Process Enactment and Coordination

Jean-Marc Andreoli, Jean-Luc Meunier, Daniele Pagani

Rank Xerox Research Centre, Grenoble, France

Abstract. This paper investigates the relationship between systems to enact software processes and systems to coordinate distributed, heterogeneous and concurrent objects. In particular, we describe in detail how one of these coordination systems systems—the "Coordination Language Facility" (CLF), developed at the Rank Xerox Research Centre—can be used to model and execute a sample software development process: bug reporting. The main advantages of using CLF are: i) language facility that allows to dynamically change both the core process model and the application-specific process templates; ii) modular architecture that allows to easily reconfigure, migrate and replicate each process component in a distributed, heterogeneous environment.

1 Introduction

The field of Software Process Technology (SPT) pioneered the research on how computer systems can support the software development process. Several systems originally designed to support software processes have been applied to the domain of office procedures and business processes (e.g., ProcessWeaver [13]). On the other hand, research in Office Information Systems (OIS) and Computer-Supported Cooperative Work (CSCW) led to a number of research prototypes and products aiming at supporting "office work" and "knowledge work" within customer-focused business processes, usually referred to as "workflow management systems". Some workflow systems have been applied to support software development processes, e.g. [23]. As Chroust argued [7], the requirements of software processes and business processes are indeed different; however, there is a large degree of overlap in the *coordination primitives* required to orchestrate people (workers, programmers, analysts, managers, etc.) and software tools (document management systems, databases, legacy applications, compilers, debuggers, etc.).

Some authors ([18], [15]) already addressed the key role of coordination in software development. On the other hand, the research on the coordination of concurrent (object-oriented) systems led to the definition of several systems to model and execute the coordination of distributed and heterogeneous software components ([9], [10]). In this paper, we investigate how one of these coordination systems —the "Coordination Language Facility" (CLF) [1], developed at the Rank Xerox Research Centre—can be used to support knowledge work processes in general, and software development processes in particular.

The main advantages of using the CLF are:

i) flexibility in the work processes that can be specified and enacted. The CLF language provides sort of "process assembly language" that allows programmers to dynamically change both the core process model (i.e., activity states and each activity is executed) and the application-specific process templates (i.e., the network of dependencies among activities for a specific user process).

ii) support for coordination in a truly distributed, wide-area computing environment. The stateless, resource-based approach of the CLF and the reflection feature of the CLF allow to design a process enactment environment with a very modular architecture, where each component can be dynamically migrated to a different machine and/or replicated for reliability or performance reasons, even during execution.

The paper is structured as follows: section 2 starts with a more detailed discussion of the coordination requirements of systems to support business processes and software processes. Then section 2.2 compares the coordination approach of the CLF with other SPT systems and section 2.3 presents a short outline of the CLF. In section 3, we show how the CLF can be used to implement, and also enhance, the IPO process model; then we

discuss the benefits of the CLF approach. In section 4 we discuss a sample application in the domain of software development: the software defect report process. Finally, section 5 concludes the paper.

2 Process and Coordination Models

2.1 Coordination in Knowledge Work and Software Development Processes

Most systems to support knowledge work and software development processes share a common underlying paradigm: input-process-output (IPO) [6]. A process is modelled as a "black box" that transforms inputs into outputs (Figure 1). The IPO approach exploits hierarchical decomposition to specify at increasing levels of detail how a process is made up of sub-processes. At the most detailed level, the process is made up of atomic activities or tasks executed by one worker (activity is the name standardised by the Workflow Management Coalition [26]). Processes, sub-processes and activities are interconnected in a network of dependencies that represents the flow of control and information.

Each individual activity is modelled as a black box with three basic states:
- inactive: the activity cannot be executed because the previous dependencies have not been satisfied yet;
- active: the previous dependencies are satisfied, the activity can be processed;
- done: the activity has produced the desired output and the following dependencies can be triggered.
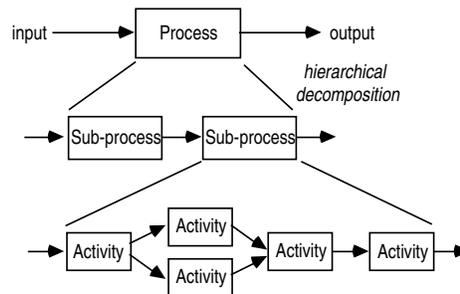

Figure 1: *Traditional IPO paradigm*

Most workflow and software process systems also have other states to represent special situations such as: activity ready to be executed but not yet assigned to a user; activity no longer reachable because a branch has been skipped; activity offered but not accepted; activity delegated, etc. These additional states, however, can be seen as minor refinements to the IPO paradigm; the inactive-active-done state diagram is the common denominator to most IPO process enactment systems.
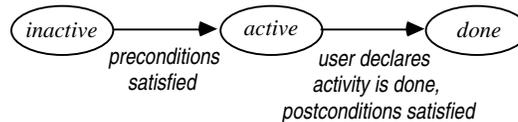

Figure 2: *Simplest state diagram of an activity*

An IPO-based process enactment engine performs two main coordination functions:

- **coordinating the flow**: pacing the activation of activities as specified in the network of dependencies among activities. Typical flow control primitives include sequence, parallel execution, branching, looping, etc.

- **coordinating the work**: monitoring the execution of each activity. In most process enactment systems available on the market, executing an activity involves the following functions: role resolution to find the users eligible to perform the task; offering the task to the eligible users through some kind of task list manager (G)UI and assigning the task to one (and only one) of them; sending the relevant data and

documents to the users' desktop; launching the relevant software tools, e.g. word processor and legacy applications; displaying some kind of (G)UI for instructing the user about the task, let the user access to data and documents, let the user send in the new data/documents s/he produced, and let the user notify the system that the activity has been performed; perform consistency checks on the data and documents submitted by the user.

In the next section, we will describe how the coordination of flow and work can be modelled and enacted with the CLF, a platform to coordinate distributed, heterogeneous, and concurrent objects.

2.2 Coordination Models

The need for coordination tools arose in the early days of operating systems, to fill the gap between the low level primitives offered by operating systems and the high level constructs required by application programmers. For applications developed from scratch, traditional programming languages such as C or Fortran were sufficient to achieve this goal, but they proved too rigid for applications manipulating and integrating existing software components developed independently. Coordination can be either implicit or explicit.

- In implicit coordination, the programmer is not aware of the other applications which may run concurrently with his/her own program. The system only guarantees that, overall, the concurrent applications do not conflict in the modifications they perform on their environment. Typically, the environment is modeled as a database and applications are coordinated for consistency through traditional ACID transactions.

- At the other hand of the spectrum stand scripting languages where the coordination of potentially concurrent applications is fully explicited in a program (script) and the system only supports communication (usually message passing) between the applications.

Process centered development environments usually provide a mixture of these two aspects of coordination. On the one hand, they store the various artifacts involved in a software process (pieces of code, requirements, documentation, test cases, etc.) into a database on which applications can perform transactions. On the other hand, they usually incorporate a scripting language in which the different constituent activities of a process can be explicitly related, thus going beyond the traditional assumption of total isolation between transactions. The coordination constructs offered by such languages derive from traditional existing programming language, either imperative (Saga [14] or Contracts [25]) or declarative (Adaptors [27]).

Rule based languages have been particularly successful in software process modelling, using either production rules (a la OPS5) or deduction rules (a la Prolog) or a mixture of both, combining both forward and backward chaining, as in many Expert System shells. Marvel/Provence [3] and Merlin [17] are typical representatives of rule based software development environments. Rules may be used either to impose relations between the objects (or object attributes) of the data model representing the software artifacts (Inference rules), or to react to events generated by these objects (Event-Condition-Action rules [11]). This requires a deep integration of the data and the process model, realised in the underlying database.

Thus, it appears that a database (or, possibly, multi-database) system is a central component of most process centered software development environments. Such relatively heavy-weight solutions are appropriate for large size, centrally managed projects developed in a homogeneous environment. However, for smaller size projects, involving geographically distributed teams with heterogeneous platforms, a more light-weight solution may be more appealing. The system Oz [5], for example, tackles the issues of distribution but still assumes the heavy infrastructure of a database system. We believe that flexible software development tools, requiring minimal infrastructure in a weakly integrated environment, still deserve attention. Such tools would be able to loosely coordinate, in the workflow sense, the activities of different agents (both human and software) distributed over different, autonomous sites. For this purpose, we have developed the Coordination Language Facility (CLF), a light-weight Process Oriented Middleware aimed at supporting various kinds of work processes in a distributed, heterogeneous setting.

The CLF uses rules as its main process modelling tool, but CLF rules are not connected to any specific data model (e.g. encoded in a database) and aim at coordinating any kind of applications. They do not differentiate between the applications which hold the state of the process and the applications which enact the activities of the process: they are all treated at the same level, as (active) objects to be coordinated. This provides a more uniform view of the different actors of a process, and better modularity. CLF rules are production-like kinds of rules

which act on consumable resources and not simply on facts, as in the Expert-system kind of approach taken in, e.g., Merlin [17]. They do not assume any Linda-like shared tuple space infrastructure as in Extended Shared Prolog [8] used in Oikos [19]: the only required infrastructure is an object request broker. Finally, CLF rules are pro-active, as opposed to re-active ECA rules used in Adele [4]. With pro-active rules, subscription to triggering events may be done not only when the rule is activated, as with re-active rules, but also dynamically while the rule is executing: progression in the rule execution may generate new subscriptions. As in most systems, CLF rule based processes can be modified dynamically, since CLF rules can be manipulated within a process which can itself be defined by rules (full reflectivity).

2.3 The Coordination Language Facility (CLF)

The Coordination Language Facility (CLF), developed at the Rank Xerox Research Centre, is a typical representative of the declarative approach to coordination [1]. It provides a high level scripting language for the coordination of autonomous active objects, but, unlike traditional scripting languages (e.g AppleScript [2]), it relies on a richer object model offering more refined communication primitives.

2.3.1 The CLF Object Model and Protocol

The CLF object model assumes that each object can be viewed as a *resource manager*, which accepts basically two types of operations: removal and insertion of specified resources. Resources are not visible directly from the outside world but can be accessed through their properties, defined in the interface of the object. The underlying architecture is client/server where the client tries to insert/remove resources from the server. This is achieved through the following protocol:

Inquiry: the client *inquires* whether the server holds (or can produce) a resource satisfying a given property of its interface. The server returns a stream of actions that it could perform to make such a resource available and remove it. The returned stream may be empty (no action can produce a resource satisfying the given property, neither now nor in the future) or infinite (a new action is added to the stream each time the server changes its state and a new resource satisfying the given property becomes available). Hence, this phase of the interaction is "defered synchronous".

Reservation: the client asks the server to *reserve* one of the actions returned during the Inquiry phase. If an action is successfully reserved, the server commits to perform it on request (i.e. remove the corresponding resource). Failure may occur when an action returned in the Inquiry phase is no more available at the time of the Reservation. Notice that the server must also handle conflicting reservations on "one-shot" actions (e.g. actions sharing unique resources). This phase of the interaction returns one result (success or failure) and is therefore "synchronous".

Confirmation/Cancellation: the client may either *confirm* or *cancel* an action it has succesfully reserved. If the reservation is confirmed, the action must be executed, leading to the deletion of the corresponding resource; if cancelled, the action may become available again to other reservation requests. No result is expected in this phase of the interaction, which cannot fail: it is purely "asynchronous".

Insert: the client requests to *insert* into the server some resource satisfying a given property. This is also purely asynchronous.

There are other operations in the protocol, which we omit here for simplicity, and in particular **Check** and **Kill** which allow garbage collection, in a server, of Inquiries which a client is no longer interested in.

Notice that the Inquiry operation starts a long lived process on the server, in charge of warning the client each time a resource satisfying the given property becomes available. This subscription mechanism is usually found in message oriented middleware (e.g. ToolTalk [24] or MQ-Series [16]). The Reserve/Confim/Cancel operations of the protocol provide the basis for elementary transactions based on a two-phase commit protocol: a set of actions can be performed atomically by reserving each of them separately, then either confirm them all if all the reservations have suceeded or cancel the successful ones if some reservation has failed[1]. Thus, the CLF object

---

[1] In fact, a reservation may neither succeed nor fail, but return a special value indicating that deadlock is suspected; if some of the reservations return that value while the others succeed, the successful ones are cancelled and the transaction may be retried later.

model combines at the lowest level and in an abstract form the facilities offered by both message oriented architectures and transactions.

The CLF protocol can be implemented on top of any transport layer (currently, Corba [20] and HTTP are supported, but other protocols such as Microsoft COM/OLE and OpenDoc OpenEvents are considered). A CLF object can be implemented in any language, and can encapsulate any kind of resources, as long as it accepts the CLF protocol. A library of predefined general purpose classes of objects (e.g. various flavors of bags of simple tuple structures) are available and can be derived to produce more sophisticated classes. Furthermore, legacy applications can easily be encapsulated inside CLF objects, once the resources they manipulate and the properties by which they should be made visible are clearly identified. In the case of databases, the obvious choice is to identify resources with records, or objects in the case of OODB, but less fine grained resources (e.g. relations or meta-information) can also be considered.

2.3.2 The CLF Scripting Language

The CLF scripting language can be used to program (possibly remotely) a special kind of CLF objects, called *coordinators*, whose role is simply to coordinate operations (i.e. resource manipulations) on other objects, called *participants*. Basically, the resources of a coordinator are scripting rules. Each rule is divided into left- and right-hand sides. Each side consists of a list of symbolic tokens which represent different properties of the resources held by the participants. The tokens on the left-hand side characterise resources which are intended to be removed from the participants, while the tokens on the right-hand side are intended to be inserted into the participants, *after* those of the left-hand side have been successfully removed.

Formally, the left- and right-hand sides of a rule are separated by the symbol **<>-** (read *becomes*), and are composed of lists of tokens separated by the symbol **@** (read *par*). The abstract syntax for rules is:

Rule	=	*TokenList* **<>-** *TokenList*
TokenList	=	*Token* | *Token @ TokenList*
Token	=	*TokenName (ParameterList)* | *TokenName*
ParameterList	=	*ParameterName* | *ParameterName, ParameterList*

Thus, a rule of the form

```
p(X,Y) @ q(Y,Z) <>- r(X,Z)
```

will try to (*i*) find a resource satisfying the property `p(X,Y)` and a resource satisfying the property `q(Y,Z)` for consistent values of `X,Y,Z`, then (*ii*) extract these two resources atomically, and finally (*iii*) insert a resource satisfying `r(X,Z)`. We assume here that the token names `p,q,r` refer, through some name service, to properties declared in the interface of some CLF objects.

Apart from rules, coordinators also handle signature resources. The signatures indicate how each rule should combine the different Inquiry operations needed to determine resources to be assigned to the tokens in its left-hand side. The naive solution would be to launch in parallel all the inquiries for the tokens of the left-hand side of the rule. However, shared parameters between tokens may involve conflicts, hence the need of some form of sequentialization. Signatures specify implicit sequentialization constraints: a signature for a given token name specifies which of its parameters must be assigned a value for an Inquiry on this parameter to be authorized. The abstract syntax for signatures is:

Signature	=	*TokenName* **(** *ParameterList* **)** **:** *ParamListNil* **->** *ParamListNil*
ParamListNil	=	*ParameterList* | ε

For example, the signature

```
q(X,Y): X -> Y
```

characterizes a property `q` of resources, such that resources satisfying the property `q(X,Y)` for some values of `X,Y`, can be CLF-Inquired through that token only when `X` is assigned, and, in this case, a possible value for `Y` is returned with each answer to the Inquiry. Similarly, if the signature for `p` is

```
p(X,Y): -> X,Y
```

then the sample rule above will systematically inquire `p` first, and, for each action returned by that Inquiry, assigning values to `X` and `Y`, it will inquire `q` with the value assigned to `Y` as first argument.

3 Implementing IPO Process Models in CLF

We show here how to encode process specifications into CLF scripts. The CLF script can be produced in a number of ways: it can be hand-written by a programmer; or it can be automatically translated from an existing process definition language such as PIF [22] or WAPI 1 being defined by the Workflow Coalition [26], or it can be generated by a graphical process editor that allows business users to draw processes with a graphical interface. The CLF coordinator enacts the process encoded in the script by managing the interaction and negotiation among a number of software participants, such as: task list managers; process visualisation and administration tools; persistent process state servers; user directory servers; document management systems; relational database systems; desktop applications; legacy applications; etc. These software participants can be developed specifically to support the CLF protocol, or can be existing off-the-shelf network services encapsulated behind a CLF-compliant wrapper. In both cases, the software participants are accessed through homogeneous symbolic tokens in the CLF scripts that are manipulated by the CLF coordinators.

In the rest of this section we will focus on the design of the participants that manage the process state and the process flow. The most important choice to make when implementing a process enactment engine using the CLF (or any other coordination system) concerns the granularity of the components of the workflow we want the coordination system to manipulate. If the granularity of the components is very fine then it will be possible to fine-tune the workflow in the CLF script, however the CLF script will be long and harder to maintain. We distinguish two kinds of participants:
* those which hold the state of the different processes in the workflow;
* those which support the process state transitions.

According to the IPO paradigm, we assume that the state of each process `P` is characterised by, *at least*, two resources which can be consumed or produced during the process transitions. These resources, written `P-in` and `P-out`, state that, respectively, the Input and Output conditions attached to process `P` have been fulfilled. The participants holding these resources are supposed to act as simple bags: when an Inquiry request is performed, the participant starts to scan the bag till the resource(s) corresponding to the criterion of the Inquiry appear. The Reserve/Confirm/Cancel operations commit or abort the removal of the requested resources from the bag. The bag is asynchronously filled by Insert requests.

Typically, a workflow specification cannot be expressed directly in terms of the specific state resources `P-in` and `P-out`, for each individual process `P`, but rather in terms of classes of these resources, corresponding to classes of processes behaving in the same way. This is naturally accounted for in the CLF by the possibility of querying the same participant, holding the state resources, through different tokens and with different signatures, corresponding to different views on these resources. In each token, the predicate characterises the view and the parameters characterise the individual process state resources available through this view. For example, the state of a process in a system for authoring-translation of large document bases could be accessed through different tokens, designating the same state resource, such as

* `translate-in(Sec)` in a translation-specific transition interested only in the input availability of a translation process; `Sec` is, for example, the number of the section to be translated;

* `translate(State,Sec)` in a translation-specific transition which applies a-priori to any state of a translation process and where the state is visible and, hence, can be filtered out by some other token; `State` is any of the translation specific process states (including input and output availability);

* `job(Proc,State)` in a transition of a large workflow which applies a-priori to any of its processes, not only translation, and where the process and state are visible and, hence, can be filtered out by some other token; `Proc` is a data-structure characterising the exact process being accessed (including its category, such as "translate" and specific parameters, such as a section number).

Notice that all these tokens which encode the same state resources, but viewed differently, can co-exist within the same CLF program. In the sequel, we will introduce rule skeletons which mention directly resources. This gives an abstract view of what each rule does in term of resource manipulations. These rule skeletons should be distinguished from the actual rules which appear in CLF programs, and which mention tokens instead of resources. What an actual rule does in terms of communication between coordinators and participants (according to the CLF protocol) heavily depends on the tokens which are used for each resource, and their signature.

3.1 Flow Control Primitives in CLF

The flow of execution of a process consists of transitions which remove resources corresponding to the output availability of some processes and insert resources corresponding to the input availability of other processes. Such transitions can straightforwardly be encoded by CLF rules. As an example, we show below how to model traditional imperative coordination constructs. As explained above, the rules introduced here are in fact abstract skeletons, which need to be concretised by appropriately choosing the tokens for each resource.
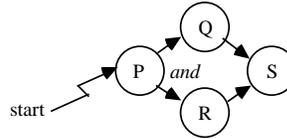
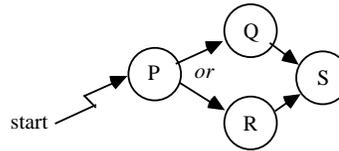**Sequential execution**

```
P-out <>- Q-in
```
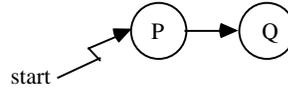
Parallel execution, rendezvous

```
P-out <>- Q-in @ R-in
Q-out @ R-out <>- S-in
```

Branching

```
P-out @ case1 <>- Q-in
P-out @ case2 <>- R-in
Q-out <>- S-in
R-out <>- S-in
```

Here `case1` and `case2` represent resources which are supposed to be mutually exclusive and allow to discriminate between the two branches. Typically, the tokens for `P-out` and `case1/case2` share a common variable (output for the former, input for the latter), and the selection tokens simply perform mutually exclusive tests on this variable (e.g. a test and its negation).

The IPO model also allows a process to be hierarchically decomposed into sub-processes. This again can easily be achieved by CLF rules:

Decomposition

```
P-in <>- Q-in
Q-out <>- R-in
R-out <>- P-out
```
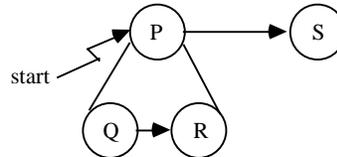
3.2 The Activity Level

3.2.1 Rules for Executing Activities

The execution of activities, the atomic units of process decomposition, may go through various transitions of the state of the activity, which can be implemented as CLF rules taking as input the input state of a transition, possibly combined with access to other attributes of the activity, and producing as output the output state of the transition, possibly with some notification. Thus, the general skeleton of an activity state transition rule is:

```
old-state @ {pre-conditions}  <>- new-state @ {post-conditions}
```

We will start by considering the simplest IPO workflow management system, where activities have only two states corresponding to the Input and Output availability of the activity seen as an atomic process (see Figure 3).
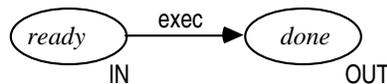
Figure 3: *Simplest state diagram of an activity*

7

Other states can also be defined, depending on the degree of detail in the description of the activity and its transitions, for example: released, delegated, re-routed, skipped, re-done, looped, etc. These elaborations will be discussed later, but the working principle of mapping state transitions into CLF rules is basically the same.

In this very coarse grain approach, a single resource `exec-A` is attached to each activity `A`. It denotes an abstract resource the removal of which corresponds to one complete execution of the activity. The transition attached to an activity is then simply implemented with the following CLF rule skeleton:

```
A-in @ exec-A <>- A-out
```

Of course, the behaviour of an actual rule materialising this skeleton would depend on the token used for the abstract resources `exec-A` and the "real" work performed by the activity `A` must be distributed over the different phases of the CLF protocol for the token corresponding to the resource `exec-A`.

Notice that the work performed in the Inquiry operation for the resource `exec-A` can be scheduled with respect to the availability of the resource `A-in` by sharing arguments between the tokens corresponding to these two resources and setting appropriate signatures for these tokens. For example, in the rule

```
translate-in(Sec) @ exec-translate(Sec) <>- translate-out(Sec)
```

the signatures could be:

```
translate-in(Sec): -> Sec
exec-translate(Sec): Sec ->
```

so that the real work, i.e. the translation, in the Inquiry for `exec-translate(Sec)` will not start before the section number, `Sec`, which is at an input position in this token, is provided as a result to the Inquiry for `translate-in(Sec)` where `Sec` is at an output position. Let's detail the behaviour of the simple rule above. The translation is done locally, upon Inquiry of the execution token, on the current version of the requested section; this amounts to making the assumption that the section will not change during the translation work.

- If the assumption is correct, i.e. the section has not been modified between the time translation started and the time the execution token is reserved, then the reservation prepares to update the section in the translated document, i.e. puts the appropriate locks. Actual update occurs on confirmation.

- If, exceptionally, the section has been modified when the reservation is attempted, two cases may occur: if the modification is small and can be accounted for immediately, it is incorporated in the translation; otherwise, the reservation fails, the translation is reworked and a new answer to the Inquiry of the execution token is produced.

What we described above models simple, deterministic workflow systems, where each activity is executed in a rather rigid way. The CLF, however, allows to represent and execute in a simple way much more diverse and flexible schemes. The diversity of execution models can be represented at two levels:

- Rule level: several rules of the form above may correspond to several static competing alternatives to perform an activity; rules can also involve several activity resources, to model cooperating activities.
- Reply level: an Inquiry request on the token for `exec-A` could yield several replies, corresponding to various dynamically determined alternatives to perform an activity. By default, the selection among the alternatives is made non-deterministically, on a "first available - first selected" basis; however, it is possible to get some form of control through explicit tokens which filter out the replies.

3.2.2 Accounting for Roles and Users

The coarse grain approach described in section 3.2.1 concentrates into one participant all the stages in the execution of the activity. However, traditional workflow models make visible at the level of the workflow various intermediate states of the activities, which the enactment engine can take into account. We propose here a simple extension of the previous coarse grain approach, which can be pushed further, and which makes explicit at the level of the rules the mechanism of role resolution.

Figure 4: *State diagram of an activity with role resolution*

Each activity `A` can now go through an intermediate state, denoted by the resource `A-acq-by-U`, meaning that the activity has been assigned to a user `U` (see Figure 4). The execution of the activity proceeds in two steps; first, a user is assigned to the activity and then the activity is performed by the selected user. This behaviour can be achieved by the following rule skeletons:

```
A-in @ assign-A-to-U <>- A-acq-by-U
A-acq-by-U @ exec-A-by-U <>- A-out
```

This solution implements a rather rigid management scheme. The flexibility of the CLF can be easily exploited to add richer ways to interact with users and to execute activities. For example, users may be given a chance to change their mind about taking an activity, or users may propose different ways to perform the activity. Such elaborations of the basic mechanism are described below.

3.3 Discussion of the CLF Approach

The previous section only provides the skeleton of the CLF rules needed to implement a workflow according to the IPO model. A lot more needs to be added to obtain a complete workflow system. The participants holding the activity execution resources must be implemented and must offer the appropriate tokens and signatures. Document management systems may have to be integrated to the coordination.

One advantage of the CLF approach is that its architecture is highly decentralised. The participants holding the state of the different processes or the activity resources can be distributed appropriately over a network. If reliability is an issue, each of them can use its own strategy to ensure reliability. Even the transport mechanisms used to access these participants can be heterogeneous (currently, the CLF supports both Http and Corba as transport layers). Coordinators can be started and stopped any time during the workflow execution. Starting several overlapping instances of the same coordinator is not a problem, so that some rudimentary form of migration can be achieved easily (asynchronously interrupt a coordinator at one site and start it at another site). Of course, if the coordinator uses private participants to hold an internal state, one would probably want these participants to migrate with the coordinator, but this problem can be treated separately at the level of the concerned participants.

The other main advantage of the CLF approach is that developers can easily change the core process model, i.e. add new activity states, new dependencies between activity states, new activity allocation mechanisms, etc. We show here an example of such extensions aiming at monitoring the activity assignment mechanism.

The basic role resolution rule,

```
A-in @ assign-A-to-U <>- A-acq-by-U
```

lets the rule engine pick one user, without external control, among the potential candidates for the task, returned by the Inquiry operation for resource `assign-A-to-U`. One may want the selection of the candidate to be controlled by a manager. This means that the activity results in fact of the negotiation of two actors, the manager of the activity and the candidate performer. Let's first restrict the work of the manager to the selection of an appropriate performer: the work of a manager `M` selecting a user `U` for an activity `A` is an activity of its own whose execution is represented by an abstract resource `manage-A-by-M-for-U`. The skeleton of the resolution rules becomes:

```
A-in @ assign-manage-A-to-M <>- A-man-by-M
A-man-by-M @ assign-A-to-U @ exec-manage-A-by-M-for-U <>- A-acq-by-U
A-acq-by-U @ exec-A-by-U <>- A-out
```

In fact, these rules model a deep coupling between the base activity `A` and the management activity `manage-A`:
- First, the input availability of `A` triggers the role resolution for the activity `manage-A` (selection of a manager) leading to a new intermediate state `A-man-by-M` where `M` is the selected manager.
- Then, the role resolution of `A` is triggered, but controlled by manager `M` executing the activity `manage-A`, leading to the intermediate state `A-acq-by-U` where `U` is the selected user.
- Finally, user `U` executes the activity `A` as before.

If we assume that the resource `exec-manage-A-by-M-for-U` is accessed by a token *exec-manage-a(A,M,U)* with signature `A,M,U->`, then the different stages of the CLF protocol for that token could be interpreted as follows:

Inquiry: User U is added to a pool of potential users for activity A. The pool is available to the manager M for him to pick from. The Inquiry is suspended till user U is picked by the manager (if ever), in which case a positive reply is returned.

Reserve: The manager is informed that his choice is taken into account and is asked to prepare to commit to it. Failure occurs if the manager refuses.

Confirm/**Cancel**: The final choice of user is notified to the manager (confirm) or the manager is informed that his choice is not available anymore, maybe because the user refused to commit (cancel).

4 The Software Defect Report Process

This section illustrates how CLF can be used to model and enact a typical software engineering process: software defect reporting. We first describe the process and then we show how it can be supported using the CLF.
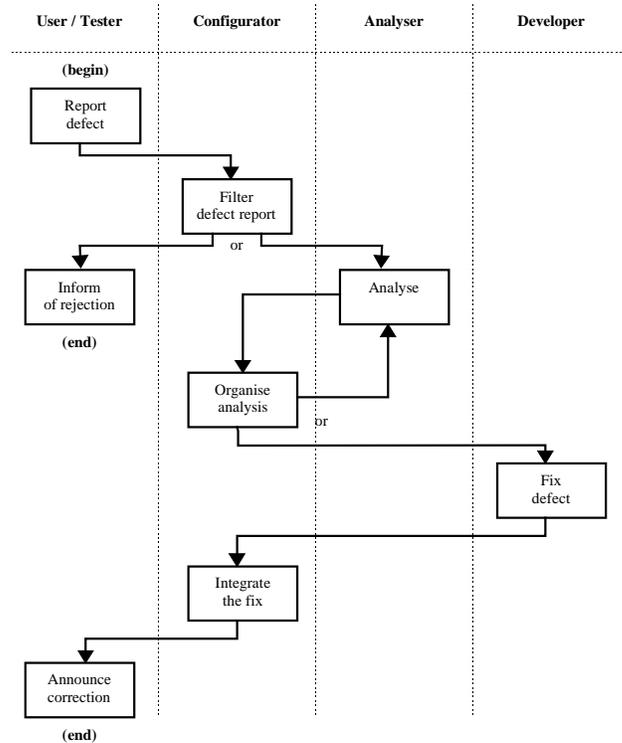

Figure 5: *The software defect report process map*

4.1 Description

The sample process we are interested in concerns the testing and maintenance phases of large systems in software engineering. Informally, the process starts when a user or a tester discovers a defect in the system. He describes the defect through an electronic form. Then relevant experts carry out the analysis. Once they have identified a fix to the problem, a developer makes the necessary modifications to the code. The availability of the fixed version of the system is then announced to the author of the defect report.

So, the different roles involved in this process are:
1. the testers or users of the code: they report the defect.
2. the configurator: s/he manages the software sources. S/he registers defects, integrates sources provided by the developers, builds (compiles) the software, manages software versions and delivers them to users/testers.
3. the analysers: they analyse defects, propose corrections and identify impacts of correction.
4. the developers: they modify software sources and deliver them to the configurator.

Figure 5 gives a more precise view of the process using a sample formalism: boxes represent tasks, and columns give the associated roles.

Let's detail each of the activities involved in this process:

Report defect: the tester describes the defect through an electronic form, including when, how, who, what, etc. S/he also evaluates the defect severity, which can range from 'catastrophic', to 'essential' or 'cosmetic'. At the next stage the configurator may modify this evaluation to ensure a better uniformity of severity among defects.

Filter defect report: The configurator receives the defect report, and either rejects it for incompleteness or inappropriateness, or accepts it. Complementary information can be requested and added. The configurator then chooses one of the analysers to initiate the analysis loop.

Inform of rejection: The author is informed of the rejection of his report.

Analyse: The analyser receives the report and carries out the defect analyse. S/he has to propose corrections and to estimate their costs or their possible impacts. Impacts could be on the documentation, on other parts of the software. S/he can also provide only a partial analysis, which must be completed by some other analyser.

Organise analysis: The configurator receives the report and its attached set of analysis. S/he may choose to send again the report for further analysis or consider that the analysis is completed. In the latter case, s/he selects a developer to fix the defect.

Fix defect: The developer receives the defect report and the associated analysis, applies the proposed correction. S/he then delivers the modified modules to the configurator.

Integrate the fix: The configurator receives a set of modified modules together with the defect report and the analysis. S/he has to integrate these modules into the software.

Announce correction: This inform the report writer that the defect should no longer exist in the new version of the system.

4.2 Modelling in the CLF

To simplify, we assume that a single participant manages the activity state resources, such as `A-in`, `A-out`, `A-acq`, and that a document management system stores the defect reports. The nature of the activity (`A`, i.e. *analyse*, *filter*, *fix* etc.) and the reference to the defect report identifies fully the resource. Thus, for example, the resources concerning the *analyse* activity could be accessed through the following tokens:

```
analyse-in(Report):          -> Report
analyse-out(Report):         -> Report
analyse-acq(User, Report):   -> User, Report
```

As an exit status may be needed, for instance to determine which activity to perform next, we introduce a variant of the activity output token which retrieves a report reference given an activity nature and an exit status, for example:

```
filter-out'(Report, Status):    Status  -> Report
```

The workflow graph of Figure 5 can then straightforwardly be translated into CLF rules, as explained in section 3.1:

```
report-out(R) <>- filter-in(R)
filter-out'(R, "rejected") <>- inform-in(R)
filter-out'(R, "accepted") <>- analyse-in(R)
```

Another participant is in charge of activity enactment and it supports in fact the state transitions, as explained in section 3.2.2. It manages the `assign`, `exec` resources the signatures of which are:

```
assign(Task, Role, User):           Task, Role -> User
exec(User, Task, Input):            User, Task, Input ->
exec'(User, Task, Input, Status):   User, Task, Input -> Status
```

Notice that the `exec` token has two variants depending on whether we are interested in the exit status of the activity, or not.

As only one person plays the role of configurator, a `config` resource is provided to retrieve the configurator name. This resource is provided infinitely and can therefore be consumed many time. Its signature is:

```
config(User): -> User
```

Thus, the beginning of the process can be modelled by:

```
report-out(R) <>- filter-in(R)
filter-in(R) @ config(C) @ exec'(C,"filter",R,Status) <>-
     filter-out'(R,Status)
filter-out'(R, "rejected") <>- inform-in(R)
filter-out'(R, "accepted") <>- analyse-in(R)
```

The assignment rule is skipped here, as only the configurator may perform the *filter* activity.

Also, to allow him to monitor the assignment of the *analysis* activity we introduce a special activity called *select*, which consists in accepting or rejecting the couple (analyser, report). This activity takes as input not just a report but a "proposal" to perform a given task on a given report by a given user. Proposals are created "on the fly" by accessing a virtual resource through a token of the following form:

```
build-proposal(Task,User,Report,Proposal): Task,User,Report ->Proposal
```

Assignment monitoring (see section 3.2.1) is done by the following rule:

```
analyse-in(R) @ config(C) @ assign("analyse", "analyser", U)
@ build-proposal("analyse",U, R, I) @ exec(C, "select", I)
      <>- analyse-acq(U, R)
```

The other parts of the process graph can be modelled in a similar way, for example:

```
analyse-acq(U, R) @ exec(U, "analyse", R) <>- analyse-out(R)
analyse-out(R) <>- organise-in(R)
organise-in(R) @ config(C) @ exec'(C, "organise", R, S) <>-
      organise-out'(R, S)
organise-out'(R, "need more analysis") <>- analyse-in(R)
organise-out'(R, "enough") <>- fix-in(R)
```

The *fix* activity is interesting, because we may imagine that the last analyser would like to monitor the way it is to be performed. We therefore introduce a variant of the assignment token which returns a plan to perform a task, and a corresponding variant of the proposal building token which takes into account that plan:

```
build-proposal(Task,User,Report,Plan,Prop):Task,User,Report,Plan->Prop
assign'(Task,Role,User,Report,Plan): Task,Role,Report -> User,Plan
```

The corresponding rules are:

```
fix-in(R) @ last-analyser(R,A) @ assign'("fix","developer",U,R,Plan)
@ build-proposal("fix",U,R,Plan,I) @ exec(A,"select",I)
      <>- fix-acq(U,R)
fix-acq(U, R) @ exec(U, "fix", R) <>- fix-out(R)
```

At this point we have one participant dealing with states, another dealing with state transitions (activity assignment and performance) and one coordinator enacting the above set of rules. If we imagine that the testers are geographically distributed, we can manage activities devoted to testers in a different way by providing each of them with one participant and one coordinator. The private participant of each user provides the following tokens:

```
name(User):              -> User
assign(Task):            Task ->
exec(Task,Report):       Task -> Report
```

The `name` resource is provided infinitely and holds the name of the user to whom the participant is devoted. The `assign` and `exec` tokens are specialised versions of the corresponding general tokens to the case of the user at hand.

Each personal coordinator enacts the following set of rules:

```
name(U) @ in(T, R) @ assign(T) <>- acq(T, U, R)
name(U) @ acq(T, U, R) @ exec (T, R) <>- out(T, R)
name(U) @ exec("report", R) <>- report-out(R)
```

This assumes that the state participant also provides the following interface:

```
in(Task,Report):         -> Task, Report
out(Task,Report):        -> Task, Report
acq(Task,User,Report):  User -> Task, Report
```

This is yet another illustration of the possibility of accessing the same resource through different tokens corresponding to different properties of interest.

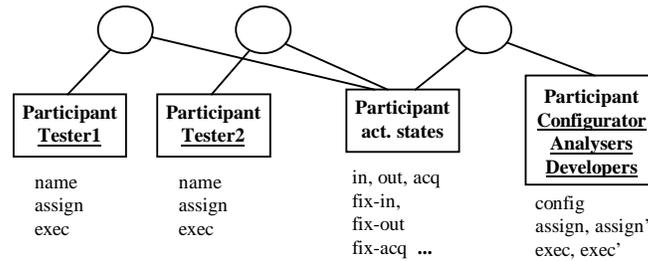Finally the architecture is as follow:

Figure 6: *An architecture with distributed testers*

4.3 Benefits of the CLF approach in the software defect reporting process

The previous section shows how to model and enact the 'Software Defect Report' process; here we highlight the benefits of using CLF, that can be summarized in two main points: i) modular distributed architecture, and ii) flexible language support.

The architecture of a process enactment system implemented with CLF is highly decentralised: the various process servers can be freely distributed, migrated and replicated over the network, according to the needs. For instance, several task list managers may coexist, either to be closer to the user location (as explained in section 3.2.2), or to obtain redundancy for performance or fault-tolerance reasons. Figure 12 shows two possible redundant configurations: a) shows a configuration with one task list manager and two redundant coordinators; b) shows two redundant coordinators each controlling a redundant task list manager. In both cases, the rules in the two redundant coordinators compete to consume the resource denoting any activity state, but only one will succeed thanks to the transaction phase.
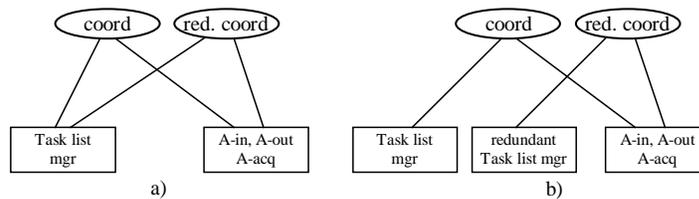

Figure 7: Two redundant configurations

Also the server managing the activity states can be distributed so that the state of each activity is stored on a different machine. For instance, in our process the testers are in charge of the *report*, *inform* and *announce* activities. Therefore a server can be dedicated to the management of these three activities and this server can be geographically located close to the testers; if the users move to another location, the activity server can be migrated without any modification in the CLF rules. We could also decide to have one activity server for each supported process definition, thus gaining a good independence among processes of different kind.

The second main advantage concerns the flexibility in changing both the core process model and the application-specific process template. The core process model is expressed using CLF rules, therefore it is possible to change them safely and cost effectively. For instance, as shown is section 3.3, it is easy to change the mechanism to propose an activity to the user. In a standard process enactment engine the task allocation mechanism is usually a built-in function, therefore the modification requires strong programming skills and recompiling the source code.

Concerning the application-specific process templates, CLF provides a great deal of flexibility. The user can define a process either with a graphical user interface that automatically generates CLF rules, or by writing directly CLF rules. Then a CLF coordinator interprets the rules to enact the process. We think that CLF rules, while being directly executable, are still understandable by a developer and provide a good level of abstraction for a process designer. A key feature is that CLF rules can be added or removed dynamically at run-time, giving a good potential to change at run-time the process definition. Furthermore, the CLF rules associated to a process definition can be dynamically partitioned among several coordinators. For instance, in the case of two activities

managed in the same geographic location, we can execute the CLF rule expressing their synchronisation on the same location, independently of the other rules constituting the process definition. This allows us to manage distributed processes in a distributed way. Finally, we can exploit the reflection feature of CLF--i.e., a CLF rule can manipulate another CLF rule as a resource--in order to express using the CLF how to dynamically distribute, migrate and replicate CLF rules among several coordinators.

5 Conclusion

Coordination has nowadays become the topic of a large corpus of research, trying to identify high level primitives, independent of any specific architecture, needed to build large scale application involving heterogeneous subsystems and legacy applications. Workflow is an obvious candidate domain of application for coordination primitives, where the coordinated entities are business resources and processes which, in large organisation, may be widely distributed and involve many heterogeneous components. In this paper, we have illustrated the link between workflow and coordination with a concrete instance of coordination system (the CLF). We have shown how workflow requirements are handled in a straightforward way in the CLF, offering a lot of flexibility at different levels (distribution, exception handling). We have concretised our discussion on a typical workflow in software engineering, namely the software defect report process.

6 References

[1] Andreoli, J-M., Freeman, S. and Pareschi, R. The Coordination Language Facility. To appear in the Journal of Theory and Practice of Object Systems.

[2] Apple Computer Inc. *AppleScript manual*, 1993.

[3] N.S. Barghouti and G.E. Kaiser. Scaling up Rule Based Software Development Environments. Proc. of the *European Software Engineering Conference*, Milan, Italy, 1991. Springer, LNCS 550 pp 380-395.

[4] N. Belkhatir, J. Estublier and W. Melo. Adele/Tempo: An Environment to Support Process Modelling and Enaction. In A. Finkelstein, J. Kramer and B. Nuseibeh, editors, *Software Process Modelling and Technology*, John Wiley & Sons Inc., New-York, N.Y., U.S.A., 1994.

[5] I. Ben-Shaul and G. Kaiser. A Paradigm for Decentralized Process Modeling and its Realization in the Oz. Proc. of the *16th International Conference on Software Engineering*, Sorrento, Italy, 1994.

[6] S. Carlsen. Organizational Perspectives of Workflow Technology, University of Trondheim, Norway, 1995.

[7] G. Chroust. Interpretable Process Models for Software Development and Workflow. In EWSPT '95, Noordwijkerhout, The Netherlands, April 1995.

[8] P. Ciancarini. Coordinating Rule-Based Software Processes with ESP. *ACM Transactions on Software Engineering and Methodology*, 2(3) 203-227, 1993.

[9] P. Ciancarini, O. Nierstrasz, A. Yonezawa. Object-Based Models and Languages for Concurrent Systems. Proc. of ECOOP '94 Workshop on Modelas and Languages for Coordination of Parallelism and Distribution, Bologna, Italy, July 1994, Springer.

[10] P. Ciancarini and C. Hankin (editors). Coordination Languages and Models. Proceedings of the First Int. Conf. COORDINATION '96, Cesena, Italy, April 1996, Springer.

[11] U. Dayal, M. Hsu and R. Ladin. Organizing Long-running Activities with Triggers and Transactions. In M. Stonebraker, editor, *Readings in Database Systems*, 324-334, Morgan Kaufmann Publishers, San Francisco, 1995.

[12] A. Elmagarmid, editor. *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, San-Mateo, Ca, U.S.A., 1993.

[13] C. Fernström. ProcessWeaver: Adding Process Support to Unix. In Proc. of the Seconf Int. Conf. on Software Process (ICSP-2), pp. 12-26, Berlin, Germany, Feb. 1993. IEEE-CS Press.

[14] Garcia Molina, H. and Salem, K. Sagas. Proc. of ACM SIGMOD conf., 1987.

[15] M.R. Greenwood. Coordination Theory and Software Process Technology. In EWSPT '95, Noordwijkerhout, The Netherlands, April 1995. Springer

[16] IBM Corporation. MQ Series, 1993.

[17] G. Junkermann, B. Peuschel, W. Schäfer and S. Wolf. Merlin: Supporting Cooperation in Software Development Through a Knowledge-Based Environment. In A. Finkelstein, J. Kramer and B. Nuseibeh, editors, *Software Process Modelling and Technology*, John Wiley & Sons Inc., N.Y. U.S.A., 94.

[18] R.E. Kraut and L.A. Streeter. Coordination in Software Development. Communications of the ACM, March 1995, Vol. 38 No. 3, pp. 69-81.

[19] C. Montangero and V. Ambriola. Oikos: Constructing Process Centered SDEs. In A. Finkelstein, J. Kramer and B. Nuseibeh, editors, *Software Process Modelling and Technology*, John Wiley & Sons Inc., N.Y., U.S.A., 1994.

[20] Object Managenent Group, *The Common Object Request Broker: Architecture and Specification*, 1991.

[21] Object Management Group, Object Transaction Service, document 94.8.4.

[22] PIF working group, The PIF process interchange format and framework, 1995, http://soa.cba.hawaii.edu/pif/

[23] K. D. Swenson. Visual Support for Reengineering Work Processes. In Proc. of Conf. on Organizational Computing Systems (COOCS '93), Milpital, CA, pp. 130-141.

[24] Sun Microsystems Inc., ToolkTalk Programmer's Guide, 1992.

[25] H. Wächter and A. Reuter. The ConTract Model. In A. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, Morgan Kaufmann Publishers, San-Mateo, Ca, U.S.A., 1993.

[26] Workflow Management Coalition, Glossary, http://www.aiai.ed.ac.uk/WfMC/

[27] D. Yellin and E. Strom. Interfaces, Protocols and the Semi-automatic Construction of Software Adaptors. In *Proc. of OOPSLA'94*, Portland, Or., U.S.A., 1994.