Chemistry-Inspired Adaptive Stream Processing

Javier Rojas Balderrama, Matthieu Simonin, Cédric Tedeschi

Chemistry-inspired Adaptive Stream Processing

Javier Rojas Balderrama, Matthieu Simonin, and Cédric Tedeschi

IRISA laboratory
Inria / University of Rennes 1, France

Abstract. Stream processing engines have appeared as the next generation of data processing systems, facing the needs for low-delay processing. While these systems have been widely studied recently, their ability to adapt their processing logics at run time upon the detection of some events calling for adaptation is still an open issue.
Chemistry-inspired models of computation have been shown to ease the specification of adaptive systems. In this paper, we argue that a higher-order chemical model can be used to specify such an adaptive SPE in a natural way. We also show how such programming abstractions can get enacted in a decentralised environment.

1 Introduction

In the quest to reducing data processing delays for demanding applications, industry is shifting from the traditional user-driven *store-and-process* approach to a system-driven *on-the-fly processing* approach. The reduction of processing delays is particularly crucial in domains such as social networking, where current trends need to be figured out quickly before they get outdated, environmental systems (e.g., climate and traffic), or military applications (e.g., missile or target detection).

This reduction of processing delays has led to a new generation of Stream Processing Engines (SPEs) addressing the processing of continuous streams of data, while minimizing the end-to-end processing delay. SPE tools and approaches [1, 2, 8, 10, 13] share a common ground in their programming model: the programmer needs to specify a set of *operators* every data item is supposed to traverse. The operators are combined in a directed acyclic graph (DAG). SPEs are closely related to the field of *workflow computing* in which the applications are specified as a DAG of tasks.

For many reasons, this workflow of operators processing an incoming continuous stream of data may have to be adapted at some point at run time. Imagine for instance a weather monitoring system based on a workflow W. In regular conditions of operation, W stays the same, and data sent from the set of sensors allowing to monitor the weather systematically follows the same path. Imagine further that, some particular pattern in the data is detected, meaning that a storm is coming. This calls for a different processing pipeline, specialised in emergency situations. In this state, W needs to be adapted to, say, W' in order to reflect this new processing pipeline. In other words, the program itself

needs to be changed at run time, upon the detection of some particular (possibly complex) event. This kind of adaptiveness cannot afford stopping and restarting the system as it would require too much time.

Chemistry-inspired models of computation have been shown to ease the specification of adaptive systems. In this paper, we argue that a higher-order chemical model can be used to specify such an adaptive SPE in a natural way. We also show how such programming abstractions can get enacted in a decentralised environment.

Section 2 presents related work. In Section 3, the basics of our programming model is introduced. In Section 4, the abstractions for the specification of adaptive workflows are presented and illustrated. In Section 5, our software prototype for decentralised workflow execution is briefly discussed, as well as how to include the support for the new concepts, and Section 6 concludes this work.

2 Related work

There is a longstanding effort to provide suitable frameworks to the scientific community in order to design and enact workflows [16]. However, most of current solutions are not designed to handle stream processing. Moreover, workflow adaptiveness is rarely targeted in scientific workflows. In this section, we present some works describing analogous techniques and models.

In [11], authors propose a framework to compensate for the impedance mismatch between scientific workflows and continuous data streams. They also propose workflow semantics to incorporate stream in scientific workflows. They aim at extending the support for workflow execution in a way that satisfies the following requirements: preserve the workflow programming model for the user; make changes transparent to the workflow engine; and define workflow patterns to use them as new workflow semantics. In a similar way, the work in [19] addresses the lack of integrated support for data models to support emerging applications that are streaming oriented. They propose a scientific workflow framework supporting files, structured collections and data streams. Both approaches place the emphasis upon the programming model rather than the execution model. They clearly state the need for streaming support in scientific workflows for applications that responds to events in the environment at real time, but distributed execution and adaptiveness are not addressed in these works.

Most workflow manager systems ensure enactment flexibility at infrastructure level. Nevertheless, the work presented in [17] proposes an adaptive exception handling at definition level that is comparable to our programming abstractions defined for adaptiveness. The authors propose two patterns to manage the exception handling based on the Reference Nets-within-Nets formalism: propagation and replacement. In spite of mechanisms for dynamically adapting the workflow structure at run time, the resulting representation with their reference model suggests a complex workflow definition, where the original scenario and the alternative path are mixed (expressed in the same description artifact).

Our work envisages the workflow execution as an autonomous process evolving in time according to the requirements and dependencies without bounding to any preset constraint. A similar approach described by Verma et al. [18] proposes a workflow manager system inspired by P-Systems. Nevertheless, they are focused on the elasticity properties of their framework and the associated formalism. They do not cover features such as adaptiveness or stream processing support. In terms of architecture, our approach takes its roots in the work presented in [7]. Although there is an important evolution due to the adaptiveness introduction and the continuous data streams management detailed hereafter.

The idea of using chemical programming to enact workflows autonomously is not new [4, 5, 12]. These works, however, remain abstract, and only few clues are given concerning how to implement such approach, or if it would include centralised or decentralised settings. Again, they do not consider neither streams nor adaptiveness.

3 Preliminaries

In this paper, we rely on the Higher-Order Chemical Language (HOCL) [3].

3.1 HOCL

HOCL is a rule-based language. In HOCL, data is left unstructured in a multiset on which a set of rules is applied concurrently. The role of the programmer is to write this set of rules, which given a particular input multiset will output another multiset containing the results. In other words, the initial multiset of data, containing the input, is *re-written* by the rules, to produce the final multiset, containing the output. Such a programming approach allows users to concentrate on the problem to be solved without having to worry on some external constraints on data structures and control. Let us illustrate the expressiveness of HOCL through the classic *max* problem, which extracts the highest values from a multiset of values. In HOCL, the *max* problem is solved by the following program, given a particular input:

$$\textbf{let } \max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in } \langle 2, 3, 5, 8, 9, \max \rangle$$

The *max* rule consumes two integers x and y when $x \geq y$ and replaces them by x. Initially, several reactions are possible in the provided multiset (between symbols $<$ and $>$), *max* can use any couple of integers satisfying the condition: 2 and 3, 2 and 5, 8 and 9, etc. At run time, the rule will be applied in some order (unknown, and left to the interpreter's developer). Whatever the order is, the final content of the multiset will be $\langle 9 \rangle$.

Looking carefully, we observe that *max* is part of the program. HOCL provides the higher order: rules are first-class citizens in the multiset. In fact, *max* is present in the solution from the beginning to the end of the execution. Also,

a rule can apply on other rules. For instance, removing *max* can be done by structuring the multiset and adding a rule in the initial program.

$$\textbf{let } \max = \textbf{ replace } x, y \textbf{ by } x \textbf{ if } x \geq y \textbf{ in}$$
$$\textbf{let } \text{clean} = \textbf{ replace-one } \langle \max, \omega \rangle \textbf{ by } \omega \textbf{ in } \langle \langle 2, 3, 5, 8, 9, \max \rangle, \text{clean} \rangle$$

The program has been restructured to put our initial program in an outer multiset containing it and a new *clean* rule which will extract the result from the inner multiset, and remove *max* at the same time. However, to be sure that the final (outer) multiset contains the correct result, we need to apply this new rule only when the execution of the inner multiset is completed. This is what the HOCL execution model assumes. Note that the latter rule is a **replace-one** rule. It is one-shot: it will disappear from the multiset once triggered (and completed). The ω symbol has a special connotation as it can match any molecule. In this case, it will match the result.

The chemical analogy is as follows: the multiset is a *solution* in which data *atoms* float and react according to *reaction* rules when they meet. In the following, we adopt the chemical vocabulary to designate artifacts of the programming model. Note that the terms *solution* and *multiset* can be used interchangeably. An atom can be either a simple one (such as a number or a string), or a structured one, such as a subsolution, denoted $\langle A_1, A_2, \ldots, A_n \rangle$, or a tuple denoted $A_1 : A_2 : \cdots : A_n$.

The previous example shows how the program's behaviour can change dynamically through the injection or removal of some rules. It also suggests that the multiset is a container for the state of the program, on which possibly distributed engines can apply rules.

For the sake of simplicity, in the following, we will use the notation $A \to B$ to simplify the specification of the n-shot rule **replace** A **by** B. A one-shot rule will be written $A \to_1 B$. Some of them can be named using the following syntax:

$$\text{RULENAME} : A \to B$$

Ordered collections are manipulated as lists using the following functions:

- `first(l)` returns the first element of the list `l`,
- `rest(l)` returns `l` deprived of its first element,
- `cons(e,l)` returns `l` with the element `e` added at its end, and
- `concat(l1,l2)` returns the concatenation of `l1` and `l2`,

Note that ω denotes any combination of atoms. It is used as a *wildcard molecule*, and `[]` denotes the empty list.

3.2 HOCL and P-systems

Built on top of the principles of chemical programming, *membrane computing*, also called *P-systems*, relies on a structure of nested membranes [15]. The elements floating in them are called molecules. The membranes form a hierarchical

structure, and the membrane containing another membranes is called its parent. An element can move from one membrane to another one which is either its parent or one of its child membranes. These movements between membranes can be used to model communications. In terms of execution model, and following a discrete-time approach, one of their primary objective is to consume as many molecules as possible at each step, in order to try to minimise the global execution time. This execution model constitutes a difference with the execution model of chemical computing where the actual level of parallelism is left to the engine implementor. Other peculiarities of HOCL compared to P-systems, is its ability to model sequential behaviours through *subsolutioning*, and the higher-order. Note however, that many flavours of execution models, especially regarding level of parallelism, have been discussed for P-systems in literature [14]. Let us finally mention the series of work about the MGS system, which is another good example of a series of work where rule-based programming has been investigated in conjunction with membrane systems [9].

4 Programming abstractions for workflows

4.1 Workflow description

We now devise a set of abstractions based on HOCL to program adaptive workflows. Each service taking part in the workflow is represented as a subsolution, and each of these subsolutions will contain a set of atoms modeling queues storing incoming or result data. Let SRC be the set of sources of one given service. Each of these queues is a list ℓ tagged by the parametric keyword IN_i. Then, the set of queues can be written:

$$\{\text{IN}_i : \ell_{\text{IN}_i} \mid i \in \text{SRC}\}$$

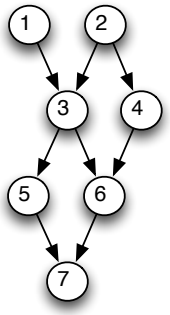
Secondly, a service is equipped with a queue where to put the results of the computation. This queue is unique for each service and is tagged by RES:

$$\text{RES} : \ell_{\text{RES}}$$

Note that a service implementation (the actual binary program producing the output) needs to be specified. This will be simplified as a function name tagged by SRV in the following. To sum up, a service S having the set of sources SRC and the implementation "`func`" can be denoted as follows:

$$S : \langle \{\text{IN}_i : \ell_{\text{IN}_i} \mid i \in \text{SRC}\}, \text{RES} : \ell_{\text{RES}}, \text{SRV} : \text{“}\texttt{func}\text{”} \rangle$$

Let us now give an example of a workflow. Consider the workflow depicted in Fig. 1. It is given in its graphic form, on the left, and in its HOCL description form (i.e., the initial multiset) on the right. Each subsolution S_i acts as a container of the information related to a given service. IN and RES queues, as well as the actual service to be called, are specified within each service's subsolution. All S_i subsolutions act as *contexts*, and delimit the scope of atoms it contains.



$S_1 : \langle$ IN:ℓ_{IN}, RES:[ ], SRV:"`func1`"$\rangle$
$S_2 : \langle$ IN:ℓ_{IN}, RES:[ ], SRV:"`func2`"$\rangle$
$S_3 : \langle$ IN_1:[ ], IN_2:[ ], RES:[ ], SRV:"`func3`"$\rangle$
$S_4 : \langle$ IN_2:[ ], RES:[ ], SRV:"`func4`"$\rangle$
$S_5 : \langle$ IN_3:[ ], RES:[ ], SRV:"`func5`"$\rangle$
$S_6 : \langle$ IN_3:[ ], IN_4:[ ], RES:[ ], SRV:"`func6`"$\rangle$
$S_7 : \langle$ IN_5:[ ], IN_6:[ ], RES:[ ], SRV:"`func7`"$\rangle$

Fig. 1. A workflow DAG and its HOCL description counterpart

For instance, the $\text{IN} : \ell_{\text{IN}}$ atom is present in both S_1 and S_2 but they are two different atoms, representing two different input queues. Note that initially, all queues are empty, except for S_1 and S_2 that are the first services of the workflow and receive their input from the external world.

4.2 Workflow enactment

Let us now describe the set of rules needed to enact this workflow, in an HOCL interpreter. We actually need two types of rules. The first one, denoted CALL, is related to processing. In other words, it applies the operator specified in the atom tagged by the SRV keyword to the set of inputs. Each service's input consists in one element taken from each of the input queues, this represents the set of parameters for the `func-`i operator. For instance, the $\text{CALL}_{1,\emptyset}$ parametric rule to be put within S_1 is:

$$\begin{gathered}\text{IN} : \ell_{\text{IN}}, \text{RES} : \ell_{\text{RES}}, \text{SRV} : \text{“}\texttt{func1}\text{”} \\ \downarrow \\ \text{IN} : \text{rest}\,(\ell_{\text{IN}}), \text{RES} : \text{cons}\,(\texttt{func1}(\text{first}\,(\ell_{\text{IN}})), \ell_{\text{RES}}, \text{SRV} : \text{“}\texttt{func1}\text{”})\end{gathered}$$

This operation takes its input in the head of the IN queue, and puts the result of the invocation of the service it encapsulates at the tail of the RES queue. This rule is specific to S_1. The equivalent for S_3, $\text{CALL}_{3,\{1,2\}}$ is the following—the only noticeable difference, compared to S_1 stands in the fact that one element is taken from both IN queues containing the results of S_1 and S_2, respectively.

$$\begin{gathered}\text{IN}_1 : \ell_{\text{IN}_1}, \text{IN}_2 : \ell_{\text{IN}_2}, \text{RES} : \ell_{\text{RES}}, \text{SRV} : \text{“}\texttt{func3}\text{”} \\ \downarrow \\ \text{IN}_1 : \text{rest}\,(\ell_{\text{IN}_1}), \text{IN}_2 : \text{rest}\,(\ell_{\text{IN}_2}), \text{RES} : \text{cons}\,(\texttt{func3}(\text{first}\,(\ell_{\text{IN}_1}), \text{first}\,(\ell_{\text{IN}_2})), \ell_{\text{RES}})\end{gathered}$$

More generally, the $\text{CALL}_{i,\text{SRC}}$ rule can be seen as a parametric template, parametric by 1) i, the service identifier and 2) SRC, the set of identifiers of the sources of the service. Then, rule $\text{CALL}_{i,\text{SRC}}$ has the following general form:

$$\{\text{IN}_j : \ell_{\text{IN}_j} \mid j \in \text{SRC}\}, \text{RES} : \ell_{\text{RES}}, \text{SRV} : \mathit{func} :: \mathit{String}$$
$$\downarrow$$
$$\{\text{IN}_j : \text{rest}\,(\ell_{\text{IN}_j}) \mid j \in \text{SRC}\}, \text{RES} : \text{cons}\,(\mathit{func}(\{\text{first}\,(\ell_{\text{IN}_j}) \mid j \in \text{SRC}\}), \ell_{\text{RES}})$$

The second type of rules, denoted PASS, enables the information transfer between services. Let us consider the transfer needed from S_1 to S_3. What is needed here is to model the transfer from S_1's queue RES to the queue IN_1 included in the S_3 subsolution:

$$S_1 : \langle \text{RES} : \ell_{\text{RES}}, \omega_1 \rangle, S_3 : \langle \text{IN}_1 : \ell_{\text{IN}_3}, \omega_3 \rangle$$
$$\downarrow$$
$$S_1 : \langle \text{RES} : [\,], \omega_1 \rangle, S_3 : \langle \text{IN}_1 : \text{concat}\,(\ell_{\text{IN}_3}, \ell_{\text{RES}}), \omega_3 \rangle$$

Note that, while the CALL rule makes sense inside the subsolution of the service invoked, in the PASS rule, several subsolutions (source and destinations) are pertained by the action. Similarly, as for the rule modeling the information transfer from S_1, the rule for S_3 is as follows:

$$S_3 : \langle \text{RES} : \ell_{\text{RES}}, \omega_3 \rangle,$$
$$S_5 : \langle \text{IN}_3 : \ell_{\text{IN}_{5,3}}, \omega_5 \rangle, S_6 : \langle \text{IN}_3 : \ell_{\text{IN}_{6,3}}, \omega_6 \rangle$$
$$\downarrow$$
$$S_3 : \langle \text{RES} : [\,], \omega_3 \rangle,$$
$$S_5 : \langle \text{IN}_3 : \text{concat}\,(\ell_{\text{IN}_{5,3}}, \ell_{\text{RES}}), \omega_5 \rangle, S_6 : \langle \text{IN}_3 : \text{concat}\,(\ell_{\text{IN}_{6,3}}, \ell_{\text{RES}}), \omega_6 \rangle$$

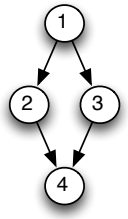
Each time this rule is invoked, it empties the result queues, and transfers all the elements that have been queued since the last application of this rule. More generally, the PASS rules are of the parametric form $\text{PASS}_{src,\text{DST}}$, the parameters being 1) src, the index of the source service, and 2) DST the set of destination services.

$$S_{src} : \langle \text{RES} : \ell_{\text{RES}}, \omega_{src} \rangle, \{S_i : \langle \text{IN}_{src} : \ell_{\text{IN}_{j,src}}, \omega_i \rangle \mid j \in \text{DST}\}$$
$$\downarrow$$
$$S_{src} : \langle \text{RES} : [\,], \omega_{src} \rangle, \{S_i : \langle \text{IN}_{src} : \text{concat}\,(\ell_{\text{IN}_{j,src}}, \ell_{\text{RES}}), \omega_i \rangle \mid j \in \text{DST}\}$$

Let us simplify our example to illustrate a workflow execution. Our simplified workflow is given in Fig. 2 (left). As detailed above, enacting the workflow consists in adding the correct set of CALL and PASS rules at the right locations within the HOCL description to make it a runnable HOCL program. Then, that description is processed by an HOCL interpreter which actually executes the workflow.

The right part of Fig. 2 represents the HOCL program to be submitted to some HOCL interpreter acting as the workflow orchestrator and the initial state of the multiset. For the sake of clarity, we omit the explicit definitions of CALL and PASS rules, as their parameters are sufficient to get fully understood. Initially, only the IN queue of the initial service contains some data: the input data.[1] For

[1] We assume, that after some time, data is sent to the workflow, filling the IN-tagged list in S_1 triggering the workflow.



$S_1 : \langle$ IN:[e_1, e_2], RES:[ ], SRV:"`func1`", $\text{CALL}_{1,\emptyset}\rangle$,
$S_2 : \langle$ IN_1:[ ], RES:[ ], SRV:"`func2`", $\text{CALL}_{2,\{1\}}\rangle$,
$S_3 : \langle$ IN_1:[ ], RES:[ ], SRV:"`func3`", $\text{CALL}_{3,\{1\}}\rangle$,
$S_4 : \langle$ IN_2:[ ], IN_3:[ ], RES:[ ], "`func4`", $\text{CALL}_{4,\{2,3\}}\rangle$,
$\text{PASS}_{1,\{2,3\}}$, $\text{PASS}_{2,\{4\}}$, $\text{PASS}_{3,\{4\}}$

Fig. 2. Workflow definition with CALL and PASS rules

this example, as illustrated in Fig. 2, we assume two inputs e_1 and e_2 have been sent to the workflow to be processed.

Initially, only the $\text{CALL}_{1,\emptyset}$ is enabled. Indeed, to get enabled, a CALL rule needs all the IN queues in the relevant services to be non-empty, which is only the case for S_1. The same applies for the PASS rules: they need the RES queue to be non-empty. So, in a finite time, $\text{CALL}_{1,\emptyset}$ is applied. This rule application encapsulates the invocation of the service, the collection of the results and the pushing of the results in the RES queue of S_1, resulting in the following updated solution (the unchanged part has been greyed out and rules removed for clarity):

$S_1 : \langle$ IN:[e_2], RES:[out_{e_1}], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN_1:[ ], RES:[ ], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN_1:[ ], RES:[ ], SRV:"`func3`"$\rangle$,
$S_4 : \langle$ IN_2:[ ], IN_3:[ ], RES:[ ], SRV:"`func4`"$\rangle$

At this point, two rules are enabled, namely $\text{CALL}_{1,\emptyset}$ and $\text{PASS}_{1,\{2,3\}}$. In other words, it is possible to process e_2 as well as to transfer a first S_1 result to S_2 and S_3. We can verify that all other rules are disabled. Assuming both rules are applied, the resulting intermediate multiset is the following (again, unchanged lines/subsolutions have been greyed out):

$S_1 : \langle$ IN:[ ], RES:[out_{e_2}], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN_1:[out_{e_1}], RES:[ ], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN_1:[out_{e_1}], RES:[ ], SRV:"`func3`"$\rangle$,
$S_4 : \langle$ IN_2:[ ], IN_3:[ ], RES:[ ], SRV:"`func4`"$\rangle$

Now, S_2 and S_3 can be called in parallel through the concurrent application of $\text{CALL}_{2,\{1\}}$ and $\text{CALL}_{3,\{1\}}$ rules, respectively ($\text{PASS}_{1,\{2,3\}}$ could also be triggered but the model does not enforce the application of all enabled rules), leading to the following multiset's state:

$S_1 : \langle$ IN:[ ], RES:[out_{e_2}], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN_1:[ ], RES:[$\text{out}_{e_{12}}$], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN_1:[ ], RES:[$\text{out}_{e_{13}}$], SRV:"`func3`"$\rangle$,

$S_4 : \langle$ IN$_2$:[ ], IN$_3$:[ ], RES:[ ], SRV:"`func4`"$\rangle$

While the second result of S_1 is transferred to S_2 and S_3, results of S_2 and S_3 can also be sent to S_4. These actions are enabled through the concurrent application of PASS$_{1,\{2,3\}}$, PASS$_{2,\{4\}}$ and PASS$_{3,\{4\}}$ with the following outcome:

$S_1 : \langle$ IN:[ ], RES:[ ], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN$_1$:[out$_{e_2}$], RES:[ ], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN$_1$:[out$_{e_2}$], RES:[ ], SRV:"`func3`"$\rangle$,
$S_4 : \langle$ IN$_2$:[out$_{e_{12}}$], IN$_3$:[out$_{e_{13}}$], RES:[ ], SRV:"`func4`"$\rangle$

We are now in a state where CALL rules can be applied, in S_2, S_3 and S_4. Assuming they are applied, the new multiset state is the following:

$S_1 : \langle$ IN:[ ], RES:[ ], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN$_1$:[ ], RES:[out$_{e_{22}}$], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN$_1$:[ ], RES:[out$_{e_{23}}$], SRV:"`func3`"$\rangle$,
$S_4 : \langle$ IN$_2$:[ ], IN$_3$:[ ], RES:[out$_{e_{14}}$], SRV:"`func4`"$\rangle$

The remainder of the execution consists in 1) transferring the second set of results from the RES queues of S_2 and S_3 to the IN queues in S_4, and 2) invoking `func4` on them, leading to the following final inert multiset (as long as no new data is injected in the IN queue of S_1).

$S_1 : \langle$ IN:[ ], RES:[ ], SRV:"`func1`"$\rangle$,
$S_2 : \langle$ IN$_1$:[ ], RES:[ ], SRV:"`func2`"$\rangle$,
$S_3 : \langle$ IN$_1$:[ ], RES:[ ], SRV:"`func3`"$\rangle$,
$S_4 : \langle$ IN$_2$:[ ], IN$_3$:[ ], RES:[out$_{e_{14}}$, out$_{e_{24}}$], SRV:"`func4`"$\rangle$

The idea behind using parametric rules is that they can be easily constructed using the rule template and the parameters given by the user in its workflow description. The idea is not necessarily to let the user write the HOCL code directly, but to generate the HOCL code from the user's own description.

4.3 Adaptiveness

When workflow reconfiguration is needed, the set of atoms (data and rules) describing the workflow needs to get updated. Let us consider the adaptive workflow depicted in Fig. 3, along with its HOCL code.

As illustrated on the left of Fig. 3, an alternate workflow is specified by the services and links in dashed lines, to replace Service 3 in case adaptation is requested. More specifically, the two services $a1$ and $a2$ are to replace Service 3. The two last lines of the initial workflow specify these two services. Services $a1$

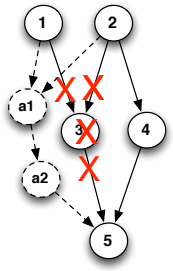
and $a2$ are initially *disabled* in the sense that no PASS rule transfers data into them. Note also the UPDATE_PASS and UPDATE_S_5 (within S_5 subsolution) rules whose purpose is to *re-branch* the workflow upon adaptation.

The basic idea is to enable the additional services only if the adaptation is required (i.e., when the reason for adapting is satisfied). As discussed above, the need for adaptations can take several forms. For the sake of simplicity, we will assume that the execution monitoring system is simply made to inject some particular atom in the multiset, so as to trigger the adaptation. In our example, we will use the ADAPT atom keyword to reflect this. The execution monitoring system will inject it where needed, namely, at the level of the multiset, and also inside the S_5 subsolution.

When the need for adaptation is declared, the multiset requires to get updated on-the-fly so as to enable services $a1$ and $a2$, and partially redirect the data flow accordingly. To modify the path of data some PASS rules need to be removed and replaced by $\text{PASS}_{1,\{a1\}}$, $\text{PASS}_{2,\{a1,4\}}$, $\text{PASS}_{a1,\{a2\}}$ and $\text{PASS}_{a2,\{5\}}$. The specific higher-order one-shot rule UPDATE_PASS, defined below, achieves this:

$$\begin{gathered}\text{ADAPT},\ \text{PASS}_{1,\{3\}},\ \text{PASS}_{2,\{3,4\}},\ \text{PASS}_{3,\{5\}}\\ \downarrow_1\\ \text{PASS}_{1,\{a1\}},\ \text{PASS}_{2,\{a1,4\}},\ \text{PASS}_{a1,\{a2\}},\ \text{PASS}_{a2,\{5\}}\end{gathered}$$

The other update to perform concerns the internals of S_5, the IN_3 queue needs to be removed and replaced by an IN_{a2} queue, in order to satisfy the indices. The



$S_1 : \langle\ \text{IN}{:}[\text{e}_1, \text{e}_2],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func1}\text{”},\ \text{CALL}_{1,\emptyset}\rangle,$
$S_2 : \langle\ \text{IN}{:}[\text{e}_1, \text{e}_2],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func2}\text{”},\ \text{CALL}_{2,\emptyset}\rangle,$
$S_3 : \langle\ \text{IN}_1{:}[\ ],\ \text{IN}_2{:}[\ ],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func3}\text{”},\ \text{CALL}_{3,\{1,2\}}\rangle,$
$S_4 : \langle\ \text{IN}_2{:}[\ ],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func4}\text{”},\ \text{CALL}_{4,\{2\}}\rangle,$
$S_5 : \langle\ \text{IN}_3{:}[\ ],\ \text{IN}_4{:}[\ ],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func5}\text{”},\ \text{CALL}_{5,\{3,4\}},\ \text{UPDATE\_}s_5\rangle,$
$\text{PASS}_{1,\{3\}},\ \text{PASS}_{2,\{3,4\}},\ \text{PASS}_{3,\{5\}},\ \text{PASS}_{4,\{5\}},$

UPDATE_PASS,
$S_{a1} : \langle\ \text{IN}_1{:}[\ ],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func-a1}\text{”},\ \text{CALL}_{a1,\{1\}}\rangle,$
$S_{a2} : \langle\ \text{IN}_{a1}{:}[\ ],\ \text{RES}{:}[\ ],\ \text{SRV}{:}\text{“}\texttt{func-a2}\text{”},\ \text{CALL}_{a2,\{a1\}}\rangle$

Fig. 3. An adaptive workflow, including its HOCL description

S_5 CALL rule needs to be updated accordingly from $\text{CALL}_{5,\{3,4\}}$ to $\text{CALL}_{5,\{a2,4\}}$, as specified by the UPDATE_S_5 rule:

$$\begin{array}{c}\text{ADAPT, IN}_3{:}\ell_{\text{IN}}, \text{CALL}_{5,\{3,4\}}\\ \downarrow_1\\ \text{IN}_{a2}{:}[\,], \text{CALL}_{5,\{a2,4\}}\end{array}$$

This rule is supposed to take place within the S_5 subsolution. These rules are initially present in the multiset, but it can react only if the ADAPT atoms have been injecting by the monitoring system. Let us review the execution of such an example. Initially, we have the multiset described in Fig. 3. The data traverses the graph similarly as for the previous workflow. At some point however, the monitoring system decides that the alternate workflow needs to be triggered, leading to the following multiset:

$S_1 : \langle\ \text{IN}{:}\ell_{\text{IN}_1}, \text{RES}{:}\ell_{\text{RES}_1}, \text{SRV}{:}\text{“}\texttt{func1}\text{”}, \text{CALL}_{1,\emptyset}\rangle,$
$S_2 : \langle\ \text{IN}{:}\ell_{\text{IN}_2}, \text{RES}{:}\ell_{\text{RES}_2}, \text{SRV}{:}\text{“}\texttt{func2}\text{”}, \text{CALL}_{2,\emptyset}\rangle,$
$S_3 : \langle\ \text{IN}_1{:}\ell_{\text{IN}_{31}}, \text{IN}_2{:}\ell_{\text{IN}_{32}}, \text{RES}{:}\ell_{\text{RES}_3}, \text{SRV}{:}\text{“}\texttt{func3}\text{”}, \text{CALL}_{3,\{1,2\}}\rangle,$
$S_4 : \langle\ \text{IN}_2{:}\ell_{\text{IN}_{42}}, \text{RES}{:}\ell_{\text{RES}_4}, \text{SRV}{:}\text{“}\texttt{func4}\text{”}, \text{CALL}_{4,\{2\}}\rangle,$
$S_5 : \langle\ \text{IN}_3{:}\ell_{\text{IN}_{53}}, \text{IN}_4{:}\ell_{\text{IN}_{54}}, \text{RES}{:}\ell_{\text{RES}_5}, \text{SRV}{:}\text{“}\texttt{func5}\text{”}, \text{CALL}_{5,\{3,4\}},$
$\quad \text{UPDATE}_{S_5}, \text{ADAPT}\rangle,$
$\text{PASS}_{1,\{3\}}, \text{PASS}_{2,\{3,4\}}, \text{PASS}_{3,\{5\}}, \text{PASS}_{4,\{5\}},$

UPDATE_PASS, ADAPT,
$S_{a1} : \langle\ \text{IN}_1{:}[\,], \text{RES}{:}[\,], \text{SRV}{:}\text{“}\texttt{func-a1}\text{”}, \text{CALL}_{a1,\{1\}}\rangle,$
$S_{a2} : \langle\ \text{IN}_{a1}{:}[\,], \text{RES}{:}[\,], \text{SRV}{:}\text{“}\texttt{func-a2}\text{”}, \text{CALL}_{a2,\{a1\}}\rangle$

At this point, due to the presence of ADAPT at both global and S_5 levels, the two UPDATE_* rules are triggered, leading to the following multiset, where PASS rules have been replaced and where S_5 has been updated—note that the UPDATE rules, one-shot, have been removed in the reactions:

$S_1 : \langle\ \text{IN}{:}\ell_{\text{IN}_1}, \text{RES}{:}\ell_{\text{RES}_1}, \text{SRV}{:}\text{“}\texttt{func1}\text{”}, \text{CALL}_{1,\emptyset}\rangle,$
$S_2 : \langle\ \text{IN}{:}\ell_{\text{IN}_2}, \text{RES}{:}\ell_{\text{RES}_2}, \text{SRV}{:}\text{“}\texttt{func2}\text{”}, \text{CALL}_{2,\emptyset}\rangle,$
$S_3 : \langle\ \text{IN}_1{:}\ell_{\text{IN}_{31}}, \text{IN}_2{:}\ell_{\text{IN}_{32}}, \text{RES}{:}\ell_{\text{RES}_3}, \text{SRV}{:}\text{“}\texttt{func3}\text{”}, \text{CALL}_{3,\{1,2\}}\rangle,$
$S_4 : \langle\ \text{IN}_2{:}\ell_{\text{IN}_{42}}, \text{RES}{:}\ell_{\text{RES}_4}, \text{SRV}{:}\text{“}\texttt{func4}\text{”}, \text{CALL}_{4,\{2\}}\rangle,$
$S_5 : \langle\ \text{IN}_{a2}{:}[\,], \text{IN}_4{:}\ell_{\text{IN}_{54}}, \text{RES}{:}\ell_{\text{RES}_5}, \text{SRV}{:}\text{“}\texttt{func5}\text{”}, \text{CALL}_{5,\{3,4\}}\rangle,$
$S_{a1} : \langle\ \text{IN}_1{:}[\,], \text{RES}{:}[\,], \text{SRV}{:}\text{“}\texttt{func-a1}\text{”}, \text{CALL}_{a1,\{1\}}\rangle,$
$S_{a2} : \langle\ \text{IN}_{a1}{:}[\,], \text{RES}{:}[\,], \text{SRV}{:}\text{“}\texttt{func-a2}\text{”}, \text{CALL}_{a2,\{a1\}}\rangle,$
$\text{PASS}_{1,\{a1\}}, \text{PASS}_{2,\{a1,4\}}, \text{PASS}_{a1,\{a2\}}, \text{PASS}_{a2,\{5\}}, \text{PASS}_{4,\{5\}}$

As soon as it is updated, the new workflow is operational, new data flows being specified by the newly injected rules.

To sum up, two types of workflow-specific one-shot higher-order rules are needed to enhance the adaptation of the streaming workflow at run time: one

to update the PASS rule and one to update the internals of the services whose sources have changed in the update. Again, these rules can be generated from a high-level description of the workflow provided by the user.

5 Architecture and implementation

5.1 Decentralised Architecture

We plan to implement the programming abstractions presented above in GinFlow[2], a software initially developed in the context of service composition, and whose architecture, implementation and experimentation has been recently detailed in [7]. These works intend to decentralise the execution of workflows and rely on a shared space to coordinate services involved in a composition. Specifically, it sits on the HOCL language to describe the service composition (*workflow* of services) and its enactment. In this architecture, services are encapsulated into agents communicating by reading and writing information in a shared data space.

The architectural model is depicted in Fig. 4. As detailed in [6], the shared space contains the description of the workflow. During enactment, each time the execution moves forward, this description is updated so as to reflect the execution progress. The service agents (SAs) are essentially workers that encapsulate the invocation of the services. This encapsulation includes an engine able to read, interpret and update the information contained in the shared space. For instance, when a SA completes the invocation of a service and collects the result, it pushes this information to the shared space, allowing another service agent, which was waiting for this result, to collect this result and use it as input to invoke its own service.

In the software prototype built and experimented in [7], each service agent taking part in the workflow is composed of three elements as shown in Fig. 5.

[2] http://ginflow.inria.fr

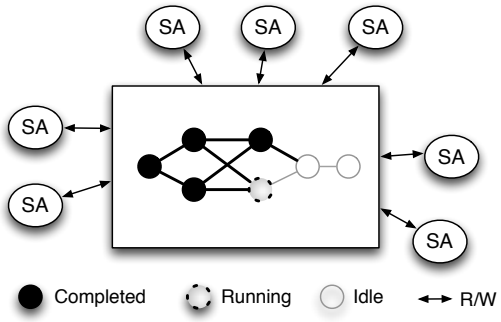


Fig. 4. Shared space-based coordination

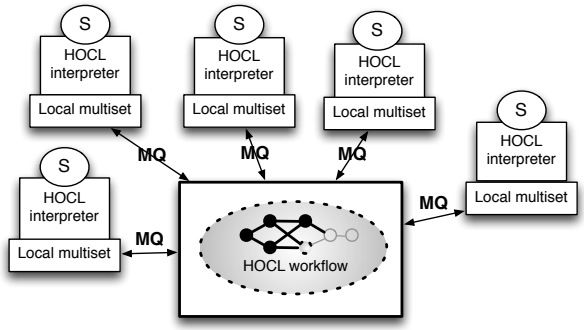


Fig. 5. Software prototype architecture

The first element is the service to invoke (S), or a wrapper of an application representing the service. The second element is a storage place for a local partial copy of the multiset. This local copy acts as a cache of the service's subsolution. The third element is an HOCL interpreter that reads and updates the local copy of the multiset each time it tries to apply one of the rule in the subsolution. Communication between the multiset and the agents is done through the use of ActiveMQ.[3]

5.2 Decentralising the adaptive stream processing rules

Let us now review how the CALL, PASS and UPDATE rules can be implemented on top of this decentralised architecture.

Firstly, when CALL rules are applied, the service is called from the HOCL interpreter and the result is injected into the local multiset. Secondly, the PASS rule is supposed to act from outside subsolutions since it requires to match the atoms from several subsolutions. To avoid the need for a monitoring system having the global view of the system and keep the control decentralised[4], the PASS rules are modified to act within a subsolution. In other words, a $\text{PASS}_{i,\text{DST}}$ rule will be placed inside the S_i service and get triggered by the HOCL interpreter of the SA encapsulating S_i. Once the result of the invocation of some service is collected and put in the local RES queue, it triggers the local version of the PASS rule which sends a message to the destination, via the multiset. When the message is received in the multiset, it is automatically pushed to the right subsolution. For instance, the $\text{PASS}_{1,\{3\}}$ rule will be within the S_3 subsolution and look like this:

$$\begin{gathered}\text{RES} : \ell_{\text{RES}}, \omega_1 \\ \downarrow \\ \text{RES} : [\,], \texttt{transfer}(\ell_{\text{RES}}, S_3), \omega_1\end{gathered}$$

[3] http://activemq.apache.org/

[4] Each SA is allowed to store only its own description.

The `transfer()` method sends a message to the multiset, which will update the state of the S_3 subsolution by adding ℓ_{RES} in its IN_1 queue and push it to the S_3 SA through ActiveMQ.

Finally, to make the UPDATE rule work in these decentralised settings, we need to slightly improve the program. We will first assume that each SA is equipped with a monitoring system able to inject the ADAPT atom within the local multiset managed by the SA. Then, we use that SA, upon the appearance of the ADAPT atom, to change the PASS rules to reflect the new dataflow. In fact, instead of only one UPDATE rule, we need one UPDATE_i rule for each service that requires to update its dataflow, and include it within each S_i subsolution:

$$\begin{gathered}\text{UPDATE}_1\colon \text{PASS}_{1,\{3\}} \rightarrow \text{PASS}_{1,\{a1\}} \\ \text{UPDATE}_2\colon \text{PASS}_{2,\{3,4\}} \rightarrow \text{PASS}_{2,\{a1,4\}} \\ \text{UPDATE}_5\colon \text{IN}_3 : \ell_{\text{RES}} \rightarrow \text{IN}_{a1}\end{gathered}$$

These three rules, initially absent from the multiset, will be injected by the SA who detected the failure in a manner similar to the PASS process. Assume that the ADAPT atom has been injected in the S_3 service. The following rule, put within the S_3 subsolution will make the UPDATE_i rules appear in the relevant subsolutions, in a fashion similar to how the PASS rule transfers results:

$$\begin{gathered}\text{TRIGGER-ADAPT: ADAPT} \\ \downarrow \\ \texttt{transfer}(\text{UPDATE}_1,\, S_1),\, \texttt{transfer}(\text{UPDATE}_2,\, S_2),\, \texttt{transfer}(\text{UPDATE}_5,\, S_5)\end{gathered}$$

As soon as the UPDATE_i rule appears in the S_i subsolution, it is triggered by the HOCL interpreter of its SA encapsulating it. Every UPDATE_i can be triggered concurrently, thus realising the update concurrently.

6 Conclusions

We have presented a set of abstractions using chemistry-inspired programming model for adaptive decentralised workflows supporting continuous dataflows. We have described the rules to define workflows, modify their behaviour when exceptions are raised, and ensure service invocation management with data streams. We also have shown the generic approach of services execution based on parametric rules. These rules enable a workflow enactment taking advantage of a distributed execution environment. The use of high-level definitions encompasses a concise and clear description of workflows delegating the complex instrumentation to the workflow engine. These definitions allow users to specify the workflow and its alternate paths at design time. Future work includes testing the framework with real use-cases and evolve towards better scheduling.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik,

S.B.: The design of the Borealis stream processing engine. In: Second Biennial Conference on Innovative Data Systems Research. Asilomar, CA (Jan 2005)
2. Apache Software Foundation: Apache Storm. https://storm.apache.org/
3. Banâtre, J.P., Fradet, P., Radenac, Y.: Generalised multisets for chemical programming. Mathematical Structures in Computer Science 16(4), 557–580 (2006)
4. Caeiro, M., Németh, Z., Priol, T.: A chemical model for dynamic workflow coordination. In: The 19th Euromicro International Conference on Parallel, Distributed and network-based Processing. Ayia Napa, Cyprus (Feb 2011)
5. Di Napoli, C., Giordano, M., Németh, Z., Tonellotto, N.: Adaptive instantiation of service workflows using a chemical approach. In: The 16th International Euro-Par Conference on Parallel Processing. Ischia, Italy (Aug 2010)
6. Fernández, H., Priol, T., Tedeschi, C.: Decentralized approach for execution of composite Web services using the chemical paradigm. In: The 8th IEEE International Conference on Web Services. Miami, FL (Jul 2010)
7. Fernández, H., Tedeschi, C., Priol, T.: Rule-driven service coordination middleware for scientific applications. Future Generation Computer Systems 35, 1–13 (2014)
8. Gedik, B., Andrade, H., Wu, K., Yu, P.S., Doo, M.: SPADE: The System S declarative stream processing engine. In: The ACM SIGMOD International Conference on Management of Data. Vancouver, Canada (Jun 2008)
9. Giavitto, J., Michel, O.: MGS: a rule-based programming language for complex objects and collections. Electr. Notes Theor. Comput. Sci. 59(4), 286–304 (2001), http://dx.doi.org/10.1016/S1571-0661(04)00293-2
10. Gulisano, V., Jiménez-Peris, R., Patiño-Martínez, M., Soriente, C., Valduriez, P.: StreamCloud: An elastic and scalable data streaming system. IEEE Transactions on Parallel and Distributed Systems 23(12), 2351–2365 (2012)
11. Herath, C., Plale, B.: Streamflow programming model for data streaming in scientific workflows. In: The 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. Melbourne, Australia (May 2010)
12. Németh, Z., Pérez, C., Priol, T.: Workflow enactment based on a chemical methaphor. In: The 3rd IEEE International Conference on Software Engineering and Formal Methods. Koblenz, Germany (Sep 2005)
13. Neumeyer, L., Robbins, B., Nair, A., Kesari, A.: S4: Distributed stream computing platform. In: The 13th IEEE International Conference on Data Mining Workshops. Sydney, Australia (Dec 2010)
14. Paun, G., Rozenberg, G., Salomaa, A.: The Oxford Handbook of Membrane Computing. Oxford University Press, Inc., New York, NY, USA (2010)
15. Păun, G.: Computing with membranes. Journal of Computer and System Sciences 61(1), 108–143 (2000)
16. Taylor, I.J., Deelman, E., Gannon, D.B., Shields, M. (eds.): Workflows for e-Science. Springer (2007)
17. Tolosana-Calasanz, R., Bañares, J.A., Rana, O.F., Álvarez, P., Ezpeleta, J., Hoheisel, A.: Adaptive exception handling for scientific workflows. Concurrency and Computation: Practice and Experience 22(5), 617–642 (2010)
18. Verma, R., Ahmed, T., Srivastava, A.: Expressing workflow and workflow enactment using P systems. In: The 15th International Conference on Membrane Computing. Prague, Czech Republic (Aug 2014)
19. Zinn, D., Hart, Q., McPhillips, T., Ludäscher, B., Simmhan, Y., Giakkoupis, M., Prasanna, V.K.: Towards reliable, performant workflows for streaming-applications on cloud platforms. In: The 11th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing. Newport Beach, CA (May 2011)