TransCoop

ESPRIT Basic Research Action 8012

DELIVERABLE IV.3

DATE OF ISSUE	21 February 1996
AUTHOR(S)	Frans J. Faase, Susan J. Even, Rolf A. de By
ORIGIN	UNIVERSITEIT TWENTE
STATUS	FINAL
CONFIDENTIALITY	TRANSCOOP
RELATED ITEMS	
FILING CODE	TC/REP/UT/D4-3/033

ABSTRACT This report introduces COCOA, the TRANSCOOP specification language for cooperative scenarios. COCOA offers organisational and transactional views of a scenario in a single language. Because it includes features related to both views, COCOA supports workflow descriptions of cooperation, as well as operational descriptions of cooperation. The examples given in this document focus on Cooperative Document Authoring as an application area. COCOA serves as a declarative, high-level front end to LOTOS/TM, which is used to formalise its constructs. The mapping from COCOA to LOTOS/TM is illustrated here by a number of examples.

KEYWORDS	COOPERATIVE SCENARIOS, COCOA
APPROVED BY	TMG
DISTRIBUTION	CEC, within TransCoop

DOCUMENT HISTORY

version	date	reason
1.0	February 21, 1996	Final.

Preface

At an earlier stage in the TRANSCOOP project, LOTOS/TM was targeted as the answer to all our linguistic requirements for cooperative scenario specification. As the project evolved, UT and the project consortium realised that a higher level language more appropriate for end-users was required. This language is called COCOA (pronounced 'koko'), which stands for *Coordinated Cooperation Agendas*. COCOA will serve as a more intuitively appealing interface to LOTOS/TM, with specific language constructs to identify users, workspaces, and cooperative activities. We envision the use of a graphical front-end for COCOA in which boxes are used as activity indications, and arrows and lines stand for forms of messaging and transfer of control. Additional textual information will be used to augment the diagrams to a full specification. This information will include data type and database type definitions, data operation specifications, actors and workspace types, communication, forms of negotiation, and forms of transaction behaviour. The language we discuss in this report concerns the *textual* representation of the eventual language we have in mind. The graphical interface will be defined at a later stage. Please note: Although we have tried to finalise the design of COCOA, parts of the language related to the cooperative transaction model may be revised.

Acknowledgements

Hereby we would like to thank the following persons for their comments and contributions to earlier versions of this document, and for their in-depth discussions about the execution rules: Thomas Tesch, Jürgen Wäsch, Justus Klingeman, Jari Veijalainen, Olli Pihlajamaa, and Henry Tirri.

Contents

1 Introduction

In this report, the language CoCoA is introduced. CoCoA (*Coordinated Cooperation Agendas*) aims at the specification of *cooperative scenarios*, by which we mean activities communally undertaken by a group of actors (users) with an agreed upon goal. As this definition of a cooperative scenario is rather general, there are many ways one could understand the concept. There are at least three dimensions involved in the definition of a cooperative scenario: actors, activities, and data. Informally, an activity can be likened to an "office procedure" that is performed by the actors.

When observing a cooperative activity, such as for example the writing of a book, there are different ways of looking at it. For example, one can look at the writing process of how the writers wrote the book. This view, which we call the *organisational view*, is especially interested in recording who had to write what, and how the premature versions of the different chapters were distributed, and finally approved by the authors. Hopefully, any reader of the final version of the book will not need to have any knowledge of how the text came into existence. There are, of course, all kinds of prerequisites to make a book readable, such as: the chapters should be numbered correctly, the page numbers in the index should be correct, and definitions should be given before their usage. A good way of meeting these requirements is to make use of constraints during the writing process, which prevent the creation of inconsistent data. The so-called *transactional view* looks at how the intermediate versions of the chapters can be combined into consistent results, and at what rules are needed for manipulating the intermediate versions. The transactional view abstracts away from the organisational aspects of the cooperative activity, such as who granted a specific writer the right to modify the text of a particular chapter.

CoCoA integrates the organisational and transactional views of a cooperative scenario into a single language. The organisational view looks at the activities as they are performed by the actors in real time, and focuses on the organisational aspects of the cooperation. It deals with aspects like user roles, communication, and the temporal-ordering of user activities.[1] The transactional view looks at how shared data can be manipulated such that consistency is preserved.[2] It deals with the ordering of data operations, and the combination of partial results through import and export operations. These views emphasise complementary aspects of a scenario: the organisational view determines who can perform data manipulations in the transactional view, whereas the status of intermediate results in the transactional view influences the flow of control in the organisational view.

In this report, the semantics of CoCoA is explained by showing how the two views can be mapped to LOTOS/TM [EvFB95], a formal specification language based on the process-algebraic language LOTOS [BoBr87,ISO87,VSSB91], and the database specification language TM [BBBB95,BaBZ93,BaFo91]. We assume that the reader is familiar with LOTOS, TM, and LOTOS/TM.

[1] In the organisational view, activities comprise all kinds of operations that the users can perform. See Section 2.5 for an overview of these.

[2] By consistency, we mean consistency as defined by database constraints in the schema specification, as well as consistency as defined by the execution rules of the scenario specification.

The remainder of this report is structured as follows. Section 2 gives an example of a cooperative scenario in which the organisational and transactional views are explained in more detail. We also give an intuitive introduction to the cooperative transaction model. In Section 3, a skeleton specification is given to show all of the necessary components of a CoCoA specification; the skeleton is enhanced in later sections, as we introduce the CoCoA language constructs one by one. We focus on language constructs to support the organisational view in Sections 4 and 5. Some advanced features are investigated in Section 6, including communication patterns for negotiation. Language constructs to support the transactional view are then discussed in Section 7, which looks at execution rules, commutativity rules, and data exchange rules for combining partial results. Section 8 discusses the mapping of language constructs found in the organisational and transactional views of CoCoA to LOTOS/TM. Section 9 makes some concluding remarks. The complete grammar for CoCoA is given in Appendix A. The complete text of the example specification is given in Appendix B.

1.1 Scope and usage of the language

The development of a cooperative application can be divided into the following three phases: *conceptual design*, *technical design*, and *implementation*. CoCoA is intended to be used as a specification tool during the conceptual and technical design phases. The users of the CoCoA language will thus be the designers who develop a cooperative application. The end-users of the resulting application will not write specifications in CoCoA, although they may be aware of the language because of its use in documentation.

CoCoA is designed for the specification of the conceptual behaviour of a cooperative application. CoCoA does not cover the design of the user-interface (e.g., which key is to be pressed for which function); nor does it deal with the definition of the cooperative transaction model underneath. CoCoA is designed with a specific transaction model underneath: the CoAct (Cooperative Activities) model [RKTW95,WäKl96]. CoCoA includes features for the specification of data model-specific information that is needed by the CoAct model, such as execution rules and commutativity relations.

CoCoA is designed with TM as the data model specification language. In later phases of the TransCoop project, namely the demonstrator implementation in Work Package VII, TM data types and operations will be mapped to VODAK, which will serve as an implementation platform [GMD-I95].

1.2 Data model assumptions

Each actor in a cooperative scenario has a private workspace. Conceptually, a workspace contains copies of those database items that an actor is working on. Update and retrieval (read-only, query) operations can be applied to a workspace. The sequence of data operations that is applied to a workspace is called a *workspace history*. The data operations that we consider in CoCoA are *atomic, consistency-preserving* state transitions on workspaces (i.e., update

operations). Workspaces are formally specified (by the scenario designer) as TM-typed values.[3] Each data operation is formally specified as a TM method that operates on a workspace value.[4] The workspace is an *implicit* input and output parameter to all data operations. The granularity of the data operations on the workspace is up to the scenario designer. (The level of abstraction is not fixed by TM.)

During the execution of a cooperative scenario, retrieval operations can be applied at any time to an actor's private workspace. In contrast, the application of update operations is controlled by operation enabling in the organisational view, and by execution rules in the transactional view. We assume that only successful (non-failed) operation invocations are kept in a workspace history. Although this assumption is not essential, it allows us to think of a history as the consistency-preserving composition of those operations that it contains.

1.3 Overview of language features

The organisational view of a cooperative scenario divides the work into a number of steps. A step may be either composite or elementary. An elementary step enables a number of data, data exchange, and communication operations for the actor (or actors) filling a certain user role while the step is active. Composite steps describe the temporal ordering of cooperative activities. The orderings include step sequencing, step repetition, parallel step activation, and the parallel activation of (indexed) groups of steps. These orderings introduce dependencies that are specific to the organisation of the scenario, and may address "human" aspects of cooperation. Language features to support the organisational view include: the specification of data, data exchange, and communication operations; elementary and composite step definition; and operation enabling constructs.

Steps have no transactional properties. A step is not a computation. A step describes in a declarative manner what can be done by different actors at a particular time. A step grants permissions for data, data exchange, and communication operations to be executed. Data operations are atomic computations. How data operations are sequenced together is up to the actor(s). However, to avoid nonsensical computations, some operation ordering guidelines are needed. These are provided by the transactional view.

The transactional view of a cooperative scenario is concerned with "consistency" aspects of the work—consistency for the data in single-user (private) workspaces as well as in the multi-user (shared) workspace. The cooperative transaction model that supports CoCoA relies on some data model-specific information about the scenario to ensure that operations are applied to workspace data in a consistency-preserving manner. This information is defined in CoCoA and used by the transaction manager. Language features to support the transactional view include: the specification of data operation ordering rules; data exchange protocols for importing and exporting; and operation commutativity relations.

[3] Please note: Values in TM may be arbitrarily complex, ranging in complexity from an integer or character, up to an entire database. Consult the previous Work Package IV deliverables for more details [EvFa94,EvFB95].

[4] Methods can be arbitrarily complex in TM, just like the values they operate on.

2 An example CDA scenario

The example cooperative scenario that we use throughout this document deals with an editor who, with the help of some co-authors, must write a document. This example is based on an example cooperative scenario that was presented in [FaBy95]. We have tried to keep the example as concise as possible. It's simplicity should not be regarded as typical of cooperative scenarios. The editor plays an important role in the organisation of the work: She decides who will be the co-author(s), and assigns tasks to them. Each task consists of writing a chapter, where the writing can be assigned to a group of authors.

In the following sections, we look at the organisation of the scenario, the working environment of the authors, some data operations and some data exchange operations.

2.1 Organisation of the scenario

The overall organisation of the cooperative scenario consists of three steps: the *proposal step*, the *writing step*, and the *finalisation step*. These steps are distinguished to highlight important organisational boundaries in the way the work to be done is set up, and to establish protocols for performing the work. We point out that the scenario organisation we describe here is only one of many ways to organise a cooperative document writing task.

During the proposal step, the editor is responsible for writing the title page and the introduction. These need to be approved by an external referee, before the writing step can start. There is also a possibility that the whole scenario is cancelled by the referee before any writing of the chapters has started, or that the referee tells the editor to revise the introduction, but (in this example) once the writing step has started, the scenario can be completed successfully.

During the writing step, the editor will assign tasks for writing a chapter to a group of authors. The authors are taken from a set of authors that is determined at the *start* of the scenario. Whenever the authors agree that they have completed their work, they inform the editor about this, after which the task is completed. During the writing step, any number of tasks can be in progress, and any author can be involved in any number of tasks. The authors of a chapter are responsible for the organisation of the writing; authors take turns writing and annotating the chapter. The editor determines when the writing step is completed.

The purpose of the finalisation step is to complete the document; it includes things like: fixing the introduction, completing the conclusion, correcting the bibliography and so on. The editor has the responsibility for these tasks, and when she determines that the document is ready, the scenario terminates.

In the above description, a number of decision points are mentioned where one of the actors (the editor, the referee, or an author) influences the *control flow* of the cooperative scenario. For this purpose, the following *communications* are used:

`ed : introWritten()`	`r : introOkay()`
`r : reviseIntro()`	`r : abortWriting()`
`ed : startTask(c,ag)`	`a : completeTask(c)`
`ed : completeWriting()`	`ed : documentReady()`

In the above, '`r`', '`ed`', and '`a`' stand for the referee, the editor, or the author that issues the communication. The parameter '`c`' stands for the chapter, and '`ag`' is used to specify the group of authors that is to write the chapter.

Each communication is initiated by a single actor and changes the state of the scenario, usually by causing some step (or steps) to be started or terminated. Section 6.3 describes how the communications can be used as a basis for specifying cooperative decision making. Communications should not be understood as being communication events between different actors, as the name might suggest; they should be understood as events by which actors effectuate their decisions in a running scenario.

We assume that if the authors working on a certain chapter agree that the chapter is okay, only one of them will issue the `completeTask` operation to inform the editor. The negotiation process for coming to an agreement is not formally supported by this scenario.

To summarise, we draw attention to all the different *organisational* aspects that occur in the CDA example:

- Sequencing of steps (e.g., like the three steps of the overall scenario: proposal, writing, finalisation).
- Subdivision of steps. (The proposal step can be thought of as two smaller steps, namely: "writing the introduction" by the editor, and "approving" by the referee.)
- Repetition of steps. (This occurs in the proposal step if the referee requires the editor to revise the introduction.)
- Parallel execution of steps (e.g., if we consider the task of writing a chapter as a separate step).
- Dynamic creation of steps.
- Dynamic allocation of actors to a step.
- Both "strict" and "free" organisation of the work within a step. (The proposal step is an example of a strictly organised step, whereas the task of writing a chapter is an example of a freely organised step.)
- Communication by which actors influence the control flow.

2.2 Working environment

The working environment is based on the Cooperative Activity Model (CoAct) [RKTW95], which allows actors to work independently on partial results, and provides merging abilities to combine these partial results into a final result. We describe the working environment in terms of the example.

The document is stored in a shared data structure that is accessible to all co-authors. The co-authors, however, work in their own workspaces, where a full version of the document also lives. All changes that an author makes to the document are, at first, only applied to the local version that resides in the author's workspace.

Authors can exchange their work by special import and export operations. COCOA provides facilities for "querying" the sequence of data operations that have been applied to a local version of the document (i.e., a workspace history). This allows the scenario designer to consider the exchange of partial results either as the exchange of data operations (as it is defined in the cooperative transaction model), or as the exchange of workspace data (as is the case when an import or export query over the workspace history is defined to retrieve all changes to a data item).

Local changes to a chapter can be exchanged directly between two author's workspaces, by importing or exporting data operations, or indirectly, by first exporting data operations to the shared workspace, after which other authors can import the operations. Local changes to a private workspace must be explicitly exported to the shared workspace to be made part of the final result of the scenario.

2.3 Data operations

Besides writing a chapter, the authors also have the ability to annotate chapters, i.e., they are granted the right to make marginal notes. These annotations are exchanged with the chapters, or exchanged separately. Of course, there is a subtle relationship between the version of the chapter and its annotations, when we realise that annotations are removed once integrated with the text of the chapter: combining the text of the chapter with the annotations results in a complex kind of version, because annotations can be added and removed independently from changes to the text.

For simplicity, we consider the title page, the introduction, the conclusion, and the bibliography as chapters in our data model. The following operations are provided for writing activities[5]; each of these can be executed in the workspace of a co-author:

`a : addChapter(a,c,t)`	`a : addAnnotation(a,c,an)`
`a : delChapter(a,c)`	`a : remAnnotation(a,c,an)`
`a : editChapter(a,c,t)`	

[5]Please note: This is a simplified example. Cooperation would obviously be increased if the operations were defined on paragraphs instead of chapters.

All of these operations affect a *single* workspace only. Each operation takes parameters: `a` stands for the author who issues the operation, `c` stands for a chapter identification, `t` stands for some text, and `an` stands for an annotation. The text parameter probably has type string; the annotation parameter probably has a record type and includes a number of components.

To support an efficient cooperation, information about who issued an operation is kept, so that it is always known who edited a chapter for the last time, who added an annotation, and so on. For this reason, the value of the `a` parameter is stored with the operation invocation in the workspace history, as one of its parameters.[6] Actor annotations come in useful when ordering rules (execution rules) are defined for operation invocations. For example, by using actor annotations, it is possible to express the restriction that the actor who deletes a chapter must be the same actor who added it. (See Section 7.1.)

2.4 Data exchange operations

For combining intermediate results, the CoAct model uses a merging algorithm that is based on the history of operations that an actor has issued in her workspace. Whenever data is imported from one workspace into another, the operations that were executed on this data are to be "re-executed" in the destination workspace. Problems can arise if these operations work on the same data (e.g., the same objects) as operations already executed in the destination workspace. Therefore, the operations that are to be re-executed are first checked against the operations already executed in the destination workspace, and if there are no "conflicts" they can be re-executed. A COCOA specification will contain the rules that the CoAct model requires for determining whether or not two data operations conflict. (See Section 7 for details.)

Continuing with the example scenario, the following operations are given for exchanging versions of chapters and chapter annotations:

`a :` **import** `Chapter(c)` **from** `w`	`a :` **import** `Annotations(c)` **from** `w`
`a :` **export** `Chapter(c)` **to** `w`	`a :` **export** `Annotations(c)` **to** `w`

Each of these data exchange operations takes parameters: `a` stands for an author executing an operation, `c` stands for a chapter identification, and `w` stands for a workspace or the shared workspace identification. These parameters will receive values dynamically, during the execution of the scenario. The scenario designer specifies a range of acceptable values that the parameters may take. (See Section 5.) The private workspace of an author can contain data related to more than one chapter. Depending on the data exchange protocol (see Section 7.2), each of the above operations may have no effect (i.e., nothing is imported or exported).

From the perspective of an author, her workspace contains versions of the chapters she is working on. However, from the perspective of the transaction model, the workspace contains a history of operations that were executed. The data exchange operations are defined

[6] We point out that using explicit actor annotations in the operation signatures is a design choice.

in terms of selecting data operations from the history in the source workspace, and merging them with the history in the destination workspace. The data exchange operations are defined in such a way that they always transfer all dependent data operations from a workspace history, which contributed to a result. (See Section 7.2 for a precise definition.) Thus, from the perspective of the author, the data exchange operations appear to import or export data.

2.5 Summary

In the previous sections, three kinds of operations have been mentioned:

communications These operations influence the execution of the scenario at an *organisational level*. They usually influence the rules that determine *who* is allowed to execute the data operations and data exchange operations.

data operations These operations manipulate the data in the workspaces. Data operation invocations are kept in the workspace histories.

data exchange operations These operations allow intermediate results to be exchanged among workspaces.

The transactional view deals with the ordering of data operations in a workspace (viewed in isolation), and with the merging of intermediate results from one workspace into another with a minimised loss of work. The shared data repository is also viewed as a workspace. The organisational view deals with the ordering of all three kinds of operations on a global scale. However, because of its global nature, the organisational view typically does not address the ordering of individual data operations in a single workspace.

3 Specification skeleton for CoCoA

Although the organisational view and the transactional view are dealt with in separate syntactic parts of CoCoA, it is obvious that they are not entirely independent. A specification consists of a total of nine components, which we order according to the *definition-before-use* principle. We have chosen to organise the language in an "inside-out" way, meaning that data types, data operations, workspace types, and user types are identified first, since they form the definitions common to both views. The organisational view of the language takes shape at an "outer" level, where communications, data exchange operations, and the protocol (e.g., control flow) of the scenario are identified. The transactional view of the language is reflected in the ordering rules for data operations, and in the directives for history management in the case of merges.

A CoCoA specification consists of the following components:

1. **Data types**
   This component is a list of relevant TM data types. Types can be defined here; and they can also be imported from a TM schema specification.

2. **Data operations**
   This component is a list of invocable data manipulation operations with their signatures. These operations are applied to the local workspaces, and to the shared workspace.[7] The code for these operations is provided elsewhere in the database schema.

3. **Workspace types**
   This component is a list of relevant workspace types. The shared workspace and the private workspaces are identified here. For each workspace type, the set of data operations that it allows is given. Data operations can appear in more than one workspace type. (See Section 5.2 and Section 7.)

4. **User types**
   This component is a list of user roles. The actual users are determined at execution time (i.e., when the scenario is instantiated). For each user type, a workspace type is specified; an optional data type may be specified as a user identification. Workspace types help support the idea of user roles.

5. **Communications**
   This component defines a list of end-user communication patterns that can be used by actors to influence the control flow of the scenario. (See Section 6.3.)

6. **Data exchange operations**
   This component is a list of operations that allow the shipping of partial results (in terms of data operations) to and from private workspaces and/or the shared data structure. Operation signatures and definitions are specified here. (See Section 7.2.)

7. **Procedure**
   This component specifies the organisation of the steps and events that may occur in the scenario. In a typical usage of the language, we would expect this part to be (at least partially) generated from a graphical tool. This component is explained in detail in Sections 4, 5, and 6.

8. **Data operation order**
   This component specifies ordering rules for the data operations. These rules govern the ordering of operations within a workspace. Breakpoints and termination constraints are also defined here. This component is explained in Section 7.1.

9. **History rules**
   This component is a list of constraints in the form of non-commutativity statements for the operations in the named history. The rules express the so-called mergeability criteria. History rules are discussed in Section 7.2.

For the first five components, the example CDA specification is as follows:

[7] Data operations only manipulate the shared workspace when they are re-executed there via an export operation.

scenario `write_document` **includes** `document_schema`

data types
```
chapter, text, annotation
```

data operations
```
addChapter(actor, chapter, text)
delChapter(actor, chapter)
editChapter(actor, chapter, text)
addAnnotation(actor, chapter, annotation)
remAnnotation(actor, chapter ,annotation)
```

workspace types
```
cda = { addChapter, editChapter, delChapter,
        addAnnotation, remAnnotation }
```

user types
```
actor,
editor isa actor using cda,
author isa actor using cda,
referee using cda
```

communications
```
introWritten()
startTask(chapter, ℙ author)
completeTask(chapter)
completeWriting()
documentReady()
```

A CoCoA specification is given with respect to a TM database specification[8]; all **data types** and **data operations** that are not specified here refer to TM types and methods specified in the included TM schema. Thus, the data types `chapter`, `text`, and `annotation` are imported from the TM module `document_schema`. For ease of presentation, TM type definitions for the **user types** `actor` and `referee` have been omitted. Definitions for both of these types should either be given here, or be defined in the TM database schema. The '**using** `cda`' construction is CoCoA-specific, and is used to declare the workspace type associated with the user role.

A TM specification does not distinguish between data types and user types; they are merely TM types. However, from a cooperative scenario perspective, data types and user types are not interchangeable and must be distinguished. CoCoA adds a user dimension to a TM specification. This is done is via the **data operations** component of a CoCoA specification, which lists TM methods, together with an additional user parameter when user information

[8] TM was discussed extensively in the two previous TransCoop deliverables for Work Package IV [EvFa94, EvFB95].

is required for the cooperation.[9] Scenario-specific data can also be added as additional parameters to the operations imported from a TM schema. We do not investigate this possibility here.

The grammar for a CoCoA specification is given in Figure 1; it includes all of the components described above. The syntax notation used in the figure is explained in Section 3.1. The unexpanded non-terminal symbols (those without rules) are defined in the later sections of the report. We choose to expand the non-terminal symbols of the grammar as each new feature is introduced, rather than overwhelm the reader with all features at an early stage. The complete grammar is given in Appendix A.

3.1 Syntax notation

To make the grammar descriptions more compact, an extended form of BNF is used. This syntax uses a string notation for all terminal symbols that would otherwise be defined separately, e.g., we use `"begin"`, instead of `begin_symbol`. All non-terminals that end with '`_name`' represent identifier terminal symbols, which are defined in the usual way.

The syntax notation uses a number of meta-symbols for often used constructions such as options, sequences, comma-separated lists, and chains. For optional parts, the meta-symbol `OPT` is used, such that rule '`a ::= b OPT`' is equivalent to '`a ::= b | `'. For a sequence of one or more language elements, the meta-symbol `SEQ` is used. This means that the rule '`a ::= b SEQ`' is equivalent to '`a ::= b | b a`'. For a comma separated list, the meta-symbol `LIST` is used, such that '`a ::= b LIST`' is equivalent to '`a ::= b | b "," a`'. A generalisation of the comma separated list is a list that is separated by another symbol or expression. The meta-symbol `CHAIN` is used for this, which is defined such that the rule '`a ::= b CHAIN c`' is equivalent to '`a ::= b | b c a`'. These meta-symbols can used at any place in a grammar rule in combination with round-brackets for grouping, if needed. As this cancels out the use of many auxiliary rules, a compact syntax notation is achieved.

4 Organisation of cooperative activities

The following sections explain how the organisation of the user activities of a scenario can be specified using *steps*. A step is a singled out piece of work within the scenario with a well-defined goal or purpose. Thus it is an organisational concept. The steps are specified in the part of the specification that starts with the keyword **procedure** (component 7 of the specification skeleton in Section 3). First, we show a more static organisation of scenario activities using the sequential and parallel ordering of steps. Next, we show a more dynamic organisation of scenario activities using step repetition and the parallel creation of steps.

[9] We opt to keep the examples in this report simple and omit typing information from these operations. Complete details will be worked out in Deliverables IV.4 and IV.5.


```
Cocoa_spec ::=
   "scenario" scenario_name
   ("includes"
       TM_module_name LIST)OPT
   ("data" "types"
       data_type_name LIST)OPT
   ("data" "operations"
       data_operation_signature LIST)OPT
   "workspace" "types"
       workspace_type_def SEQ
   "user" "types"
       user_type_def LIST
   "communications"
       communication_prim_def LIST
   "data" "exchange" "operations"
       data_exch_oper_def SEQ
   procedure_def
   "data" "operation" "order"
       order_rule_def SEQ
   "history" "rules"
       history_rule_def SEQ
   "end" scenario_name.

data_operation_signature ::=
   data_oper_name "(" data_type_name LIST ")".

workspace_type_def ::= workspace_type_name "="
                       "{" data_oper_name LIST "}".

user_type_def ::= user_type_name
                  ("isa" data_type_name)OPT
                  ("using" workspace_type_name)OPT.

communication_prim_def ::=
   comm_prim_name "(" data_type_name LIST ")".
```

Figure 1: Grammar of a specification skeleton.

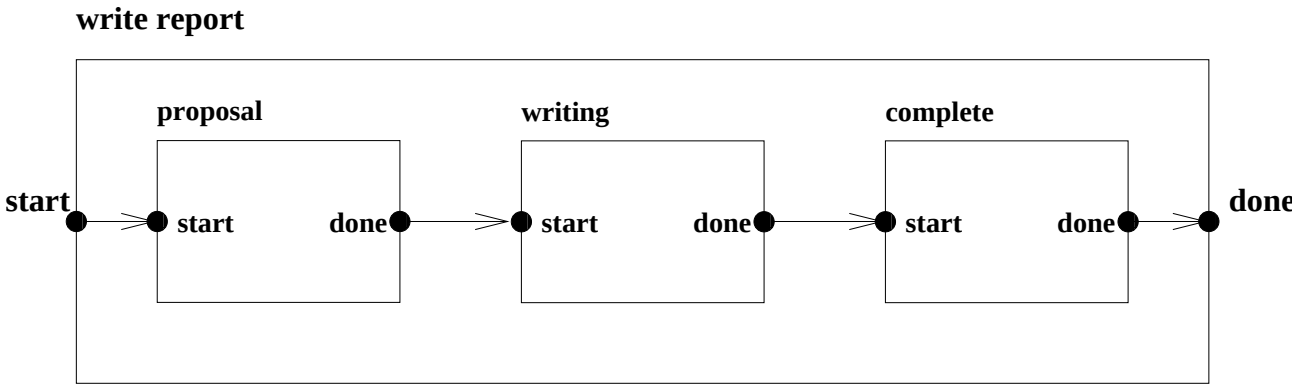


Figure 2: Graphical representation of the CDA scenario.

4.1 Sequential ordering of steps

The organisation of user activities discussed in Section 2.1 divided the CDA scenario work into three major steps. The graphical representation that we have in mind to specify this is shown in Figure 2. The surrounding box stands for the whole scenario; the three smaller boxes stand for the three steps of the scenario. (The reader should be aware that the steps organise the activities of the scenario, not the users or their workspaces. More than one user can be "working" in the same step.) The lines connecting the boxes are used for control flow. The filled circles on the left side of a box represent its entry points (called *in points*); those on the right side represent its exit points (called *out points*). Both kinds of point are called the step's *interaction points*. The four arrows in the figure that connect the points represent the four *transitions* between the steps. These concepts will be explained in the remainder of Section 4.

A textual representation of the figure is given in the following (top-level) scenario specification:

```
procedure write-report[in start out done]
begin

  step proposal[in start out done]
  begin ... end

  step writing[in start out done]
  begin ... end

  step complete[in start out done]
  begin ... end

  on start  do proposal.start
  on proposal.done  do writing.start
  on writing.done  do complete.start
```



```
  on complete.done  do done
end
```

In place of '. . .' in the above specification, we can (statically) either fill in a more detailed description of the step in terms of smaller (nested) steps, or we can specify which operations can be executed by whom while this step is in progress. For example, some operations might only be possible during the writing step. Section 5 discusses the specification of elementary steps. The mapping to LOTOS/TM will ensure that each sequential step in this example is only performed once, as depicted in the diagram. (See Section 8.)

The textual specification might appear verbose considering what it specifies, namely the successive execution of the three steps (proposal, writing, and completion). The importance of the details in the notation will become clear when we look at more complex examples in later sections.

Syntax The procedure component of the skeleton grammar in Figure 1 is enhanced with the following rules:

```
procedure_def ::= "procedure" point_list
                  "begin" step_stmt SEQ "end".
step_stmt ::= (step_def | trans_def ).
point_list ::= "[" "in" point_name
                   "out" point_name "]".
step_def ::= "step" step_name point_list
             "begin" step_stmt SEQ "end".
trans_def ::= "on" point "do" point.
point     ::= (step_name ".")OPT point_name.
```

Step activation A step is *activated* when one of the incoming transitions occurs; it stays active until one of the outgoing transitions occurs. We will call an active step an *instance* of the step. The same terminology will be used for the top-level scenario. Which of the scenario steps are active depends on what transitions have occurred. We assume, for the time being, that the transitions happen atomically. The above specification can thus be understood as a description of a state transition system, in which four different states are possible: none of the steps are active, only the first step is active, only the second step is active, and only the third step is active.[10] The state enabling notation will be mapped to LOTOS/TM, allowing properties of the transitions to be formally investigated. (See Section 8.)

Due to the nature of steps and elementary steps, deactivation of a step is analogous to revoking permissions for those operations that were enabled during the step. Elementary steps

[10]The state where none of the steps are active will not be considered a valid state of an instance of this scenario, since such a state is equivalent to a scenario instance that does not yet exist, or has ceased to exist. As a consequence, the first and the last of the four transitions indirectly imply creation and destruction, respectively, of the scenario instance.

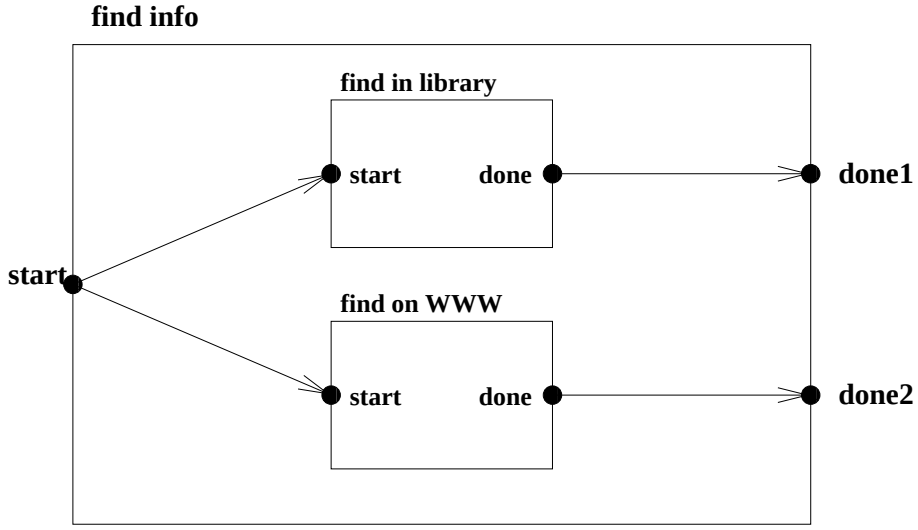


Figure 3: Graphical representation of a parallel information retrieval step.

are discussed in Section 5. It is possible for several (non-nested) steps to be active at the same time. Multiple activations are discussed in Section 4.4. Transitions that do not involve step activation or deactivation are possible. These are also discussed in Section 4.4. Incoming transitions that activate more than one step are discussed in Section 4.2. Outgoing transitions that "wait" for the completion of several active steps are discussed in Section 6.2.

4.2 Parallel steps

In addition to sequential ordering, steps can also occur in parallel. Because there is no simple example of parallel ordering in our CDA scenario, we give an information retrieval example to illustrate parallel steps. The graphical representation of this example is shown in Figure 3; its textual representation is given below:

```
step find_info[in start  out done1, done2]
begin

  step find_in_library[in start  out done]
  begin ... end

  step find_on_WWW[in start  out done]
  begin ... end

  on start do find_in_library.start
  on start do find_on_WWW.start
  on find_in_library.done do done1
  on find_on_WWW.done do done2
end
```


In the example, there are two (electronic) means of searching for information: via an on-line library catalogue, and via the WWW. When step `find_info` is activated, both search steps (`find_in_library` and `find_on_WWW`) are activated. As soon as one of these steps exits on its `done` interaction point, a transition to the corresponding **out** interaction point (i.e., `done1` or `done2`) of the `find_info` step occurs. The `find_info` step is then deactivated. This deactivation causes the other (unfinished) search step to be deactivated as well. The deactivation of a step does not result in the loss of work, it merely means that those operations that were enabled by the step are disabled.[11] (See Section 5.) Step deactivation is translated to the LOTOS disabling combinator '[>' by the mapping from CoCoA to LOTOS/TM. Thus, although there is no graphical link between the out transitions of the two parallel steps, deactivation information is communicated in the LOTOS/TM representation when one of the steps terminates. (See Section 8.)

The completion of one search step need not cause the `find_info` step to terminate, as it does in this example. More complex forms of transition are discussed in Section 6.

This example illustrates that steps are not merely a syntactical notion; "execution scopes" are activated and deactivated. The data, data exchange, and communication operations that can be performed within a step are only possible while the step is active. Steps can be used to grant and revoke permissions for users to do certain operations. Steps offer a means to manage the cooperation in a scenario. This idea is explored in Section 5.

4.3 Step repetition

Steps of a scenario can be repeated. For example, the proposal step in our example could contain an *acceptance cycle*, in which the editor's proposal is accepted by the management. Of course, it might be possible for the proposal to be rejected. Figure 4 shows how such an acceptance cycle can be specified in a graphical manner. Note the backward transition from the point 'rev' on the right side of the second box to the point 'revise' on the left side of the first box.

We can specify the acceptance cycle in textual form as follows:

```
step proposal[in start  out done, rejected]
begin

  step write_proposal[in start, revise  out done]
  begin ... end

  step accept_proposal[in start  out acc, rev, rej]
  begin ... end
```

[11] The work can become inaccessible if the remainder of the scenario specification does not enable imports from the workspaces that were modified during the step. A tool for checking for the potential inaccessibility of work might be desirable.

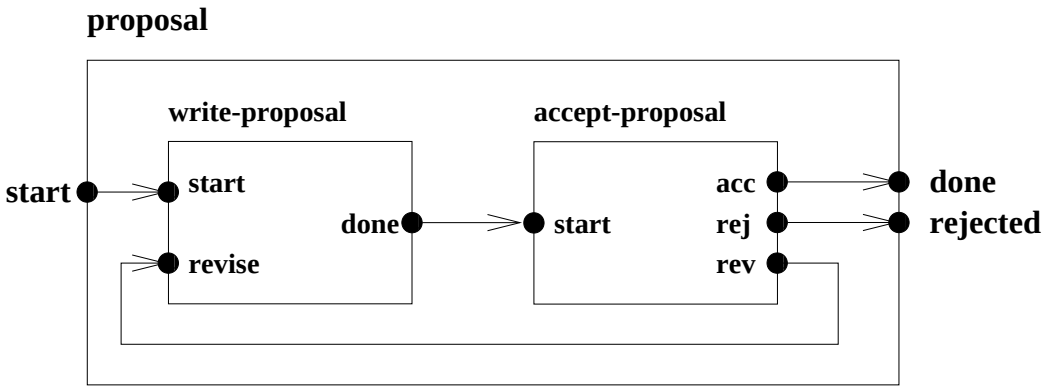


Figure 4: Graphical representation of an acceptance cycle.

```
    on start do write_proposal.start
    on write_proposal.done do accept_proposal.start
    on accept_proposal.acc do done
    on accept_proposal.rev do write_proposal.revise
    on accept_proposal.rej do rejected
end
```

This specification extends the proposal step definition given previously, and it adds an extra possibility for exiting.[12]

The above example illustrates that a step can have more than one entry, and more than one exit point. Whether a step has one or more entry or exit points is a matter of design. For example, we could modify the above specification by removing the `revise` point, and connecting the transition from the `rev` point to the `start` point (assuming that this does not conflict with what is inside the write proposal step). However, when values are added to transitions in Sections 4.4.2 and 5, we will see that interaction points can be typed. Thus, it may not be possible to connect two transitions to the same interaction point, because of different parameter types.

Step repetition is not the same as having multiple activations of a step (discussed in the next section). With multiple activations, the "earlier" step activations do not need to be deactivated before another activation can take place, whereas with repetition, a step is deactivated and then activated again. Parameters to a repeated step are determined by the transition that activates it. (See page 26.)

Syntax The grammar rule for `point_list` is enhanced as follows to accommodate multiple in and out points:

[12] We remark that because of the `rejected` point, this proposal box no longer fits inside the top-level scenario box given in Figure 2.


```
point_list ::= "[" "in" point_name LIST
                   "out" point_name LIST "]".
```

To be correctly formed, each step should have at least one entry point (otherwise it can never be activated), and at least one exit point (otherwise it can never be deactivated).

4.4 Multiple activations

In our CDA example, it should be possible for several chapters to be written at the same time. At the start of the scenario, however, it is not known *a priori* how many chapters the document will contain. Thus, the mechanism introduced in Section 4.2 for parallel steps does not suffice. A general mechanism is introduced in this section. Additional constructs are proposed for signals and interrupts, and the identification of multiply active steps.

To specify that more than one instance of a step can be active, the code will be enclosed by the keywords **parallel** and **endpar**. For example, parallel activations of the `task` step can be specified within the `writing` step as follows:

```
step writing[in start  out done]
  ...
  parallel
    step task[in start, out compl]
    begin ... end
  endpar
  ...
```

Several steps can be placed within a parallel clause definition. Whenever there is a transition to a step that is within a parallel clause, a new instantiation of the parallel clause becomes active; there are new activations of the transitioned-to steps. Before we continue with our CDA example, we introduce two new kinds of interaction points for steps.

4.4.1 Interaction points

In the CDA scenario, the editor can initiate several writing tasks in parallel. The editor starts up each writing task by means of a transition. This transition should not terminate the editor's own activity. To accomplish this, we introduce a new kind of interaction point for outgoing transitions that do not deactivate the step that caused the transition. We call such an interaction point a *signal point*. Furthermore, because the editor should be informed whenever a writing task has been completed, it should be possible to make an incoming transition to a step that is already active (i.e., the transition does not activate the transitioned-to step). We call such an interaction point an *interrupt point*. There are now four kinds of interaction points for steps in CoCoA:

1. **in points** indicate incoming transitions that activate a step,
2. **out points** indicate outgoing transitions that deactivate a step,
3. **signal points** indicate outgoing transitions that do not deactivate the step, and
4. **interrupt points** indicate incoming transitions to an already active step.

If we classify transitions by their starting and ending interaction points, we come up with four different kinds of transition, and four different kinds of arrow in the graphical representation:

1. *relay transitions* connect **out points** to **in points** (this represents a transfer of control),
2. *interrupt transitions* connect **out points** to **interrupt points**,
3. *signal transitions* connect **signal points** to **in points**, and
4. *message transitions* connect **signal points** to **interrupt points** (neither step is deactivated).

We remark that it is also possible to connect **in points** to **in points** and **out points** to **out points** to specify either the simultaneous activation or deactivation of a step. In previous sections, we have referred to these transitions as *in transitions* and *out transitions*, respectively. The other potential connections are not possible. For example, it is not possible to form a transition that *starts* at an **interrupt point** or *ends* at a **signal point**, due to the directions of these kinds of point. Nor is it possible to connect an **in point** to an **interrupt point**, or a **signal point** to an **out point**, due to the activation rules for steps.

Returning to the CDA example, the graphical representation of the writing step is given in Figure 5. Signal and interrupt points are placed at the top or bottom side of the boxes that represent the steps. A dashed box is used to indicate a parallel component. The textual representation of this figure is given below:

```
step writing[in start  out done]

  step editors_work[in start  out done,
                    signal start_task,
                    interrupt task_compl]
  begin ... end

  parallel

    step task[in start, out compl]
    begin ... end

  endpar
```

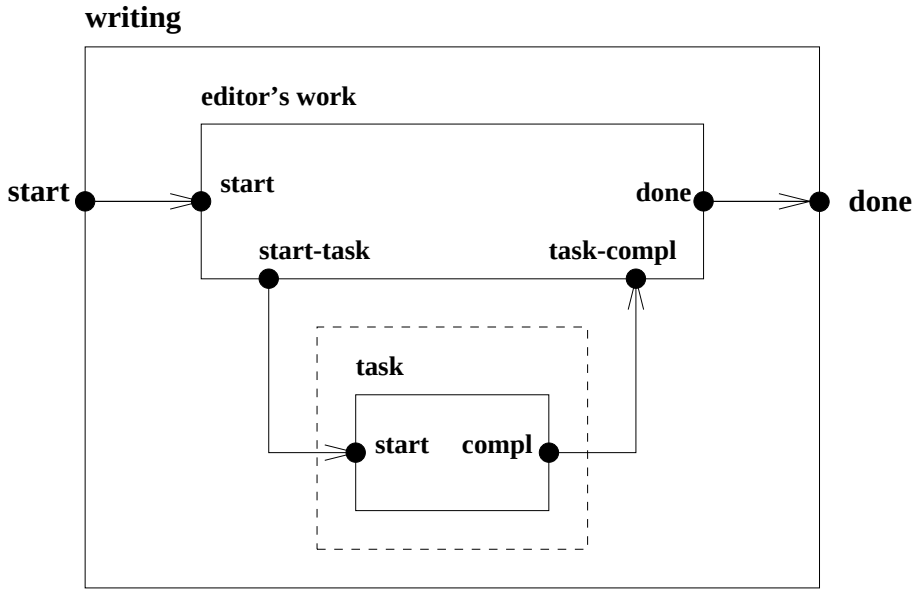



Figure 5: Graphical representation of the writing step.

```
    on start do editors_work.start
    on editors_work.start_task do task.start
    on task.compl do editors_work.task_compl
    on editors_work.done do done
end
```

Here, we see examples of two new kinds of transition: a signal transition, which starts up a new writing task:

```
on editors_work.start_task do task.start
```

and an interrupt transition, which informs the editor that a writing task has been completed:

```
on task.compl do editors_work.task_compl
```

Neither transition deactivates the `editors_work` step. Observe that there is currently no way to distinguish the parallel writing tasks from one another. This problem is remedied in the next section.

4.4.2 Identification of activations

The specification given above allows the parallel execution of multiple instances of the `task` step, but there is no way to discriminate between the different instances. If identification is

desired, we can attach values to each instance of the parallel clause. To do this, we allow a non-empty TM type list to be specified for identification purposes. If a parallel clause has an explicit identification, the types will be given after the **parallel** keyword, inside of parentheses.[13]

Any reference to a step activation that is inside of a parallel clause (from outside of the clause) will consist of the name of the step, followed by identifying values (of the appropriate TM types) in parentheses. This illustrates the need for value passing with the transitions. To add value passing, types can be optionally specified after the interaction points; the types are given in a comma-separated list between parentheses. The syntax for step and transition identification is given at the end of this section. First, we look at an example.

The specification given earlier is repeated below, with an identification of parallel writing tasks by the chapter:

```
step writing[in start  out done]

  step editors_work[in start  out done,
                    signal start_task(chapter, ℙ author)
                    interrupt task_compl(chapter)]
  begin ... end

  parallel(ch : chapter)

    step task[in start(ℙ author)  out compl]
    begin ... end

  endpar

  on start do editors_work.start
  on editors_work.start_task(ch, as) do task(ch).start(as)
  on task(ch).compl do editors_work.task_compl(ch)
  on editors_work.done do done
end
```

Let's look at the value passing annotations on the transitions. As required by the parallel clause identification, all transitions mentioning the `task` step are parameterised by the chapter. This can be seen in the second and third transitions. Consider the second transition:

```
on editors_work.start_task(ch, as) do task(ch).start(as)
```

The `start_task` signal point of the `editors_work` step requires values for the chapter and the set of authors; the chapter value is used as an identification for the `task` step, and the list of authors is passed as a parameter to the `start` point of the `task` step. The third transition also uses values for identification:

[13] We put the types here, because there can be more than one step within a parallel clause; *each step* within the clause is identified by values of these types.


```
    on task(ch).compl do editors_work.task_compl(ch)
```

Here, the chapter value is used to identify the parallel task step, and to identify the interrupt transition made on its completion.

4.4.3 Syntax

Because the grammar changes for adding values involve many parts of the grammar, we give the complete, adapted grammar for steps below:

```
procedure_def ::= "procedure" point_list
                   "begin" step_stmt SEQ "end".
step_stmt ::= (step_def | trans_def | par_clause).
point_list ::= "[" "in" point_def LIST
                    "out" point_def LIST
                  ( "signal" point_def LIST) OPT
                  ( "interrupt" point_def LIST) OPT "]".
point_def ::= point_name
               ("(" tm_type_name SEQ ")") OPT.
step_def ::= "step" step_name point_list
              "begin" step_stmt SEQ "end".
trans_def ::= "on" point "do" point.
par_clause ::= "parallel" ("(" tm_type_name LIST ")") OPT
                 step_stmt SEQ
                "endpar".
point ::= (step_name "." ("(" par_key_var_name LIST ")") OPT) OPT
          point_name ("(" point_var_name LIST ")") OPT.
```

5 Elementary steps

Elementary steps are those steps that are not divided into smaller steps. An elementary step specifies a set of data, data exchange, and communication operations, and identifies the actors who can execute these operations while the step is active. Before we reveal how the set of operations is specified, we first show how actors and workspaces are identified for a step.

5.1 Actor identification

Actors can be identified statically, when the scenario is instantiated, or they can be identified dynamically, during the execution of the scenario. When an actor is identified dynamically, an actor identification is passed as part of the transitions, using value passing. This was shown earlier, in the example in Section 4.4.2.

Static actor identifications are specified as parameters to a scenario. For example, to specify actors statically in the CDA scenario, the **procedure** heading syntax is changed to allow:

```
procedure (ref : referee,
           ed : editor,
           authors : ℙ author)[in start  out done]
```

Whenever `ed` is used in the specification of a step or operation, it indicates the actor who was assigned to be the editor of the document when the scenario was instantiated. The set of possible authors who the editor can choose to assign the writing task to is given by `authors`, a variable declared to have TM type ℙ `author`. We point out that when the referee, the editor, and the authors are specified statically in this way, it is not possible to assign unidentified actors to these roles during the scenario's execution.

The syntax of `procedure_def` is changed as follows to include the static identification of actors:

```
procedure_def ::= "procedure" "(" user_list ")" point_list
                    "begin" step_stmt SEQ "end".
user_list ::= ( user_name ":" user_type_name
              | user_set ":" P user_type_name) LIST
```

5.2 Workspace identification

Our model makes use of a private workspace for each user that is involved in the scenario, and a single shared data repository, which we call a shared workspace. All data operations performed by an actor are executed in the actor's private workspace. Actors do not execute data operations in the shared workspace; an actor may only export or import data operations to or from the shared workspace.

Because a shared workspace is defined for the whole scenario, it is specified at the beginning of the procedure component of the scenario. Because our CDA example uses a shared workspace to store the document, we need to add a workspace declaration to our specification:

```
procedure (ref : referee,
           ed : editor,
           authors : ℙ author)[in start out done]
begin
  workspace  document : cda
  ....
end
```

Here, a shared workspace named `document` is declared with workspace type `cda`.

The CoCoA grammar is extended as follows to include shared workspace declaration:

```
procedure_def ::=
   "procedure" "(" user_list ")" point_list
   "begin" workspace_def
           step_stmt SEQ "end".
workspace_def ::=
   "workspace" shared_workspace_name ":" workspace_type_name.
```

To refer to a workspace elsewhere in the specification, the name of the shared workspace is used, or the name of an actor is used to indicate the actor's private workspace. Private workspaces are implicitly declared in the **user types** component. (See Page 15.)

5.3 Operation enabling

An elementary step identifies which data operations, communications, and data exchange operations can be issued by which actors while the step is active. When a user is identified as an actor in several active steps, the user can issue any of the operations that are enabled in these steps. Each incoming transition to an interaction point defined for a step may enable different operations. For each transition, we need to specify which operations it enables. The values that are passed with a transition may be used to further identify the operations.

We illustrate operation enabling with the CDA example. To specify which operations can be executed by the editor during the `write_proposal` step, the following statements are used:

```
step write_proposal[in start, revise  out done]
begin
  on start enable
    /* The data operations the editor can do: */
    ed : addChapter(ed, "title", _),
         addChapter(ed, "intro", _)
  endon
  on start, revise enable
    ed : editChapter(ed, "title", _),
         editChapter(ed, "intro", _),
    /* Export operations the editor can do: */
         export Chapter("title") to document,
         export Chapter("intro") to document
    /* Other actions the editor can do, e.g., leave this step: */
    when ed issues introWritten() do done
  endon
end
```


A transition to the `start` interaction point enables four data operations that the editor (identified by `ed`) can perform in her private workspace: `addChapter` (for the title and the introduction), and `editChapter` (also for the title and the introduction). A transition to the `revise` interaction point enables the two editing operations only. The symbol '_' is used to indicate that no restrictions are placed on the value of a parameter in that position. Thus, for these data operations, any text value may be given as a parameter.

Two data exchange operations are enabled on transitions to the `start` and `revise` interaction points. (The comma separating the two point names indicates that a transition to *either* of the two points enables the same operations.) Using these operations, the editor can export the contents of her private workspace to the shared workspace (called '`document`'). Further enabling is done for communication operations. In the example, the editor can use the `introWritten()` communication operation to exit the step. This initiates the outgoing transition marked `done`, which then deactivates the step. Observe that no ordering constraints are imposed by the enable construct on the operations that are enabled. Ordering constraints are specified by execution rules, which are discussed in Section 7.1.

Now let's look at operation enabling for authors in the writing task:

```
step task[in start(ℙ author)  out compl]
begin
  on start(as) enable
    for a in as allow
      a : addChapter(a, ch, _),
          editChapter(a, ch, _),
          addAnnotation(a, ch, _),
          remAnnotation(a, ch, _),
          import Chapter(ch) from document,
          export Chapter(ch) to document,
          when a issues completeTask(ch) do compl
    for a1, a2 in as allow
      a1 : import Chapter(ch) from a2,
           import Annotations(ch) from a2
  endon
end
```

Transitions to the interaction point `start` must provide a value of type '`ℙ author`' (a set of authors) as a parameter. This value will be bound to variable '`as`' of the '**on** `start(as)` **enable** . . .' clause when such a transition takes place. A `start` transition enables a number of operations for the set of authors given as a parameter. As before, the '_' symbol is used for text and annotation parameters when their value is not restricted. Authors can import and export chapter text to and from the shared workspace (the `document`). Furthermore, each author writing a chapter may import chapter text and annotations from other authors' private workspaces.

Syntax The syntax for the constructions used in the above examples is given by the following grammar:


```
step_stmt ::= (step_def | trans_def | par_clause | enable_expr).
enable_expr ::=
    "on" point_with_params "enable"
    ( for_part OPT
      ( (user_name | for_param_name)
        ":" ( en_data_operation
            | en_data_exch_oper ) LIST)
      | en_comm_oper ) SEQ
    ) SEQ.
    "endon"
for_part ::= "for" (for_param_name LIST "in" TM_expr) LIST
             ("where" TM_expr)OPT  "allow".
en_data_operation ::=
   data_oper_name "(" (TM_expr | "_") LIST ")".
en_data_exch_oper ::=
    "import" data_exch_oper_expr "from" workspace_expr |
    "export" data_exch_oper_expr "to" workspace_expr.
data_exch_oper_expr ::=
    data_exch_oper_name "(" TM_expr | "_") LIST ")".
workspace_expr ::=
   user_name | for_param_name | shared_workspace_name | "_".
en_comm_oper ::=
   "when" (user_name | for_param_name)
   "issues" comm_prim_name "(" (TM_expr | "_") LIST ")"
   ("iff" TM_expr) OPT
   "do" point.
```

An enable expression can sometimes be used instead of a step definition. To illustrate this, the writing step of our CDA specification can be rewritten to eliminate the `editors_work` step. Instead, an enable expression and communications are used in the body of the writing step's specification. Eliminating the `editors_work` step leads to the following specification:

```
step writing[in start  out done]

  parallel(ch : chapter)

    step task[in start(ℙ author)  out compl]
    begin
      ...
    end

  endpar

  on start enable
     ed : import Chapter(_) from _,
          export Chapter(_) to document,
```

```
            ...
        when ed issues completeWriting() do done
        when ed issues startTask(ch, as)
             iff as subset authors do task(ch).start(as)
    endon
end
```

In the above, **subset** is a TM operation on sets.

6 Some advanced features

6.1 Disabling

A step can consist of several substeps, and more than one of these substeps can be active at the same time. When a step is completed (deactivated), it seems logical to assume that none of its substeps remain active.[14] This means that when a step is deactivated by an **out point** transition, all of its active substeps are *disabled* (i.e., deactivated). Disabling a substep also means disabling all of its active substeps, and so on. Operation results are not lost when a step is disabled; disabling a step only means that any operations that were enabled by this step are no longer enabled (by this step).

6.2 Complex transitions

It is possible to define transitions that start at the same interaction point, or end at the same interaction point. In Section 4.2, we saw an example of parallel step activations. This was accomplished by specifying two transitions that start at the same interaction point. In this section, we extend the CoCoA syntax to allow the specification of multiple starting and ending points in a single transition rule. We explain complex transitions in terms of an example.[15] Consider the following specification:

```
step split_and_join_or(in start  out done)
begin
   step a(in start  out done)
   begin .... end
   step b(in start  out done)
   begin .... end
   on start do a.start
```

[14] Note: This is a design decision.

[15] Please note: At the time of writing this document, we do not have a graphical representation for complex transitions.


```
    on start do b.start
    on a.done do done
    on b.done do done
end
```

When two or more transitions start at the same interaction point (like the first two transitions shown above), both transitions take place. To make this situation easier to specify, a list of interaction points can be given after the **do** keyword. For example, the following transition is equivalent to the above pair of transitions on the `start` point:

```
    on start do a.start, b.start
```

The last two transitions in the example are independent. If step '`a`' exits on point '`a.done`', then step '`split_and_join_or`' will exit on point '`done`', and as a side effect, step '`b`' will be deactivated. The same thing happens when step '`b`' exits on '`b.done`'. The two transitions can be grouped together to form a *single* complex **or** transition as follows:

```
    on a.done or b.done do done
```

This transition has the same semantics as the pair of transitions above.

In the above example, steps '`a`' and '`b`' are not synchronised on their completion. To specify that the two substeps should *both* complete, we introduce an **and** transition:

```
    on a.done and b.done do done
```

An **and** transition can be used instead of the **or** transition in the above example:

```
step split_and_join_and(in start  out done)
begin
    step a(in start  out done)
    begin .... end
    step b(in start  out done)
    begin .... end
    on start do a.start, b.start
    on a.done and b.done do done
end
```

Although this seems like a natural construct, its semantics is not obvious. Earlier, we claimed that transitions are atomic. However, now steps `a` and `b` must *both* exit via outgoing transitions on `a.done` and `b.done` to enable this transition. Therefore, this transition cannot be atomic, unless `a` and `b` are made to synchronise when they exit. We see that complex transitions might hide complex LOTOS/TM protocols underneath. The details of mapping complex transitions to LOTOS/TM will be worked out in Deliverables IV.4 and IV.5.

6.3 Negotiation protocols

The *communications* that where explained in Section 2.1 serve the purpose of letting the users of a scenario influence the flow of control. They are always issued by a single user, and can cause several steps to be started or terminated, and thus influence the operations that the users of the scenario can perform.

COCOA does not specify who is informed of the changes in the organisational state of the running scenario. This would be necessary if one would like to specify both directions of the communication between the users and the running scenario. This limitation can be overcome by adding an (optional) inform-clause to transitions, which indicates who is to be informed about the occurrence of the transition.

By adding a notification mechanism, it becomes possible to specify complicated interaction between the users of the scenario. The drawback, however, is that this might involve the usage of many small steps.

Decisions within a scenario are usually taken by a single person or a group of persons with equal rights. In case a decision is made by a group with equal rights, this is usually done by means of voting. A special case of voting is where everybody has to agree.

The syntax of the when-part of communications can be extended such that the above kind of protocol can be specified more easily. Instead of mentioning just a single user, as it is now, we allow an expression specifying who has to agree. The logical operators '**and**', '**or**', '**not**' and '**implies**', in combination with brackets can be used to combine references to individual users, and sets of users grouped in voting expressions. The voting expressions are of the form '**all** `(...)`', and '**vote** `(...)`', where between the brackets individual users or sets of users are enumerated. The **all**-voting expression requires all mentioned users to agree. The **vote**-voting expression requires only half of the users to agree. The grammar thus becomes:

```
en_comm_oper ::=
   "when" user_expr
   "issues" comm_prim_name "(" (TM_expr | "_") LIST ")"
   ("iff" TM_expr) OPT
   "do" point.
user_expr ::=
     (user_name | for_param_name)
   | "all" "(" (user_name | for_param_name) LIST ")"
   | "vote" "(" (user_name | for_param_name) LIST ")"
   | "not" user_expr
   | user_expr "and" user_expr
   | user_expr "implies" user_expr
   | user_expr "or" user_expr.
```

The semantics of this extended syntax is that the users can issue the communication primitive mentioned in the rule, and will get a message if the requirements for the primitive to be

executed are not met yet. In case the rule contains an **iff**-clause, this clause is checked once a required group of users has issued the communication primitive. If the expression following the **iff**-clause is not met, all users who issued the communication primitive are informed of this, and required to issue the primitive again in order to have it happen. In the expression is met (or if there is no **iff**-clause), the transition will occur.

7 The transactional view

Sections 4, 5, and 6 of this report introduced CoCoA features to support the organisational view of a scenario specification. In this section, we introduce features to support the transactional view of a cooperative scenario. There are three important aspects in the transactional view of a scenario: *execution rules*, *data exchange rules*, and *history rules*. All three types of rule are used to ensure that data consistency is preserved in the workspaces. Execution rules are important for workspaces viewed in isolation (both private workspaces and the shared workspace), although the operations in a workspace history may have originally been done by different actors. (This is the case when work is imported or exported.) Data exchange rules deal with the exchange of partial results between the workspaces. The rules allow the specification of a **select** statement over a workspace history, which is used to describe the imported or exported work in terms of data operations that can be re-executed in the destination workspace. History rules concern state-independent relations over data operations, such as commutativity. The history rules are used by the transaction model to determine whether the operations done in one workspace are independent of the operations done in another workspace. Independent operations can be imported or exported without problems. The history rules are used in combination with the execution rules by the transaction manager to determine whether an ordering of operations in one history is equivalent to a reordering of the operations in another history.

Execution rules consist of 1) rules which govern the ordering of the data operations that can be executed in a workspace, and 2) rules which describe when certain termination or breakpoint requirements are met. The ordering rules are checked whenever operations are executed in a workspace. The termination requirements are checked whenever there is an outside reason for it. At other points during the execution of a scenario, there can be organisational reasons why certain breakpoint requirements are enforced. For example, in the CDA scenario, we might want to specify that the editor should only be allowed to complete the write proposal step after the title page and the introduction are exported to the shared workspace. Ordering rules and termination constraints such as this are explored in Section 7.1.

To describe the exchange of data between workspaces, it is necessary to identify which data is to be exchanged, and how the data is to be integrated with the destination workspace. The integration is based on a merging algorithm (part of the cooperative transaction model) that merges the data operations from two histories into a single consistent history. For an effective merge algorithm, information about the independence of the data operations is needed. CoCoA provides language constructs that allow this information to be supplied to the transaction model. Data exchange protocols are discussed in Section 7.2.

7.1 Execution rules

Several approaches can be taken to define ordering constraints on the data operations in a workspace history. First, we mention LOTOS/TM as a means to specify ordering constraints. LOTOS/TM allows the formal specification of the temporal ordering of events. Although LOTOS/TM is very powerful, we believe it is too low-level for the description of cooperative scenarios. (See [EvFB95].) The intention of COCOA is to provide something that is easier to use than LOTOS/TM, but which is still easy to map to LOTOS/TM.

Another approach that can be taken is to use grammars to describe the ordering of all possible sequences of operation. (See the discussion of transaction groups in Section 9.3.1.) However, if a single grammar is used to describe all ordering constraints, the grammar soon becomes large and difficult to handle. For this reason, we specify a number of ordering rules that place constraints on the occurrences of the operations they contain. The ordering rules are enforced in combination. Operations not mentioned in the rules can occur freely. Our notation is based on regular grammar operators [AhUl77]. The notation is enhanced with a mechanism for defining a parameterised collection of rules.

7.1.1 Basic rules

For the examples in this section, we assume that we have data operations `a`, `b`, `c`, and `d`. For the moment, we ignore operation parameters. If no ordering rules are given, any sequence of operation instances (with appropriately typed parameters) is allowed. Thus, `a`'s, `b`'s, `c`'s, and `d`'s can appear freely intermixed. When there are no ordering rules, each operation may be repeated any number of times (with the same, or with different parameter values).

Each ordering rule enforces restrictions on the possible occurrences of operations in the sequences, and on the values of their parameters. The ';'-symbol is used for specifying a sequential order. What follows the ';'-symbol in a rule can only happen after the operations that precede the symbol have happened. We explain the sequencing operator with the following rule:

order `a; b`

To check this rule for a given sequence of operations, we look at its maximal subsequence that contains operations `a` and `b` only, and check whether this subsequence is a prefix of the sequence '`a; b`'. (A more precise definition follows in Section 7.1.2.) The following table shows a number of example sequences:

sequence	{a,b} subsequence	comments
		okay
a	a	okay
b	b	b not allowed
a; b	a; b	okay
b; a	b; a	sequence not possible
a; a	a; a	second a not allowed
c; d; a; c; b	a; b	okay
c; b; d; c; a	b; a	sequence not possible

Two of the above sequences are not possible, because operation `b` is not allowed until operation `a` has been executed; these sequences are included for illustration purposes only.

The following table shows which operations can be added to a history after the given sequence of operations has already been done:

sequence	allowed operations
	a c d
a	b c d
c	a c d
a; b	c d
a; c; d	b c d

Here, we see that `c` and `d` can be done at any time, whereas `b` can only be done after `a` has been done.

Figure 6 shows Deterministic Finite Automaton (DFA) representations of the ordering rule examples given in this section [AhUl77]. Observe that there are no "accept" states in the DFA representations; termination and breakpoint constraints for execution rules are discussed in Section 7.1.5.

The '*' and '+' symbols are postfix operators used for repetition: '*' stands for zero or more occurrences of the ordering expression it annotates, whereas "+" stands for one or more occurrences. Both symbols have a higher priority than the ';' symbol. (Parentheses can be used to override precedences when necessary.) We explain these operations with the following rule:

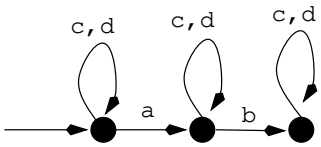
order `a+; b*`

Intuitively, the rule states that no '`a`' can be done after a '`b`' has been done, and that at least one '`a`' must be done before a '`b`' can be done. There are no restrictions on `c`'s and `d`'s. The following table shows which operations are allowed after a given sequence of operations:

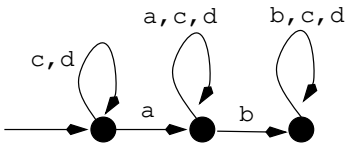
sequence	allowed operations
	a c d
a	a b c d
c	a c d
a; b	b c d
a; c; d	a b c d

The '|' symbol is used for specifying alternatives. An empty expression may be used to make the other expression optional. The '|' symbol has a lower-priority than the '*' and '+' symbols. The following rule illustrates this:

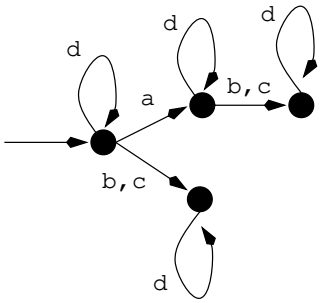
(a) **order** `a;b`



(b) **order** `a+ ; b*`



(c) **order** `(a | );(b | c)`



(d) **order** `a;{b,c};d`

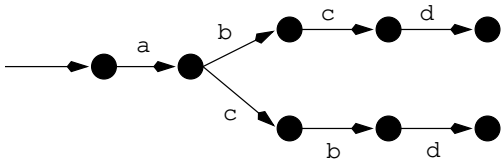


Figure 6: DFA representations of the CoCoA ordering rules. The DFA representations are based on a finite alphabet of operations: `a`, `b`, `c`, and `d`.

order `(a | ); (b | c)`

The following table shows which operations are allowed after a given sequence:

sequence	allowed operations
	a b c d
a	b c d
b	d
d	a b c d
a; b	d
a; c; d	d

We require that the ordering rules be deterministic.[16] The reason for this requirement is that we want to have a simple method for testing the rules when an operation is invoked.

As a notational convenience, we use curly brackets to group a set of operations that can occur in any order, but only once. This is illustrated by the following rule:

order `a; {b, c}; d`

The following table shows which operations are allowed after a given sequence according to this rule:

sequence	allowed operations
	a
a	b c
a; b	c
a; c	b
a; b; c	d
a; c; b	d

7.1.2 Semantics of ordering rules

In the basic case, the semantics of ordering rules can be understood as a filtering technique. If we let α be the set of allowed operation names (the alphabet), then α^* is the set of all finite strings of elements of α. A specific element of α^* is the empty string Λ. Obviously, an element of α^* is interpreted as a sequence of operations.

Each ordering rule will serve as a filter on α^* in the following way. Consider a rule R_i described through syntactic constructs introduced in the previous section. By $ops(R_i)$ we mean

[16] For example, the following non-deterministic rule specification is not allowed: **order** `(a;b) | (a;c)`. This does not pose restrictions on the expressibility of the rules, because any non-deterministic expression in the rules we allow can always be rewritten into a deterministic rule, using common compiler techniques [AhUl77].

the set of operation names, thus elements of α, that occur in R_i. The rule R_i obviously defines a regular language, let us denote it by $L(R_i)$, consisting of operation name sequences. These sequences are intended as the allowed sequences of operations *within* the smaller alphabet $ops(R_i)$. In other words, if we look at an actual sequence of operations σ, and we take its maximal subsequence of operations in $ops(R_i)$, then this sequence should be allowed by $L(R_i)$.

We state here 'should be allowed by $L(R_i)$' instead of 'is an element of $L(R_i)$' because the sequence σ may not be complete yet, and it is only its completion that should be an element of $L(R_i)$. For this reason, we checked whether a subsequence of operations was a prefix of a sequence allowed by an ordering rule in the previous section. The issue of completion is another matter, by the way, and we will not require that all relevant ordering rules have been completed by a scenario. Termination is dealt with in a later section.

As a consequence, we need two further notions: *maximal subsequence* and *is allowed by*. Without going into precise, formal definitions, we provide a sketch of a formalisation. Suppose σ is a sequence of operation names, and O is a subset of α, then by '$\sigma \upharpoonright O$' we mean the sequence that contains just those elements of σ which are members of O, in the same order as they appear in σ.

Also, a sequence of operation names σ is allowed by a language L if σ coincides for its full length with an element τ in L. The sequence τ may be longer than σ, and we say that σ is a *prefix* of τ.

We are now in the position to define what restrictions the rule R_i imposes. Define

$$\mathcal{U}_i = \{\, \sigma \in \alpha^* \mid \sigma \upharpoonright ops(R_i) \textit{ is allowed by } L(R_i) \,\}.$$

Thus, we allow those operation sequences that are well-behaved up to this point in time with respect to the operations mentioned in the rule. What remains is to define how several ordering rules combine their semantics. Let us assume we have n such rules R_i. The set $\mathcal{U}$ is the set of prefixes allowed by all rules:

$$\mathcal{U} = \bigcap_{i=1}^{n} \mathcal{U}_i.$$

From this definition, we can observe that the semantics of every ordering rule is enforced at the same time.

7.1.3 Ordering rules with parameters

Up to here, we have ignored the fact that operations may have parameters. First of all, we allow values to be used in the ordering rules. The following rule specifies which operations the editor can execute for editing the introduction:[17]

[17] For simplicity, we use strings as value parameters in this example. We remark, however, that typical values in these circumstances will be identifiers—typically object identities.


```
order addChapter(ed, "intro", _);
      editChapter(ed, "intro", _)*
```

This rule specifies that the editor first has to create an introduction, before she can edit it (arbitrarily often). The rule also implicitly states that the introduction can be created only once by the editor. The "_"-symbol is used to represent parameters that can have an arbitrary value. The global `ed` parameter was already in scope, and we remark that this is an example of the first of two kinds of variables that we allow in the rules.

It should be carefully understood what the addition of parameters gives us. The above rule, for instance, limits the scope of checked operations to the two mentioned operations, and to *only* those operation invocations that concern the editor and the introduction chapter together. The rule states nothing about other authors working on the introduction, nor about the editor working on other chapters. Thus, the more parameters are mentioned, the more specific the rule becomes.

To check the above rule against a given sequence of operations, we will consider the maximal subsequence of operation invocation patterns that fit one of the patterns in the rule. This fit concerns both operation name and parameter list.

As we sometimes would like to have a rule like the above for each chapter, such that the chapter can be edited by any (arbitrary) actor, we introduce the following **forall** construct.

(1)
```
forall c : chapter
order addChapter(_,c,_); editChapter(_,c,_)*; delChapter(_,c)
```

This rule states that a chapter `c` should first be added before it can be edited and finally deleted. These actions can be performed by different actors. The **forall** construct defines a collection of rules, one for each type-correct value of the quantified variable. A universally quantified variable is the second kind of variable that we allow in the rules.

To illustrate the intention of the **forall** construct in the above rule, consider the sequence:

```
addChapter("a1", "c1", "t1"); editChapter("a1", "c3", "t2");
editChapter("a2", "c1", "t3"); editChapter("a2" "c2", "t4")
```

The essence is that there are three versions of the given ordering rule that are important here, namely one for each chapter. This sequence obeys the rule for the `c1` chapter, but not for the `c2` or `c3` chapter, and thus this sequence of operations is not allowed. In fact, the second operation invocation would already have been flagged down as it violates the rule for chapter `c3`.

We allow the use of several universally quantified variables in a single ordering rule, but such usage is not trivial, and can be difficult to understand at times. Whenever more than a single such variable is used, the specifier should carefully check the semantics of the rule. As an illustrative example of semantic subtlety, consider the following rule:

```
forall c : chapter, an : annotation
order addChapter(_,c,_);
      (addAnnotation(_,c,an); delAnnotation(_,c,an))*;
      delChapter(_,c)
```

This rule states that arbitrary annotations can be added to and deleted from a chapter, after the chapter is added and before it is deleted. The rule allows several annotations to be added to a chapter, because each annotation is covered by a separate rule instantiation, and all rule instantiations are enforced together. The rule requires that any annotation that is added to a chapter must be deleted before the chapter itself can be deleted. To drop this restriction, the following *two* rules can be used instead:

(2)
```
forall c : chapter
order addChapter(_,c,_);
      (addAnnotation(_,c,_) | delAnnotation(_,c,_))*;
      delChapter(_,c)
```

(3)
```
forall c : chapter, an : annotation
order addAnnotation(_,c,an); delAnnotation(_,c,an)
```

Each of these rules makes different restrictions on the allowed orderings. The first rule captures the fact that addition and deletion of annotations may only happen during the existence of a chapter. The second rule imposes restrictions on the ordering of annotation operations: an annotation must have been added before it can be deleted. This rule is specific to the context of a given chapter, so that we are allowed to add the same annotation to several chapters. Rule (3) does not allow the same annotation to be added more than once to the same chapter.

Consider the first of the above two rules, and observe that the first parameter in all four operations of that rule identifies the author. There are no restrictions on authors in the rules—we have used the anonymous '_' placeholder—and thus a chapter added by one author can be deleted by another. Suppose we do not want to allow this and require that chapter ownership is respected. Thus, the author who adds the chapter is the only author allowed to delete it. To do this, we specify an additional rule:

(4)
```
forall a : author, c : chapter
order addChapter(a,c,_); delChapter(a,c)
```

The immediate consequence of rule (4) is that the `addChapter` operation can now be performed many times, once per chapter by each potential author. However, recall that all of the ordering rules are enforced at the same time. Therefore, rule (1) ensures that the `addChapter` and `delChapter` operations are done at most once for a given chapter, and rule (4) ensures that a chapter is deleted by the same author who added it.

The grammar for ordering rules with parameters consists of the following rules:

```
order_rule_def ::=
   rule_name OPT
   ("forall" (param_name LIST ":" tm_type_name) LIST) OPT
   "order" order_expr.
order_expr ::= operation_elem
             | order_expr CHAIN ";"
             | "(" order_expr ")"
             | order_expr "*"
             | order_expr "+"
             | order_expr CHAIN "|"
             | "" order_expr LIST "".
operation_elem ::= operation_name "(" (param_name | "_") LIST ")".
```

We have already alluded to the somehow 'definitional' role of the first occurrence of a universally quantified variable. That occurrence determines the filter for maximal subsequences of operation invocations at the value level. Second and later occurrences can be viewed as equality conditions. For the reader familiar with LOTOS, we mention that first occurrences will be mapped onto '?'-variables, and later occurrences will be mapped onto '!'-expressions. This also makes clear that second and later occurrences in our CoCoA-rules could be allowed to be used in complex value expressions. We do not address this topic any further here because it is felt that such a provision would be too cumbersome to use and implement.

As pointed out earlier, the meaning of the combined usage of several universally quantified variables in a single ordering rule can sometimes be difficult to understand. To prohibit the specification of some forms of rule with unintuitive semantics, we are currently investigating syntactic restrictions on the use of parameters in ordering rules. Our investigation will also consider the ease with which the ordering rules can be mapped to LOTOS/TM.

7.1.4 Semantics of ordering rules with parameters

Our approach towards a semantics of CoCoA ordering rules in the parameter-augmented case is essentially that for the basic rules, given in Section 7.1.2, but with an important distinction. Ordering rules are, by nature, specification-time entities, whereas operation invocation sequences are run-time notions. This distinction plays a role when parameters arrive at the scene, and when parameter instantiation becomes an issue.

To bridge the gap, we define an environment $\mathcal{E}$ to be a function that assigns to each typed variable a value from the domain of the variable's type. The environment captures the values to which the variables are bound, or may become bound at run-time. By '$[x \mapsto v]\mathcal{E}$' we mean the environment identical to $\mathcal{E}$, except for its assignment to the variable x, which is now assigned the value v.

Our alphabet α in the parameter-augmented case has *operation invocations*, i.e. operation name with complete parameter list, as elements. The parameter list, in turn, consists of input values with types appropriate for the operation. Output is ignored in our description of the semantics of ordering rules.

The definition of α^* is likewise extended: elements of this set are now arbitrary finite sequences of operation invocations. Observe once more that α and α^* are run-time notions. This is not the case for our redefinition of $ops(R)$: this is based on *invocation patterns*. An invocation pattern consists of an operation name together with a parameter list pattern. This pattern, as in the syntax of our ordering rules, allows parameters to be:

- a global variable (determined by the scenario),
- a variable universally quantified over in the rule,
- an explicit value, or
- an anonymous placeholder ('_').

The environment $\mathcal{E}$ provides values for global as well as universally quantified variables.

An invocation pattern *matches* an operation invocation under a given environment $\mathcal{E}$ whenever:

1. they agree on operation name and cardinality of the parameter list,
2. a universally quantified variable x at position n in the invocation pattern coincides with value $\mathcal{E}(x)$ in the operation invocation,
3. a value v at position n in the pattern coincides with the same value in the operation invocation, and
4. an anonymous placeholder at position n in the invocation pattern matches any value in the operation invocation.

The matching relation is used to define the filter $\upharpoonright_{\mathcal{E}}$, which selects operation invocations from a sequence for consideration by an ordering rule. By $ops(R)$ we now mean the set of invocation patterns that occur in R. Like before, '$\sigma \upharpoonright_{\mathcal{E}} ops(R)$' is the maximal subsequence of elements of σ that match—in the above sense—some invocation pattern in $ops(R)$ under environment $\mathcal{E}$.

We define the collection of sequences $L_{\mathcal{E}}(R)$ as the regular language defined by R under $\mathcal{E}$. This collection contains all completed and properly instantiated operation invocation sequences that comply with the rule when all of its variables have been instantiated.

The semantic organisation of an ordering rule is precisely described by the following two clauses:

- **[basic rule]** If R is a quantifier-free rule, we have

$$[\![R]\!]_{\mathcal{E}} = \{\, \sigma \in \alpha^* \mid \sigma \upharpoonright_{\mathcal{E}} ops(R) \text{ is allowed by } L_{\mathcal{E}}(R) \,\}$$

- **[∀-rule]** A rule of the form '**forall** $x : \tau$ R' has the following semantics:

$$[\![\textbf{forall}\, x : \tau\ R]\!]_{\mathcal{E}} = \bigcap_{v \in [\![\tau]\!]} [\![R]\!]_{[x \mapsto v]\mathcal{E}}$$

The second clause is inductively defined. Both clauses are parameterised on the environment. We use double square brackets around a language construct to denote its semantics. Thus, $[\![\tau]\!]$ stands for the domain of values of type τ, and $[\![R]\!]$ stands for the semantics of rule R, the definition of which is given above.

As for the case without parameters, we define the semantics of a collection of n ordering rules to be the intersection of the semantics for each ordering rule:

$$\mathcal{U} = \bigcap_{i=1}^{n} \mathcal{U}_i, \text{ where } \mathcal{U}_i = [\![R_i]\!]\mathcal{E}_0$$

The intersection ensures that all ordering rules are enforced together. The environment $\mathcal{E}_0$ contains correct mappings for all globally defined variables.

7.1.5 Termination constraints

In the previous section, we explained how specific ordering constraints can be specified. Ordering constraints are, however, only descriptive and not prescriptive, or in other words, they don't express what *must* be done. *Termination constraints* are used to this end. They will be added to step definitions when needed, and make use of value labels that are added in-line to the ordering rules. A value label essentially assigns a state value to the ordering rule at a certain position, and can be viewed as a means of rule progress administration. We refer the reader to the example given below. There, we have augmented the ordering rule `chapter_rule` with a value label, given in square brackets, that is valid after each successful invocation of `editChapter`. This value label is consecutively used in the termination constraint of the `write_proposal` step. Value labels are allowed in positions immediately following invocation patterns in the ordering rules.

Predicate expressions consequently may use these labels to express conditions under which a transition of a step is allowed to occur. These predicates are allowed to query the rule state, i.e. whether a value is the current state value of the rule. If the transition for which the predicate is defined is exiting the step, we have defined a condition that must be satisfied before the step can finish. Hence the link to step definitions. The approach that COCOA takes allows both the formulation of conditioned break-points and conditioned termination points.

To allow reference to a particular ordering rule by a predicate, we may name ordering rules. Two examples of value-labelled ordering rules are:

```
chapter_rule :
  forall c : chapter
  order addChapter(_,c,_);
        editChapter(_,c,_) ["edited"]*;
```



```
       delChapter(_,c)

annotation_rule :
  forall c : chapter, an : annotation
  order addChapter(_,c,_);
        (addAnnotation(_,c,an);
           delAnnotation(_,c,an) ["processed"])*;
        delChapter(_,c)
```

To express the condition that the editor can only terminate the write proposal step when the introduction and the title page appear in the shared database, we could have added the following condition:

```
    step write_proposal[in start, revise  out done]
    begin
      ....
      on start, revise enable
        .....
        when ed issues introWritten()
        iff query document on chapter_rule("intro")
                          = "edited"
           and query document on chapter_rule("title")
                          = "edited"
        do done
      endon
```

Here, a query expression is used to check the status of the execution rule with respect to the respective chapters. The query expression uses an extension of the TM-expression syntax, with the following grammar:

```
query_expression ::=
     "query" workspace_expr
     "on" rule_name ( "(" TM_expr LIST ")" ) OPT.
```

The expressions following the rule name indicate the values for the parameters mentioned in the **forall**-clause. In case the current position of the rule is not labelled, the query expression returns an empty string value.

If we would like to state that a task for writing can only be completed if all the annotations have been removed from the chapter (and hopefully processed as well), this can be done by adding the following **if**-clause to the expression governing this communication operation:

```
  when a issues completeTask(ch)
  iff forall an : annotation
        exists annotation_rule(ch, an) in document
        implies query annotation_rule(ch, an) on document
                    = "processed"
  do compl
```


Here, we also use an **exists**-clause to check whether there is a subsequence for given values of the parameters mentioned in the **forall**-clause of the rule.

7.1.6 Grammar of the ordering rules

The grammar that we use here consists of:

```
order_rule_def ::=
   (rule_name ":") OPT
   ("forall" (param_name LIST ":" tm_type_name) LIST) OPT
   "order" order_expr.
order_expr ::= operation_elem state_mark OPT
             | order_expr CHAIN ";"
             | "(" order_expr ")"
             | order_expr "*"
             | order_expr "+"
             | order_expr CHAIN "|"
             | "{" order_expr LIST "}".
operation_elem ::= operation_name "(" (param_name | "_") LIST ")".
state_mark ::= '[' string ']'.
```

Note that the implicit priority of the symbols in the order expression is: first ";", then "*" and "+", and finally "|".

7.2 Data exchange protocols

A data exchange protocol should identify the operations to be imported or exported from one history into another history. COCOA provides a **select** construct for querying a workspace history to select such a set of operations. It is possible for these operations to depend on other operations in the history. In such a case, these other operations should be imported or exported as well.

Given a set of operations $\mathcal{I}$ to be exchanged, the transaction model identifies all depended-on operations in the workspace history [WäKl96]. Call this set of operations $\mathcal{P}$.[18] The operations in $\mathcal{P}$ constitute a consistent, atomic unit of cooperative work. Which operations are grouped into a unit of work is determined dynamically (at run-time) by the transaction model. To maintain consistency, all (or none) of the operations in $\mathcal{P}$ should be imported or exported. Through interaction with the transaction manager, it may be feasible to decrease the size of set $\mathcal{I}$, if some of the operations in $\mathcal{P}$ cannot be exchanged.

[18] The set $\mathcal{P}$ may be further partitioned into a collection of pairwise independent subhistories to decrease the likelihood of conflicts with operations in the destination workspace.

To calculate the set $\mathcal{P}$, the transaction model uses a *compatibility relation*, denoted by '$\oplus$', which is defined over operation invocations [WäKl96]. Two operation invocations a_1 and a_2 are said to be compatible (i.e., $a_1 \oplus a_2$ holds) if they are semantically independent. A typical situation for semantic independence is when the two operations manipulate different objects, or different attributes of the same object, but no shared subcomponents. Often, *commutativity* is an acceptable compatibility relation. That is, $a_1 \oplus a_2$ holds if it doesn't matter whether the two operation invocations are ordered as a_1 before a_2, or as a_2 before a_1 in a workspace history, because a_2 does not depend on the results of a_1, and vice versa. In transaction model research based on the semantics of data operations, relations such as $\oplus$ are user-defined or otherwise assumed to exist [BaRa92,Herl90,Levy91,LMWF94,Weih88].

CoAct defines $\mathcal{P}$ as follows. Let $\mathcal{H}$ be a (finite) workspace history, and let $\ll_{\mathcal{H}}$ be the ordering relation defined over the operation invocations in $\mathcal{H}$, such that $a_1 \ll_{\mathcal{H}} a_2$ holds whenever a_1 precedes a_2 in $\mathcal{H}$. The set of operations $\mathcal{P}$ is defined iteratively as follows:

$$\begin{aligned} \mathcal{P}_0 &:= \mathcal{I} \\ \mathcal{P}_i &:= \mathcal{P}_{i-1} \bigcup \{p \mid \exists\, q \in \mathcal{P}_{i-1} : p \ll_{\mathcal{H}} q \ \wedge \ not(p \oplus q)\} \end{aligned}$$

($\mathcal{P}$ is defined to be $\mathcal{P}_n$, where n is the smallest value for which $\mathcal{P}_n$ equals $\mathcal{P}_{n-1}$.) $\mathcal{P}$ includes all operation invocations in $\mathcal{I}$, plus all operation invocations on which they depend. In the worst case, $\mathcal{P}$ includes all operations in the history. Intuitively, $\mathcal{P}$ represents the set of operation invocations that are to be exchanged. These invocations will be re-executed by the transaction model in the destination workspace (if not already (re-)executed in the destination workspace) as part of the import or export, provided there are no conflicts with the operations in the destination workspace.

The COCOA **select** construct is provided to identify the set $\mathcal{I}$. This construct is discussed in Section 7.2.1. The COCOA **non-commutative** construct is provided to define the relation $not(p \oplus q)$ that appears in the definition of $\mathcal{P}$ above. This construct is discussed in Section 7.2.2.

The CoAct transaction model relies on a second data model-specific compatibility relation, called $\diamond$, which is used to relate operation invocations in *different* workspace histories. Relations $\oplus$ and $\diamond$ are similar: $\oplus$ is used to determine independencies between operations in the *same* history, whereas $\diamond$ is used to determine independencies between operations in *different* histories. In the examples given in [WäKl96], it is assumed that $\diamond$ is equal to $\oplus$.

Commutativity is often a sufficient criterion for semantic independence. That is, $p \oplus q$ holds if p and q commute backward, and $p \diamond q$ holds if p and q commute forward.[19] However, there may be cases where $p \oplus q$ or $p \diamond q$ holds, but p and q do not commute. This situation typically arises when commutativity is defined at the implementation level, not at the specification level. (For example, if a set type is implemented as a list, then changing the order of insertion of its elements can result in a "different" set.) The situation also arises when only the read and write sets of the operations are considered, instead of their semantics. (For example, two increment operations may read and write the same variable, but their cumulative effect is to add two to the variable's value, regardless of their order.)

[19] Forward and backward commutativity are subtly different notions, which take into account the preconditions of operations [Weih88].

Data operations in COCOA are defined in TM, which is a formal specification language, not an implementation language. TM is used to specify the semantics of data operations. Therefore, COCOA offers the designer the ability to specify commutativity as the basis for semantic independence.

Two operation invocations that are related by $\diamond$ can both appear in a merged history, because they are independent. If $\diamond$ does not hold, then the operations are said to conflict. A data exchange operation can be classified as *safe*, meaning that if there are conflicts it does nothing, or as *destructive*, meaning that conflicting operations are undone in the destination history. There may be several ways to resolve a conflict, by undoing different operations. Which strategy is followed, and how conflicting operations are undone, is determined by the transaction manager.[20] In the specification of the data exchange protocol, we do not specify whether the exchange operation is classified as safe or destructive; this will be a parameter of the execution environment.

7.2.1 Selecting operations from a history

In Section 5.3, we saw some examples of import and export operations, such as the following, which can be done by an author:

```
export Chapter(ch) to document
import Annotations(ch) from a2
```

Here, '`ch`' stands for the chapter being edited, '`document`' stands for the shared workspace, and '`a2`' stands for a second author's workspace. The two expressions '`Chapter(ch)`' and '`Annotations(ch)`' refer to sets of operations that are to be exchanged (called $\mathcal{I}$ above). This set can be identified using the **select** construct in COCOA. For example, to identify a set of operations '`Chapter(ch)`', the following definition can be used:

```
data exchange operations
   Chapter(c : chapter) =
   select addChapter(_,c), editChapter(_,c,_), delChapter(_,c)
```

This code specifies that all add, edit, and delete operations on the chapter parameter '`c`' are selected from the current history. For a given chapter '`ch`', when the expression '`Chapter(ch)`' is evaluated for a particular history, the transaction model calculates the set of operations that is returned, using the definition of $\mathcal{P}$ given earlier. The semantics of the **select** construct is dependent on this calculation, and thus on the transaction model.

The syntax for the specification of a select operation on a history is given by the following grammar rule:

```
data_exch_oper_def ::=
   data_exch_oper_name
   "(" par_name ":" tm_type_name LIST ")" "="
       "select" operation_elem LIST
```

[20] Please note: At the time of writing this report, no transaction model primitives were available for user interaction with the merging algorithm.

7.2.2 History rules

As discussed earlier, to maximise the possibilities for data exchange between workspaces, semantic information about data operations is needed. In CoCoA, this information is given as *history rules*, which are used by the transaction model during data exchange. In this section, we look at history rules for specifying the semantic independence of data operations.

In CoCoA, we use *commutativity* as our notion of semantic independence, and *failure to commute* as our notion of semantic dependence. Commutativity is a stronger relation than both $\oplus$ and $\diamond$, but we can precisely state what it means for data operations that are state transitions.[21] Each data operation in CoCoA is formally specified as an update method in TM. Thus, each data operation invocation takes an implicit input parameter that is the workspace state, and returns an implicit output value that is the modified workspace state. Data operations are functions on the workspace. We can precisely define what is meant by commutativity of data operation invocations in this context.

In CoCoA, we consider state-independent, but *parameter-dependent* notions of commutativity. Commutativity can be specified as a function of the parameter values of the operations involved, but must be independent of the workspace state. A boolean expression may be defined over the parameters; two operation invocations are said to commute if the expression evaluates to true for their parameter values.

Commutativity can be specified "positively" or "negatively." A positive specification enumerates all cases in which two operations commute, whereas a negative specification enumerates all cases in which they *do not* commute. The CoCoA ordering rules (see Section 7.1) influence our choice here. It is clear that all operation orderings specified by the ordering rules involve a dependency (regardless of whether the operations commute). For example, according to rule (2) in Section 7.1.3, an invocation `addChapter(a1,c,t)` must precede the invocation `delChapter(a2,c)` in a history, since the operations are on the same chapter. Because the ordering rules already specify many cases for which operation dependencies exist, we use a negative specification of commutativity in CoCoA.

Unless two operations are ordered by the execution rules, we will assume that they are independent (unless they are specified as non-commutative, as discussed below). With this in mind, let us think about possible dependencies not implied by the ordering rules given in Section 7.1.3. Suppose we want to import or export an edit operation on a chapter. From the ordering rules, we can observe that this editing operation is dependent on an `addChapter` operation, also in the history. Now consider trying to construct a consistent subhistory (i.e., the set $\mathcal{P}$ defined earlier), based on this ordering rule. The rule states that an arbitrary number of edit operations can be done after a chapter has been added. However, the rule does not tell us that a consistent subhistory should include all of the editing operations that precede (in the workspace history) the one that we want to exchange. Therefore, we must provide the transaction model with a history rule to say that two edit operations on the same chapter are not independent. This is specified using the **non-commutative** construct as follows:

[21] We point out that $\diamond$ is difficult to define precisely unless it is assumed that, in addition to the workspace state, data operations also output some indication of the data items they modify.


```
forall a1, a2 : author, c : chapter, t1, t2 : text
non-commutative editChapter(a1,c,t1) and editChapter(a2,c,t2)
```

This rule can also be written as:

```
forall a1, a2 : author, c1, c2 : chapter, t1, t2 : text
non-commutative editChapter(a1,c1,t1) and editChapter(a2,c2,t2)
           iff c1 = c2
```

using an optional expression over the parameter values.

Additional history rules are needed to relate annotation operations to editing operations, since neither of the ordering rules that mention annotations (rules (3) and (4) on page 42) mention editing operations. The following history rules are used:

```
forall a1, a2 : author, c : chapter, t : text, an : annotation
non-commutative remAnnotation(a1,c,an) and editChapter(a2,c,t)
```

```
forall a1, a2 : author, c : chapter, t : text, an : annotation
non-commutative editChapter(a1,c,t) and addAnnotation(a2,c,an)
```

Observe that the specification of commutativity in this section is given in terms of operation invocations. The scenario designer may utilise the TM proof tool to check whether the history rules actually make sense [Spel95]. This possibility will be further explored in Deliverables IV.4 and IV.5.

The CoCoA syntax introduced in this section is as follows:

```
commutability_rule ::=
   forall_part
   "non-commutative" operation_elem "and" operation_elem
   ("iff" TM_expr) OPT.
```

As stated earlier, the expression `TM_expr` may not be a function of the workspace state.

8 Mapping to LOTOS/TM

The following sections show how the different features of CoCoA can be mapped to LOTOS/TM. The mappings given are not complete, but rather serve as a first attempt to give a formalisation of the semantics of the language. The mapping of the organisational aspects of the language is discussed first, followed by the mapping of the transactional aspects. Familiarity with LOTOS/TM is assumed in this section.

8.1 Mapping the steps

In this section, we sketch how the specification of the organisational aspects can be mapped to LOTOS/TM. The general idea is to map each step to a process. For clarity, additional local process definitions are used. With respect to the transitions, the general idea is to use one gate for each transition. Because we consider transitions to be atomic events, this also means that we try to map each transition to a single gate event. There are cases, however, where it is convenient to use more than one event. According to this general idea, the following simple step definition:

```
step s1[in i1  out o1]
begin .....
end
```

is mapped to a process with the following structure:

```
process s1[i1, o1] : exit :=
   i1;
   ....
   o1; exit
endproc
```

Transitions on the **in** and **out** interaction points are mapped to events at their respective LOTOS/TM gates. These events "border" the other actions performed by the process.

8.1.1 Sequential steps

If a step consists of a sequence of smaller steps, then these internal steps can be chained using hidden gates for each internal transition. For example, the step specification of the whole scenario shown in Figure 2 is represented by the following specification fragment:

```
process write_report[start, done] : exit :=
   hide t1, t2 in
     proposal[start, t1]
      |[t1]|
     writing[t1, t2]
      |[t2]|
     complete[t2, done]
where
   .....
endproc
```

The transition between the `proposal` step and the `writing` step is made on gate `t1`; the transition between the `writing` step and the `complete` step is made on gate `t2`.

8.1.2 Repetitive steps

In the `proposal` step of the example scenario, the referee can ask the editor to revise the introduction, which means that the two smaller steps that the step consists of can be repeated any number of times.

The CoCoA specification of the proposal step, is given by:

```
step proposal[in start  out done, rejected]
begin

  step write_proposal[in start, revise  out done]
  begin ... end

  step accept_proposal[in start  out acc, rev, rej]
  begin ... end

  on start do write_proposal.start
  on write_proposal.done do accept_proposal.start
  on accept_proposal.acc do done
  on accept_proposal.rev do write_proposal.revise
  on accept_proposal.rej do rejected
end
```

At first sight, it may look like this step can be mapped to the following LOTOS/TM specification:

```
process proposal[start, done, reject] : exit :=
  hide t1, t2 in
    write_proposal[start, t2, t1]
       |[t1, t2]|
    accept_proposal[t1, done, t2, reject]
where
   process write_proposal[start, revise, done] : exit :=
        start; .... ; done; exit
     [] revise; .... ; done; exit
   endproc
   process accept_proposal[start, acc, rev, rej] : exit :=
     start; .... (   acc; exit
                  [] rev; exit
                  [] rej; exit)
   endproc
endproc
```

Gate `t1` is used to synchronise events on gate `done` of process `write_proposal` and gate `start` of process `accept_proposal`; gate `t2` is used to synchronise events on gate `revise` of process `write_proposal` and gate `rev` of process `accept_proposal`. Unfortunately, the above specification *does not* work correctly, once an event on gate `t2` occurs. Both local processes should be defined recursively, so that the substeps can be repeated. This leads to the following specification:

```
process proposal[start, done, reject] : exit :=
  hide t1, t2 in
    (write_proposal[start, t2, t1]
     [> (done; exit [] reject; exit)
    )
       |[t1, t2, done, reject]|
    accept_proposal[t1, done, t2, reject]
where
   process write_proposal[start, revise, done] : noexit :=
        start; .... ; done; write_proposal[start, revise, done]
     [] revise; .... ; done; write_proposal[start, revise, done]
   endproc
   process accept_proposal[start, acc, rev, rej] : exit :=
     start; .... (   acc; exit
                  [] rev; accept_proposal[start, acc, rev, rej]
                  [] rej; exit)
   endproc
endproc
```

Observe that process `write_proposal` is disabled (using `[>`) whenever the `proposal` step terminates on gate `done` or `reject`. Although the correct LOTOS/TM specification is complicated, it is not difficult to generate it automatically.

8.1.3 Parallel steps

Parallel steps do not use any new mapping techniques, other than those we have already seen. Mapping the parallel steps shown in Figure 3 (page 20) results in the following process definition:

```
process find_info[in, done1, done2] : exit :=
   (find_in_library[in, done1] [> done2; exit)
     |[done1, done2]|
   (find_on_WWW[in, done2] [> done1; exit)
where
   process find_in_library[in, done] : exit :=
     in; .... ; done; exit
   endproc
```

```
    process find_on_WWW[in, done] : exit :=
      in; .... ; done; exit
    endproc
endproc
```

Completion of either subprocess disables the other one. Both subprocesses synchronise when they exit.

8.1.4 Multiple activations

The last part we have to deal with, with respect to the mapping of the steps to LOTOS/TM, is the parallel sections. As there could be an infinite number of possible instances of the steps within a parallel section, it is not very practical to create the instances before they are actually needed. There are two kinds of parallel section: those without a key value, and those with a key value. Although the syntax for these two kinds of parallel section look much the same, their mappings are slightly different. We first deal with mapping parallel clauses without key values.

An example of a parallel section without a key value is:

```
    parallel
       step one [in i1  out o1]
       begin .....
       end
    endpar
```

This can be mapped to:

```
process par_sec [i1, o1] : noexit :=
     i1;
     (one[o1]
       |||
      par_sec[i1, o1])
where
   process one[o1] : exit :=
      ....;
      o1; exit
   endproc
endproc
```

Observe that we moved the `i1` gate event outside process `one`. For this example, moving the event does not pose any problems. However, this mapping is not general. A better solution would be:

```
process par_sec [i1, o1] : noexit :=
     i1;
     ((hide e_i1 in
       one[e_i1, o1] |[e_i1]| e_i1; exit)
        |||
      par_sec[i1, o1])
where
   process one[i1, o1] : exit :=
      i1; ....;
      o1; exit
   endproc
endproc
```

In this solution, there are two events at two different gates needed for the transition to complete. This solution, however, is not sufficient in the case that there are several steps inside a parallel section that can be repeatedly activated.

Now let's look at a more complicated example. Assume we have the following parallel section:

```
step main [in .. out ...
                signal start  interrupt ready]

parallel
   step one [in i1  out o1]
   begin .....
   end
   step two [in i2  out o2, o3]
   begin .....
   end
endpar

on main.start do one.i1
on one.o1 do two.i2
on two.o2 do one.i1
on two.o3 do main.ready
```

Notice that the first and third transitions both end at point `i1` of step `one`. It is therefore not possible to map both transitions onto an event at the same gate, so we have to use two gates for the incoming interaction point `i1` of step `one`. This leads to the following mapping for the parallel section, in which interaction point `main.start` is bound to gate `i1`, and interaction point `main.ready` is bound to gate `o1`:

```
process par_sec [i1, o1] : noexit :=
     i1;
```

```
        ((hide e_i1 in
          (hide t1, t2 in
             one[e_i1, t2, t1] [> t2 ; exit
               |[t1, t2]|
             two[t1, o1, t2]
          ) |[e_i1]| e_i1; exit)
           |||
         par_sec[i1, o1])
   where
      process one[i1_ext, i1_int, o1] : noexit :=
         (i1_ext; exit [] i1_int; exit) >>
         ....;
         o1; one[i1_ext, i1_int, o1]
      endproc
      process two[i2, o2, o3] : exit :=
         i2;
         ....;
         ( o2; two[i2, o2, o3]
           []
           o3; exit
         )
      endproc
   endproc
```

Let us turn to the kind of parallel section that does have a key value. In the parallel section without a key value, each incoming transition causes a new copy of the steps inside the parallel section to be instantiated. But in a parallel section with a key, an incoming transition should not create any new instances, if an instance with the same key already exists. This means we have to keep track of all instances that have been created so far. To illustrate how this is done, we use the following example, which adds an integer key to the first example of this section.

```
   parallel(v : int)
      step one [in i1  out o1]
      begin .....
      end
   endpar
```

In the LOTOS fragment that follows we use the parameter `vs` to contain the set of all key values for which an instances has been created. This parameter is used to determine whether a new instances is to be created. Because instances that have been created, do not exit, they can be re-activated. In order to distinguish between all the different instances, we have to add the value of the key as an attribute of the gate event. This results in the following LOTOS definition:

```
process par_sec[i1, o1](vs : ℙ int) : noexit :=
      i1 ?v [not (v in vs)]
      ((hide e_i1 in
        one[e_i1, o1](v) |[e_i1]| e_i1 !v ; exit)
         |||
       par_sec[i1, o1](vs union v))
where
   process one[i1, o1](v : int) : exit :=
      i1 !v ; .... ;
      o1 !v ; one[i1, o1](v)
   endproc
endproc
```

Of course, `par_sec` should be called with the empty set as the argument for the `vs` parameter. If the key consists of a number of values, we need to generate the appropriate TM sort to contain them.

8.1.5 Elementary steps

Adding values to transitions is done by mapping them to attributes of the gate event to which the transition is mapped. All three kinds of operation (data, communication, and data exchange) need to be mapped. Our approach is to map all operation invocations to events at a single gate, called `oper`, which will be passed as a parameter to all processes of the LOTOS/TM specification. We point out that the LOTOS/TM behaviour expression resulting from the mapping of the organisational view does not specify what operations do. The behaviour only specifies which operations can happen, based on the organisational constraints. Additional constraints on the occurrence of some data operations will arise as a result of the execution rules in the transactional view. These are discussed in Section 8.2.

In the mapping of operation invocations, the first event attribute on gate `oper` is used to identify the actor performing the operation. The second attribute indicates the kind of the operation: `"data"` is used for a data operation, `"import"` for an import operation, `"export"` for an export operation, and `"com"` for a communication. The third attribute is the name of the operation, and the remaining attributes are used for arguments to the operation. The "other" workspace for data exchange operations (i.e., not that of the actor performing the operation) is identified by an additional attribute.

To illustrate the mapping, suppose we are given the following operations in an elementary step:

```
a1 : addChapter(a2,ch1,t1)
a3 : import Chapter(ch2) from document
a4 : on completePrepare() do out
```


According to the above rules, these operations are mapped to the following LOTOS/TM event offers:

```
oper !a1 !"data" !"addChapter" !a2 !ch1 !t1
oper !a3 !"import" !"Chapter" !ch2 !document
oper !a4 !"com" !"completePrepare" ; out
```

An elementary step is mapped to a LOTOS/TM process. Because the operations mentioned in an elementary step can be repeated any number of times, the process needs to be tail-recursive. Consider the following, slightly simplified specification of the `write_proposal` step given on page 30[22]:

```
step write_proposal(in start  out done]
begin
  on start enable
    /* The data operations the editor can do: */
    ed : addChapter(ed, title, _),
         addChapter(ed, intro, _),
         editChapter(ed, title, _),
         editChapter(ed, intro, _),
    /* Export operations the editor can do: */
         export Chapter(title) to document,
         export Chapter(intro) to document,
    /* Other actions the editor can do, e.g., leave this step: */
         on introWritten() do done
  endon
end
```

This step is mapped to the following LOTOS/TM process:

```
process write_proposal[start, done, oper](ed : actor) : noexit :=
    start;
    write_proposal_operations[done, oper](ed) ≫
    write_proposal[start, done, oper](ed)
where
  process write_proposal_operations[done, oper](ed : actor) : exit :=
      oper !ed !"data" !"addChapter" !ed !"title" ?t:text;
      write_proposal_operations[done, oper](ed)
    [] oper !ed !"data" !"addChapter" !ed !"intro" ?t:text;
      write_proposal_operations[done, oper](ed)
    [] oper !ed !"data" !"editChapter" !ed !"title" ?t:text;
      write_proposal_operations[done, oper](ed)
    [] oper !ed !"data" !"editChapter" !ed !"intro" ?t:text;
```

[22] Its `revise` interaction point has been removed.

```
        write_proposal_operations[done, oper](ed)
    [] oper !ed !"export" !"Chapter" !"title" !"document";
        write_proposal_operations[done, oper](ed)
    [] oper !ed !"export" !"Chapter" !"intro" !"document";
        write_proposal_operations[done, oper](ed)
    [] oper !ed !"com" !"introWritten" ; done ; exit
  endproc
endproc
```

An auxiliary, local process definition is used to group the operations. Observe that '_' parameters are mapped to '?'-attributes on event offers.

Elementary steps with more than one enabling rule are a little more complicated to map to LOTOS/TM, but the same principles applied above are used in their mapping.

The use of a **for**-construct in combination with conditions is mapped to a selection predicate on the event offers at the `oper` gate. For example, consider the following **for**-construct, taken from the task step on page 30:

```
for a1, a2 in authors_of_task where a1 <> a2 allow
  a1 : import Chapter(ch) from a2
```

This construct is mapped to the following event offer:

```
oper ?a1:actor !"import" !"Chapter" !ch ?a2:actor
  [a1 in authors_of_task
   and a2 in authors_of_task
   and a1 <> a2
  ]
```

The selection predicate imposes constraints on the values of the two actor parameters to the event.

The last example in Section 5.3 includes a mixture of steps and enabling constructs within steps. Its mapping to LOTOS/TM involves a combination of the techniques shown above with those given in Section 8.1.4.

8.2 Mapping the execution rules to LOTOS/TM

In this section, we sketch how the specification of the execution rules can be mapped to LOTOS/TM. Because only deterministic execution rules are allowed (see Section 7.1), the set of rules can be mapped to a Deterministic Finite Automaton (DFA). The mapping to LOTOS/TM uses knowledge of the DFA representation of the rules, and in fact generates the

LOTOS/TM specification of such a DFA. To keep the discussion simple, the details of some steps of the mapping (e.g., those related to the DFA) are omitted from the examples. The reader is referred to Deliverables IV.4 and IV.5 for the complete definition of the mapping algorithm.

The general idea of the mapping is to use a separate gate for each data operation.[23] Each ordering rule is mapped to a single LOTOS/TM behaviour expression, and all of these behaviour expressions synchronise on the data operations that they have in common. The mapping is defined in a syntax-directed manner: each subexpression in an ordering rule is mapped to a local process definition; instantiations of these processes are composed using synchronisation combinators. The gate list for each process definition includes all gates associated with operations that occur within the ordering (sub)expression. Our explanation in this section closely follows the description of the execution rules given in Section 7.1.

The most basic element of an ordering expression is a single data operation. At first, we assume that data operations have no parameters. The mapping is extended to include parameters in Section 8.2.1. An ordering expression such as '**order** `a`', which consists of a single data operation `a`, makes use of the following LOTOS process:

```
process e1[a] : exit :=
      a; exit
endproc
```

The ordering rule '**order** `a`' is mapped to an instantiation of this process: '`e1[a]`'. The process allows one `a` event to take place and then exits.

The ';' operator in an execution rule is closely related to the LOTOS sequencing operator ';'. The ordering expression '`<e1> ; <e2>`', where `<e1>` and `<e2>` are ordering expressions, is mapped to a process '`e3`' by taking the process definition for '`<e1>`', adding additional gates for `<e2>`, and replacing all occurrences of '**exit**' by (an instantiation of) the process definition for '`e2`', followed by the proper gate list. The definition of process '`e2`' is made local to that of '`e3`'. As an example, the ordering rule '**order** `a ; b`' (where '`<e1> = a`' and '`<e2> = b`') is mapped to the following behaviour expression:

```
e3[a,b]
where
process e3[a,b] : exit :=
    a; e2[b]
where
  process e2[b] : exit :=
     b; exit
  endproc
endproc
```

[23] The mapping for data operation invocations in this section uses a different approach than that given in Section 8.1.5. Refer to Section 8.3 for comments on their integration.

The composite behaviour allows one `a` operation to take place, followed by one `b` operation, and then exits.

The translations of the '*' and '+' operators, because of their repetitive nature, make use of tail-recursive process definitions. We only need to translate the '*' operator, because '`<e>+`' is merely an abbreviation for '`<e>; <e>*`'. The expression '`<e1>*`' is mapped to a process `e2` in which each occurrence of '**exit**' is replaced by '`e2[..]`'; an extra choice for '**exit**' is added for the empty sequence. As an example, consider the definition of '`e1`' for operation '`a`' given above. The ordering rule '**order** `a*`' is mapped to the following behaviour:

```
e2[a]
where
process e2[a] : exit :=
       (a; e2[a])
   [] exit
endproc
```

The process allows the repetition of zero or more `a` operations.

The '`|`' operator is used to specify alternatives and can thus be mapped to the LOTOS choice operator '`[]`'. For example, the ordering expression '`<e1>|<e2>`' is mapped to a behaviour expression with the form '`e1[..] [] e2[..]`'. The rule '**order** `(a| );(b|c)`' is mapped to the following behaviour:

```
e1[a,b,c] [] (e2[b] [] e3[c])
where
process e1[a,b,c] : exit :=
   a; (e2[b] [] e3[c])
endproc
process e2[b] : exit :=
   b; exit
endproc
process e3[c] : exit :=
   c; exit
endproc
```

To translate a group of ordering expressions, which can be executed in any order, the LOTOS interleaving combinator '`|||`' is used. For example, the ordering expression '`{<e1>,<e2>}`' is mapped to a behaviour expression with the form '`e1[..] ||| e2[..]`'. The rule '**order** `a;{b,c}`' is mapped to the following behaviour:

```
e4[a,b,c]
where
process e4[a,b,c] : exit :=
```

```
    a; (e2[b] ||| e3[c])
endproc
process e2[b] : exit :=
    b; exit
endproc
process e3[c] : exit :=
    c; exit
endproc
```

After operation `a` takes place, operations `b` and `c` can take place in either order.

8.2.1 Parameters in execution rules

The mapping of execution rules that make use of a '**forall**'-clause with parameters is slightly more complicated. This is due to the fact that an execution rule with a '**forall**'-clause represents a potentially infinite set of rules, for all possible values of the parameters. In practise, however, only a finite subset of these rules need to be enforced at run-time. For this reason, a universally quantified execution rule will be mapped to a single recursive process (with local, auxiliary process definitions). One instantiation of the process definition will be used for each value of its quantification.

To illustrate the mapping, the following two rules are used as examples:

(1) **forall** `i : int`
order `a(i); b(i,_)`

(2) **forall** `i : int, j : int`
order `a(i); b(i,j); c(i,j)`

Rule (1) makes use of a single universally quantified parameter. The first operation in the rule, '`a(i)`', has a *defining* occurrence of this parameter. (See Section 7.1.3.) We use '?'-attributes in LOTOS/TM gate event offers to model defining occurrences of universally quantified parameters. The second operation in rule (1), '`b(i,_)`', has a *using* occurrence of parameter '`i`'. We use '!'-attributes for the using occurrences of a parameter. The second parameter to operation '`b(i,_)`' is the anonymous placeholder, which can have any value. A '?'-attribute will be used on the LOTOS/TM event offer. The following gives the result of the mapping for the first rule, which is similar to the mapping of **order** `a; b` given earlier:

```
e3[a,b](emptyset(int))
where
process e3[a,b](all_i : ℙ int) : noexit :=
    a ?i:int [not (i in all_i)] ;
     ( e2[b](i)
```

```
          |||
        e3[a,b](all_i union i))
where
  process e2[b](i: int) : exit :=
     b !i ?k:int ; exit
  endproc
endproc
```

Note the use of TM sets here. The process is defined recursively, with each new instantiation responsible for a different value of the universally quantified 'i' parameter. For each operation a that has been executed, an operation b is allowed with the same i parameter value. This is indicated by the '!'-attribute on the b event, offered by process e2. Any value is permitted for b's second attribute: '?k:int'.

Rule (2) makes use of two universally quantified parameters. The first quantified parameter, 'i', is defined at its occurrence in operation 'a', and used at its occurrences in operations 'b' and 'c'. The second quantified parameter, 'j', is defined at its occurrence in operation 'b', and used at its occurrence in operation 'c'. The rule is mapped to the following LOTOS/TM behaviour expression:

```
e5[a,b,c](emptyset(int))
where
process e5[a,b,c](all_i : ℙ int) : noexit :=
    a ?i:int [not (i in all_i)];
     ( e3[b,c] (i, emptyset(int))
        |||
       e5[a,b,c](all_i union i))
where
  process e3[b,c](i: int, all_j: ℙ int) : noexit :=
      b !i ?j:int [not (j in all_j)];
       ( e2[c](i, j)
          |||
         e3[b,c](i, all_j union j))
  where
    process e2[c](i:int, j:int) : exit :=
       c !i !j ; exit
    endproc
  endproc
endproc
```

The value of the i parameter is "chosen" by the a event of process e5. The selection predicate ensures that previous parameter values do not get used again. Process e3 is given the value of parameter i as an argument, since it is needed for the using occurrences of this parameter in operations b and c. Process e3 is also given an empty set of values for j, since for this particular choice of i value, all j values are possible.

8.2.2 Termination constraints

In Section 7.1.5, the labelling of states is introduced. The values of the labels can be queried from the organisational part of the specification, and for this reason, they should be made known on an external gate. To accomplish this, we add an extra gate to each process of a rule that is labelled, and specify that a gate event occurs at this gate after each operation that is accepted. (We also need to specify such a gate event after each operation that has not been labelled in the ordering rule.) As an example, consider the following execution rule, which includes labelling expressions:

```
ab_rule :
forall i : int
order a(i); b(i) ["done"]
```

Because the above example introduces a collection of rules, one for each value of `i`, we need to use the `i` parameter as an attribute on the gate events. In the following LOTOS/TM behaviour expression, we use events on the '`s`' gate for signaling the so-called state of the execution rule:

```
e3[a,b,s](emptyset(int))
where
process e3[a,b,s](all_i : ℙ int) : noexit :=
    a ?i:int [not (i in all_i)] ;
    s !i !"" ;
     ( e2[b](i)
        |||
       e3[a,b,s](all_i union i))
where
  process e2[b](i: int) : exit :=
     b !i ;
     s !i !"done" ;
     exit
  endproc
endproc
```

The above process definition assumes that an event at the '`s`' gate always happens immediately after a data operation invocation. The empty string is signaled after the '`a`' operation, which has no label in the ordering rule.

8.3 Combining the two views

In the previous sections, we have shown how the specification of the two views, the organisational view and the transactional view, can be mapped to LOTOS/TM. An obvious question

that comes to mind is: can these two mappings be glued together? To answer this question, we first remark that we only showed how the execution rules can be mapped to LOTOS/TM; we did not show mappings for the data exchange protocols and the history rules. To describe these mappings, we would need to provide a specification of the transaction model in LOTOS/TM. Second, we point out that data operations are mapped differently in the mappings for the two views. In the mapping of the organisational view, we use one gate for all kinds of operation (data, import/export, and communication); attributes are used to determine the user, the kind of operation, the name of the operation, and (optionally) the parameter values. In the mapping of the execution rules, we use one gate for each data operation; the parameter values are given, but the user is not mentioned.

It is clear that some kind of interface is needed between the two LOTOS/TM parts. This might require that the results of both mappings are adapted. For example, it may not be possible to map each data operation to the occurrence of a single event on a gate. More details of this will be provided in TRANSCOOP Deliverables IV.4 and IV.5.

In our TRANSCOOP specification environment, a simulator for LOTOS/TM will be provided. The simulator will allow the scenario designer to test the code generated by the mappings. In particular, the designer will be able to determine whether the sequences of operations permitted by the ordering rules are indeed what was intended.

9 Conclusions

This report describes the COCOA language, a language aimed at the *specification* of cooperative applications. Such applications involve data, users and activities, operations and communications in a wide spectrum of application areas, ranging from relatively static work flow descriptions to relatively dynamic design environment descriptions such as cooperative document authoring. The target area, thus, is a highly complex field, and it is this complexity that provides the main motivation for having a specification/implementation distinction. It is our aim, eventually, to develop a specification environment that will allow the specifier to derive correct implementations from correct specifications, but given the complexity of the field, this is a long-term goal. A preciser definition of such an environment will follow in the project, but we may mention our plans for a simulator, a commutativity proof tool (based on operation semantics), and possibly other verifications.

COCOA was developed as—and was targeted to be—a user-friendly front-end to LOTOS/TM, the specification formalism chosen in the TRANSCOOP project. We believe additional clarity can be obtained in specifications for the target field by using COCOA, but we acknowledge the fact that we have left some issues untouched. We hope to address these at a later stage, but well within the context of the project. Some design issues that will be resolved only during the implementation of COCOA are:

integration with TM A better integration with TM is wanting, with respect to the schema definition, as well as the syntax used. TM expressions are widely used in COCOA, although this fact is somewhat underrepresented in the current document. We have concentrated on the novelties of the language, not on the features of the languages that it is composed of (semantically).

negotiation and awareness Aspects like negotiations and awareness have not been fully addressed. Negotiations could play a role during data exchange operations. Awareness could be defined by stating the manner in which the actors are to be informed about conflicting data operations. Note that these concepts will depend on the existence of similar facilities in the transaction model.

Other topics may need to be addressed in the language, which are not present in its current version. We mention the following possible extensions:

local value declarations Local, volatile data could be added to CoCoA at the level of steps. This kind of data could then be used to control the execution of the scenario. The reason for not adding it to the language is more of a design question than it is a technical problem.

timing constraints There are several ways in which timing constraints can be added to CoCoA. The easiest way to add timing to the language would be to add a delay (or wait until) clause to the transition specification as follows:

```
on a.done do b.start after(2 min)
```

This is, admittedly, an ad hoc solution to a more general problem, but we include it here to bring the intuitive idea across.

9.1 Comparison of CoCoA and LOTOS/TM

Our goal in designing CoCoA has been to provide a user-friendly interface for LOTOS/TM, more appropriate for describing cooperative scenarios. In this section, we first look at LOTOS/TM, and discuss its drawbacks for use in modelling cooperative scenarios. We then summarise the features of CoCoA that were introduced to provide domain-specific declaration mechanisms for the language.

9.1.1 Modelling cooperative scenarios in LOTOS/TM

In a way, the concepts available in LOTOS/TM are much more abstract than the concepts needed to describe cooperative scenarios. The latter is just one (good) use of the former. This becomes clear if we think about describing actors, activities, and data (the three facets of a cooperative scenario) in LOTOS and TM. In LOTOS, the specification mechanisms available are processes, gates, and events, and the means to order the events at the gates. In TM, the specification mechanisms available are data and operations on that data. Neither language offers specification mechanisms for the actors who are responsible for causing events to occur, or data operations to be performed. Both languages lack a notion of 'active participants', namely, the actors in a cooperative scenario.

Actors can be *modelled* in a variety of ways in LOTOS/TM, but there is no way to explicitly declare them, i.e. the notion of actor itself is not present in the language. For example, to model actors, a scenario designer could specify them as *data values*, which are passed around as parameters to processes. Unfortunately, this seems to contradict the idea of cooperation. Actors should be active participants in the things that happen; they should be responsible for making event selections. If we consider two other specification mechanisms available in LOTOS/TM, events and data operations, we realise that neither of these is appropriate for modelling actors either. (For example, it makes no sense to say an actor is an event.) The typical LOTOS-perspective would be to consider an actor as a specific kind of process, able to make autonomous decisions. It is possible to abstract away from the internal decision-making of such a process, and thus autonomy can be somehow modelled.

Another drawback of using LOTOS/TM to specify cooperative scenarios the lack of facilities to distinguish concepts such as communications and data operation invocations. It is obvious that both concepts are dynamic notions. But the only primitives that are dynamic in LOTOS/TM are events, so both concepts would likely be modelled in the same way.

9.1.2 Evaluation of CoCoA

The philosophy of the CoCoA language design is fourfold. During the design process, our goals have been: the ability to describe both the static and dynamic organisation of cooperative activities, the identification of language features that are appropriate for describing cooperation, the ability to write declarative specifications in the language, and the existence of a well-defined interface to LOTOS/TM. We elaborate on these ideas below.

CoCoA has been designed as a specific language to describe arbitrary situations of cooperation. Some cooperative situations require *a priori* prescribed procedures or protocols for the way the work is organised. By procedure, we mean a planned activity by multiple actors. The level of nesting of procedures is not fixed by CoCoA, but is chosen by the specifier according to what the specific cooperation seems to require. Other situations demand a level of freedom of activity for the actors involved; CoCoA accommodates this by allowing the specifier to list the possible actions, but without further restrictions with respect to their ordering or interleaving. Any mix of prescriptive procedures and 'freehand' activities can be described in the language: the CoCoA steps are somehow prescriptive, and allow to organise the work, but the activities within an elementary step can be considered 'freehand'. This freedom can be fine-tuned by execution rules, which also provide the link to the transaction model.

A second language design consideration was that we wanted to identify sensible features for cooperation and make sure that provision for them was made. To this end, we identified data, users and steps, data operations, communications, data exchange operations, and execution rules as important concepts for a language like CoCoA. We expect that this set of concepts fits well with the TransCoop application areas, and also that it is general enough. However, we imagine that after some realistic use of the language we may want to improve upon its design. For example, we certainly envision proposals for improved communication, awareness, and negotiation mechanisms. As mentioned earlier, this will depend to some extent on the existence of similar facilities in the transaction model.

Another consideration for CoCoA was to have, as much as possible, declarative specifications. There is good reason for declarativeness, as long as the mechanism remains intuitively appealing: declarative rules are understandable and hence verifiable to some extent by the specifier, they provide a level of modularity to the specification, and finally, they bring automatic verifications within reach. Our work on the TransCoop Specification Environment in Deliverables IV.4 and IV.5 will hopefully reveal what we can provide for the support of such automatic verifications, and surely also what we cannot support. (This, in turn, may lead to some enhancements to the language.)

Last, but not least, we should mention the use of LOTOS/TM as the basis for CoCoA's design. We are dedicated to build on our previous experiences with LOTOS and TM, and the tool sets that have in the past been built for them. During the TransCoop project, these tool sets will be exploited towards tools for cooperation design. The design of CoCoA has obviously been influenced by the language characteristics of LOTOS and TM. We have also taken the point of departure that openness and modularity—for example of the data definition part of CoCoA, which is now fixed to the TM language—is not a research issue to be taken up in the TransCoop project. Such openness and modularity we believe to be more an exploitation issue of the project's results.

9.2 Comparison to Workflow approaches

It is the procedural part of CoCoA that comes most close to the flow graphs that are found in workflow approaches. There are a number of differences that we feel should be emphasised. In CoCoA, there is a less strict connection between steps and actors than is typically found in workflow approaches. A step can have any number of actors, and actors can be involved in more than one step (of one and the same scenario instance) at the same time. This difference also becomes apparent when we realise that CoCoA promotes less rigid styles of specification. Suppose, for example, that in the CDA scenario, we want to specify that the editor can also terminate a task for writing a chapter. Following a flow graph approach, one is tempted to add another interrupt transition from the editor's work step to the task step (both inside the writing step), which will cause the task step to be terminated. A more appropriate way of specifying this is to add an enabling expression inside the task step definition, which specifies that the editor can also cause an outgoing transition on the `compl` out point, and thus terminate the task step.

In CoCoA, much of the control flow typically seen in a flow graph is moved inside the steps definitions. Workflow approaches may allow transitions to take time, or to perform conditional branching that is dependent on certain values. This can lead to complex or ad hoc semantics. In CoCoA, conditional transitions are specified by adding an **iff**-clause to an enabling rule that can cause the transition to occur.

9.3 Related work on execution rule specification

In this section, we compare our work to related transaction model approaches that offer mechanisms at the specification level for the definition of execution rules. We do not discuss cooperative transaction models, only proposed language mechanisms.

9.3.1 Comparison to Transaction Groups

[NoRZ92] describes *transaction groups* as a formal notation for the specification of cooperative transactions. An LR(0) grammar is used to describe a transaction group's correctness criteria in terms of valid histories. The alphabet of the language defined by the grammar consists of operation invocations, which are issued by members of the transaction group. The LR(0) grammar specifies a number of rules, which define *patterns* (interleavings of operations that must happen) and *conflicts* (interleavings of operations that are not allowed). A history is considered correct when it conforms to all of the pattern rule specifications and contains no conflicts. Checking a history with respect to an LR(0) grammar can be done in linear time.

Like our execution rules, the LR(0) grammars are meant to be enforced *online* to avoid throwing away work after it has been done (i.e., an operation cannot be permitted and then later rejected). The algorithm is thus able to recognise correct histories as well as valid prefixes of correct histories. This is called a *viable prefix property* of the grammar.

LR(0) grammars are more expressive than our ordering rules in CoCoA. For example, the following transaction group pattern cannot be expressed in CoCoA:

$$
\begin{aligned}
&S \rightarrow B \\
&B \rightarrow AB \mid A \\
&A \rightarrow \langle any, inc, count\rangle\, B\, \langle any, dec, count\rangle \\
&\qquad\quad \mid \langle any, inc, count\rangle\, \langle any, dec, count\rangle
\end{aligned}
$$

This context-free grammar rule describes sequences of operation invocations that contain an equal number of increment and decrement operations. The string of symbols '$\langle any, inc, count\rangle$' specifies an increment operation done by any member of the transaction group on object *count*; the string of symbols '$\langle any, dec, count\rangle$' specifies a decrement operation, also done by any member of the transaction group on the same object. The above example cannot be expressed by CoCoA ordering rules, which are regular, not context-free. It can, however, be described by LOTOS/TM.

The following behaviour expression allows increment and decrement operations to be applied to a count object:

```
B[count](0)
where
   process B[count](v: int) : exit :=
         count !"inc" ; B[count](v+1)
      [] [v>0] → count !"dec" ; B[count](v-1)
      [] [v=0] → exit
   endproc
```

Operation invocations are modelled as gate events. The process can terminate the sequence of events whenever the value of the count is zero.

There is a trade-off involved here. The transaction group grammar rule given above describes histories which have an equal number of increment and decrement operations. However, an LR(0) grammar is not sufficient if we modify the example slightly. Suppose we include in the operations an additional parameter value that indicates an amount by which the counter is to be incremented or decremented. The set of operation invocation histories for which the counter is left at zero on completion can no longer be described by a context-free grammar. It can, however, be described by a LOTOS/TM process, very similar to the one above.

9.3.2 Comparison to intertask dependency primitives

Two primitives have been proposed by [Klei91] for the specification of *intertask dependencies*:

1. $e_1 < e_2$: If both e_1 and e_2 occur, then e_1 must precede e_2.
2. $e_1 \rightarrow e_2$: If e_1 occurs, then e_2 must also occur. No ordering is implied on the occurrences of e_1 and e_2.

These primitives are used to specify constraints on the occurrence and temporal order of certain *significant events* [ASSR93]. We refer to these primitives as the *Klein primitives*. Well-known dependencies between two transactions, A and B, can be expressed using these primitives: the temporal ordering primitive is used to express a *commit dependency*, `commit_A` $<$ `commit_B` (if both transactions A and B commit, then A commits before B commits), and the existence dependency primitive is used to express an *abort dependency*, `abort_B` $\rightarrow$ `abort_A` (if B aborts, then A must also abort).

To compare the Klein primitives with the ordering rules of CoCoA, we first make an observation about significant events in CoCoA, and then we look at an example that uses repetition. The Klein primitives are used to specify dependencies between so-called significant events. As pointed out above, significant events are typically operations such as the commit or the abort of a transaction. However, there are no commit or abort operations at the specification level in CoCoA. Of course, the Klein primitives are applicable to other kinds of significant events. In [ASSR93], the following is suggested:

> Note that we allow dependencies of the form $E_1 \rightarrow E_2$, where E_1 and E_2 are general expressions. An expression E can be formally treated as an event by identifying it with the first event occurrences [sic] that makes it definitely true. For example, $e_2 \rightarrow e_3$ is made true as soon as e_3 or the complement of e_2 occurs.

In this more general situation, significant events in CoCoA are the occurrences of operation invocations (data operations, as well as communication operations and data exchange operations). The ordering of data operations is guided by the execution rules of a CoCoA specification, but actors are also free to sequence together the operations dynamically—provided that consistency (as formally defined in the database schema) is preserved, and provided that

the operation invocations are permitted by the organisational view. The Klein primitives may be used as an alternative specification mechanism to describe execution rules (as example is given below). However, just as the execution rules of a CoCoA specification do not fully describe the behaviour of a cooperative scenario, additional specification mechanisms would also be needed if Klein primitives were used instead.

The Klein primitives offer a declarative way of specifying the ordering of events. However, the primitives are not well suited for environments in which operations can be repeated. From [ASSR93], it is assumed that an event can occur at most once in any possible execution of the system. To get around this, the authors propose that new identifiers can be assigned to re-executed events during system operation. Compared to the CoCoA ordering rules, the Klein primitives are less declarative when repetition is involved. An example will illustrate this. Consider the following ordering rule for editing a specific chapter value `c`:

order `(addChapter(_,c,_); editChapter(_,c,_)*; delChapter(_,c))*`

To express this rule using the Klein primitives, we use integer subscripts to identify each operation. We use the same '_' notation for anonymous parameters. A list of dependencies such as the following is needed to describe the above CoCoA rule using Klein primitives:

$$
\begin{aligned}
&\forall i \geq 0 : \\
&\quad \texttt{delChapter(\_,c)}_i < \texttt{addChapter(\_,c,\_)}_{i+1} \\
&\wedge\ \texttt{addChapter(\_,c,\_)}_{i+1} \rightarrow \texttt{delChapter(\_,c)}_i \\
&\wedge\ \texttt{addChapter(\_,c,\_)}_i < \texttt{editChapter(\_,c,\_)}_{i,0} \\
&\wedge\ \texttt{editChapter(\_,c,\_)}_{i,0} \rightarrow \texttt{addChapter(\_,c,\_)}_i \\
&\wedge\ (\ \forall j \geq 0\text{: } \texttt{editChapter(\_,c,\_)}_{i,j} < \texttt{editChapter(\_,c,\_)}_{i,j+1}\ ) \\
&\wedge\ (\ \forall j \geq 0\text{: } \texttt{editChapter(\_,c,\_)}_{i,j+1} \rightarrow \texttt{editChapter(\_,c,\_)}_{i,j}\ ) \\
&\wedge\ (\ \forall j \geq 0\text{: } \texttt{editChapter(\_,c,\_)}_{i,j} < \texttt{delChapter(\_,c)}_i\ ) \\
&\wedge\ \texttt{addChapter(\_,c,\_)}_i < \texttt{delChapter(\_,c)}_i \\
&\wedge\ \texttt{delChapter(\_,c)}_i \rightarrow \texttt{addChapter(\_,c,\_)}_i
\end{aligned}
$$

The semicolon operator ';' in a CoCoA ordering rule translates to existence and ordering primitives. It should be obvious that the CoCoA rule is more concise. Of course, macro definitions can be used for combinations of the Klein primitives.

In [ASSR93], the translation of an intertask dependency into a dependency automaton that enforces the dependency is described. To be run-time enforceable, it is prerequisite that the significant events in a dependency have certain properties. For example, to enforce the dependency $e_1 \rightarrow e_2$, event e_2 must be *forcible* (it must be forced to happen when e_1 has happened first), or e_1 must be be *delayable and rejectable* (it must be delayed until e_2 happens, and rejected if e_2 never happens).[24] Dependency automata may be constructed manually, or by a synthesis technique, described in [AtSR92], that first translates the dependencies to the temporal logic CTL [Emer90,EmCl82]. For the specification designer, the temporal logic translations and the resulting dependency automaton are comparable in complexity to the processes that result from our LOTOS/TM translation of the CoCoA execution rules.

[24] We point out that these properties take on different meanings in CoCoA, when we realise that the operations in a workspace history are invoked by an actor. Such operations are not "forcible." However, termination constraints can be used in CoCoA to check whether an operation has been done, before allowing a transition out of a step to occur.

References

[AhUl77] A. V. Aho & J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977.

[AtSR92] P. C. Attie, M. P. Singh & M. Rusinkiewicz, "Specifying and enforcing intertask dependencies," MCC Technical Report Carnot-245-92, December 1992.

[ASSR93] P. C. Attie, M. P. Singh, A. Sheth & M. Rusinkiewicz, "Specifying and enforcing intertask dependencies," *Proceedings of the 19th International Conference on Very Large Databases*, Dublin, Ireland (August 1993).

[BaRa92] B. R. Badrinath & K. Ramamritham, "Semantics-based concurrency control: Beyond commutativity," *ACM Transactions on Database Systems* Vol. 17 (March 1992), 163–199.

[BBBB95] R. Bal, H. Balsters, R. A. de By, A. Bosschaart, J. Flokstra, M. van Keulen, J. Skowronek & B. Termorshuizen, "The TM Manual; version 2.0, revision e," Universiteit Twente, Technical report IMPRESS / UT-TECH-T79-001-R2, Enschede, The Netherlands, June 1995.

[BaBZ93] H. Balsters, R. A. de By & R. Zicari, "Typed sets as a basis for object-oriented database schemas," in *Proceedings Seventh European Conference on Object-Oriented Programming, July 26–30, 1993, Kaiserslautern, Germany, LNCS #707*, O. M. Nierstrasz, ed., Springer-Verlag, New York–Heidelberg–Berlin, 1993, 161–184.

[BaFo91] H. Balsters & M. M. Fokkinga, "Subtyping can have a simple semantics," *Theoretical Computer Science* 87 (September, 1991), 81–96.

[BoBr87] T. Bolognesi & E. Brinksma, "Introduction to the ISO Specification Language LOTOS," *Computer Networks and ISDN Systems* 14 (1987), 25–59.

[Emer90] E. A. Emerson, "Temporal and Modal Logic," in *Handbook of Theoretical Computer Science* #Vol. B, J. V. Leeuwen, ed., 1990.

[EmCl82] E. A. Emerson & E. Clarke, "Using branching time temporal logic to synthesize synchronization skeletons," *Science of Computer Programming* Vol. 2 (1982), 241–266.

[EvFa94] S. Even & F. Faase, "Merge options for LOTOS and TM," 7 October 1994, Universiteit Twente, Project TransCoop, ESPRIT LTR 8012, Deliverable TC/REP/UT/D4-1/009.

[EvFB95] S. J. Even, F. J. Faase & R. A. de By, "Language features for cooperation in an object-oriented database environment," Memoranda Informatica 95-40, Universiteit Twente, Enschede, The Netherlands, 28 September 1995.

[FaBy95] F. J. Faase & R. A. de By, "An Introduction to CoCoA," 15 September 1995, Universiteit Twente, Project TransCoop, ESPRIT LTR 8012, /TC/TR/UT/WP4/030.

[GMD-I95] GMD-IPSI, "VODAK V4.0 User Manual," Arbeitspapiere der GMD 910, Technical Report GMD, Darmstadt, April 1995.

[Herl90] M. Herlihy, "Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types," *ACM Transactions on Database Systems* Vol. 15 (March 1990), 96–124.

[ISO87] ISO–Information Processing Systems–Open Systems Interconnection, *LOTOS–A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, DIS 8807, 1987.

[Klei91] J. Klein, "Advanced rule driven transaction management," *Proceedings of the 36th IEEE Computer Society International Conference*, San Francisco, California, USA (March 1991).

[Levy91] E. Levy, "Semantics-based recovery in transaction management systems," Technical Report TR-91-29, University of Texas at Austin, August 1991.

[LMWF94] N. Lynch, M. Merritt, W. Weihl & A. Fekete, *Atomic Transactions*, Morgan Kaufmann Publishers, 1994.

[NoRZ92] M. H. Nodine, S. Ramaswamy & S. B. Zdonik, "A cooperative transaction model for design databases," in *Database Transaction Models for Advanced Applications*, A. K. Elmagarmid, ed., Morgan Kaufmann Publishers, San Mateo, CA, 1992, 53–85.

[RKTW95] M. Rusinkiewicz, W. Klas, T. Tesch, J. Wäsch & P. Muth, "Towards a cooperative transaction model: The cooperative activity model," *Proceedings of the 21st International Conference on Very Large Databases*, Zurich, Switzerland (September 1995).

[Spel95] D. Spelt, "A Proof Tool for TM," Universiteit Twente, 26 July 1995, M.Sc. thesis.

[VSSB91] C. A. Vissers, G. Scollo, M. van Sinderen & E. Brinksma, "Specification styles in distributed systems design and verification," *Theoretical Computer Science* 89 (1991), 179–206.

[WäKl96] J. Wäsch & W. Klas, "History merging as a mechanism for concurrency control in cooperative environments," *To appear in the proceedings of the 6th International Workshop on Research Issues in Data Engineering: Interoperability of Nontraditional Database Systems (RIDE-NDS '96)* , New Orleans, Louisiana, USA (February 1996).

[Weih88] W. E. Weihl, "Commutativity-based concurrency control for abstract data types," *IEEE Transactions on Computers* Vol. 37 (1988), 1488–1505.

A Complete grammar for CoCoA

This appendix gives the complete COCOA syntax. A preliminary parser for the language has been implemented.

```
Cocoa_spec ::=
   "scenario" scenario_name
   ("includes"
       TM_module_name LIST)OPT
   ("data" "types"
       data_type_name LIST)OPT
   ("data" "operations"
       data_operation_signature LIST)OPT
   ("workspace" "types"
       workspace_type_def SEQ)OPT
   "user" "types"
       user_type_def LIST
   "communications"
       communication_prim_def LIST
   "data" "exchange" "operations"
       data_exch_oper_def SEQ
   procedure_def
   "data" "operation" "order"
       order_rule_def SEQ
   "history" "rules"
       history_rule_def SEQ
   "end" scenario_name.

data_operation_signature ::=
   data_oper_name "(" data_type_name LIST ")".

workspace_type_def ::= workspace_type_name "="
                    "{" data_oper_name LIST "}".

user_type_def ::= user_type_name
                  ("isa" data_type_name)OPT
                  ("using" workspace_type_name)OPT.

communication_prim_def ::=
   comm_prim_name "(" data_type_name LIST ")".

data_exch_oper_def ::=
   data_exch_oper_name
   "(" par_name ":" tm_type_name LIST ")" "="
   "select" operation_elem LIST.
```

```
procedure_def ::=
   "procedure" "(" user_list ")" point_list
   "begin" workspace_def
           step_stmt SEQ "end".
user_list ::= ( user_name ":" user_type_name
              | user_set ":" ℙ user_type_name) LIST
step_stmt ::= step_def | trans_def | par_clause | enable_expr.
point_list ::= "[" "in" point_def LIST
                   "out" point_def LIST
                 ( "signal" point_def LIST) OPT
                 ( "interrupt" point_def LIST) OPT "]".
point_def ::= point_name
              ("(" tm_type_name LIST ")") OPT.
step_def ::= "step" step_name point_list
             "begin" step_stmt SEQ "end".
trans_def ::= "on" from_point_expr "do" point LIST.
from_point_expr ::=
        point
      | "(" from_point_expr ")"
      | from_point_expr "or" from_point_expr
      | from_point_expr "and" from_point_expr.
par_clause ::= "parallel" ("(" tm_type_name LIST ")") OPT
                 step_stmt SEQ
               "endpar".
point ::= (step_name "." ("(" par_key_var_name LIST ")") OPT) OPT
          point_name ("(" point_var_name LIST ")") OPT.
workspace_decl ::= "workspace"  workspace_name ":" workspace_type_name.
enable_expr ::=
    "on" point_with_params "enable"
    ( for_part OPT
      ( (user_name | for_param_name)
        ":" ( en_data_operation
            | en_data_exch_oper ) LIST
      | en_comm_oper ) SEQ
    ) SEQ.
    "endon"
for_part ::= "for" (for_param_name LIST "in" TM_expr) LIST
             ("iff" TM_expr) OPT  "allow".
en_data_operation ::=
   data_oper_name "(" (TM_expr | "_") LIST ")".
en_data_exch_oper ::=
    "import" data_exch_oper_expr "from" workspace_expr |
    "export" data_exch_oper_expr "to" workspace_expr.
data_exch_oper_expr ::=
    data_exch_oper_name "(" TM_expr | "_") LIST ")".
workspace_expr ::=
   user_name | for_param_name | shared_workspace_name | "_".
```

```
en_comm_oper ::=
   "when" (user_name | for_param_name)
   "issues" comm_prim_name "(" (TM_expr | "_") LIST ")"
   ("iff" TM_expr) OPT
   "do" point.

order_rule_def ::=
   rule_name OPT
   ("forall" (param_name LIST ":" tm_type_name) LIST) OPT
   ("exist" (param_name LIST ":" tm_type_name) LIST) OPT
   "order" order_expr.
order_expr ::= operation_elem state_mark OPT
             | order_expr CHAIN ";"
             | "(" order_expr ")"
             | order_expr "*"
             | order_expr "+"
             | order_expr CHAIN "|"
             | "" order_expr LIST "".
operation_elem ::= operation_name "(" (param_name | "_") LIST ")".
state_mark ::= '[' TM_expr ']'.
TM_expr ::=   (standard TM expressions)
          | "exists" rule_name ("(" TM_expr LIST ")") OPT
            "in" workspace_expr
          | "query" rule_name ("(" TM_expr LIST ")") OPT
            "on" workspace_expr.

history_rule ::=
   forall_part
   "non-commutative" operation_elem
   ("and" | "before") operation_elem
   ("iff" tm_expr) OPT.
operation_elem ::= operation_name "(" param_name LIST ")".
```

B Full example

This appendix gives the complete CoCoA specification of the CDA scenario. The specification does not include the examples of advanced features that were discussed in Section 6.

scenario `write_document` **includes** `document_schema`
data types
`chapter, text, annotation`

database operations
`addChapter(actor, chapter, text)`

```
  delChapter(actor, chapter)
  editChapter(actor, chapter, text)
  addAnnotation(actor, chapter, annotation)
  remAnnotation(actor, chapter ,annotation)

history types
  cda = { addChapter, editChapter, delChapter,
          addAnnotation, remAnnotation }

user types
  referee using cda,
  actor,
  editor isa actor using cda,
  author isa actor using cda

data exchange operations
  Annotations(c : chapter) =
  select addAnnotation(_,c,_)

  Chapter(c : chapter) =
  select addChapter(_,c), editChapter(_,c,_),
         delChapter(_,c)

communications
  introWritten(),
  introOkay(),
  reviseIntro(),
  abortWriting(),
  startTask(chapter, ℙ author),
  completeTask(chapter),
  completeWriting(),
  documentReady()

procedure (ref : referee,
           ed : editor,
           as : ℙ author)[in start out cancel, done]
begin
  workspace  document : history cda

  step prepare[in start  out done, rejected]
  begin

    step write_proposal(in start, revise  out done]
    begin
      on start enable
        /* The data operations the editor can do: */
        ed : addChapter(ed, title, _),
```

```
            addChapter(ed, intro, _)
      endon
      on start, revise enable
        ed : editChapter(ed, title, _),
             editChapter(ed, intro, _),
        /* Export operations the editor can do: */
             export Chapter(title) to document,
             export Chapter(intro) to document,
        /* Other actions the editor can do, e.g., leave this step: */
        when ed issues introWritten()
        iff    query document on chapter_rule("intro")
                        = "edited"
           and query document on chapter_rule("title")
                        = "edited"
        do done
      endon
    end

    step accept_proposal(in start  out acc, rej, rev)
    begin
       on start enable
          when ref issues introOkay() do acc
          when ref issues reviseIntro() do rev
          when ref issues abortWriting() do rej
       endon
    end

    on start do write_proposal.start
    on write_proposal.done do accept_proposal.start
    on accept_proposal.acc do done
    on accept_proposal.rev do write_proposal.revise
    on accept_proposal.rej do rejected
  end

  step writing[in start, out done]

    parallel(ch : chapter)

      step task[in start(ℙ author), out compl]
      begin
        on start(authors) enable
          for a in authors allow
            a : addChapter(a, ch, _),
                editChapter(a, ch, _),
                addAnnotation(a, ch, _),
                remAnnotation(a, ch, _),
                import Chapter(ch) from document,
```

```
                export Chapter(ch) to document
            when a issues completeTask(ch)
            iff for all an : annotation
                  exist annotation_rule(ch, an) in document
                  implies query annotation_rule(ch, an)
                                on document
                                  = "processed"
            do compl
          for a1, a2 in authors  allow
            a1 : import Chapter(ch) from a2,
                  import Annotation(ch) from a2
        endon
      end

    endpar

    on start enable
       ed : import Chapter(_) from _,
            export Chapter(_) to document,
            ....
       when ed issues startTask(c, as) do start_task(c, as),
       when ed issues completeWriting() do done
    endon

    on editors_work.start_task(ch, as) do task(ch).start(as)
    on task(ch).compl do editors_work.task_compl(ch)
  end

  step complete[in start, out done]
  begin
    ...
  end

  on start(ed) do prepare.start
  on prepare.done do writing.start
  on prepare.rejected do cancel
  on writing.done do complete.start
  on complete.done do done
end

data operation order
 chapter_rule :
   forall c : chapter
   order addChapter(_,c,_);
         editChapter(_,c,_) ["edited"]*;
         delChapter(_,c)
```

```
annotation_rule :
  forall c : chapter, an : annotation
  order addChapter(_,c,_);
       (addAnnotation(_,c,an); delAnnotation(_,c,an) ["processed"])*;
       delChapter(_,c)
```

history rules

```
forall a : agent, c : chapter, t1, t2 : text
non-commutative editChapter(a,c,t1) and editChapter(a,c,t2)

forall a : agent, c : chapter, t : text, an : annotation
non-commutative remAnnotation(a,c,an) before editChapter(a,c,t)
non-commutative editChapter(a,c,t) before addAnnotation(a,c,an)

end write_document
```

Index

