Ariadne and HOPLa: flexible coordination of collaborative processes

Gert Florijn, Timo Besamusca, Danny Greefhorst

Utrecht University
Department of Computer Science
P.o. Box 80.089
3508 TB Utrecht
The Netherlands
Telephone: +31-30-2531807
Fax: +31-30-2513791
E-mail: florijn@cs.ruu.nl

Abstract

The research into the Ariadne system - and its coordination language HOPLa - aims to provide generic support for hybrid collaborative processes. These are complex information processing tasks involving coordinated contributions from multiple people and tools. Ariadne should applicable for a broad spectrum of these processes and actively support people in working in these processes and in defining and managing. A key design issue is flexibility. It should be easy for users to model and perform new (kinds of) processes even if this happens during the work on the process itself.

Processes in Ariadne combine a shared workspace with the ability to define tasks and to coordinate their execution. The workspace uses a tree-like data model and can hold arbitrarily structured data. It is self-descriptive which means that it not only holds actual data but also the constraints (i.e. type definitions) that govern its structure. The definition of the way of working in a process can be blended into the workspace. Nodes in the workspace can be marked as tasks to be performed. Aditional coordination operators and constraints e.g. on time or performer of a task can be attached. Ariadne keeps track of the execution state of each process and uses this for support. When tasks are to be performed, actors are notified and results of the work are stored again in the workspace. The execution state of the process is stored in the workspace too, so that processes are self-contained.

The need for flexibility is addressed in two ways. First, process workspaces can be adapted arbitrarily within the boundaries of the type-definitions stored within them. This can make it easy, for example, to add extra tasks or to annotate data. Second, both process descriptions and running processes can be represented in textual form, i.e. as an HOPLa program. This means that a running process can be stopped, aribitrarily modified (through editing) and continued whenever exceptions give rise to this, and without the loss of any data that had already been provided.

This paper gives an overview of Ariadne and HOPLa. After some background on collaborative process we discuss the HOPLa mechanisms for the definition of workspaces and process models, and the way in which Ariadne handles processes. We discuss and evaluate several examples and describe some current research.

Ariadne and HOPLa: flexible coordination of collaborative processes

Gert Florijn, Timo Besamusca, Danny Greefhorst

Utrecht University
Department of Computer Science
E-mail: florijn@cs.ruu.nl

1. Introduction

Coordination languages provide a new perspective on constructing computer programs. Instead of describing individual computations, a coordination language is used to glue together existing computations into new systems [Carriero92]. Given this scope, a coordination language does not have to be computationally complete, e.g. it does not have to provide things like artithmetic operations. Instead it should be "coordinationally complete", providing the means to create new computations and to organize and control their execution in dimensions like time and space.

Coordination and coordination languages have become popular in several fields. Many developments over the past few years can be viewed as studies of some aspect of coordination. For example, within the field of object-oriented systems, contracts [Helm90] have been put forward to describe the interaction among cooperating objects. Similarly, protocols are used to describe the conditions under which certain actions on an object can be performed. Coordination of computations can also be viewed as an application of computational reflection: we not only consider the computations themselves, but also the way in which they are managed and organized. Several metaphores and models for coordination-based computing have been proposed, leading to languages and tools that allow us to glue together processes or commands into larger systems, e.g. shells and extensions like Kibitz [Libes93], Linda [Carriero92], Gamma [Banâtre90], Manifold [Arbab93] and others [e.g. Schill91].

In this paper we focus on another field where coordination plays an important role, i.e. that of supporting group work. In particular, we focus on how coordination can be applied to support hybrid collaborative processes. These are complex information processing activities that involve coordinated contributions by multiple users and computer agents. Typical examples of such processes are

- organizing and performing a poll among a group on a particular subject
- finding a suitable time-slot and location for a group meeting
- performing administrative procedures like travel expense reimbursement, granting loans, etc.
- keeping track of bug-reports in software and of their resolution
- carrying out a review of a draft report

All of these activities involve organized contributions by multiple users (and tools) and all of them would become easier if they were supported by adequate tools.

Instead of developing dedicated tools for specific activities, the main objective of our research in the Ariadne project is to create one generic environment that assists people in modelling, performing and managing a broad variety of such activities. This requires a more general coordination model by which

- the work involved in a collaborative process can be decomposed into smaller steps
- the steps can be assigned to and performed by various people or tools
- the execution of steps can be coordinated (e.g. ordered in time)
- the results of performing steps (e.g. data produced) are kept together for reference and use by other steps

The goal of our work is to define such a model and implement it in a system that can provide active support for users working in and with such processes. For example, users who can perform a particular step in a process should be notified of this and possibly reminded later on. Likewise, users who are responsible for a process should be supported in keeping track of it, be informed of problems with it, and have the ability to correct these.

Since we would like (end-) users to define processes, the system should be as simple as possible while expressive enough to handle the variety of activities. Flexibility is also a key issue. The model (and system) should not only work for routine processes for which the steps to be performed and the data that is shared are well-known, but also for ad-hoc processes in which the steps and the results they should produce are defined on-the-fly.

1.1. About this paper

In this paper we describe how the Ariadne system and its process definition language HOPLa, address the issues involved in supporting hybrid collaborative processes. The paper is organized as follows. Section 2 provides some background on collaborative processes and gives a brief overview of existing systems. Sections 3, 4 and 5 discuss HOPLa and Ariadne in some detail. In section 3 we discuss the data model used to store the shared data in Ariadne processes. Section 4 discusses the mechanisms to define tasks and control their execution, while section 5 describes the way in which processes are executed by Ariadne. in In section 6 we discuss some examples and identify some shortcomings of the current system while section 7 provides some conclusions and discusses current research.

2. Background

The goal of the Ariadne project is to provide generic computer support for hybrid collaborative processes. These are information processing activities involving coordinated contributions by multiple people or computer systems. In this section we will discuss the characteristics of these processes and consider what is needed to support them. Also we will look at existing systems and evaluate them briefly.

2.1. Examples of collaborative proceses

To set the scene for the rest of this paper, we first give some examples of the kind of collaborative processes we are talking about. First of all, there are many processes which are variations of the basic notion of an (electronic) conversation. Consider, for example, a negotiation-cycle in which two parties try to reach consensus on a certain topic, e.g. a supplier and a customer trying to reach agreement about the price of a product [e.g. Martial90]. The supplier submits a first proposal, to which the customer – after some consideration and within a certain time-interval - gives a reaction. The reaction can be the acceptance of the proposal - so that agreement is reached - but it can also be a counter-offer or request for changes, or a cancellation of the negotiation process. Given a counter-offer, the supplier can decide to accept this, to submit a new proposal or to cancel the negotiation. Another example of such conversation-oriented processes is the Speech-Act model of conversations used in the Coordinator [Flores88]. Here users interact via typed messages and the system imposes structure on the conversation by limiting the types of messages that can be sent in reaction to an earlier message of a given type.

Many collaborative activities focus on the coordinated development of some electronic product. Consider the Issue-Based Information System (IBIS) approach to structure group discussions [Conklin88]. In this approach, a discussion is a hypertext web of nodes that either are Issues (discussion topics), Positions (opinions about the issues) and Arguments (that support or weaken positions). In addition to the nodes, the IBIS-model also restricts the links that can occur in the web. Version management is another example of a product-centered process. It deals with the historical evolution of a product, such as a computer program. Starting out with an initial version, changes made to the product result in new versions that are successors to the original. To avoid undesired concurrent updates, developers may be required to indicate that they are going to change a particular version of the product, e.g. by performing a checkout operation which locks the version until a successor is added.

A well-known class of office processes are routine administrative processes that are part of the administrative organization [Gommans93]. As an example consider the handling of invoices in a business. When a service has been provided or a product delivered, an invoice is sent to the customer with the request to pay before a certain date. If the payment has been received within the given time-frame, the process is terminated successfully, otherwise a reminder has to be sent (typically by some information system) to the customer. If, after another period, payment is still not received, a second reminder may be sent until eventually a bailiff is called in. Other examples of such routine processes are reimbursement of travel expenses, order-handling or processes that decide on requests for loans. Routine processes also occur in software development, e.g. a process that describes how modifications to an existing system must be developed and reviewed [Heimbigner91].

2.2. Basic ingredients

While these processes appear to be very different, there are some fundamental similarities (see also [Ellis91]). First of all, all processes are collaborative: they are aimed at a particular goal and can involve contributions by multiple actors, i.e. people or computer systems (hence the term hybrid). Secondly, they all involve data that is shared among the participants. For example, the two parties involved in a negotiation need to have an up-to-date picture of the latest offers and reactions, while in an IBIS discussion all people involved need access to the same discussion database. In an order-handling process, people working on the various steps of accepting and planning the order share the information about the order, the client involved, etc. In the remainder of this paper we will use the term *workspace* to refer to the collection of data used and produced in a collaborative process.

The final similarity is that all processes are organized in some way by imposing constraints on the workspace and/or on the way of working. Constraining the workspace means that the shared data in the process has to follow some predefined structure (defined in a so-called *workspace model*), while constraining the way of working means that the work is broken down into tasks and that the execution of these tasks is coordinated. This includes defining when a task should be performed, e.g. by planning it for a particular moment in time or by defining dependencies on (specific results of) other steps. Similarly, constraints can be defined to control execution properties of tasks, such as who can/should perform it, which specific resources should/can be used, when the work should be finished (a deadline), or what the result should be. In the remainder we will use the term *process model* to refer to the decomposition in tasks and the constraints imposed on them for a particular process.

2.3. Variations

These basic similarities also allow us to classify the differences between actual collaborative processes. Here we consider three points, i.e. the mechanisms used in organizing them, the level of detail of the models, and the moment at which the models are established (see also [Kaplan92]).

As far as the use of mechanisms is concerned, some processes just impose structure on the data (e.g. IBIS) while stating little about the process. Others structure just the process (e.g. the negotiation-cycle) and still others structure both (e.g. administrative procedures). The detail of the models spans a wide spectrum, ranging from unstructured, via semi-structured to highly structured. If we consider the data in a process, for example, there can be no or hardly any constraints imposed (e.g. the versioning process does not state anything about the objects that are versioned), while on the other hand there can be very detailed constraints, as will typically be the case for the forms and databases used in an administrative process.

Finally, the moment at which the models are established can vary. For some cases, the models may be known and defined in advance, and not changed during the execution of a process. This is typically the case for routine administrative processes. In other situations, the models may be less obvious at the outset of prcess. This can be caused by a lack of understanding of what is to be done or by a lack in experience with the particular situation (ad-hoc processes), but also because the way in which the work is done depends on the context where it is performed (so-called situation action [Suchman87]). For example, one person's strategy to write a report may be funamentally different from someone else's strategy.

In these situations, the workspace and/or process models may be created, refined and adapted during the work on a process itself. It can mean that constraints are added (e.g. more tasks are defined and scheduled, more structure is imposed on the workspace), but it can also mean that existing constraints are relaxed. Note that a similar situation can occur in routine processes when some of the circumstances during the execution differ from the expectations, for example when an important client demands a different payment period or a non-standard booking of an order. Handling such exceptions [Karbe90] can imply that a slightly modified or completely revised process model has to be adopted.

2.4. Computer support

There is a large body of work on support of group work. In fact, the whole field of CSCW and groupware [e.g. Ellis91] is concerned with this. Both there and in the field of office systems, coordination of (human) activities has been studied from a theoretical perspective [e.g. Malone90] but also from a practical perspective. There are numerous models and systems aimed at supporting specific problems, like IBIS implementations [Conklin88], or tools for negotations and conversations [Flores88].

Based on the similarities identified above, however, the notion of generic support for collaborative processes is an interesting, albeit challenging, possibility. Obviously such a system should provide basic mechansisms like a shared dataspace and decomposition of processes into sub-steps. In addition, it should be possible to parametrize the use of these mechanisms by the (dynamic) definition and enforcement of constraints, e.g. on the data or on the task-decomposition. Most importantly, the system should actively support users, both when contributing to processes but also in defining and managing processes. This can also imply specific presentations and interfaces.

The idea of generic groupware has been proposed elsewhere [Ellis94]. In fact, over the years several systems have been developed which can be parametrized to some extent (of course, this paper does not provide space for a complete overview). OVAL [Malone92] and PAGES [Hämmäinen91] for example are messaging systems that allow the exchange of user-defined, semi-structured objects, while Lotus Notes provides a shared document database with the ability to define various types of semi-structured objects and related applications. Workflow management systems [Karbe90, Kreifelts91, Gulla94] can be parametrized by a process model. Similar systems have been put forward in the field of software engineering [E.g. Kaiser93]. Finally, systems like the

Conversation Builder [e.g. Kaplan92] can impose structure on both data and way of working, and thereby support different kinds of collaborative processes.

The problem with most of the existing systems is they are often focused on just one ingredient (i.e. data or process) and furthermore, they are frequently oriented biased towards routine situations. Defining new workspace or process models is not a trivial matter and adaptation of these model within the context of a running process is mostly impossible. This means that there are no adequate means to deal with exceptional situations and with situated or ad-hoc actions.

2.5. The Ariadne project

The aim of our research in the Ariadne project is to create an enviroment – called Ariadne – which provides generic support for hybrid collaborative processes. As follows from the discussion above, an important research goal is to provide support for a broad class of processes while offering the necessary flexibility to enable groups to handle exceptions and to organize ad-hoc work. In particular we want (end-) users to be able to define models for a particular collaborative activity and to adapt these models within the context of a running process. Clearly this means that both the language used to describe models as well as the overall system should be simple to understand and use. In the following sections we will discuss how Ariadne aims to do this.

3. Shared workspaces as flexible records

The shared workspace is the core of an Ariadne processes. It holds the data that is shared by the participants in the process. As we saw earlier, a fundamental requirement for the workspace is that it can hold both highly structured data (e.g. forms consisting of typed fields) but also unstructured or semi-structured objects (e.g. an E-mail message). Clearly a very flexible data model is needed. In Ariadne we use ψ–terms or flexible records to provide this flexibility. HOPLa - the language used to define Atiadne processes [Besamusca95] - is based on the idea of flexible records, but extends it along several directions. In this section we give an overview both of the ideas behind flexibile records and the way they are used in HOPLa.

3.1. Flexible records

Ψ–terms or *flexible records* [Ait-Kaci86] are a generalization of first-order terms as found in Prolog. They are record-like structures defined by the following rule:

$$d ::= X{:}s(l_1 \Rightarrow d, \ldots, l_n \Rightarrow d) \quad \text{with } n \geq 0$$

Here *s* is an element of a set *S* of sorts or typesymbols; it denotes the rootsymbol of this term. Each l_i is an denotes a feature, while *X* is an address. As a simple example of a ψ–term, consider the following `person` "record" (we have omitted the parentheses for terms without features):

```
P: person(name -> N: id(first -> F: "Gert";
                        last  -> L: string);
          addr -> A: address)
```

This defines a record with two features, one labelled `name` and one labelled `addr`. The `name` field refers to a term of type `id`, which again has two fields, `first` and `last`, while the `addr` feature denotes a value of type `address`. Typesymbols denote sets of values, such as `person, id` or `string`, or individual values, such as `"Gert"`. Individual values thus are represented as singleton-set types.

Variables denote addresses in the term. By using variable references, sub-structures in a term can be shared. Consider a slightly different definition of a person record:

```
P: person(name -> N: id;
          addr -> A: address;
          spouse -> S: person(spouse -> SS: P))
```

This defines a cyclic data structure (through the use of P) where the spouse of the spouse of a person refers to the original person.

3.2. The subtype relationship

Typesymbols are partially ordered to reflect subtyping or set inclusion, with a type `T` (or `any`) as top of the order and type ⊥ (or `none`) at the bottom. Typically the subtype relationship is (pre-) defined for individual values such as `"Gert"` and their corresponding natural type, i.e. `string`.

The subtype ordering on typesymbols also implies an ordering on terms, i.e. one term can be a subtype of another. A term *x* is a subtype of a term *y* (denoted as $x < y$) if:

- the rootsymbol of *x* is a subtype of the rootsymbol of *y*
- *x* has at least all the features defined in *y*
- the terms in *x* denoted by the features present in *y* are subtypes of the corresponding terms in *y*

While maintaining the subtype relationship, a term can be modified in two ways. First, we can add new features, e.g. we can extend a person record with arbitrary features like "`income`", "`age`" or "`employer`" without breaking the subtype relationship with the person record defined above. Second, an existing typesymbol can be replaced by a subtype. For example, we can replace an occurrence of `string` with `"a string"`.

3.3. HOPLa terms

Flexible records and the subtyping rule provide the basis for defining and manipulating the workspace of an Ariadne process. Initially, the workspace is defined as a term. During the execution of a process, the workspace can be adapted in accordance with the subtypng rules. This means that "fields" can be filled in or refined, but annotations not planned in advance are also possible by adding extra features.

Processes and their workspaces in Ariadne are defined using HOPLa. A HOPLa definition must provide all the information needed by the run-time system to initialize a workspace when a process is created and to maintain the workspace during the lifetime of a process. Obviously, this requires the definition of a term for the (initial) structure of the workspace, but we also need to know the typesymbols that are or can be used in the term, and the subtype relationships that hold among them.

To allow for all this, a HOPLa definition consists of two ingredients: a definition of a term (the initial structure) and a set of type definitions or *termtypes*. A termtype defines a set of term values by listing a typesymbol plus the set of features that terms of this type should have. Termtypes are defined as subtypes of other termtypes. Consider the following examples (note that term addresses are not required in HOPLa definitions):

```
Person(name -> string; age -> integer);
Employee<Person(manager -> Person);
```

Here we define two new termtypes (`Person` and `Employee`) where Employee is defined as a subtype of Person. Person has no explicit supertypes; it is an implicit subtype of the (predefined) termtype Any.

The HOPLa definition of a subtype can be seen as an explicit specification of the general subtype relation among terms. The <-operator indicates that values (or instances) of the subtype must have all features defined in the supertype combined with all locally defined features. If there is overlap in the list of "inherited" features, the (terms denoted by) locally defined features must be subtypes of (terms denoted by) the inherited features. Instances of the subtype in that case will have the locally defined values. In the example, an `Employee` term will have three features, i.e. `name`, `age` and `manager`. For practical purposes, some types and subtype relations are predefined. In particular, HOPLa offers predefined types like `integer` and `string`, recognizes values of these types and knows about their subtype relations.

The initial structure of the workspace is defined by a number of term definitions. These are just features that denote terms that are instances of termtypes defined earlier. When a workspace is created, the terms will be instantiated with all the features defined in the corresponding termtypes. Following that, the terms can be modified, where the subtyping constraints defined for flexible records apply. So, we can add features to a term or replace a term by a subtype. As an example, consider:

```
john -> Employee(name -> "john");
```

Here the instantiated term referred to by the feature `john` will have the three features defined above. In addition, its `name` feature is set to the string value `"john"` which is a subtype of the predefined type `string`. So, the actual term referred to by feature `john` after instantiation is:

```
john -> Employee( name -> "john"; age -> integer; manager -> Person);
```

It should be noted that HOPLa allows us to mix the definition of termtypes and of terms, as in:

```
peter -> Manager<Employee(car -> string)
```

This defines a new termtype `Manager` whose values must have a `car` feature, while at the same time defining a term of this type denoted by the feature `peter`.

3.4. Collection terms

HOPLa extends the basic mechanisms of flexible records with constructors for collections of terms. The idea is that we can define terms whose (typically unnamed) features refer to values of one particular termtype. In fact, we can distinguish various kinds of collections, like sets, bags or indexables. Some of these collection termtypes are predefined in HOPLa. The basic definition of a Set termtype for example looks like this:

```
Set(value -> set; type  -> Any)
```

where `set` is the builtin primitive type that holds the actual elements. Note that we also use the construct `{...}` to denote an empty set value. A set value can be modified by adding (or removing) an element which must be of the given type.

A collection term is created by parametrizing one of these pre-defined termtypes with the type of the elements, i.e. by redefining the `type` feature. The actual collection is accessible through the `value` feature. As an example, consider the following definition:

```
father -> Person( name -> string;
                  children -> Set(type -> Person))
```

3.5. Constraints

Constraints restrict the set of values that are allowed for a particular feature. In defining constraints for an arbitrary type, it should be possible to test for equality and inequality. Furthermore, standard comparison relations (like <= or >=) should be possible for types that have an ordering on them. For sets of terms, a membership check (in) and inclusion relations (subset, superset) could also be provided.

In HOPLa we often use *set comprehensions* to define a set of legal values, as in:

```
student -> Person(age -> { x | x <= 28 })

staff -> Team(employee -> Person(name -> Fst:string);
              manager  -> Person(name -> {p | p != Fst}))
```

In the first example the feature `age` is restricted to containing integer values less than or equal to 28. The second example states that the `name` of the manager should be different from the name of an employee. Note that we can use variables (`fst`) too to refer to parts of the term.

4. Coordinated actions

Although the data involved in a collaborative process can now be expressed in a convenient manner, we have not yet the ability to define a way of working, i.e. to define the tasks that should/could be performed within the context of a process. Of course, in addition to the decomposition, we also want to impose constraints on the performance of these tasks, and thus express who performs a particular action, or when an action should be performed.

In Ariadne we prefer to blend the definition of the workspace and of the way of working. Phrased differently, tasks to be performed by people or tools are (directly associated with) terms in the workspace. The idea is that performing a particular task will result in an update of some piece of data that is stored in a particular place in the workspace. In the remainder of this section we will discuss this approach.

4.1. Action terms

A task can be viewed as the description of an action that is (to be) performed by some actor. In HOPLa we take the following information about an action into account:

- An action is performed by one (or possibly multiple) actors.
- An action involves performing a particular function or reaching a particular goal. In the case where a computer system performs an action, this can even be a program or piece of code that should be performed.
- An action is performed on a certain location (site, processor, etc.)
- An action is started and terminated at a certain moment in time.
- An action has a certain execution state, indicating whether it is disabled, enabled, busy or done.

We can summarize this in the following termtype

```
Action<Any(  actor      -> Actor;
             started    -> Time;
             terminated -> Time;
             state      -> State;
             location   -> Place;
             function   -> Function)
```

Note that we assume here that an action is performed by a single actor and performed at a single location. `Actor`, `Time`, `State`, `Place` and `Function` refer to termtypes representing actor values, time values, state values, location values and function values, respectively. Their type definitions all follow the same pattern:

```
Actor(value -> string);
Time(value -> integer);
…
```

Instances of the `Action` termtype represent work that is to be done, is being done or has been done, depending on the progress of the work in the process. To support this, action terms are recognized and handled by the Ariadne run-time system and the `state` feature determines what is to be done with it.

When the run-time system determines that a particular action can be performed it will set its `state` from “disabled” to “enabled” and will try to find an actor that can perform it. If an actor decides to perform the activity, the system is informed, and it will mark the term as being “busy” and log the start time. Similarly when the system is informed that work on a particular action has been completed it will mark the state of the term as “done” and it will fill in all the information related to this event in the corresponding features, i.e. it will register the time the action was completed, where it was performed, etc.

In updating these features, the normal subtyping rules apply. This means that it is fairly straightforward to impose constraints on the performance of actions. For example, we could define the following term

```
m   -> Action(actor -> Actor(value -> “florijn”);
               terminated -> Time(value -> {x | x < 12345));
```

which indicates that this action can only be performed by the given actor and should be terminated before a particular deadline. Of course, the Ariadne system can use constraints like these to actively inform people of tasks they can perform and to remind them of deadlines approaching.

4.2. Combining Actions and Data

In principle, we can use action terms to define a collection of tasks and to impose constraints on their execution. However, we do not yet have a way to capture or constrain the result of work on a task in the case it should produce some data. One option would be to add a result feature to the `Action` termtype and to impose constraints on that, e.g. like this

```
n   -> ResAct<Action(result -> integer);
```

While this is perfectly legal HOPLa, we think it is more attractive to blend the definition of actions with the definition of the workspace. Then, a process definition can become similar to the definition of a data structure whose ingredients are to be provided by user-actions.

To support this, HOPLa allows us to define a termtype that is a subtype of multiple other types. It is defined by using the + operator, as in:

```
Assistant<Student+Employee;
```

The same principles as with normal subtyping apply, with some additional constraints. First the set of feature labels in each of the supertypes should be disjunct, and second, features from shared ancestors are included only once in the instances of these types and constraints added by intermediate types should be mutually compatible.

Basically, the semantics of an action term now is, that it allows an actor to modify the values of all the features in the given action term. So if the this term is also an instance of another type, the features of that type also become accessible and can be modified - in accordance with the subtyping rules defined earlier. Now it is trivial to combine the workspace definition with the definition of tasks. For example, in:

```
Form( name -> string;  age -> integer );

test      -> Form+Action
```

`test` represents a term that will be open to user-activity that can set the features `name` and `age`. Note that there is no fundamental requirement that either or both of these features are actually modified; there is just an opportunity to do so.

Turning "data" terms into actions does not work directly for builtin types like `string` or `integer`, since these have no features. However, this can be corrected easily:

```
Integer<Data(value -> integer);
String<Data(value -> string);
```

Here, `Data` refers to a predefined termtype (without features) that is used to identify data terms (as opposed to Action terms). Most of the termtypes we saw earlier are defined as subtypes of `Data`. Given these definitions, we can now define a form in which two fields can be modified by distinct user activities:

```
f      -> Form(name -> String+Action;
               age  -> Integer+Action)
```

Note that - as a special case - when a collection termtype like `Set` is combined with `Action`, the activity also applies to the `value` feature. In this case, an action can only add or remove an element to/from the set. Furthermore note that it is still possible to include constraints on `Action` properties as in:

```
f      -> Form(name -> String+Action;
               age  -> Integer+Action(
                              actor -> Actor(
                                 value -> "greefhorst")))
```

4.3. Coordination operators

The introduction of `Action` terms and the combination of them with data terms provides the basic mechanisms to define a process model for a process. What is still lacking however is the means to define dependencies among the actions. To handle this, we use a special kind of termtypes, called *coordination operators*. These operators are subtypes of termtype `Action` and are predefined. Basically, use of these types defines when the actions that are part of a term can be performed.

The three basic coordination operators and their meaning are:
- `Serie`: actions are to be performed sequentially, in the order in which they appear in a term.
- `Parl`: actions can be performed concurrently.
- `Unordered`: actions can be performed in random order, but only one action at a time can be performed

The operators are typically combined with normal terms. The reason for this is that the operators mainly manipulate the order in which data is produced, but do not influence the structure of that data. As an illustration, consider the each operator in the following examples, where `Rec` denotes a (data) termtype with no own features:

```
a -> Rec+Parl(date -> Time+Action; age  -> Integer+Action)
b -> Rec+Unordered(date -> Time+Action; age  -> Integer+Action)
c -> Form+Serie(date -> Time+Action; age  -> Integer+Action)
```

The first expression (a) defines a record in which the `date` and the `age` can be filled in concurrently. In the second expression (b), either the `date` is filled in before the `age` or vice versa but work cannot take place on both actions at the same time, i.e. not both of these actions can be "busy". The last example (c) states that the `age` needs to be filled in after the `date`, i.e. the first action should be "done" before the second is "enabled".

The coordination operators can also be used in conjunction with sets. Consider the following examples:

```
x   -> Set+Serie(type -> Form+Action);
y   -> Set+Parl(type -> Form+Action);
```

In the first case (x) the elements to be added to the set must be added one at a time. As long as work continues on a particular element (i.e. as long as its `state` feature has not been set to "done") no other element can be added. In the second example (y) this is possible and work can be going on on multiple elements at the same time.

4.4. Composing coordination operators

The coordination operators are just "special" actions themselves. This means that we can combine these operators with other operators. Consider:

```
f   -> Rec+Serie(
         f1 -> Rec+Parl(a -> Integer+Action; b -> String+Action)
         f2 -> Rec+Serie( x -> Integer+Action; y -> String+Action))
```

The two "sub-forms" of `f` must be filled in sequentially, i.e. work on `f1` must be completed before work on `f2` can start. The actions on form `f1` can occur in parallel, while for `f2` two they must occur in sequence.

To combine coordination operators, we must define when such a term is considered "busy" or "done". These definitions are fairly intuitive. A term is "busy" when one of its component actions is "busy" and "done" when all its component actions are "done". However when we combine coordination operators with collections, it is not always obvious when the terms should be marked as "done". Of course, if the elements of a collection are defined explicitly (e.g. through the use of set comprehensions), this is not a problem since the term is "done" if all its elements have been performed. However, in this case:

```
z   -> Person+Serie(name     -> String+Action;
                    children -> Set+Parl(type -> Person+Action);
                    age      -> Integer+Action)
```

it is not known to the system when the setting of the `children` feature can be considered to be complete, so that the setting of the `age` field becomes enabled. In fact it is only the user who knows this, so one possible approach to this problem is to give him control over the "completion" of this term by allowing the setting of the `state` feature to be done in a separate action:

```
z   -> Person+Serie(name     -> String+Action;
                    children -> Set+Parl(
                                  type -> Person+Action;
                                  state -> State+Action);
                    age      -> Integer+Action)
```

In fact, this situation is common enough to introduce a special termtype for user-controllable states:

```
UserTerminated<Action( state -> State+Action )
```

4.5. Choices

In some situations, several actions - each with their own associated terms - could occur at a specific point along the process, but it is not known a priori which of these alternatives will actually occur. The choice among the possible alternatives may only be known during execution, but could even depend on information that is fully outside the scope of the process itself.

To model this, a coordination operator `OneOf` can be defined which expresses that only one of its action components can actually occur during execution of a process. As soon as one of these alternatives is completed, the OneOf term is marked "done" too. As an example, consider the following form, where a choice can be made to provide a registration number or the name and address information:

```
f -> Form+OneOf(
          registrationnr -> String+Action;
          personalinfo   -> Person+Action(name -> String; ...))
```

To avoid superfluous work, it may be desirable to enforce early commitment, so that when one the alternatives is chosen (becomes "busy"), the rest of the alternatives become "disabled". Alternatively, we could let work on multiple alternatives proceed in parallel, which would allow the `OneOf` construct to be used to model an exception handler (e.g. work takes place on one alternative until it appears that something went wrong and another alternative is still chosen). Currently, the first semantics is used.

5. Processes and their execution

The previous two sections have defined all the ingredients that allow us to define Ariadne processes. It has become clear that we can view a process definition as a combination of termtypes and terms, including the definition of actions and their coordination. Technically, an Ariadne process can thus be seen as an Action term. However, to actually "run" a process some additional information is needed such as the actor who initiated the process. In addition, the run-time system must recognize processes and handle them accordingly. Therefore, we come to the following predefined termtype `Process`:

```
Process<Serie(initiator -> Actor)
```

where we have chosen that the execution sequence for the top-level features of a `Process` is that of `Serie`.

Given this definition, we can focus on the "interpretation" of processes as handled by the Ariadne system. First of all, the creation of a new process implies the instantiation of a `Process` term. Typically this is done by providing a HOPLa process definition (or program) consisting of termtype definitions and one initial term that is of type `Process`. Ariadne parses the HOPLa program, instantiates the initial term (by copying all the inherited attributes), logs some information about the process, such as the actor that initiated it, and then starts handling activities involved working in and managing this process.

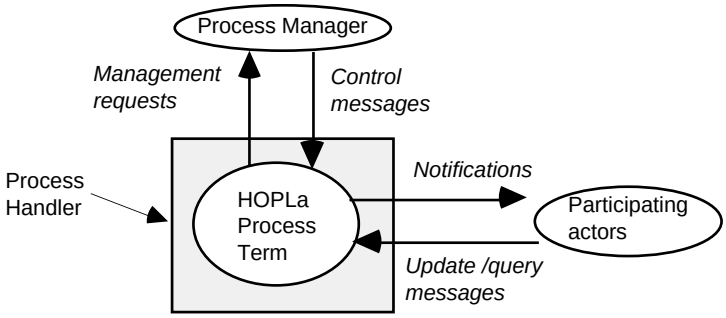

Figure 1: Ingredients of process handling

This "process handling" involves two sides. On the one hand, we have to handle incoming events that represent, for example, the results of work on a particular action. On the other hand, we must signal certain sitations in a process so that the system can actively support users, e.g. by informing actors that they can perform work on a certain action. Interaction between (the handler of) a process and users or computer pograms is thus a fundamental requirement. It is done by exchanging messages across some communication medium like electronic mail or a direct network connection. The actual medium used is determined by the handler. The (incoming) messages ultimately take the form of a term, an instance of a HOPLa termtype `message`, though the handler may create these messages from some other representation.

The process handler parses incoming messages, checks whether they are acceptable and may send a response to the sender. We distinguish various kinds of messages. A *query-message* is a request to retrieve a part of the process (workspace) term. It contains a "path", identifying the sub-term wanted, and returns this term as HOPLa text. These messages are acceptable at any time. An *update-message* on the other hand corresponds to the performance of an action. Again, a path pointing to the action term in the process must be included, but in addition, features of that term can be set by giving their labels and the new values. Also, it should be indicated whether the action has been completed or is still "busy". In either case, the handler must check whether this update is acceptable, e.g. with respect to coordination operators and the subtyping rules. If so, the term is updated

If an action is completed, the process can be reevaluated to find out whether new actions could be performed now. When new actions become enabled, we have to determine the actor that will perform the task and possibly allocate additional resources like rooms, etc. While in some cases it may be precisely known (through constraints) which actor and resources should be involved, this will typically not be the case. This means that some "process manager" will have to appoint the actor(s) and allocate the resources. Clearly, this could be done in many different ways. One approach is that the initiator of the process is considered to be in charge and should do this. Regardless of the particular approach, the process manager is sent a signal requesting the allocation, and will respond to this by means of a *control-message* that states the actual allocation. Now the appointed actor can be notified that work can be done. He can accept this by confirming that work has started (via an update message that sets the action state to "busy") or by immediately sending the results and completing the action.

Obviously, the handler has to maintain state information about the interaction with the rest of the world. For example, it should be avoided that an actor is notified repeatedly that (s)he can perform one particular action. This kind of information can again be stored in the process term in features predefined in the `Action` termtype, like:

```
allocationinitiated -> Boolean;
actornotified -> Boolean;
…
```

In addition to the interaction with actors and checking the constraints in the term, the system also provides some other basic functions, among which is the ability to save the complete process state (i.e. the complete term) in textual form on some file. This textual form is just plain HOPLa code, which combines the current state and all the information that was present in the original process definition (like the type definitions). This means that both the process can be edited - and thus adapted - easily and then restarted.

6. Examples

In the previous sections we have extensively described the HOPLa mechanisms for describing collaborative processes and the way in which they are handled in Ariadne. In this section we illustrate the current system by discussing some examples. At the same time we address some issues that cannot yet be handled and that are subject of current research. For some more examples the reader is referred to [Besamusca95].

In this discussion we will focus on (semi-) structured processes and not deal with ad-hoc processes, where the workspace and process model are defined in the course of the work. The main reason for this is lack of space, since we would have to include many "dumps" of process states at various points in time. However, HOPLa's ability to refine a term combined with the ability to freeze, edit and continue a "running" process provides adequate support for these situations.

6.1. Discussions

In this example we model an electronic discussion between a group of persons. A straightforward HOPLa definition for this would be:

```
Discussion<Process(
    group -> Set+UserTerminated( type  -> Actor+Action;
                                 value -> PS: set ));
    discuss -> Thread<Rec+Serie(
        message -> String+Action(actor -> { p | p in PS});
        replies -> Set+Parl(type -> Thread )))
```

When a process is created according to this definition, first of all the set of actors participating in this discussion has to be defined. Though we have not added it here, this would typically be limited to the initiator of this process. After the group has been defined, a string for the `message` feature has to be provided by one of the actors in the group. After that, replies can be added by members of the group, each of which is the beginning of a separate, parallel thread. Since there are no explicit termination conditions, this process will never "stop". As this example demonstrates, the Ariadne implementation must be able to handle recursive types by instantiating them lazily.

As a slight variation on this basic theme, let us consider a discussion where we don't want a person to reply to his own message. A simple definition would be:

```
Discussion<Process(
     group -> Set+UserTerminated( type  -> Actor+Action;
                                  value -> PS: set ));
     discuss -> Thread<Rec+Serie(
         message -> String+Action(actor -> S: { p | p in PS});
         replies -> Set+Parl(
                     type -> Thread(
                           message -> String+Action(
                                          actor -> {x | x != S})))))
```

Here we see that in order to model this constraint we have to override a feature in the recursive use of the type `Thread`. While this definition is intuitively appealing it does not work correctly in our current implementation, since there it is assumed that variables are global for a term (as should be the case for PS). Also, the combining the two constraints on the actor that can add the message - it should not be the previous sender but still be a member of the group - has not yet been realized.

6.2. Negotiation

Another variation on the basic conversation model is the negotiation-cycle. Remember that in a negotiation-cycle a supplier and customer try to reach consensus about some topic. The HOPLa representation of a simplified process could look like:

```
Negotiation<Process(
        prepare -> Rec+Serie(
            supplier -> S: Actor+Action;
            customer -> C: Actor+Action);
        negotiate -> Proposal<Rec+Serie(
            proposal -> String+Action(actor -> { s | s == S } );
            decision -> OneOf(
                 accept -> Action(actor  -> {c | c == C});
                 revise -> Serie(
                     comment   -> String+Action(actor  -> {c | c == C});
                     revision  -> Proposal))))
```

In the preparation phase we collect the the supplier and the customer. Following this the supplier has to create an offer. The customer can then indicate whether he is satisfied with the proposal. If so, the whole process is done, otherwise the customer can give comments and there is going to be a revised proposal. Again, we assume that the data is represented as strings.

6.3. Version Management

The purpose of a version management process is to organize the evolution of a particular data object by recording multiple versions of this object and by storing the derivation history of versions (i.e. the successor relation). Of course, historical versions should be accessible, which is possible using query messages. An important additional aspect is concurrency control. We may want to avoid that multiple new sucessors are created for a particular version at the same time. In conventional systems this is achieved by locking the version and disallowing further checkouts for updating until the new version is checked in and the lock is released. Alternatively, however, we may take an optimistic view and allow these concurrent updates, assuming that users will merge conccurrent changes later on.

First we look at the “optimistic” variant. We see that a version term holds some data (not further defined here) and a number of successors:

```
VersionTree<Process(
    root  -> Version<Rec+Serie(
                 data -> Code;
                 successors -> Set+Parl(type -> Version)))
```

The fact that we use the `Parl` operator for the successors means that multiple “updates” can be active at any given moment in time. Once a user starts working on a new version a corresponding entry in the set is created which can be operated upon in parallel with others. Of course, further information could be recorded for versions like why it was created and what was changed. Also we could restrict the users who can perform these modifications by constraining the actor field.

An intuitive approach to model “locking” behaviour would be to replace the `Parl` operator with `Serie`:

```
VersionTree<Process(
    root  -> Version<Rec+Serie(
                data -> Code;
                successors -> Set+Serie(type -> Version)))
```

However, this does not work, because in this case the successors set is locked for further additions until the complete version history for the newly added version has been added. Since this process will never explicitly terminate, this will never occur. So, this definition implies that we have linearized development. A good solution for this problem is currently still being investigated.

6.4. Electronic Voting

The purpose of the electronic voting example is to model the situation where a group of designated users are asked to vote on a particular topic. Users can make a choice from a set of options defined by the organizer of the vote.

```
ElectronicVote<Process(
        prepare -> Rec+Parl(
            topic    -> String+Action;
            deadline -> D: Time+Action;
            choices  -> Set+UserTerminated(  type  -> String+Action;
                                             value -> CH: set );
            group    -> Set+UserTerminated(  type  -> Actor+Action;
                                             value -> PS: set ));
        votes   -> Set+Parl(
            type  -> Vote<Form+Action(
                          choice -> {c | c in CH};
                          actor  -> {p | p in PS})))
```

The voting process consists of two parts; the preparation phase and the actual voting. In the preparation phase the topic, deadline, possible choices and the participants are set by the initiator of the process, though this is not explicitly enforced through actor constraints. In the voting phase, all participants as denoted by PS must cast a vote which is an element of the set CH of possible choices.

It should be noted that this definition is not precise enough, in that the constraints do not prevent actors from voting more than once (though the run-time system informs them only once). Furthermore the definition lacks the enforcement of a deadline. Basically what should happen is that the set of votes is disabled (i.e. no new entries can be made) once the deadline passes. It would be nice if this would happen automatically, but the only way that is possible now is to let the state of the set be modifiable by an action and to let some computerized actor change the state, e.g.:

```
votes -> Set+Parl(
            state -> State+Action(
              actor -> Actor(value -> “some tool”);
              terminated -> D;
              function -> Function(value -> “set state to done”)))
            type ->Vote<Form+Action(
              choice -> {c | c in CH};
              actor  -> {p | p in PS}))
```

Clearly, a more elegant solution to this problem would be desirable.

7. Conclusions

The Ariadne system - and its coordination language HOPLa - aim to provide generic support for hybrid collaborative processes. This means that a broad class of activities should be acceptable ranging from ad-hoc unstructured processes to routine, structured processes. The examples discussed in this paper show that Ariadne and HOPLa can handle this variety. The workspace and the process model can be specified in arbitrary detail while the resulting process definitions are still fairly simple and intuitive. Furthermore, the Ariadne system can provide active support for all of these processes.

We have shown that the need for flexibility is addressed in two ways. First, process workspaces can be adapted arbitrarily within the limits of the type-definitions stored within the. This can make it easy, for example, to add extra tasks or to annotate existing data. Second, running processes are fundamentally the same as their definitions. Both are HOPLa programs and can be represented in textual form. Since a process is fully self-contained, this means that a running process can be aribitrarily modified (through editing) whenever exceptions give rise to this, and without the loss of information that has already been provided.

Current research

Work on Ariadne and HOPLa is continuing both on a conceptual and a practical level. Here we describe some of the topics that are currently being addressed.

As seen in section 6, there are some issues that are not handled properly by the current HOPLa definition. One problem area is the combination of coordination operators with sets, and in particular the control over the "locking" behaviour when adding elements and the completion conditions for the set itself. Further work is needed here. In fact, we are considering a more general mechanism by which the coordination of component actions and the completion conditions of coordination operators could be described explicitly. This would mean that users could to some extent define the behaviour for the operators they want. This would also allow us to distinguish between the two different interpretations of `OneOf` (section 4).

Most of the Ariadne and HOPLa functionality described in this paper has been implemented. After an initial prototype in the functional language Gofer, the current prototype is implemented in C++ [Greefhorst95]. One extension that is currently being considered is a tool that provides a graphical representation of HOPLa programs. This tool could visualise the actual state of an Ariadne process, but would also make the definition and adaptation of processes easier.

The self-descriptive, self-contained nature of Ariadne processes offers some other potential advantages that have to be explored in more detail. For example, migration of processes is conceptually straightforward, in that a process can be sent to another Ariadne interpreter on another machine, and handled there. Users can even take a process along on a portable machine and bring it back later. Furthermore, the fact that processes can be viewed as (textual) data objects introduces the option of meta-processes, i.e. processes that operate upon other processes or process definitions [Pemberton91].

The process handler (figure 1, section 5) defines how an Ariadne process interacts with the outside world. It uses some communication mechanism, translates and handles incoming messages and turns state changes in the process into signals to the outside world. Obviously, we can envision multiple handlers that behave and present themselves in different ways. A trivial difference would be in the use of the communication mechanism, e.g. direct network connections vs. electronic mail, which might provide a basis to support both synchronous and asynchronous collaboration in the same process. But, also the information that is exchanged could be in different formats or be presented in different ways, and even the way in which process state changes are handled could be different. This leads to the notion of different *enactment styles* [e.g. Kaiser93] for a particular process. For example, a more passive handler would let users decide which task they want to work on next and then try to meet the constraints needed to make that happen. In yet other cases the existence of a process model may be implicit- users only see a visual representation of some data in the workspace and modify it. In this case, the handler should track the activities performed by the users and check whether these correspond to (enabled) actions in the procedure. It is simpe to implement different handlers explicitly and use them. But a more interesting opportunity however is to describe the enactment policy (including perhaps presentation aspects) in some high-level formalism and make them user-definable. This is an issue that is currently being studied.

Support for process management is also being explored. As indicated before, process management involves the responsibility to allocate actors and other resources to particular tasks while meeting the constraints imposed in the process model. This requires some sort of database to store all the available resources and some approach to satisfy the contraints (see [Boerboom94] for some of the issues and potential solutions). An additional issue is how to appoint the best resource when multiple candidates are available. Clearly, making such choices requires a good overview of other running processes and the resoures they (will) need. Also, the possibility of reflective processes [Bandinelli93], where Ariadne processes are used to perform these allocations, is considered.

8. References

[Ait-Kaci86] H. Ait-Kaci, "Type Subsumption as a Model of Computation", in *Proceedings of the First International Workshop on Expert Database Systems,* L. Kerschberg (Ed.), 1986.

[Arbab93] F. Arbab, I. Herman, and P. Spilling, "An overview of Manifold and its implementation," *Concurrency: Practice and Experience*, vol. 5, no. 1, pp. 23–70, February 1993.

[Banâtre90] Jean-Pierre Banâtre and Baniel Le Métayer, "The Gamma Model and its Discipline of Programming," Science of Computer Programming, 15, pp. 55-77, 1990.

[Bandinelli93] Sergio Bandinelli and Alfonso Fuggetta, "Computational Reflection in Software Process Modelling: the SLANG approach," in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, 1993.

[Besamusca95] Timo Besamusca, "Defining a Coordination Language for Hybrid Office Processes," Master's thesis, University of Utrecht, Department of Computer Science, 1995.

[Boerboom94] Benno Boerboom, "Primitiva voor het modelleren van samenwerkingsprocessen," Master's thesis, University of Utrecht, Department of Computer Science, February 1994.

[Carriero92] Nicholas Carriero and David Gelernter, "Coordination Languages and Their Significance," *Communications of the ACM*, vol. 32, no. 2, pp. 96–107, February 1992.

[Conklin88] Jeff Conklin and Michael L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," Technical Report STP-082-88, MCC, Software Technology Program, March 1988.

[Ellis91] C.A. Ellis, S.J. Gibbs, and G.L. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM*, vol. 34, no. 1, January 1991.

[Ellis94] C.A. Ellis and J. Wainier, "A Conceptual Model of Groupware," in *Proceedings CSCW'94*.

[Flores88] Fernando Flores, Michael Graves, Brad Hartfield, and Terry Winograd, "Computer Systems and the Design of Organizational Interaction," *ACM Transactions on Office Information Systems*, vol. 6, no. 2, pp. 153–172, April 1988.

[Gommans93] L.F.B. Gommans and A. Nales, "De Administratieve Organisatie in Objecten: een model en een omgeving," Master's thesis, Erasmus universiteit, vakgroep informatica, Rotterdam, August 1993.

[Gulla94] J. Gulla and O. Lindland, "Modelling Cooperative Work for Workflow Management," in *Proceedings CAiSE 94*, G. Wijers et. al. (Eds.), Utrecht, 1994.

[Greefhorst95] D. Greefhorst, "A Simulation Environment for Ariadne," Master's thesis, Utrecht University, Dept. of Computer Science, too appear, 1995.

[Hämmäinen91] Heikki Hämmäinen, et. al., "Distributed Form Management," *ACM Transactions on Information Systems*, vol. 8, no. 1, pp. 50–76, January 1990.

[Heimbigner91] Dennis Heimbigner and Marc Kellner, *Software Process Example for ISPW-7*, Anon. FTP from ftp.cs.colorado.edu, /pub/cs/techreports/ISPW7/ispw7.ex.ps, August 1991.

[Helm90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems," in *Proceedings of the 1990 OOPSLA/ECOOP Conference*, Norman Meyrowitz (Ed.), Ottawa, Canada, October 1990, pp. 169–180.

[Kaplan92] Simon M. Kaplan, William J. Tolone, Douglas P. Bogia, and Celsina Bignoli, "Flexible, Active Support for Collaborative Work with ConversationBuilder," in *CSCW'92 - Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, November 1992, pp. 378–385.

[Karbe90] B. Karbe and N. Ramsperger, "Influence of Exception Handling on the Support of Cooperative Office Work," in *Proceedings of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications*, Simon Gibbs and Alex A. Verrijn-Stuart (Eds.), Heraklion, Crete, Greece, 1990.

[Kaiser93] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul, "A Bi-Level Language for Software Process Modelling," in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, 1993.

[Kreifelts91] Thomas Kreifelts, Elke Hinrichs, Karl-Heinz Klein, Peter Seuffert, and Gerd Woetzel, "Experiences with the DOMINO Office Procedure System," in *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, L. Bannon, M. Robinson, and K. Schmidt (Eds.), Amsterdam, the Netherlands, September 1991, pp. 117–130.

[Libes93] Don Libes, "Kibitz – Connecting Multiple Interactive Programs Together," *Software – Practice & Experience*, vol. 23, no. 5, pp. 465–475, May 1993.

[Malone90] Thomas W. Malone and Kevin Crowston, "What is Coordination Theory and How Can It Help Design Cooperative Work Systems," in *CSCW'90 - Proceedings of the 1990 Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 357–370.

[Malone92] Thomas W. Malone, Kum-Yew Lai, and Christopher Fry, "Experiments with Oval: A Radically Tailorable Tool for Cooperative Work," in *CSCW'92 - Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, November 1992, pp. 289–297.

[Martial90] Frank von Martial, "A Conversation Model for Resolving Conflicts among Distributed Office Activities," in *Proceedings of the 1990 Conference on Office Information Systems*, F. Lochovsky and R. Allen (Eds.), 1990.

[Pemberton91] Steven Pemberton and Lon Barfield, "The MUSA Design Methodology," Technical Report 91/12, Software Engineering Research Centre, December 1991.

[Schill91] Alexander Schill and Ashok Malhotra, "Language and Distributed System Support for Complex Organizational Services," in *Proceedings of the Conference on Organizational Computing Systems (ACM SIGOIS Bulletin)*, vol. 12, Peter de Jong (Ed.), ACM Press, Atlanta, Georgia, November 1991, pp. 1–15.

[Suchman87] Lucy Suchman, *Plans and Situated Actions*. Cambridge University Press, 1987.