual code) that they require to execute, but references to such entities.

We represent all PMs (Events, Data Elements and Resources) as sub-solutions separated from the coordinating **main** solution by a membrane. Attributes form pairs within the sub-solution. The generic form is:

\[
\langle a_1 : v_1, a_2 : v_2, \ldots, a_n : v_n \rangle
\]

where each $a_i : v_i$ is an attribute-value pair yielding a profile for a certain entity. The exact number of attributes and their meaning is entity-dependent; we do not specify any strict format at this level of abstraction.

Such a sub-solution may contain any molecules (attributes) in any order as long as the notation is consistent. Thus, an event, for instance, may be represented as

\[
\langle event, name : A, id : 12, apple, 23, \ldots \rangle
\]

an event tag, an event name ‘A’, an ID and some other unspecified parameters. Since molecules engage in reactions by pattern matching, the presence or absence of certain patterns (values) is crucial, like the tag, name and ID but other molecules are indifferent and may be omitted, like apple and 23 in this example. As we will see, identifiers have a very important role in this model. **Data Elements** can be represented in a very similar way, e.g.

\[
d_1 = \langle data, name : X, id : 42, value : nil, \ldots \rangle
\]

the nested solution may contain data items, references to data and metadata to describe the content. One may imagine them as (self-describing) data structures in conventional languages. Analogously, **Resources** are represented in a quite self-explanatory way

\[
r_1 = \langle rsc, type : comp, proc : 16, OS : Linux, \ldots \rangle\text{ in }\text{main} = \langle r_1, e_5, d_{12} \ldots \rangle
\]

putting all the above PMs into the coordinating solution.

The **reaction rules** and **actions** of the AMs in the CWE specify matching the appropriate PMs and generating new PMs and AMs. Particularly, **Functions** have to match the appropriate input **Data Elements**, the input **Event** and the required **Resources**. When the right molecules are matched, they are all removed from the chemical solution, and as an external call,
the program (or other activity) corresponding to the Function is executed using the matched input Data Elements and the matched Resource. When the execution is finished, the output Data Elements and the output Event are generated as results and put as molecules in the chemical solution. Similarly, molecules representing the utilized Resources appear again in the solution, since these resources are available again. Thus, the generic form of a function molecule is a simple rewriting rule:

\[ \text{eq1}: \; f_i = \text{replace } d_{ix}, e_{iy}, r_{iz} \text{ by } d_{ix'}, e_{iy'}, r_{iz} \]

where \( f_i \in \text{FUNCTION} \), \( d_{ix}, d_{ix'} \in \text{DATA} \), \( e_{iy}, e_{iy'} \in \text{EVENT} \) and \( r_{iz} \in \text{RESOURCE} \). This is a simplified presentation: real function molecules may capture and yield multiple data, event and resource molecules.

The behavior of Connector AMs is similar to the Functions but they do not require any Resources. Connectors have to match the appropriate input Data Elements, the input Event or set of Events. When the right molecules are matched, the reaction is ignited: the molecules are eliminated from the solution and the corresponding Connector is processed. As a result, Data Elements and the output Event(s) are introduced again in the chemical solution. Formally,

\[ \text{eq2}: \; c_i = \text{replace } d_{ix}, e_{iy} \text{ by } d_{ix'}, e_{iy'} \]

where \( c_i \in \text{CONNECTOR} \), \( d_{ix}, d_{ix'} \in \text{DATA} \) and \( e_{iy}, e_{iy'} \in \text{EVENT} \).

This general description is a ground model, adding more details thus, refining a model may reveal four states of the molecules that would be modeled as four reaction phases (cf. Figure 1):

1. **Instantiation of Function and Connector AMs** means matching the Function and/or Connector AMs with Event PMs. This reaction yields a new molecule, named as Enabled Active Molecule (EAM) representing the corresponding Function or Connector. The Function and Connector AMs (should be really named as Disabled Active Molecules - DAMs) behave like a catalyst in this reaction: they remain intact. Therefore, the DAMs can react again with new Event PMs to generate new EAMs thus, allowing the creation of several instances of the same Function or Connector. Formally, \( \text{FUNCTION} \) and \( \text{CONNECTOR} \) sets are refined as \( \text{FUNCTION} = \text{D.FUNCTION} + \text{E.FUNCTION} \) so that \( \text{D.FUNCTION} \cap \text{E.FUNCTION} = \emptyset \) and \( \text{CONNECTOR} = \text{D.CONNECTOR} + \text{E.CONNECTOR} \) so that \( \text{D.CONNECTOR} \cap \text{E.CONNECTOR} = \emptyset \). The "disabled" molecule is the template or the definition of the function/connector whereas the "enabled" one is a cloned instance of it. The cloning is triggered by event \( e_{iy} \). Thus, the corresponding refinement of the function molecules (Eq. 1) is:

\[ \text{eq3}: \; \text{disabled} \cdot f_i = \text{replace } e_{iy} \text{ by } \text{enabled} \cdot f_i \]

where \( \text{disabled} \cdot f_i \in \text{D.FUNCTION} \) and \( \text{enabled} \cdot f_i \in \text{E.FUNCTION} \) The refinement of Eq. 2 is similar.

2. **Assignment of Data Elements to Functions and Connectors.** Once EAM instances are created, Function and/or Connector EAMs are matched with Data Element PMs. This reaction is different for Functions and Connectors. In the case of Function a new molecule, named as Ready Active Molecule (RAM), is created that represents together the Function and the matched Data Elements. The corresponding refinement, i.e. the definition of \( e_{f_i} \) in Eq. 3 is

\[ \text{eq4}: \; \text{enabled} \cdot f_i = \text{replace } d_{ix} \text{ by } \text{ready} \cdot f_i, d_{ix} \]

where \( d_{ix} \in \text{DATA} \) and \( r_{f_i} \in \text{E.FUNCTION} \times \text{DATA} \), i.e a parametrized instance. In case of Connector EAMs, the output Event and the output Data Element PMs are created. In both cases Data Element \( d_{ix} \) acts as a catalyst: it ignites the reaction but remains intact for further activations.

3. **Assignment of Resources to Functions.** After Function RAMs are created, they are matched with Resource PMs yielding an Initiated Active Molecule (IAM), representing the execution of the Function with the resources using data elements. Formally,

\[ \text{eq5}: \; \text{ready} \cdot f_i = \text{replace } r_{iz} \text{ by inst} \cdot f_i \]

where \( r_{iz} \in \text{RESOURCE} \) and \( \text{inst} \cdot f_i \in \text{E.FUNCTION} \times \text{RESOURCE} \), i.e an instance with an assigned resource.

4. **Termination of execution of Function.** IAMs are reacted and output Event, output Data Element and Resource PMs are created. Note that this reaction is external to the coordination model. We provide its chemical specification (semantic equivalence) but it is not realized by the engine itself.

\[ \text{eq6}: \; \text{external}_\text{execution} = \text{replace } \text{inst} \cdot f_i \text{ by } d_{iz'}, e_{iz'}, r_{iz} \]

As it can be seen, Eqs. 3, 4, 5 and 6 are a refinement of Eq. 1 and they can be derived similarly for connectors.

**B. Solutions of the Chemical Workflow Engine**

The four execution states of the workflow activities conceived as four reaction phases are reacted in chemical sub-solutions arranged accordingly, as represented in Figure 1.

In the chemical model, reactions occur according to laws of locality (also called as chemical law): if a reaction can occur, it will occur in the same way irrespectively to the environment; and of solution (also called as membrane law): reactions can occur in nested solutions or in other words, solutions may contain sub-solutions separated by membranes. A chemical sub-solution is a contained space of chemical molecules that can react without any interference from the outside world until no more reactions can be produced inside the sub-solution, i.e. it becomes inert [15].

The four states and corresponding sub-solutions are the followings:

- **Disabled.** This sub-solution contains Functions and Connectors as Disabled Active Molecules (DAMs) that cannot be processed yet because they have not been activated.
Fig. 1. The chemical workflow engine

- **Enabled.** This sub-solution contains Functions and Connectors that can not be processed because they have not captured the required input Data Elements.
- **Ready.** This sub-solution contains Functions ready to be executed, but that have not been assigned to any Resource.
- **Initiated.** This sub-solution contains Functions that are being executed by assigned Resources, carrying out the intended work (e.g., the execution of a program).

### C. Identifiers

The CWE is designed to support the operation of several workflows simultaneously, each one involving potentially multiple instances. It supports a scenario taken from [1]: a workflow may be dynamically (re)designed according to the results and progress of initial steps; decisions can be made at run-time how to carry on with the computation.

The support of several processes with several instances is managed in the CWE through the use of distinguished identifiers: a Process Schema Identifier (PSI) and an Instance Identifier (II). The Process Schema Identifier distinguishes the several process descriptions. This identifier is composed by two parts: process number and version number. The version number is initially fixed to zero. Each time a modified node is introduced, the version identifier may require to be changed for such node. Meanwhile, the Instance Identifier distinguishes the several executions or instances. Related to the Instance Identifier, an additional, Thread Identifier (TI) may be maintained in some special cases. This is required if a workflow instance can include several threads in parallel, involving the same elements. Therefore, it is required to distinguish between the different threads.

These identifiers are related to the molecules in the following way:

- **Function** and **Connector** DAMs include the Process Schema Identifier that represents the process description they belong to.
- **Data Elements** include the Instance Identifier and Thread Identifier that represents the instance and thread they belong to, respectively.

- **Events** include both the Process Schema Identifier, the Instance Identifier and Thread Identifier. They are the connection between the elements of the process description and the instance-thread data elements. EAMs, RAMs and IAMs take the three identifiers as they are generated from Events.

These three identifiers are used to constrain the matching among molecules. Two molecules can be matched if they share the Process Schema Identifier, the Instance Identifier and the Thread Identifier. In this way, as molecules with different identifiers can not be matched, it is supported the operation of several instances of several processes by the same CWE.

### V. REACTIONS OF THE CWE

This section presents a detailed description of Functions and (due to space limitations) some of the Connectors, in the different states.

#### A. Functions

Workflow activities represented by Functions are activated by successively traversing the four states: disabled, enabled, ready and initiated. In terms of the chemical metaphor, AMs are reacted in four successive phases, in four separate sub-solutions. Figure 2 shows the representation of a Function (top of the figure), preceded and succeeded by an Event, and its chemical processing ((i)-(iv) in the figure).

Prior to enactment, activities corresponding to Function are mapped to Function DAMs thus, entering symbolically the chemical world and forming the first solution. The chemical processing of Functions involves the following reactions:

1. the reaction of the Function DAM with an appropriate Event molecule yields the Function EAM. The condition of the reaction is the appropriate matching of the identifiers of the Function DAM and the Event. In the action the Event molecule vanishes, a Function EAM appears while the Function DAM is a catalyst;
2. the next reaction of the Function EAM with the appropriate Data Element molecules produces the Function RAM. Similarly, the condition of the reaction is the appropriate matching of the identifiers of the Function EAM and the Data Element. In the action the Function EAM molecule vanishes, a Function RAM appears while the Data Element molecules remain is a catalyst; the reaction of the Function RAM with an appropriate Resource molecule produces a new Function IAM while both matched molecules disappear; and
3. the reaction of the Function IAM generates a new Event, Data Element and Resource molecules. This last reaction is external: the chemical model coordinates but does not execute the actual Function. In this way, the workflow engine takes the Function IAMS and assigns the corresponding task (actual code to be executed) to physical resource(s) and stages the necessary matched data elements. The resulted new molecules: Event represents the state after the execution of the activity, Data Element is the result of the execution and Resource stands for the released resource(s).
B. AND-Split and AND-join Connectors

Figure 3 shows the representation and execution behavior of the AND-split and AND-join Connectors. Prior to enactment, AND-split and AND-join Connectors of the workflow are translated into AND-split and AND-join DAMs of the chemical world, respectively.

The behavior of these two Connector DAMs is developed as follows. (For the sake of simplicity we present the HOCL terms in an drafty way with not all details. Also, we do not specify all identifiers (PSI, II, TI), just a symbolic id representing that the identifiers must match – unless the precise identifier has an importance.)

i The AND-split DAM reacts with Event A and produces a corresponding AND-split EAM. The condition of the reaction is the matching of appropriate identifiers of AND-split DAM and Event A. Event A vanishes (the very same event cannot trigger any further splits of the same instance) whereas the AND-split remains intact as a catalyst.

\[ \text{disabled}_a\text{split} = \text{replace} (\text{event}, \text{name} : A, \text{id} : q, \omega) \text{ by } \text{enabled}_a\text{join} \]

ii Subsequently, AND-split EAM reacts any Data Element to generate the specified number of Events B and a special Data Element molecule, the Event Counter, that maintains the number of Events B generated. This value is needed by the AND-join Connector. Each Event B generated has to include a different Thread Identifier.

\[ \text{enabled}_a\text{split} = \text{replace} \{\text{data}, \text{id} : q, \omega\}, \]
\[ \text{by} \{\text{event}, \text{name} : B.1, \text{id} : q, \text{ti} : x, \ldots\}, \]
\[ \{\text{event}, \text{name} : B.2, \text{id} : q, \text{ti} : x + 1, \ldots\}, \]
\[ \ldots \]
\[ \{\text{event}, \text{name} : B.n, \text{ID} : q, \text{ti} : x + n, \ldots\}, \]
\[ \{\text{data}, \text{e}_\text{count}, \text{value} : n, \text{id} : q\ldots\}, \]
\[ \{\text{data}, \text{id} : q, \omega\} \]

iii The AND-join DAM would react Events C. Each time the AND-join DAM matches one of these Events C a new Enabled Decrement (1) EAM is resulted, while the matched Event C vanishes. Obviously, AND-join DAM is a catalyst, since it must react multiple times.

\[ \text{enabled}_a\text{join} = \text{replace} (\text{event}, \text{name} : C, \text{id} : q\ldots), \]
\[ \text{by } e.\text{dec}(1) \]

iv The new Decrement (1) EAM reacts with the Event Counter Data Element and results the Event Counter Data Element with a decreased value by 1. In addition, when the value of the counter is zero it generates an Event D molecule. In this way, the synchronization is realized when the number of threads captured is equal to the number of threads initiated in the AND-split Connector. Here we have two molecules:

\[ e.\text{dec} = \text{replace} \{\text{data}, \text{e}_\text{count}, \text{value} : x, \text{id} : q\ldots\} \]
\[ \text{by} \{\text{data}, \text{e}_\text{count}, \text{value} : x - 1, \text{id} : q\ldots\} \]
\[ \text{if } x > 0 \]

and

\[ e.\text{dec} = \text{replace} \{\text{data}, \text{e}_\text{count}, \text{value} : 0, \text{id} : q\ldots\} \]
\[ \text{by} \{\text{event}, \text{name} : D, \text{id} : q\ldots\} \]

C. OR-Split and OR-join Connectors

During the compilation, both OR-split and OR-join Connectors are mapped to OR-split and OR-join DAMs, respectively. The behavior of these two DAMs are very similar to the previous AND DAMs in many sense, see Figure 4 hence, they are presented informally.
i The OR-split DAM reacts with an input Event A and produces a corresponding OR-split EAM. The OR-split DAM is a catalyst.

ii Then, the OR-split EAM reacts with appropriate Data Element molecules to generate (depending on the evaluation of particular conditions) none, one or more of several output Events B. Each Event B generated has to include a different Thread Identifier. In addition, a special Data Element molecule, the Event Counter is generated, that represents the number of Events B actually generated. The captured Data Element molecules are catalysts in this reaction.

iii This Event Counter is captured by the OR-join that proceeds in the same way as the AND-join connector.

iv Synchronization takes place when the number of threads captured is equal to the number of threads initiated. The workflow resumes with the next state (Event D) after the join.

Fig. 4. The execution behavior of the OR-split and OR-join Connectors

Similar to the previous case, this solution is correct for well-structured OR-split - OR-join pairs but with minor considerations it is also valid for unstructured splits.

VI. DISCUSSION

Albeit, workflow modeling and execution has been extensively studied, our approach is novel in a sense that it tries to combine three aspects. First, support for a wide range of dynamic requirements. Scenarios that have been considered include scientific applications where experiments are formulated as workflows and are refined as tests require different combinations of functions, parameters, simultaneous execution of multiple instances in an iterative, potentially interactive process [1], [21]. Another class with strong dynamic requirements are the ill-defined problems where certain tasks or parameters are not known or undefined and can be obtained at runtime only. Such problems require adding new or modified subtasks and dependencies, i.e. changing the workflow structure dynamically [22]. Some applications may be event-driven or data-intensive where (potentially real-time) data must be processed that require dynamic adaptation, structural changes and response to various input sources [23]. Scientific problems naturally require efficient data and computation management and therefore parallelization and distribution of the tasks is a key feature that may involve dynamic restructuring [24]. An obvious requirement for dynamic behaviour is adaptation to the environment where availability, reliability, capacity, performance and various other characteristics of resources may change.

Second, the chemical model. As it was introduced, many nature inspired approaches or algorithms were applied to solve some problems related to workflow management. Our approach is different in a sense that we do not aim at particular problems rather, to put the entire process into a nature inspired model. We do not intend to simulate chemical processes but take them as a metaphor. The similarities between workflow enactment and chemical reactions have been explained, it makes possible to establish such a model. The other question is if it is worth establishing such a model and what the chemical metaphor would add. We especially try to exploit the notion of the solution that represents the state of the computation. Since the solution is a set (contrary to a structure), elements can be easily added, withdrawn and modified at runtime. This enables dynamic modification of data and, due to the higher order property of the language, the functions themselves. Self-modification is a large potential to be explored. Similarly, the set can hold much more data as actually needed: partial or older results of a computation can be reused in a different computation without any explicit data manipulation. By reusing partial results, successive refinements can be obtained in the computation and also, the computing procedure can be refined, too. The notion of reactions makes possible realizing parallel execution in a simple and elegant way. Ultimately, we expect that chemistry inspired mechanisms for dynamic adaptation and a range of self-* properties can be specified in a simpler way in this framework.

Third, the model itself defines the execution in a mathematically precise way and in this particular aspect our approach is similar to Petri-net based models [25]. A workflow model described in HOCL can be enacted by simply interpreting the HOCL code. As it is common with functional languages, the HOCL interpreter can be considered as a virtual machine. This virtual layer has interfaces towards the environment so that it can get input and send output to the embedding world thus, reacting and controlling it. In such a way the chemical engine is detached from an actual implementation. The HOCL semantics is different but reasonably close to that of production systems hence, advanced pattern matching algorithms of production systems, like RETE [26], may serve as a foundation for implementation. Workflow constructs presented in this paper
have been tested in an experimental setting using CLIPS [27], a framework for construction expert systems. Based on the results, a multithreaded, Java based full HOCL interpreter has been designed and realized on a modified production system that can serve as an elementary chemical virtual machine.

VII. CONCLUSIONS AND FUTURE WORK

The concept of a workflow engine based on the chemical paradigm was presented in this paper in an ongoing effort to take advantage of the concurrent nature of the chemical computation for the execution of workflows; this proposal refines and advances the results of an earlier feasibility study on static workflows [13]. The main contributions of this paper are (i) support for a general workflow notation; (ii) separated data and control flow; (iii) established the concept and components of CWE; (iv) analyzed the most important constructs. Apart from the main goals CWE exhibits two interesting features: instead the common push-oriented behavior of existing workflow coordination systems, where the system distributes tasks among the resources, our proposal follows a pull-oriented mode, where resources are actively seeking for tasks furthermore, it provides an automatic parallelization for the execution of workflows: several instances or different workflows can be executed at the same time in the same CWE.

The CWE is however, a conceptual work that is continued in two directions. One is towards realization, there is an ongoing implementation of a fully concurrent, multi-threaded HOCL interpreter that can serve as elementary chemical virtual machines. The CWE model is just a framework that offers the possibility to develop adaptive and self-* mechanisms. The other direction is aimed at investigating and exploring these mechanisms and model further chemical analogons.

ACKNOWLEDGMENT

The research work presented in this paper was partly supported by the European Community’s Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (SCube).

REFERENCES