TransCoop

ESPRIT Basic Research Action 8012

DELIVERABLE IV.4:
THE TRANSCOOP SPECIFICATION ENVIRONMENT

DATE OF ISSUE	May 14, 1996
AUTHOR(S)	Susan J. Even, Frans J. Faase, Olli Pihlajamaa, Rolf A. de By
EDITOR(S)	Susan J. Even
ORIGIN	UNIVERSITEIT TWENTE
STATUS	FINAL
CONFIDENTIALITY	TRANSCOOP
RELATED ITEMS	TransCoop Technical Annex
FILING CODE	TC/REP/UT/D4-4/032
ABSTRACT	This document describes the modules of the TRANSCOOP Specification Environment (TSE).
KEYWORDS	
APPROVED BY	TMG
DISTRIBUTION	CEC; within TRANSCOOP

DOCUMENT HISTORY

version	date	reason
2.0	14 May 1996	Final.

Preface

This document describes the components of the TRANSCOOP Specification Environment (TSE). Our strategy in preparing this deliverable has been to describe the tools that we plan to implement, rather than an idyllic collection of tools in the clouds. Because of this, we have made an effort to describe the tool set components in a tangible way, by making use of TM and LOTOS/TM formal specifications when appropriate. As an added benefit, these specifications can later be simulated as part of the cooperative application design process.

While investigating the tool set components, we have been required to take a closer look at the COCOA language description in [FaEB96]. In doing this, we have become aware of a number of typos in the appendices of that deliverable. For this reason, we include revised versions of the CDA scenario specification example, and the COCOA syntax in the appendices of this deliverable. The COCOA syntax also reflects our recent refinement of the language constructs, and our work on the parser/type-checker for the language.

[FaEB96] posed the restriction that ordering rule expressions in COCOA should be deterministic. We claimed in a footnote (Page 39) that this did not impose restrictions on the expressiveness of the rules. We further claimed that any non-deterministic rule could be rewritten into a deterministic rule. This is, however, not true. (See [BrWo92].) Although the deterministic rule restriction is not applicable if Finite State Automaton (FSA) representations are used, we believe that the readability of the ordering rule expressions in textual format outweighs the limitations imposed by this restriction.

Please note: This deliverable is intended to be read as a follow-up to Deliverable IV.3 [FaEB96]. Familiarity with the features of COCOA is assumed; this avoids repetition.

Acknowledgements

Thanks to all TRANSCOOP partners for helpful comments on earlier drafts of this document. Thanks to David Spelt for discussing the internals of his TM Proof Tool with us. Thanks to Jan Flokstra for demonstrating the TM Abstract Machine to us.

Contents

1 Introduction

This document describes the design of the TRANSCOOP Specification Environment (TSE); its purpose is to identify the TSE components, and their functionalities. The components of the TSE are intended to help the specifier of a cooperative application work with the COCOA language in the early (conceptual), as well as the late (testing) phases of the application design. The TSE includes a graphical specification editor, a parser/type-checker, a verification toolbox, a simulation environment, and compilers to the TRANSCOOP run-time environment. The compilers are described elsewhere (see [BLPV96]); the remaining components are described, respectively, in Sections 2, 3, 4, and 5.

The approach we have taken in the TSE design is to use executable semantics descriptions as a means to verification. The verification toolbox includes mappings from COCOA to TM and LOTOS/TM. (Refer to the previous deliverables for Work Package IV for descriptions of these languages [EvFB95,FaEB96].) The mappings formalise various aspects of a COCOA specification in such a way that they can be analysed independently. (See Section 4 for details.) The TM and LOTOS/TM specifications that result from the mappings are given as input to the simulation environment, which includes a TM abstract machine, and a LOTOS/TM simulator. Correctness of the mapping specifications is left to the implementation phase of the tool set design, due to time constraints on the deliverable completion.[1]

The TSE described in this deliverable differs in some respects from our initial vision of it in [BLPV95], primarily because of the introduction of COCOA on top of LOTOS/TM. Refinement of the original TSE design and the overall structure of the new TSE are discussed in Section 1.1. Some language constructs of COCOA have also been refined since their introduction in [FaEB96]. These refinements are discussed in Section 1.2.

1.1 Refinement of the TSE

The original Technical Annex of the TRANSCOOP project identified the language combination LOTOS/TM as the specification language platform for the project. An initial architecture of the TRANSCOOP Specification Environment was at that time also foreseen, and was in fact defined in [BLPV95] (pp. 45, 52–55). This architecture is shown in Figure 1. This initial architecture was conceived at the time when the need for a higher-level and more user-friendly language such as COCOA had just been recognised. For this reason, it only identified the different components and the (possible) information flows between them. Now that COCOA has been established, and its properties are better understood, we present a more detailed architecture in Figure 2. This figure reflects how the tools within the toolset are used, rather than how the software components relate to each other.

Walk-through The designer of a cooperative scenario can use the Graphical Specification Editor to produce a graphical representation of a scenario specification. This specification

[1] As an aside, we point out that the TSE implementation phase involves formal specification, as well as the more typical C and C++ coding.

can serve as an input to the CoCoA Parser/Type-checker, which performs static semantics checks. The output of the Parser/Type-checker, the CoCoA specification in the figure, serves as an input to most of the other tools.

A scenario specification can be divided into three parts, which can be analysed independently to a certain extent. These parts are: the organisational view, the execution rules, and the commutativity rules. (The latter two of these are part of the transactional view.) The toolset allows the specifier to analyse these parts in isolation, which, in view of the complexity of the combined behaviour, is a benefit for locating errors.

The specifier can investigate the behaviour of the organisational aspects by using the tool for mapping the organisational view to LOTOS/TM. The LOTOS/TM Simulator can be used to see which operations are allowed by the organisational view at each point during the execution of a scenario instance.

In order to verify the combined behaviour of the execution rules, the execution rules can be mapped to a LOTOS/TM specification. The LOTOS/TM Simulator allows the specifier to view the tree of all allowable operation invocation sequences, to perform a reachability test in order to detect possible deadlocks, and to check whether certain breakpoints can be reached, starting with a given history.

The commutativity rules of a specification need to be correct with respect to the underlying database schema. The Commutativity Analysis Tool can help the specifier to verify this.

Finally, after the different aspects of a scenario specification have been analysed in isolation, they can be analysed in combination, using the View Integration Tool, and the LOTOS/TM Simulator.

After the specification has been sufficiently verified, the CoCoA to CoAct and CoCoA to CM Script compilers can be used generate code for the run-time environment.

In the remaining sections of this report, we elaborate on the functionality of:

- the Graphical Specification Editor,
- the Parser/Type-checker,
- the Verification Toolbox (inclusive of the CoCoA2LTM compiler in Figure 1), and
- the Simulation Environment.

We include the discussion of the CoCoA2LTM compiler in the Verification Toolbox components, because its initial design has indicated that it should be integrated in the respective verification tools. Indeed, this compiler will not exist as a stand-alone component, but rather as a front-end that allows to extract the relevant parts of a CoCoA specification for the specific verification purpose. The compiler algorithms are thus scattered over a number of tools in the toolbox, namely the execution rules analysis, the organisational view analysis and the

transactional view analysis. We stress that the Specification Environment has as its goal *not* a complete simulation of the TRANSCOOP run-time system, but rather simulations of those parts of a scenario that are difficult to understand fully by a human specifier, with the objective to improve that understanding.

The two compilers to the run-time environment (COCOA to CoAct [KTWP96], and COCOA to CMScript) are not discussed in this document. Although these compilers will be included as components of the TSE, they are highly dependent on the TRANSCOOP run-time environment, which is not the subject of this deliverable. The reader is referred to [BLPV96] for details.

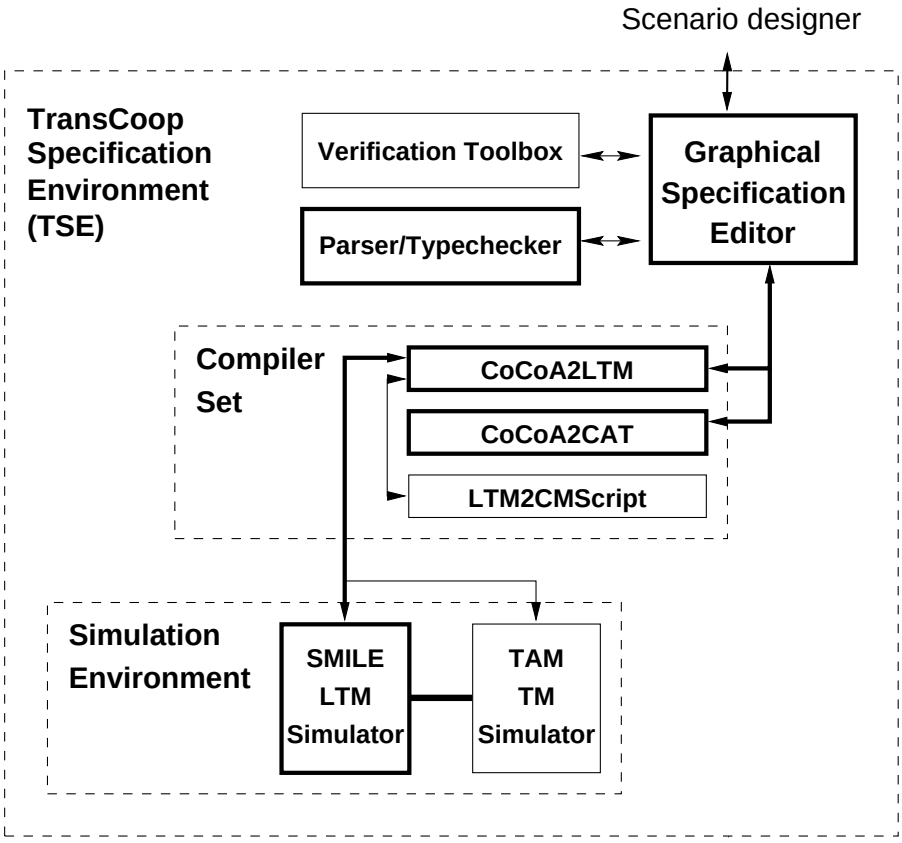


Figure 1: Components of the original TSE (as described in Deliverable III.1).

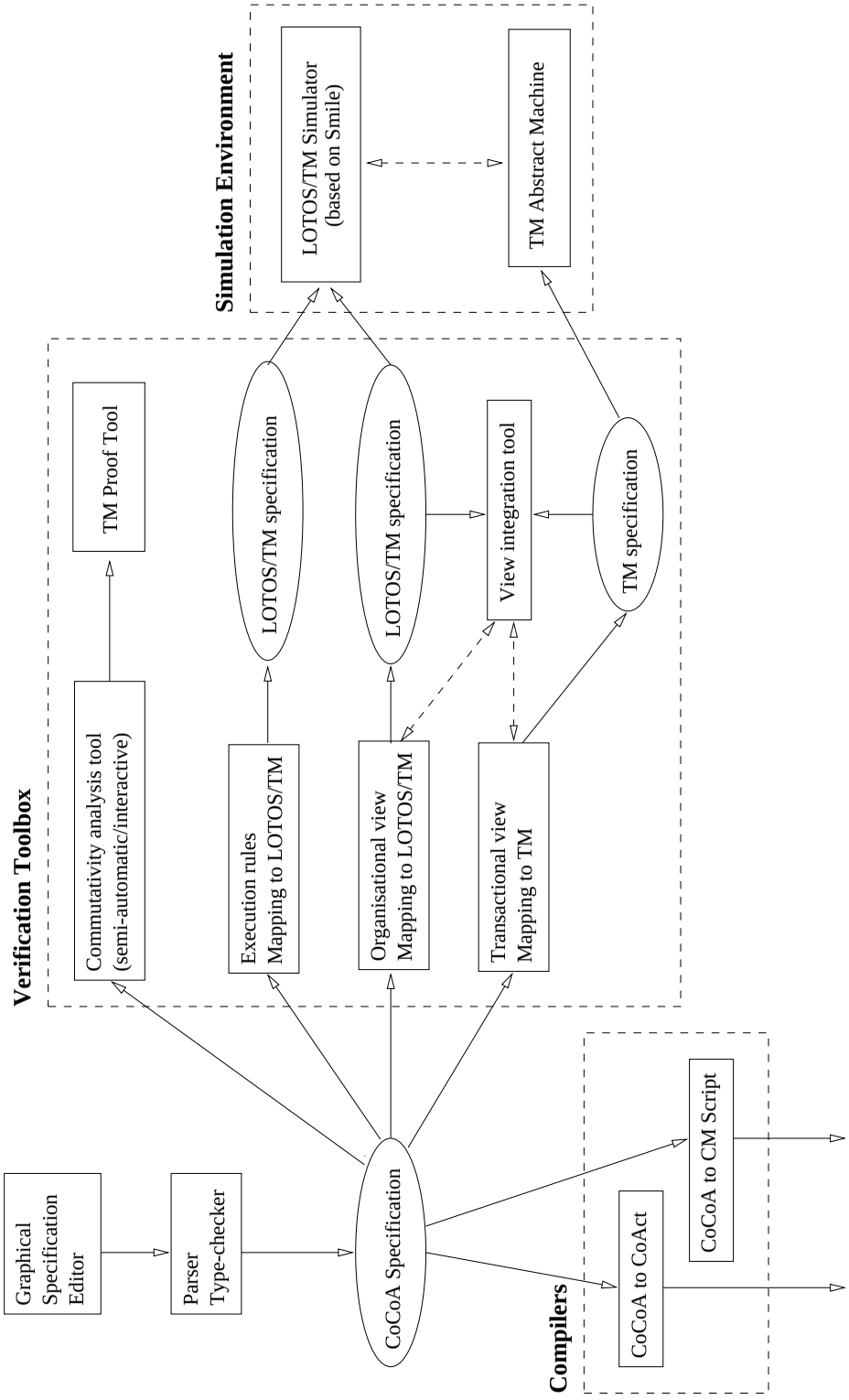


Figure 2: Inside the refined TSE.

1.2 Refinement of CoCoA

The definition of CoCoA has undergone several refinements since the completion of Deliverable IV.3 [FaEB96]. For example, some minor changes to the syntax have been made due to conflicts with the TM syntax. As a precursor to the descriptions of the tool set components, we mention the more significant language refinements here.

Parallel clause identification In [FaEB96], we permitted the definition of parallel clauses with and without value identifications. In the case of parallel clauses without a value identification, all activations of the parallel clause are necessarily identical. This causes problems if a communication operation is invoked from within a parallel clause, since the scenario does not specify which clause activation the communication is part of. To avoid such cases of ambiguity, we now require that all parallel clauses use a value identification. A value identification serves as a mechanism to statically define well-formedness requirements on transitions into and out of (a family of) step activations at run-time.

Labels and optional subexpressions in ordering rules The syntax for specifying labels (termination constraints) and optional subexpressions in ordering rules has been revised. The square brackets are no longer required around labels; a string value may simply be concatenated to the operation pattern that it annotates, as follows:

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

Because the square brackets are no longer used for labels, they will be used for optional subexpressions. Previously, an optional part of an ordering rule was specified using the choice operator ('|') and an empty subexpression, as follows [FaEB96]:

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

Now the optional part of this rule is specified using square brackets instead:

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

Quantification in ordering rules To enhance the expressiveness of the CoCoA execution rules, we introduce a construct for universal quantification over a finite set. This construct takes the following form:


```
forall param_name in TM_set_expr
order ...
```

It can be used as follows:

```
forall c in { "intro", "conclusions" }
order addChapter(ed,c,_) ; editChapter(ed,c,_)
```

We count on the set `TM_set_expr` being finite. Indeed, a different mapping to LOTOS/TM is used, depending on whether the quantification is finite or infinite. (See Section 4.1.) It is easy to see that enumerated sets can be used in this context, as shown above. We cannot allow a general TM expression, complete with retrieval (query) method applications to be used, because the expression's value must not change once the scenario has started execution.

Complex transitions In [FaEB96], we deferred giving a semantics to so-called **and** transitions, which have the following structure:

```
on A.done and B.done do done
```

Apparently, steps `A` and `B` should both make transitions on their `done` interaction points before this transition can take place. (This transition would be defined in their parent step.) At this point in the development of CoCoA, the semantics of the **and** transition seems to be problematic. Although such a construct seems natural at first glance, closer inspection reveals that there is no "correct" way to interpret it: its interaction with the other features of the language causes problems. The reasons for this are explained below.

Different situations may give rise to the use of **and** transitions. Typically, an **and** transition is used to join (synchronise the completion of) two parallel steps. For this to happen, both steps must be ready to terminate before the transition can take place. But most control flows in a scenario definition are initiated by a communication operation. In this case, the communications that enable the source interaction points of an **and** transition should take place before it can happen. Two choices are possible regarding the originating communications and the subsequent **and** transition: the originating communications might be forced to synchronise, or they might be allowed to happen asynchronously. In the former case, the **and** transition is subsumed by a complex communication protocol. In the latter case, the asynchronous communication situation interferes with the semantics of transitions. Observe that between a communication (inside one of the parallel steps) and an interaction point (named in the **and** transition), a series of transitions may be enabled. Such a series of transitions, once initiated by a communication, could be interrupted by an **and** transition, if the other interaction point of the **and** transition is not enabled. For this reason, an asynchronous semantics is not desirable.

We must clarify whether the **and** transition is necessary as a control flow structure (as it was introduced in [FaEB96]), or as a communication/negotiation protocol. An **and** transition

seems to indicate a *conditional* control flow, dependent on the state of the scenario. However, the idea of "state" can be understood in a number of ways: (i) in terms of what data operations have been done, (ii) in terms of what communication (or data exchange) operations have taken place, or (iii) in terms of the data in the workspaces. In any case, the state must be determinable from the data structures maintained by the scenario. The first case is similar to the idea of breakpoint and termination constraints used in the execution rules. In this case, the **and** transition is dependent on certain operations having been done in the parallel steps leading up to the transition. The ability to query breakpoint and termination constraints is already provided by the **iff** clause of the **when** construct for communications. The second case is analogous, but it would involve knowledge of the communications and/or data exchanges done so far. As it is, communication and data exchange operations are not logged in CoCoA (i.e., they are not stored in the workspace histories). The third case would involve querying the workspace data directly; this is currently not supported in CoCoA.

From the above discussion, it should be clear that the semantics of an **and** transition might be any of a number of things. We conjecture that in some cases, an **and** transition can be rewritten (by using additional steps, interaction points, and transitions) to avoid using it in the specification. In other cases, extended communication protocols (see [FaEB96]) can be employed to achieve the necessary synchronisation.[2] In summary, it is the specifier who must determine what an **and** transition means (i.e., what exactly is being "and-ed"), before the specification can be revised.

[2]Extended communication protocols are not included in CoCoA at this stage of development. We plan to add them after a first prototype of the TSE has been implemented.

2 The Graphical Specification Editor

The graphical modelling method to be supported by the Graphical Specification Editor (GSE) is called GRACOOP *(A Graphical Method for Cooperative Processes)*. It will serve as a front-end to COCOA [FaEB96]. Not all of the COCOA constructs can be presented graphically, and thus a graphical specification will be annotated with textual information. A goal in the GSE prototype design is to integrate graphical specification and various mechanisms for textual input in order to produce COCOA specifications as easily as possible. Hence, GRACOOP is developed with computer support in mind. Section 2.1 defines the graphical modelling method GRACOOP. Section 2.2 shows how the constructs of the modelling method are mapped to COCOA constructs. Finally, Section 2.3 outlines our planned implementation.

2.1 Description of GraCooP constructs

GRACOOP introduces a basic set of constructs that are used to build a scenario definition. Some of these constructs have a graphical representation which is referred to with the term *graphical construct*. The GRACOOP constructs are chosen and named so that they correspond as close as possible to their associated COCOA constructs. (The COCOA syntax is given in Appendix B.) First, we present the graphical part of GRACOOP; then we show how a graphical specification is annotated with textual information, such as TM types.

2.1.1 Graphical constructs

The core of GRACOOP is the **step diagram** that represents the skeleton of the organisational view of a cooperative scenario. This view consists of *steps* that have a set of **interaction points**, and **transitions** between the points. A step is a control structure that defines a logical unit of work in a scenario by declaring what can be done, when it can be done, and by whom. A step's points identify possible interactions with the step. Transitions define a dependency between the points of two or more different steps. Transitions depict control flow between steps, as well as signal and interrupt events. Points and transitions can be annotated to allow values to be transmitted.

Steps Step structure may be hierarchical. Thus, steps are divided into three basic step types:

1. A **procedure** is the top-level step representing the whole scenario. A scenario consists of exactly one procedure.
2. A **composite step** is a step that consists of constituent substeps, which may be steps or elementary steps (see below).
3. An **elementary step** is a step that is at the leaf-level in the step hierarchy (i.e., it cannot be divided into constituent substeps).

In GRACOOP, the structure of each step (or procedure) is specified by its own step diagram. Thus, step nesting is not possible in a step diagram. A hierarchical definition is achieved by decomposing composite steps into new subdiagrams. Figure 3 illustrates this; it shows the graphical representations of an elementary step and a composite step.

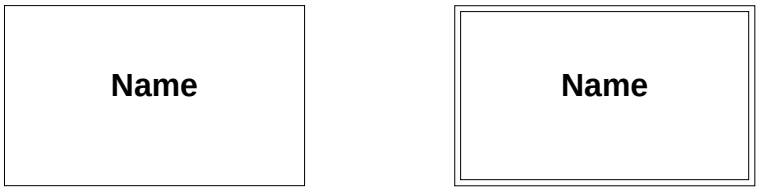


Figure 3: Graphical constructs for an **elementary step** (left) and for a **composite step** (right).

Points Four different kinds of interaction point can be defined in COCOA:

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

The graphical constructs for the four point types listed above (see Figure 4) are attached to the step by placing them on the outer edge of the box that represents the elementary or composite step.

Figure 4: Graphical constructs for the four point types: **in point**, **out point**, **interrupt point** and **signal point** (from left to right).

Transitions Transitions between interaction points are depicted by a solid line with an arrowhead. Transitions can be identified by the types of their start and end points. Because each point type has its own symbol, different kinds of line are not needed. However, in addition to transitions between two points (**binary transitions**), there is a need to define transitions in which more than two points are involved, for splitting and joining control flows. A control flow is split when two or more transitions start from the same point (**split transitions**). Similarly, control flow is joined when two or more transitions end at the same point. GRACOOP currently provides only a (disjunctive) **OR-Join** transition, which is used to model convergence: any of the multiple incoming transitions causes the control flow to move on.[3]

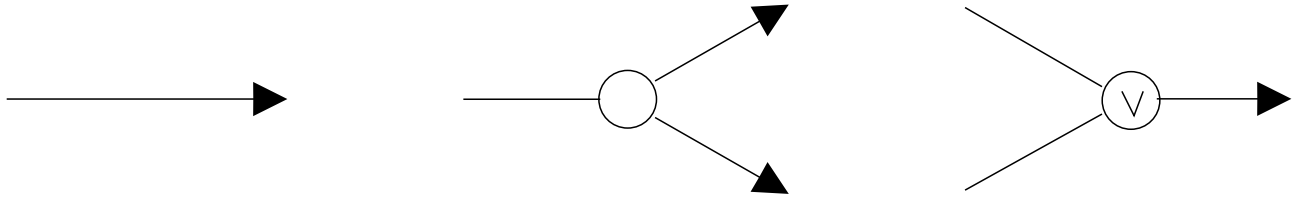


Figure 5: Graphical representation of transition structures in GRACOOP: **binary transition**, **split transition** and **OR-join transition** (from left to right).

The graphical representations for the possible transition structures are presented in Figure 5.

Above, we have shown graphical constructs for defining transitions between steps (points) in the same step diagram (i.e., at the same "level" in the hierarchy). Because a scenario specification may be hierarchical, and thus consist of several interrelated step diagrams (created through the decomposition of composite steps), a means for defining dependencies (i.e., transitions in this context) between different hierarchy levels is needed. For example, when the activation of a parent step results in the activation of a certain substep. For this, we introduce the following:

- **horizontal transitions** to define dependencies between steps inside the same parent step, and
- **vertical transitions** to define dependencies between a (parent) step and its immediate substeps.

Examples of both forms of transition are shown in Figures 7, 8 and 9. Vertical transitions are drawn in GRACOOP by placing the points of the parent step (**parent points**), and hierarchical transitions to those points in the step diagram. The points of the parent step are depicted graphically as described above, with the following distinctions:

- they are placed freely in the step diagram, not attached to any step edges, and
- they are drawn larger than the points that are attached to step edges.

Multi-instance steps A multi-instance step is a step construct that allows multiple parallel activations of the step. In order to distinguish between different step activations in the multi-instance step, a set of identifying TM-typed parameters is given when a step-instance is activated. A multi-instance step may be either elementary or composite. The graphical construct for a multi-instance step is shown in Figure 6. The graphical representation of a multi-instance step differs slightly from its representation in COCOA. In the graphical notation, if more than one step is to be defined in a parallel (**par**) clause, an extra composite step must be drawn around the steps. How the specifier annotates this graphical composite step with transitions may affect the step activation rules. Whether or not restrictions must be placed on transitions in this context requires further study.

[3] A corresponding conjunctive construct may be added later, when the semantics of such a structure in the context of COCOA is clear (see Section 1.2.)

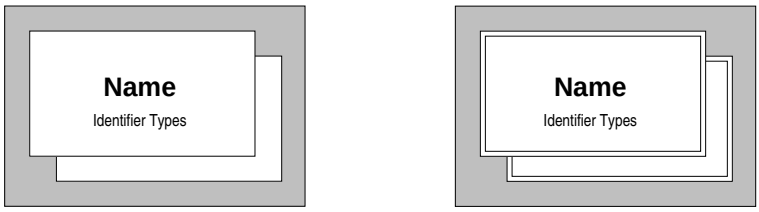


Figure 6: Graphical construct for a **multi-instance elementary step** (left) and a **multi-instance composite step** (right).

Examples of step diagrams We now provide some examples to illustrate the usage of the graphical constructs presented above. The examples are adapted from [FaEB96]. (A CoCoA specification of this scenario appears in Appendix C.)

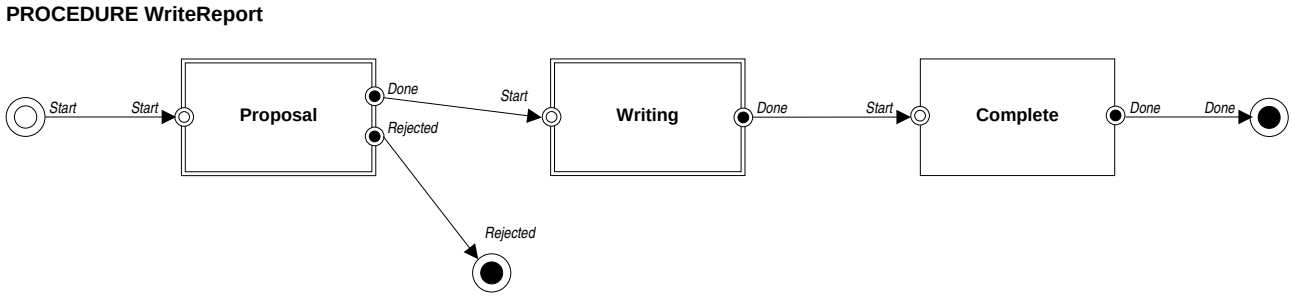


Figure 7: A step diagram depicting the sequential processing of the `WriteReport` step of the CDA Scenario.

The first example (Figure 7) shows a step diagram that depicts the `WriteReport` CDA scenario at its highest hierarchy level. The scenario consists of two composite steps and one elementary step, which are sequentially ordered. Each step has an **in point** called `Start` and an **out point** called `Done`. The `Proposal` step has an additional **out point** called `Rejected` to depict the fact that the proposal may be rejected. In this case, the high-level scenario is rejected. The three **parent points** in the step diagram depict the one **in point** and two **out points** of the whole scenario.

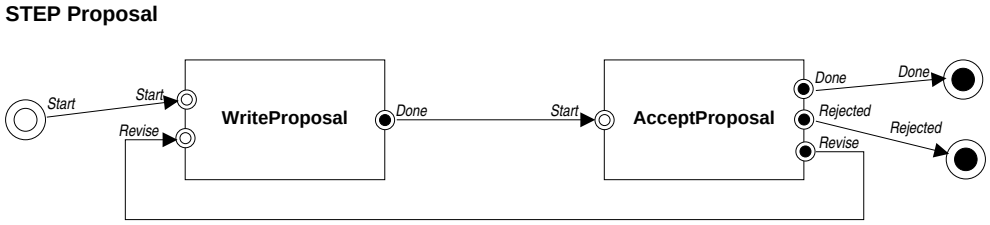


Figure 8: A step diagram depicting step repetition in the `Proposal` step of the CDA scenario.

Figure 8 is a decomposition of the composite step `Proposal`. The **parent points** `Start`, `Done` and `Rejected` are copied from the step diagram depicting the upper hierarchy level. This step diagram also illustrates how to define repetitive steps.

Figure 9 illustrates the use of a multi-instance (elementary) step to define the dynamic activation of the writing tasks (`WritingTask`), which is under the control of the master step

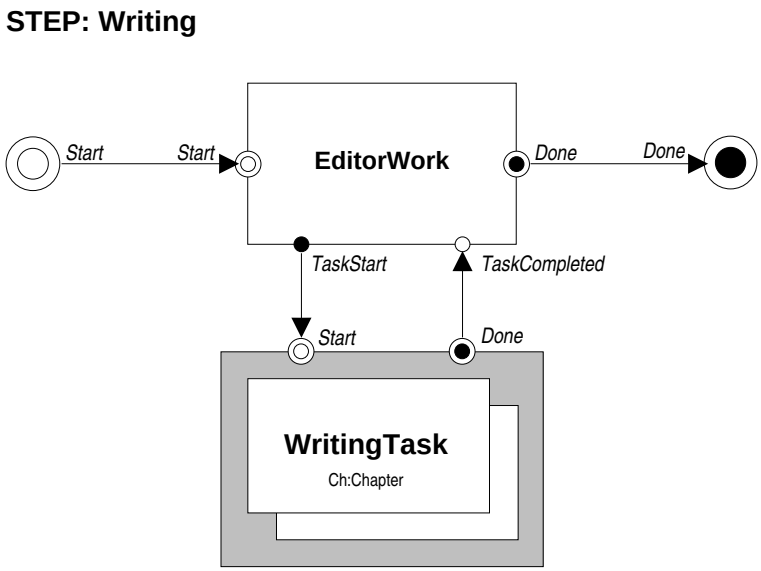


Figure 9: A step diagram depicting the dynamic activation of the writing tasks in the `Writing` step of the CDA scenario.

`EditorWork`. Here, the transition from the **signal point** `TaskStart` to the **in point** `Start` is used to define a transition from the `EditorWork` step that allows the step to continue to be active after the transition takes place. The interrupt point `TaskCompleted` of the same step is used to inform the editor that the task is completed. In this case, an identifying parameter '`Ch`' of TM type `Chapter` is used to distinguish the dynamically started `WritingTask` instances. How such typing information can be added to a graphical scenario specification is discussed next.

2.1.2 Entering textual annotations

The GRACOOP modelling method is developed with computer support in mind. Because of this, we assume that we have a windowed, form-based user interface, which can be used to add annotation (non-graphical) information. The annotation input is defined with the help of *dialogs*, *tables*, and *matrixes*, which are filled with the needed information. Dialogs are electronic forms with labelled input fields in which information can be added. To support correct field entry, a dialog may provide lists of allowed values for a field. Dialogs can be nested: some action on a dialog may cause another dialog to appear. Tables and matrixes are a special type of input, used to handle lists of multiple (graphical) objects, and the relationships between them.

Simple dialog forms are sufficient for entering non-structured properties (e.g., the name of a step) and lists (e.g., a list of type parameters annotating a point). In the remainder of this section, we concentrate on ways to give values for some of the more complex properties of a scenario, such as the:

- enable statements of an elementary step,

- execution rules of a scenario,
- data exchange operations of a scenario, or
- history rules of a scenario.

Enable statements of an elementary step An enable statement binds three entities to the elementary step itself: a **user**, a **point**, and an **operation invocation pattern**. The set of enabled operations may be dynamically bound to a set of users with the help of an actual parameter of an incoming transition. This binding is done by an optional **for**-construct in COCOA.

Entering enable statements in GRACOOP is done separately for each **in point**, and each possible **interrupt point** of the elementary step. The following information is provided by the specifier using (nested) dialogs:

- the name of the point,
- a list of point parameters,
- lists of enabling statements for different operation types (i.e., data, data exchange, and communication)
- an optional list of **for**-statement annotations, each item of which consists of:
  - a list of variable names,
  - a list of variable types, and
  - lists of enabling statements for different operations types.

An enabling statement for a data operation consists of:

- a user name (which may be a variable), and
- a data operation signature.

An enabling statement for a data exchange operation consists of:

- a user name (which may be a variable),
- a selection of the exchange operation type (**import** or **export**),
- a data exchange operation signature, and
- a workspace expression that may be a user name (indicating the user's workspace), a parameter (variable) name, a shared workspace name, or a '_' symbol to indicate no restrictions.

An enabling statement for a communication consists of:

- a user name (which may be a variable),
- a communication signature,
- a point name, and
- an optional **query**-statement.

For each enabling statement, the textual fields requested by the dialog correspond to those mentioned in the COCOA syntax. (See Appendix B.)

Execution rules Each scenario may have a set of named ordering rules, which restrict the execution orderings of the data operations enabled by elementary steps. Since the ordering rules may be presented as deterministic finite automata, a graph representation of these rules is possible. We feel, however, that the ordering rules are easier to give as regular-expression-like ordering expressions, similar to the expressions in the COCOA language.

An ordering rule is presented as a list in which every item is filled through a dialog form. In such a dialog form, the following details are entered:

- the name of the ordering rule,
- a parameter list for an optional **forall**-statement, and
- an ordering expression.

Again, the dialog provides a way to enter textual attributes for the graphical constructs, according to the COCOA syntax.

Data exchange operations The declaration of a data exchange operation is presented as a list in which each item is entered with a simple form that contains input fields for:

- a signature for the data exchange operation, and
- a list of data operation invocation patterns to be selected for exchange.

History rules History rules are defined for pairs of data operations, hence a matrix representation of the rules is a natural solution. The rules, however, cannot be defined simply by putting a 'yes/no' symbol into the matrix, because the rules can be parameter-dependent. Thus, each cell of the matrix is a record that is filled in by using a form, where all the details of the history rule can be entered. These details are as follows:

- an optional parameter list for the **forall** declaration, and
- an optional **iff**-statement for the parameter-dependent part

The exact format and behaviour of the dialogs, tables, and matrixes that are used to add textual annotations to a graphical specification will be worked out in the implementation phase of the TSE development.

2.2 Mapping GraCooP constructs to CoCoA

GRACOOP is developed to allow its constructs to be easily mapped to COCOA constructs. Table 1 summarises this mapping.

2.3 Outline of planned implementation

Although a prototype fulfilling the above requirements could be implemented using C/C++ or Smalltalk, with their visual tools and libraries, we have studied another approach that will enable much quicker development of the GSE prototype. The MetaEdit+ CASE tool by the Finnish MetaCase Consulting supports the creation of graphical editors [Meta]. After experimenting with it, we consider it to be a convenient alternative to the traditional programming approach for implementing graphical tools. In this section, we briefly introduce MetaEdit+ and its Method Workbench, which will be used to implement the GSE prototype.

2.3.1 Introduction to MetaEdit+ and MetaEdit+ Method Workbench

MetaEdit+ is a multi-platform CASE tool to support multiple modelling methods and methodologies. Currently supported methodologies include: business modelling, architecture development, object-oriented modelling, and structured modelling. MetaEdit+ provides diagraming, matrix- and table-based tools for creating and viewing definitions. In addition, MetaEdit+ enables restricted code generation and reporting with its Report Browser.

What makes MetaEdit+ interesting for the GSE prototype implementation is its Method Workbench, which provides tools that enable the creation of new (graphical) modelling methods and the modification of existing ones. This kind of high-level CASE tool for creating graphical editors quickly is ideal for our purposes. Figure 10 shows a screen-save snapshot of the GSE to be developed using MetaEdit+.

GRACOOP **construct**	COCOA **construct**	**Note**
Procedure	Procedure	
Elementary and composite step	Step	There is no keyword in the COCOA grammar to distinguish elementary and composite steps; they are distinguished by their contents.
Point	(Interaction) Point	All the point types in COCOA can be found in GRACOOP.
Transition from point to point	(same)	
Split transition	multiple outgoing transitions from one interaction point (parallel steps)	
Transition from point to point (in a cyclic way)	(same)	
Multi-instance (elementary or composite) step	Parallel clause	Each multi-instance GRACOOP step is mapped to a COCOA step in its own parallel clause.
OR-join transition	(same)	
`EnablingLists`-property of the elementary step	Enable statements given in a step definition	
`ExecutionRuleList`-property of the procedure	Execution rules	
`DataExOpList`-property of the procedure	Data operations	
`HistoryRuleMatrix`-property of the procedure	History rules	

Table 1: Mapping the graphical constructs of GRACOOP to COCOA.

The MetaEdit+ metamodel is the basis of all the modelling methods developed with the MetaEdit+ Method Workbench.[4] (See Figure 11.) We now briefly introduce the MetaEdit+ metamodel, since it provides the framework in which the GRACOOP concepts will be realised in the GSE prototype implementation.

The MetaEdit+ metamodel consists of five *metatypes*:

An **OBJECT**[5] represents a thing that exists on its own. Examples of OBJECTS are a 'data store' or a 'process' in a Data Flow Diagram, or a 'class' or an 'object' in an Object Diagram.

A **RELATIONSHIP** is an explicit connection between a group of OBJECTS. It is attached to an OBJECT via ROLES (described below). A RELATIONSHIP can be a 'data flow' belonging to a Data Flow Diagram or an 'inheritance' RELATIONSHIP in an Object Diagram.

[4]To be precise, we should call this a "*meta*metamodel", since all of the modelling methods are themselves metamodels for the actual models.

[5]To avoid confusion, when we intend the MetaEdit+ metatype meaning of a word, we write the word in capital letters. Furthermore, the metatype names are used in two ways in the text: depending on the context, we refer to a metatype (cf. a class) itself, or an instantiation of the metatype (cf. an object belonging to the class).

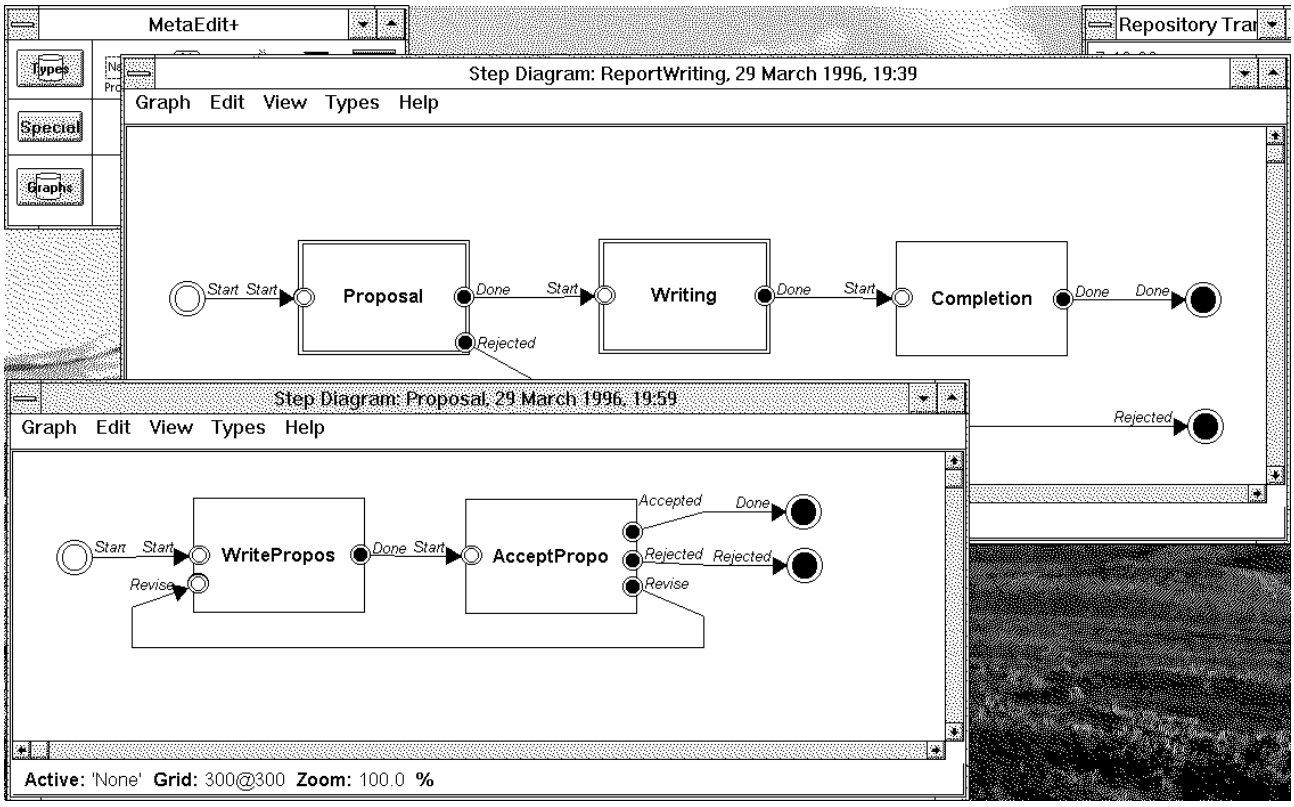


Figure 10: Defining step diagrams with the GSE to be implemented using MetaEdit+.

A **ROLE** specifies how an OBJECT "participates" in a RELATIONSHIP. Examples are 'flows from' and 'flows to' ROLES that specify which 'process' OBJECT is the producer, and which is the consumer of data in a Data Flow Diagram. Similarly, in an Object Diagram, there are two possible ROLES for an OBJECT in an 'inheritance' RELATIONSHIP: 'ancestor' and 'descendant'.

A **GRAPH** is a collection of OBJECTS, RELATIONSHIPS, ROLES, and *binding information* to show which OBJECTS a RELATIONSHIP connects via which ROLES (for an example, see Figure 12). A GRAPH also maintains information about the graphs to which its elements explode, and how its elements are decomposed. Examples of graphs are Data Flow Diagram and Object Diagram. The name GRAPH does not indicate that the presentation should consist of graphical elements. A GRAPH need not be a graphical diagram; it may be a matrix or a table. Furthermore, alternative representations of a particular GRAPH may be used to display the information in different ways.

A **PROPERTY** is a describing or qualifying characteristic that can be associated with the four *non-property* metatypes described above. Examples of PROPERTIES are: a 'name', an 'identifier', or a 'description'. Possible data types for properties are: string, number, boolean, text, ordered collection of items (i.e., list), selection list of items, and any of the non-property metatypes. By allowing a non-property (i.e., OBJECT, RELATIONSHIP, ROLE or even GRAPH) to be the type of a PROPERTY, MetaEdit+ allows the definition of composite objects and arbitrarily complex object hierarchies.

All of the metadata, as well as the instances created using the modelling methods, are stored

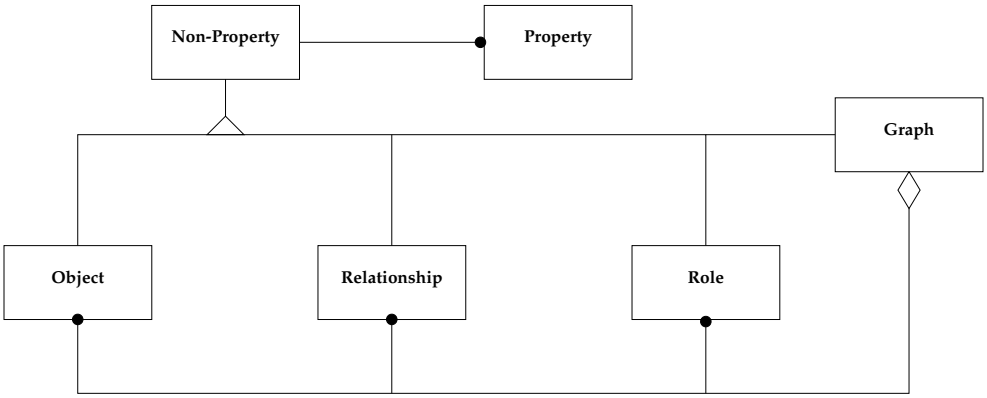


Figure 11: The MetaEdit+ metamodel.

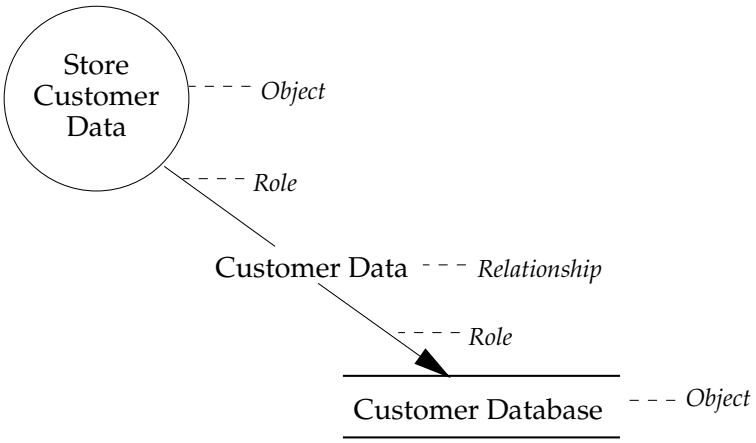


Figure 12: An example of bindings in a MetaEdit+ graph.

in the (proprietary) MetaEdit+ database. For each GRAPH, it is possible to create reports for retrieving the data in the database. This data is queried by user-defined reports. The reporting capabilities can also be used to generate simple program code, such as C/C++ header files.

2.3.2 Mapping GraCooP constructs to MetaEdit+ metamodel

A metamodel that captures the constructs of both GRACOOP and COCOA can be conveniently created within the framework of the MetaEdit+ metamodel sketched above. Next, we outline how this can be done.

A COCOA scenario is defined as a MetaEdit+ GRAPH of type `Scenario_Def`. Its properties are shown in Table 2. The related GRACOOP representation of a scenario (step diagram) uses the the graphical constructs shown in Table 3.

The decomposition feature of MetaEdit+ may be used to define the hierarchical structure of steps. For example, the `Scenario_Def` GRAPH can be exploded to related `Data_Op_Ordering`

CoCoA Construct	**MetaEdit+ Object Type**
TM modules	ordered collection of strings
Data types	ordered collection of strings
Data operations	ordered collection of OBJECTS of type `Data_Op`
Workspace types	ordered collection of OBJECTS of type `WS_Type`
User types	ordered collection of OBJECTS of type `User_Type`
Communications	ordered collection of OBJECTS of type `Comm_Spec`
Data Exchange Operations	ordered collection of OBJECTS of type `Data_Ex_Op_Spec`
Data Operation Order	GRAPH of type `Data_Op_Ordering`
History Rules	GRAPH of type `History_Rules`

Table 2: Properties of a scenario definition.

GraCooP Construct	**MetaEdit+ Object Type**
Composite Step	OBJECT of type `C_Step`
Elementary Step	OBJECT of type `E_Step`
Binary Transition	RELATIONSHIP of type `BinaryTransition`
Split Transition	RELATIONSHIP of type `SplitTransition`
OR-Join Transition	RELATIONSHIP of type `ORTransition`
From out point	ROLE of type `FromOutPoint`
From in point	ROLE of type `FromInPoint`
From signal point	ROLE of type `FromSignalPoint`
From interrupt point	ROLE of type `FromInterruptPoint`
To out point	ROLE of type `ToOutPoint`
To in point	ROLE of type `ToInPoint`
To signal point	ROLE of type `ToSignalPoint`
To interrupt point	ROLE of type `ToInterruptPoint`

Table 3: Graphical constructs of a scenario definition.

and History_Rule GRAPHS. With the help of the Data_Op_Ordering GRAPH, the scenario specifier can define the ordering rules of the data operations (possibly graphically). Similarly, the History_Rule GRAPH is used to construct the **noncommutative** matrix.

Each element of metatype OBJECT, RELATIONSHIP, ROLE, or GRAPH, specified for the GRACOOP method (e.g., elements sketched for the Scenario_Def GRAPH) can have PROPERTIES, whether or not the element has a graphical representation. The PROPERTIES can be defined with the help of other OBJECTS. Thus, object hierarchies and complex data structures are possible. For example, we can define Point_List to be an 'ordered collection' of OBJECTS of type Point; then we can add a PROPERTY of type Point_List to the E_Step or C_Step OBJECT. The C_Step OBJECT may itself be decomposed into a new STEP_DEF GRAPH, which defines the inner structure of the step (i.e., its constituent steps or elementary steps, and their ordering).

2.3.3 Other implementation considerations

Some parts of a scenario specification are defined graphically (e.g., with nodes and connectors); other parts require textual input. For PROPERTY values, such as type annotations on interaction points, MetaEdit+ provides simple data entry dialogs, with a customisable appearance. Matrices and tables provide another alternative to GRAPHS for describing OBJECTS and their RELATIONSHIPS. During implementation of the GSE, we will investigate what representations are most appropriate for displaying the constructs of COCOA.

2.3.4 CoCoA code generation from GraCooP specification

The scenario specification constructed using the MetaEdit+ and GRACOOP modelling method may be queried from the MetaEdit+ data repository. For this, it is possible to use the reporting and code generation tool, Report Browser. The tool generates formatted information about the OBJECTS and their relationships with other OBJECTS defined in the same GRAPH.

Although it is possible to create complex objects and object hierarchies, producing one common document recursively from an (arbitrarily deep) hierarchical specification is not possible with the MetaEdit+ Report Browser. Furthermore, decompositions or explosions in a GRAPH cannot be detected during report generation. Thus, the final COCOA specification may require the post-processing of several related partial specifications to produce a single (final) specification.[6]

In a multi-part specification, there will be one master file containing the highest level of the hierarchical specification, and references to the partial specifications contained in the next level of the (step) hierarchy, and so on. The references to the partial specifications are `include` statements (just like in C). Processing the master specification produces a final COCOA specification as a single file, with all of the partial specifications included (see Figure 13).

[6] In spite of being cumbersome, this approach has the benefit that it allows the reuse of parts of a specification.

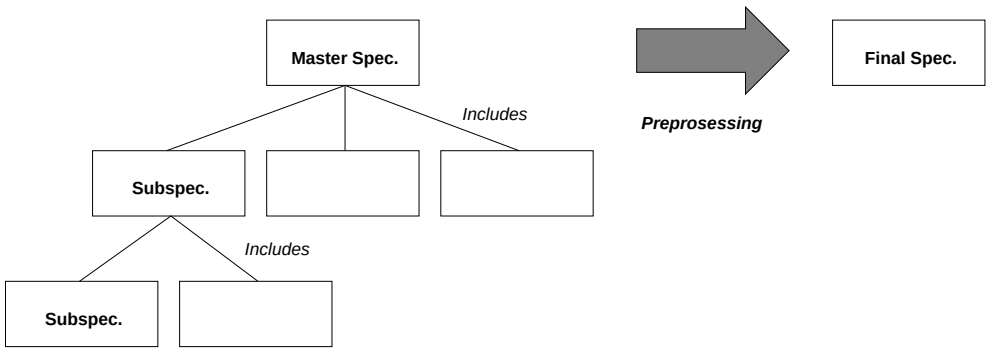


Figure 13: Producing a single COCOA specification file from a multi-part specification.

3 The Parser/Type-checker

The CoCoA parser/type-checker will be built upon work previously carried out for LOTOS and TM. For instance, CoCoA is defined with respect to a specific TM database schema. For parsing reasons, this schema must be read in first so that the data structures it constructs are resident when the CoCoA specification is parsed/type-checked. When the TM schema data types are resident, it makes sense to type-check CoCoA data expressions. Section 3.1 describes the TM parser/type-checker. Section 3.2 comments on the internal representation of a CoCoA specification. Section 3.3 looks at the static semantics checks that will be performed.

3.1 The TM Type Checker

The TM type checker is implemented on a number of platforms, and consists of a single executable that accepts a TM specification text file, and outputs a report in a variety of formats [BBBB95]. The type checker is implemented as a flexible type-checking tool that allows translation from TM to other languages. To this end, the type-checker is strictly divided into a front-end and a back-end, with a well-defined internal representation functioning as an interface between the two.

The front-end consists of a number of passes—currently three—that check the syntax and build the internal representation of the specification text. The number of passes taken can be influenced by a command line argument, but passes are always carried out in order. Knowledge of the TM internal representation and the TM type-checker architecture is essential for building TM type-checks into the CoCoA parser/type-checker. The reason is that we want to use partial type-checks on single expressions embedded within a specification text, and for this, a proper TM representation needs to be present inside the CoCoA type-checker.

The back-end of the TM type-checker can be viewed as a switch for generating different output formats; this switch can also be set as a command line argument. An essential switch for our purposes is one that generates code for the TM Abstract Machine (TAM), which we will be able to use for simulation purposes. Knowledge of this back-end part of the software is less critical for our purposes.

3.2 Internal representation of a CoCoA specification

It is essential to the TSE design to have an internal representation for a CoCoA specification that can be easily integrated with the other tool set components. The static semantics checks and the verification analysis mappings will be defined to manipulate this internal representation. Because CoCoA builds on a TM database schema and TM type structures, the internal CoCoA representations will be designed as an extension of the internal representations used in the TM parser/type-checker. An added benefit of this approach is that the TM representations are well-designed and thus provide a strong foundation for the CoCoA software design.

The internal representation of a COCOA specification is stored in a C++ class hierarchy, and will not be stored externally. This implies that the parser/type-checker will have to be integrated with the back-end parts (i.e., the verification analysis mappings) into a single program. This approach has also been used in the design of the TM tool set [FlKS94].

3.3 Static semantics checks

A COCOA specification is defined with respect to a TM database schema; it refers to data operations (methods) and data types that are defined in the TM schema. In this section, we discuss the static semantic checks that must be performed on a COCOA specification. These include constraints on the use of types, variable declarations, and operation parameters, and restrictions on the declaration, scoping and use of interaction points in transitions. Besides this, there are also well-formedness constraints on the execution rules. (These are described in Section 4.1.3.)

3.3.1 Typed variable declarations

Typed variables are introduced by the following COCOA constructs:

1. as parameters to the **procedure** statement of a scenario

   procedure $(x_1 : \tau_1, \ldots, x_n : \tau_n)$ [. . .]

2. as parameters to data exchange operations

   data exchange operations
   `op_name`$(x_1 : \tau_1, \ldots, x_n : \tau_n)$ = **select** . . .

3. as parameters to the **forall** construct of **order** rules and **noncommutative** rules

   forall $\ldots, x : \tau, \ldots, y$ **in** $\mathcal{Y}, \ldots$
   order . . .

   forall $x_1 : \tau_1, \ldots, x_n : \tau_n$
   noncommutative . . .

4. as parameters to the **parallel** clause

   parallel $(x_1 : \tau_1, \ldots, x_n : \tau_n)$ . . . **endpar**

In each of the above cases, one or more TM-typed variables are introduced. All of the different kinds of variable are bound to actual values at run-time. The scopes of the variables are as follows: (1) declares global scenario variables, which are bound to values when the scenario is instantiated; (2) and (3) declare local variables that are used within the bodies of the constructs (i.e., within the **select** statement of a data exchange operation, and within the

body of an ordering or history rule); (4) declares a variable to identify activations of a **parallel** clause; its scope is the steps declared inside the clause.

An untyped variable is introduced by the '**for** *variable* **in** *set-of-values* **allow** *operations*' construction of an **enable** statement:

for a_1 **in** $\texttt{Expr}_1$, ..., a_n **in** $\texttt{Expr}_n$ **allow** ...

This construct declares local variables a_1 through a_n, which are used in the list of operations specified after the **allow** keyword. Expressions $\texttt{Expr}_1$ through $\texttt{Expr}_n$ must have set types, and the variables are given the corresponding element types.

3.3.2 Operations

Data operations are declared in the TM schema. Communication operations and data exchange operations are declared in the COCOA specification. Restrictions on the use of operations that take parameters are as follows:

- Data operations and data exchange operations must be used with the correct number and types of parameters. The variables used as value parameters to an operation are checked against the signature of the operation.
- Any data exchange operation referred to by an **import** or **export** command must be defined as a data exchange protocol in the **data exchange operations** section of the specification.
- Communications are used with variable names for the parameters. The variable names may be bound to values elsewhere, or they may be free (i.e., not bound to values). In the former case, the variables should be of the appropriate type. This can be statically checked. In the latter case, values are bound to the variable names during execution.[7]
- A communication operation must appear exactly once in the **procedure** part of the specification. Each communication is associated with a transition, and this association must be unambiguous.

Anonymous values The anonymous placeholder '_' matches any type; it is used to indicate that any actual parameter value is allowed. It can be used as a parameter to data operations in all contexts, and as a parameter to data exchange operations in **enable** statements. The anonymous placeholder may not be used in communications.

[7] Note: COCOA is a specification language, not an implementation language, so we do not address whether a run-time exception is raised if an argument is mis-typed.

3.3.3 Interaction points and transitions

Before discussing interaction points, we first introduce some terminology that will aid the explanation. For illustration purposes, it is convenient to draw a box around a step definition to show the locality of step activations, and the extent of their interaction point declarations. The term *step box* will be used to refer to a step's definition block. The term *procedure box* will be used to refer to a similar box drawn around the scenario's procedure definition block. The term *encapsulating step* will be used to refer to the step box that immediately surrounds substep definitions, parallel clause definitions, and transition definitions. To illustrate this, consider the scenario structures in Figures 14 and 15. In the first figure, step box `A` is the encapsulating step for steps `B` and `C`, and any transitions at position (1). In the second figure, step box `A` is the encapsulating step for the parallel clause (and its substeps `B` and `C`), substep `D`, and any transitions at position (1).

The static semantic requirements for the declaration of interaction points, scoping of interaction point declarations, and the use of interaction points within transitions are as follows:

- each substep within an encapsulating step must have a unique name; this also applies to the substeps of its parallel steps
- the interaction point names declared in a step header (or in the **procedure** header) must be unique
- within transitions, interaction point names are prefaced by the step name in which they are declared, except in the following cases: interaction points of the **procedure** statement are not prefaced, and interaction points declared in the header of an encapsulating step box are not prefaced when they are referred to by transitions declared in the step box
- a transition may refer to the interaction points of its enclosing step, and to the interaction points declared in the headers of its enclosing step's substeps (this is explained in detail below); the first interaction point of a transition may not equal its second interaction point.
- interaction points must be used with the appropriate number and types of value parameters; in transitions, value parameters to an interaction point are indicated by (free) variable names, which are implicitly typed.
- any steps declared within a parallel clause with a value identification are explicitly given a value identification as a parameter whenever the step name is used in a transition outside the parallel clause: a variable name (with an implicit type) is attached to the step name for identification

Figure 14 illustrates a number of nested steps. Variables `a`, `b`, and `c` may be referred to anywhere within the steps. Three places in the specification are identified by numbers. Transitions written at (1) may refer to the following interaction points: `B.p1`, `B.p2`, `C.p1`, `C.p2`, `p1` (of step `A`), and `p2` (of step `A`). Transitions written at (1) may not refer to interaction points

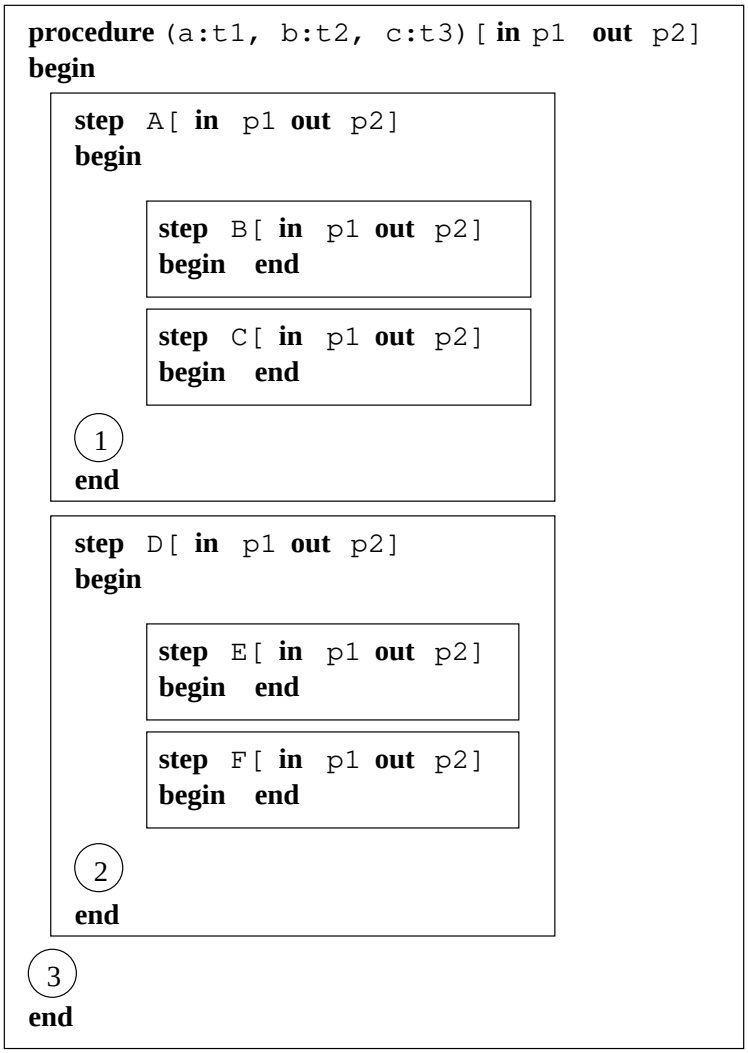


Figure 14: Scoping of interaction point declarations.

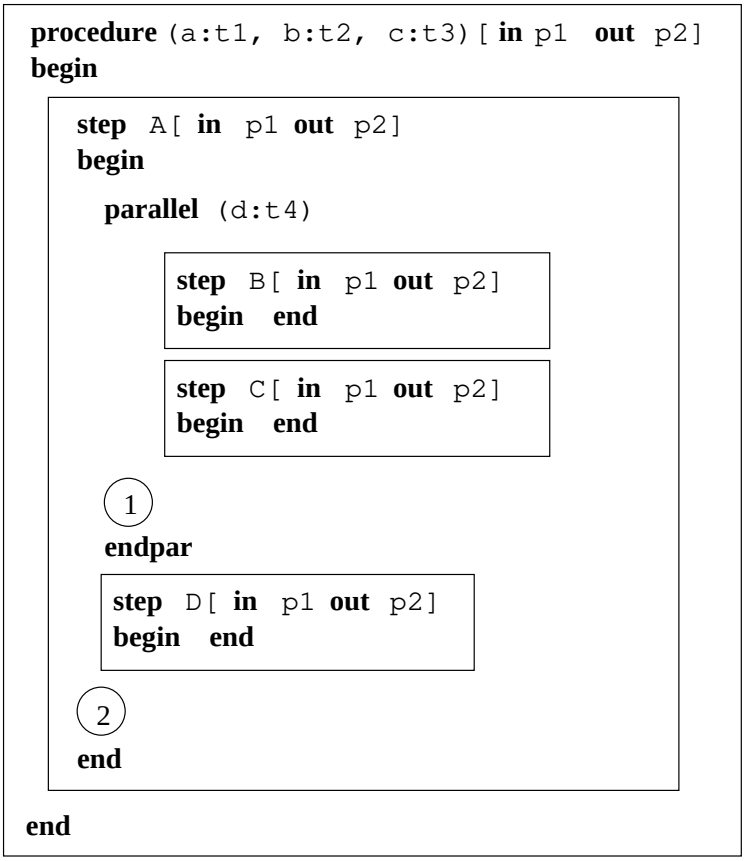


Figure 15: Scoping of interaction points within parallel steps.

declared within step `D`, and they may not refer to `p1` and `p2` of the procedure. Transitions written at (2) may refer to the following interaction points: `E.p1`, `E.p2`, `F.p1`, `F.p2`, `p1` (of step `D`), and `p2` (of step `D`). Transitions written at (2) may not refer to interaction points declared within step `A`, and they may not refer to `p1` and `p2` of the procedure. Transitions written at (3) may refer to the following interaction points: `A.p1`, `A.p2`, `D.p1`, `D.p2`, `p1` (of the procedure), and `p2` (of the procedure). Transitions written at (3) may not refer to interaction points declared within steps `B`, `C`, `E`, and `F`.

Figure 15 illustrates a number of nested steps, along with a parallel clause. Two places in the specification are identified by numbers. Transitions written at (1) may refer to the following interaction points: `B.p1`, `B.p2`, `C.p1`, and `C.p2`. Transitions written at (1) may not refer to interaction points of step `A`, step `D`, or `p1` and `p2` of the procedure. Transitions written at (2) may refer to the following interaction points: `D.p1`, `D.p2`, `B(v).p1` (where `v` is a variable name that will be bound to a value, used to identify an activation of the parallel clause), `B(v).p2`, `C(v).p1`, `C(v).p2`, `p1` (of step `A`), and `p2` (of step `A`). Transitions written at (2) may not refer to interaction points `p1` and `p2` of the procedure. Observe that transitions at (1) do not need to identify an activation of the parallel clause: since they are inside it, they implicitly refer to an activation.

3.3.4 Deterministic ordering expressions

The ordering expressions in the execution rules are required to be deterministic. This property is based on the ordering expressions themselves, not on an underlying finite state automaton. To check whether an ordering expression is deterministic, for every data operation invocation pattern in the ordering expression, we have to check whether all of the invocation patterns that directly follow it in the ordering expression are unique with respect to their data operation names. The parser/type-checker will include a procedure for checking this property.

4 The Verification Environment

The previous section discussed static semantics checks on a CoCoA specification. In this section, we look inside the verification component of the TSE, which concerns itself with the dynamic semantics of a CoCoA specification. Our approach relies on the LOTOS/TM simulation environment underneath for the symbolic execution of various facets of a specification.

As can be seen in Figure 2, five classes of verification tool are identified in the Verification Toolbox. We do not rule out other verifications, but within TransCoop we will be focusing on the five mentioned. The implementation of all semantic verifications will to some extent be based on output of a mapping from CoCoA to LOTOS/TM, specific for the various verifications that will be simulated. The five types of semantic verification are:

- *Execution rules analysis*, in which the specification freedom in execution rules is tested by illustrating their effects by way of simulation to the specifier. Some straightforward syntactic analyses like deterministic rule sets will be built into the parser/type-checker.
- *Organisational view analysis*, in which we allow the specifier to study the effects of her/his specification by simulating the step activation and operation enabling administration in a LOTOS/TM context.
- *Transactional view analysis*, in which we allow the specifier to study the effects of the TransCoop Transaction Model [KTWP96] by simulating (symbolically) its history administration. This, in effect, will provide a symbolic simulation of the TransCoop transaction model.
- *View integration analysis*, in which we will combine the results of organisational and transactional view simulation, and to study their interaction.
- *Commutativity analysis*, in which we interactively, or semi-automatically support the specifier in proving that two given data operations are commuting, possibly under certain conditions.

The technology behind these analysis tools are various CoCoA to LOTOS/TM mapping algorithms (for analysis classes 1–4), the LOTOS/TM simulator (for analysis classes 1, 2 and 4), the TM Proof Tool (for analysis class 5), and the TM Abstract Machine (for analysis classes 3 and 4). Note that the LOTOS/TM simulator invokes the TM Abstract Machine. We discuss each of the above classes in the following sections.

Executable semantics Our Verification Toolbox is intended to offer the scenario designer executable semantics for CoCoA, which the designer can experiment with using the LOTOS/TM Simulation Environment. The designer may define additional constraints on the generated TM parts of the specification, and then employ the TM Proof Tool to determine whether the dynamic aspects of the scenario (i.e., methods generated by the various verification tools) preserve the constraints.

In general, correctness proofs about a CoCoA specification would start off with a workspace history for which a certain property holds. For example, given a history $\mathcal{H}$, a claim about property P such as '$P(\mathcal{H})$ *holds, and operation* a *is allowed by the specification, implies that* $P(\mathcal{H}; a)$ *holds*' should ideally follow from the formal semantics of a CoCoA specification. In the Verification Toolbox, we include tools that allow us to investigate three types of properties:

1. whether an actor is allowed by the organisational view to do a particular operation at a particular point in the scenario,

2. whether an actor is allowed by the execution rules to do a particular operation at a particular point in the scenario, and

3. whether an actor is allowed by both the transactional and organisational views to do a particular operation at a particular point in the scenario.

The third type of property considers aspects of the scenario that depend on both the organisational and transactional views, such as breakpoint constraints. The tools in the verification environment make use of simulation. The tools generate TM data structures and LOTOS/TM behaviours, which formally define different aspects of the CoCoA specification. These formal specifications are manipulated by the simulation environment.

TM is used to describe the data domains of the dynamic semantics of CoCoA. Because TM allows the definition of constraints on data type structures, our design rationale will take advantage of this facility. Execution rules can be considered properties about the histories allowed by a CoCoA specification. If we define a TM data type for workspace histories, then we can incorporate the execution rules within proofs directly (as properties that can be inferred upon and composed) by expressing an execution rule in the form of a predicate (i.e., constraint) over a history (i.e., the history data type). This is analogous to the way that constraints on data values are expressed in TM. Similarly, by defining TM data structures that maintain the set of active steps of a scenario (these data structures are manipulated by the organisational view), we can express organisational properties we would like to hold for a specification as constraints on the TM data types. In the future, the TM Proof Tool could be used to examine whether the dynamic semantics of the specification (as defined by the TM methods that manipulate the active steps and the workspace histories) indeed preserves the properties that are expressed as constraints.

4.1 Execution rules analysis tool

In this section, we discuss the semantics of the execution rules and how they are mapped to TM and LOTOS/TM for analysis. The semantics of the execution rules is explained in [FaEB96]. From the explanation given there, it is obvious that certain restrictions need to be enforced to make run-time checking feasible. These restrictions will be described in a separate section below.

4.1.1 Semantics of execution rule checking

To check an execution rule against a given history, a subhistory has to be constructed that contains only the relevant operations that are mentioned in the execution rule. The execution rule (in the form of a regular expression) is then used to check the ordering of operations in the constructed subhistory. The history does not need to match the whole rule; it only needs to be a prefix of a history that matches the whole rule.

In the case that the execution rule has a **forall**-clause, the history has to be checked for all possible combinations of values for the universally quantified parameters. In the algorithm, this check can be restricted to only consider a subset of value combinations, based on the those that occur in the history.

The algorithm to check a given history against an execution rule thus consists of the following three steps:

1. Construct a subset of parameter value combinations to test, based on all possible combinations that are found in the given history.
2. For each of these combinations, construct a subhistory that contains only those operations that have the given parameter values.
3. For each of these subhistories, check the subhistory against the execution rule, with the associated combination of parameter values.

In the case that there are is no **forall**-clause, the second and third steps are done once for an "empty" combination of values.

We illustrate the algorithm by means of a small example that uses the following execution rule:

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

The history we want to validate is `[a(1), b(1,3), c(1,3), b(1,4)]`. This history has to be checked against all possible combinations of parameter values for `i` and `j`. However, many combinations of values entail checking the same subhistory. For this example, there are only four different value combinations/subhistories that need to be checked. These are shown in the table below:

i	j	subhistory
other	any	
1	other	`a(1)`
1	3	`a(1), b(1,3), c(1,3)`
1	4	`a(1), b(1,4)`

The abbreviation 'other' stands for any other value (in the domain of the parameter) not mentioned explicitly in the column (i.e., 'not `1`' in the first row, and 'not `3` or `4`' in the second row); 'any' stands for any value in the parameter's domain. We get an empty subhistory for each value of `i` that is different from `1`, independent of the value of `j`. This is illustrated by the first row of the table. This row can be ignored during checking, because it results in an empty subhistory, which is valid for any execution rule.

The value combinations for the remaining cases match the value combinations found as parameters to the operation invocations in the history. If we take `i` to be `1`, and `j` to be different from `3` and `4`, we obtain a subhistory with only the first operation, `a(1)`. For `j = 3` or `j = 4` (with `i = 1`), we obtain the other two cases. Observe that if a parameter has the value 'other', all operations in the history that include this parameter are left out of the selected subhistory (because values mentioned explicitly do not match 'other').

It turns out that this approach, with a slight extension, can be generalised for any execution rule; it can thus be used for the first step of the algorithm. The extension deals with cases where quantified variables are used independently (but not together) in operation patterns. In our example, it is impossible to have an 'other' value for `i` when `j` has a known value. But suppose some other execution rule contains the operation pattern `d(k)`, where `k` is a **forall**-quantified variable (of type `int`) in that rule. If there were an operation `d(5)` in the history we are checking, then we would also need to check this rule rule for `i = 1` and `j = 5`, even though this combination of values is not given as parameters to any data operation invocation in the history.

The table shown above illustrates the result of the second step of the algorithm for the given example. All of the subhistories shown in the table are a prefix of the ordering expression of the execution rule when its quantified variables are instantiated with the values of that row. From this, we conclude that the given history is valid for the given execution rule.

Appendix A gives a TM specification of a generic algorithm for checking a history against a set of execution rules. This specification defines a database that contains a set of execution rules, and defines a retrieval method that for a given history returns whether it conforms to all of the execution rules stored in the database. The primary purpose of this specification is to define the algorithm, not to give a realistic representation of the history. For this reason strings are used for the names of the operations and values of parameters. When combining it with the TM specification given in Section 4.3.3, the essential parts of the algorithm are not affected.[8]

4.1.2 Algorithm using Finite State Automata

Although it imposes no restrictions on the execution rules, the algorithm described in the previous section (and in Appendix A) is not very pragmatic: it traverses the history and the ordering expression several times. With some auxiliary storage and rewriting, the algorithm

[8]The specification of the execution rules must be adapted to contain a union type of all types that are used as parameters for the data operation invocation patterns in the ordering expressions of the execution rules.

can be optimised slightly, but its inherent complexity remains. In this section, we introduce an alternative algorithm, based on a collection of Finite State Automata (FSAs). This algorithm generates, for a given set of execution rules, a LOTOS/TM behaviour expression that describes the valid histories for these rules. The algorithm imposes restrictions on the execution rules. This makes it less complex than the previous algorithm.

Because operations are added one-by-one to a history, we would like to check if an operation can be added to a history, without having to keep track explicitly of all previous operations that were added. Placing restrictions on the execution rules makes this possible. One such restriction has already been mentioned in [FaEB96], stating that the rule specifications should be deterministic.

Our approach is to use a set of FSAs to "remember" the history that has been seen so far. A feasible implementation would make use of one FSA for each combination of parameter values. For each new operation added to the history state, transitions in the set of FSAs would occur. When an operation introduces a new value combination for the parameters, some of the FSAs may be 'cloned'. This cloning would mean that an existing automaton (in its current state) is copied to deal with the new values introduced. Ideally, it should be possible to determine *a priori* how the FSAs are cloned in all of the possible histories.

Below, the implementation using a set of FSAs is explained, using the example shown in the previous section. To represent the FSAs, we will use the order expression, and mark the operation invocations that can be chosen next by underlining them. Thus, for an empty history we have:

i	j	FSA
any	any	a(i); (b(i,j); c(i,j))*; d(i)

From this we can conclude that the operation `a` can be added for any value of `i`. In our example, the operation `a(1)` is added as the first operation. Because this operation is the first operation using the parameter `i`, the single automaton in the set has to be cloned into a rule for `i = 1`, and the original automaton needs to be restricted to accept "any value other than `1`." After `a(1)` has been added to the history, the automata state[9] of the rule becomes:

i	j	FSA
any\{1}	any	a(i); (b(i,j); c(i,j))*; d(i)
1	any	a(i); (b(i,j); c(i,j))*; d(i)

Now, we want to consider the next operation in the history, `b(1,3)`. Because the first automaton applies to values of `i` different from `1`, we only look at the second automaton. It is clear that the operation `b(1,3)` can be accepted. Adding this operation to the history causes the second automaton to be cloned again (because we now have the patterns '`3`' and '`other`' for `j`), resulting in the following automata state:

[9] The computation state maintained by the algorithm is the collection of automata shown in the table.

i	j	FSA
any\{1}	any	a(i); (b(i,j); c(i,j))*; d(i)
1	any\{3}	a(i); (b(i,j); c(i,j))*; d(i)
1	3	a(i); (b(i,j); c(i,j))*; d(i)

In the example history, the next operation encountered is `c(1,3)`. The operation can be added to the history, because it can be accepted by the third automaton, which is the only automaton that handles the values `i = 1` and `j = 3`.

Looking at the automata state shown above, the underlined operation patterns illustrate operation invocations that are permitted next. We see that the second automaton allows another `b` operation, with a value for `j` different from `3`, *or* a `d` operation, with `1` as its `i` parameter. However, although the operation `d(1)` is allowed by the second automaton, it is not allowed by the third automaton. (It must be accepted by both the second and the third automaton, because both of these are defined for the parameter value combination with `i = 1`.)

The previous example illustrates the interactive simulation that will be provided by the execution rules analysis tool we have in mind.

In the previous example, the set of automata could be managed by cloning a single automaton, whenever a new value was introduced. This is not the case for all execution rules. Consider, for example, the following rule:

forall `i : int; j : int`
order `[ a(i) ]; b(j); c(i,j)`

Suppose we want to check the history '`a(1); b(4); c(2,4)`.' After adding the operation `a(1)` to the history, we obtain the following automata state:

i	j	FSA
any\{1}	any	[a(i)]; b(j); c(i,j)
1	any	[a(i)]; b(j); c(i,j)

It is clear that the operation `b(4)` can be added to this history, because both automata accept it. But to add it to the history, both of the automata have to be cloned, resulting in the following automata state:

i	j	FSA
any\{1}	any\{4}	[a(i)]; b(j); c(i,j)
any\{1}	4	[a(i)]; b(j); c(i,j)
1	any\{4}	[a(i)]; b(j); c(i,j)
1	4	[a(i)]; b(j); c(i,j)

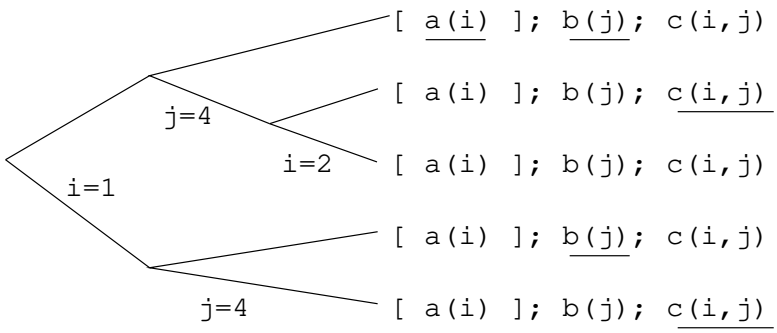


Figure 16: Tree representation of an execution rule automata state.

At this point, it is not possible to perform an `a` operation for any parameter value. Adding an operation `a(1)` is not allowed by the last two automata; and for any other value for parameter `i`, the first two automata need to accept it, but the second one does not. Adding the operation `c(2,4)` causes the *second* automaton to be split, resulting in the following automata state:

i	j	FSA
any\{1}	any\{4}	[a(i)]; b(j); c(i,j)
any\{1,2}	4	[a(i)]; b(j); c(i,j)
2	4	[a(i)]; b(j); c(i,j)
1	any\{4}	[a(i)]; b(j); c(i,j)
1	4	[a(i)]; b(j); c(i,j)

Note that this state does also not allow any `a` operations. It does allow a `b(j)` operation for `j` unequal to 4, and any operation `c(i,j)` for `j` equal 4, and `i` unequal 2.

Tree representation The example given in the previous paragraph shows the complicated bookkeeping that is needed to deal with all the separate combinations of values that have occurred. Observe, for example, that the first two automata in the table above use different sets of values for `i`. As an alternative, the automata can be kept in a tree structure, where each branch in the tree represents a cloning. The tree for the above table is given in Figure 16. In this figure, we only mark the branches that establish a value for one of the quantified variables.

General description of the implementation. We assume that the ordering expressions of the execution rules are deterministic (see Section 3.3.4). For each execution rule, we use a tree structure, where the leaf nodes contain automata, and the intermediate nodes keep track of the values for the **forall**-parameters that have been used so far. Each intermediate node contains the name of one of the **forall**-parameters, and the branches to the underlying nodes are marked with values. A special value 'other' is used to represent those values that have not been used so far.

To check whether a certain operation can be added to the history according to an execution rule, we have to check whether it is accepted by all the applicable automata in the tree. First of all we have to determine the set of all possible value combinations for the operation. If the operation is mentioned more than once in the ordering expression with different **forall**-parameters as arguments, then there might be more than one combination of values for the parameters we have to check.[10] For each of the value combinations thus found, we have to traverse the tree recursively, starting with the root node according to the following rules:

- If the node is a leaf node, the operation should be possible according to the state of the automaton, using the values of the parameters found while traversing the tree.
- If the node is an intermediate node, with a parameter that occurs in the value combination, check the tree below the branch in accordance with the corresponding value.
- If the node is an intermediate node, but the parameter does not occur in the value combination, then check all the nodes below this node.

If the operation can be added to the history, the tree has to be modified by traversing the tree recursively from the root node and applying the following rules:

- If the node is a leaf node, and the parameters with the transition in the automaton are all contained in the set of parameters that were found while walking down to this node, then change the state of the automaton by following the transition, otherwise the tree has to be extended first. The leaf node has to be replaced by an intermediate node, where the parameter at the node is one of the 'new' parameters introduced, and two leaf nodes have to be placed below the node containing the cloned automata of the original leaf node. One branch should be labelled with the value of the parameter, and the other with the 'any other' value. Only the first of the two has to be traversed recursively.
- If the node is an intermediate node, with a parameter that occurs in the value combination, traverse the node with the correct value. If such a node does not exist, then such a node has to be created first by copying the whole tree under the node with the 'any other' value.
- If the node is an intermediate node, but the parameter does not occur in the value combination, then repeat the procedure for all underlying nodes recursively.

4.1.3 Restrictions on the execution rules

When automata need to be cloned in a complex manner, it is often the case that the execution rules they represent have unintuitive semantics. For this reason, posing additional restrictions on the execution rules is beneficial from the user's viewpoint as well. In this section, we discuss the requirements that are needed to map the execution rules to LOTOS/TM behaviour expressions.

[10] In a real implementation, we could use a look-up table instead of having to traverse the complete ordering expression for each operation.

Used/defined occurrences. In order to determine *a priori* when an FSA has to be cloned, we need to know when new combinations of values are introduced. For this reason, the following restriction is placed on the execution rules:[11]

Restriction 1 *For each variable occurrence in an operation invocation pattern in the ordering expression, it must be known whether it is a defining or a using occurrence.*

A *defining occurrence* of a variable in an ordering expression is a place where a new combination of parameter values is introduced. Some example execution rules are shown below, in which the defining occurrences of the parameters are underlined:

```
forall i : int
order  a(i); b(i); a(i)

forall i : int, j : int
forall a(i, j); b(i, j)

forall i : int
order  a(i); [ b(i,j); c(i,j) ]; d(i)

forall i : int
order  a(i); c(i) | b(i); c(i)
```

There are rules for which it is not possible to determine whether each occurrence of a variable is a defining or a using occurrence. For example, consider the following rule that uses an optional construct is:

(1)
```
forall i : int
order [ a(i) ]; b(i)
```

An italic font is used for the ambiguous variable. Here, the occurrence of variable `i` as a parameter to operation `b` is either a defining or using occurrence, depending on whether an operation `a` with the same value for `i` appears before it in the history or not. This rule can be transformed into an equivalent rule that does not have this problem:

```
forall i : int
order    a(i); b(i)
       | b(i)
```

The fact that `[a(i)]` is equivalent to `(a(i) | )` is first used to rewrite the rule; then the part that follows the optional part is copied to both alternatives.

A second kind of execution rule expression that does not meet the restriction on using and defined variable occurrences is the grouping construct. Consider the following rule:

[11] This restriction does not follow from the syntax of the rules.

(2) **forall** `i : int`
order {`a(i), b(i)`}

Whether variable `i` is a defining or using occurrence depends on which operation is done first. Because the grouping construct is syntactic sugar for a special kind of (nested) choice construct, this kind of rule can be rewritten to remove the problem.

A third kind of construct that can cause problems is the one-or-more repetitions construct, as in the following rule:

(3) **forall** `i : int`
order `(a(i); b(i))+`

This rule states that whenever an operation `a` with a certain integer value occurs in the history, it can be followed by alternating invocations of operations `a` and `b` with this same value as a parameter. Each first occurrence of an `a` operation with a certain integer value *defines* the value for the above rule; all of the following invocations of operation `a` *use* the value. Thus, variable `i` in the operation invocation pattern is both a defining and a using occurrence. The rule can be transformed into the equivalent rule, by unfolding the inner expression once:

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

The zero-or-more repetitions ($*$) construct is equivalent to an optional one-or-more repetitions ($+$) construct; for this reason, it can be rewritten using the transformations we have already shown.

We have illustrated some general rules for transforming execution rules that do not satisfy Requirement 1 into forms that do. But there are some complicated rule expressions, such as the following that cannot be rewritten:[12]

(4) **forall** `i : int, j : int`
order `(a(i) | b(j))*`

An attempt to transform this rule using the transformations shown for the choice and the zero-or-more repetitions constructs fails: the transformation process does not terminate, and the expression becomes larger with each transformation step. However, there exists an equivalent execution rule that satisfies Requirement 1:

```
forall i : int, j : int
order  [ a(i); a(i)*; [b(j); (a(i) | b(j))*]
       | b(j); b(j)*; [a(i); (a(i) | b(j))*]]
```

We will accept execution rules that can be transformed by one of the above transformation rules into an execution rule that meets Requirement 1. Execution rules for which the rewriting does not terminate after a certain number of steps (a value set in the COCOA compiler) are refused. In the remainder of this section, we assume that the ordering expressions being considered meet Requirement 1.

[12] This rule does not actually enforce any restrictions on the occurrences of the operations in a history.

Restrictions on using occurrences of a variable In the mapping to LOTOS/TM, each (cloned) automaton is mapped to a process[13] that synchronises with the other processes on the operation patterns they have in common. This means that all of the automata that are cloned need to synchronise on those operation patterns that do not use the **forall**-parameter that induced the cloning. For this reason, we have to place restrictions on all operation patterns that *could use* a certain defined parameter. These comprise all operation patterns that can be followed by the operation that had the defined occurrence of the parameter. In the following example, there are two defining occurrences of the parameter `i`, and two using occurrences:

(5)
```
forall i : int
order    a(i); b(i)
       | c(i); d(i); d(53)
```

The only operation pattern that uses (can use) the first defining (underlined) occurrence of `i` is the pattern `b(i)`. The operation patterns that use (can use) the second defining occurrence are `d(i)` and `d(53)`. The automata cloned at the second occurrence need to synchronise on those operation patterns that do not use the parameter `i`. It now appears that for the operation name `d`, this is not possible because `d(i)` uses the parameter, and `d(53)` does not use it.

As illustrated by the above example, the following restriction needs to be placed on the execution rules:

Restriction 2 *All operation invocation patterns for each possible operation name, which follow a certain operation invocation pattern that has a defining occurrence of a certain parameter, either all use the parameter, or none at all.*

In the above example the first alternative of the choice construct meets this requirement, but the second does not, so this execution rule does not meet Requirement 2.

Restrictions on defining occurrences of a variable We pose additional restrictions on the defined occurrences in the ordering expression inside each choice construct, where some or all of the alternatives start with an operation invocation pattern that has a defined occurrence. If none of the operation invocation patterns at the start introduce new parameters, cloning at this point of the expression[14] cannot occur. An example of such a choice expression is found in the following example:

```
forall i : int
order  a(i); (b(i) | c(i))
```

[13] Or more precisely, a thread of tail-recursive LOTOS/TM process invocations.
[14] Actually, the state of the automaton associated with the subexpression.

But if new parameters are introduced, we have to clone along the values of these parameters, and we have to administrate for which (combinations of) values automata have been cloned. In case the defined parameters are the same for all alternatives, the administration is simple, and there are no problems. Examples of such a choice expression are:

```
forall i : int
order a(i) | b(i)

forall i : int, j : int
order a(i); b(j) | b(i); a(j)
```

In case the defined parameters are not the same for all alternatives, certain restrictions have to be posed, which have to enable us to keep the administration of the (combinations of) values at the start of the expression. Consider the following example:

```
forall i : int, j : int
order a(i); b(j) | b(j); a(i)
```

We see that each of the alternatives has a different defining parameter, e.g., the parameter `i` for the first alternative, and the parameter `j` for the second. Note that we have to keep track of all combinations of values for `i`, and `j`, not just which values have been used for `i`, and `j` independently of each other. In the previous example, cloning only took place at the start of the rule, but in this example, it also takes place in the middle. Suppose we want to check the history of operations `[a(1), b(2)]`, then after `a(1)` has been accepted, we have the following state:

i	j	FSA
any\{1}	any	a(i); b(j) | b(j); a(i)
1	any	a(i); b(j) | b(j); a(i)

Adding the operation `b(2)` causes both of the automata to be cloned, resulting in the following state:

i	j	FSA
any\{1}	any\{2}	a(i); b(j) | b(j); a(i)
any\{1}	2	a(i); b(j) | b(j); a(i)
1	any\{2}	a(i); b(j) | b(j); a(i)
1	2	a(i); b(j) | b(j); a(i)

Both of the clonings follow naturally from observing the current state of the automata, e.g., automata are split for a parameter that is found in the operations that are accepted by it.

The generalised requirement becomes:

Restriction 3 *If a choice contains parameters that appear as defining occurrences in some, but not all operation invocation patterns at the start of the alternatives, then all defining occurrences of this variable occurring in any alternative should have the same signature as one of the operation patterns found at the start of an alternative.*

With the same signature, we means that the operation mentioned in the ordering expression, has the same parameters, the same values, and the same any-values at the same place.

There seem to be examples of expressions that do not meet the above requirement but can still be mapped to LOTOS/TM. Because checking rules by hand takes a lot of time, we propose that these restrictions are researched further when a prototype of the mapping is implemented.

Additional checks There are some situations in which certain combinations of operation invocation patterns do not make sense because of the use of infinite sets. Consider, for example, the following rule:

```
forall i : int
order a(i); b
```

Because this rule has to be enforced for any value of `i`, it requires that an operation `a(i)` is invoked for every possible value of `i`, before operation `b` can occur. If the type `int` has an infinite domain, then operation `b` can never occur. But even if `int` were a finite domain, this is probably not the intention of the specifier.

4.1.4 Mapping to LOTOS/TM

The mapping algorithm presented in this section generates, for a given execution rule ordering expression, a LOTOS/TM behaviour expression that describes the valid histories for this rule. The algorithm assumes that its input meets the requirements stated in the previous section, and that any transformations needed to meet the requirements have been applied. The algorithm can be generalised for a set of execution rules. In this case, all of the behaviour expressions synchronise on those operations they have in common.

The first step of the algorithm is normalisation. Normalisation removes redundant operators from the parse tree to simplify the code generation step. Normalisation removes superfluous parentheses, eliminates redundant operators (such as nested combinations of the zero-or-more repetitions operator: e`**`), simplifies the usage of associative operators (such as '`;` ' and '`|`'), and so on.

The second step of the algorithm assigns a unique name to each subexpression in the ordering expression; these will be used as a name for the generated LOTOS/TM process that represents the expression. The top-level ordering expression is denoted by E; its subexpressions

name(E) The function *name* returns the name assigned its parameter subexpression.

gates(E) The function *gates* returns the names of the operations that are used in operation invocation patterns the given ordering expression.

params(E) The function *params* returns the **forall** and global scenario parameters that are used in the given ordering expression.

call(E) The function *call* returns the code for calling the LOTOS/TM process that represents the parameter subexpression, including a list of arguments and gate names.

pparams(E) The function *pparams* returns the list of arguments used by each LOTOS/TM process that represents the given subexpression.

body(E) The function *body* returns the body of the LOTOS/TM process definition for the given ordering expression.

cont(E) The function *cont* returns for a given subexpression the set of subexpressions that follow it (i.e., its continuation).

calls(E) The function *calls* is applied to a *set* subexpression; it returns a LOTOS/TM choice expression, where each alternative is a call to the process with an element of the set. For example, $calls(\{e_1, e_2\})$ would return $(call(e_1)\texttt{++"[]"++}call(e_2))$. The ++ operator is used to concatenate code fragment strings. When the set consists of a single element, only a single call is returned; for an empty set, "**stop**" is returned.

Figure 17: Auxiliary functions used in the translation from ordering expressions E to LOTOS/TM behaviours.

typeof(o, i) The function *typeof* returns the type of the i-th parameter of data operation o.

setparam(x) The function *setparam* returns the name of the variable (generated by the algorithm) that contains the set of values for which a cloning has taken place.

sync(x, e) The function *sync*(p, e) returns the list of gates associated with the operation invocation patterns found in the set of subexpressions e that do not use the variable x.

Figure 18: More auxiliary functions used in the translation of ordering rules to LOTOS/TM.

are denoted by $E_1, \ldots, E_n$. To aid the definition of the mapping, a number of auxiliary functions are defined to manipulate the syntactic representations of the ordering expression and its subexpressions. These are listed in Figure 4.1.4.

The generated LOTOS/TM behaviour expression for the ordering expression E is of the following form:

e[*gates*(E)](*pparams*(E))
where
process e[*gates*(E)](*pparams*(E)) : **noexit**
    body(E)
endproc
process *name*(E_1)[*gates*(E_1)](*pparams*(E_1)) : **noexit**
    body(E_1)
endproc
...
process *name*(E_n)[*gates*(E_n)](*pparams*(E_n)) : **noexit**
    body(E_n)
endproc

The *body* function takes one arguments, which is the expression for which the body must be generated. In the following generation for the different kinds of ordering expressions e, where e is the parameters of the function *body*. If the function is called for an order expression between brackets — the case where e is of the form `(`e_1`)` — it returns *call*(e_1), and *cont*(e_1) is equal to *cont*(e).

For an **optional ordering expression**, where e is of the form `[`e_1`]`, the *body* function returns *call*(e_1)++`"[]"`++*calls*(*cont*(e)), and *cont*(e_1) is equal to *cont*(e).

For a **zero-or-more ordering expression**, where e is of the form e_1`*`, the *body* function returns *call*(e_1)++`"[]"`++*calls*(*cont*(e)), and *cont*(e_1) is equal to $\{e\}$.

For a **one-or-more ordering expression**, where e is of the form e_1+, the *body* function returns *call*(e_1), and *cont*(e_1) is equal to $\{e_1\} \cup$ *cont*(e).

For a **sequence ordering expression**, where e is of the form e_1`;` e_2, the *body* function returns *call*(e_1), and *cont*(e_1) is equal to $\{e_2\}$. Furthermore, *cont*(e_2) is equal to *cont*(e).

For a **data operation invocation pattern**, the code that has to be generated depends on the fact whether there are defined parameters. In case there are no defined parameters, the following code is generated:

o !p_1 ...!p_m; *calls*(*cont*(e))

assuming that e is of the form $o(p_1, \ldots, p_m)$. In case there is only one defined parameter in the invocation pattern, the code that has to be generated depends on whether the domain of the parameter is finite or not. In case it has a finite domain, we have to keep track of the values for which cloning has taken place. Assuming that p_i is the defined parameter, the following code is generated:

$o\ !p_1\ \ldots ?p_i : typeof(o,i)\ \ldots !p_m$
  [**not**(p_i **in** $setparam(p_i)$)];
  ($name(e)[gates(e)](setparam(p_i)$ **union** $\{p_i\}, params(e))$
    $|[sync(p_i, cont(e))]|$
  $calls(cont(e))$)

The $setparam(p_i)$ parameter is listed as one of the parameters to the LOTOS/TM process $name(e)$. This means that $pparams(e)$ should return the parameter list given by:

$setparam(p_i)$++**","**++$params(e)$,

The function application $call(e)$ returns:

$name(e)[gates(e)]$(**emptyset**($typeof(o,i)$), $params(e)$)

In case the parameter p_i has a finite domain, we keep track of the values that have been used so far, and deal with case when all values have been used. The following code is generated to fit this:

$o\ !p_1\ \ldots ?p_i : typeof(o,i)\ \ldots !p_m$
  [p_i **in** $setparam(p_i)$];
  ([**count**($setvar(p_i)$) > 1]
    $name(e)[gates(e)](setparam(p_i)$ **minus** $\{p_i\}, params(e))$
      $|[sync(o, p_i, cont(e))]|$
    $calls(cont(e))$)
    []
    [**count**($setvar(p_i)$) $= 1$]
    $calls(cont(e))$

Note that the function $call(e)$ should return:

$name(e)[gates(e)](domain(typeof(o,i)), params(e))$

where *domain* returns the finite set of values in the domain of the given type.

In case there is more than one defined parameter, we have to introduce additional LOTOS/TM processes for each additional parameter. We only explain the code generation for two parameters with infinite domains, as cases with more than two defined parameters follow the same principle. Lets assume that these two parameters are p_i and p_j, then the following code will be generated by the function *body*:

$o\ !p_1 \ldots ?p_i : typeof(o,i) \ldots ?p_j : typeof(o,j) \ldots !p_m$
[**not**(p_i **in** $setparam(p_i)$)];
($name(e)[gates(e)](setparam(p_i)$ **union** $\{p_i\}, params(e))$
|[$sync(p_i, cont(e))$]|
($name(e)\_1[gates(e)](p_i,$ **emptyset**($typeof(o,j)$), $params(e))$
|[$sync(p_j, cont(e))$]|
$calls(cont(e))$))
where
process $name(e)\_1[gates(e)](p_i, setparam(p_j), params(e))$: **noexit** =
$o\ !p_1 \ldots !p_i \ldots ?p_j : typeof(o,j) \ldots !p_m$
[**not**(p_j **in** $setparam(p_j)$)];
($name(e)\_1[gates(e)](p_i, setparam(p_j)$ **union** $\{p_j\}, params(e))$
|[$sync(p_j, cont(e))$]|
$calls(cont(e))$))
endproc

For a **choice ordering expression**, the code that has to be generated depends upon the set of defined parameters found at the first operations of the alternatives. If there are no defined occurrences, and supposing that e is of the form $e_1 \mid \ldots \mid e_m$, the following code is generated:

$call(e_1)$
[]
…
[]
$call(e_m)$

If there are defined occurrences at the first operation patterns in the alternatives of the constructs, then there are several possibilities whether they occur with all, or only some of the alternatives. The first possibility is that all the defined parameters are found at all of the first operation patterns of the alternatives. If this is the case, code can be generated just like the case where there are defined occurrences with a operation invocation pattern on its own, as described above. The second possibility is that all the defined parameters are not found at all of the alternatives. In this case, code has to be generated in a different way. The third possibility is when there is a mixture of the first two possibilities, where some defined parameters occur with all alternatives, and some do not. In this latter case, we have to generate nested LOTOS/TM processes, which first deal with the defined parameters that are used with all

alternatives, and then with those that are not used with all alternatives. Below, we discuss the code generation for the different possibilities in the order we mentioned them.

For a choice construct where there is one defined parameter p with an infinite domain for all alternatives, the following code is generated:

o_1 $!p_{1,1}$ ...$?p$: $typeof(o, i_1)$ ...$!p_{m,k_1}$
    [**not**(p **in** $setparam(p)$)];
    ($name(e)$[$gates(e)$]($setparam(p)$ **union** $\{p\}$, $params(e)$)
        |[$sync(p, cont(e_1))$]|
    $calls(cont(e_1))$)
    []
...
    []
o_m $!p_{m,1}$ ...$?p$: $typeof(o, i_m)$ ...$!p_{m,k_m}$
    [**not**(p **in** $setparam(p)$)];
    ($name(e)$[$gates(e)$]($setparam(p)$ **union** $\{p\}$, $params(e)$)
        |[$sync(p, cont(e_m))$]|
    $calls(cont(e_m))$)

This assumes that each alternative starts with an operation invocation pattern, which is valid, because nested choice constructs and group constructs can be folded out, and other constructs cannot occur because of the defined/used requirements. Note that LOTOS/TM processes defined for e_1 to e_m are never called. The functions *pparams* and *call* should return values to accommodate $setparam(p)$. For a choice construct with more than one defined parameter, and/or having finite domains, the code generation is adapted like the case for a single data operation pattern as discussed above.

For a choice construct where each alternative has its own defined parameter (here p_1 to p_m), the following code is generated:

o_1 $!p_{1,1}$ ...$?p_1$: $typeof(o, i_1)$ ...$!p_{m,k_1}$
    [**not**(p_1 **in** $setparam(p_1)$)];
    ($name(e)$[$gates(e)$]($setparam(p_1)$ **union** $\{p_i\}$, . . ., $setparam(p_m)$, $params(e)$)
        |[$sync(p_1, cont(e_1))$]|
    $calls(cont(e_1))$)
    []
...
    []
o_m $!p_{m,1}$ ...$?p_m$: $typeof(o, i_m)$ ...$!p_{m,k_m}$
    [**not**(p_m **in** $setparam(p_m)$)];
    ($name(e)$[$gates(e)$]($setparam(p_1)$, . . ., $setparam(p_m)$ **union** $\{p_m\}$, $params(e)$)
        |[$sync(p_m, cont(e_m))$]|
    $calls(cont(e_m))$)

The functions *pparams* and *call* should return values to accommodate $setparam(p_1)$ to $setparam(p_m)$.

For a choice construct where some alternatives have a defined parameter in common, it means that they have to be joined together. If for example p_3 is equal to p_5, then the functions *pparams* and *call* should return values to accommodate $setparam(p_1)$ to $setparam(p_4)$, and $setparam(p_6)$ to $setparam(p_m)$. Also the *body* function should generate the union of p_5 and $setparam(p_5)$ at the place of $setparam(p_3)$.

In case more than one defined parameter occurs at an alternative, which does not occur at each alternative, we have to introduce set with records for these values.[15] Also the function *sync* has to be extended as to deal with sets of parameters, instead of just a single parameter. It should return the gates that belong to the operation that do not use all of the parameters in the set. Note that there are restrictions when the sets of depending parameter occurrences do overlap.

We explain this with a concrete example of an execution rule. The execution rule is:

```
forall i : int, j : int, k : int
order {a(i,j), b(j,k), c(k,i)}
```

This has to be rewritten into the following rule to meet the defined/used requirements:

```
forall i : int, j : int, k : int
order  a(i,j); (b(j,k); c(k,i) | c(k,i); b(j,k))
     | b(j,k); (a(i,j); c(k,i) | c(k,i); a(i,j))
     | c(k,i); (a(i,j); b(j,k) | b(j,k); a(i,j))
```

The code (with simplified TM syntax) that should be generated for this execution rule, should look like:

```
e1[a,b,c]({}, {}, {})
process e1[a,b,c](a_i_j, a_j_k, a_k_i) =
  a ?i ?j [ not(<i,j> in a_i_j) ];
       (e[a,b,c](a_i_j union {<i,j>}, a_j_k, a_k_i)
            |[b,c]|
        e2[b,c](i,j,{}))
  []
  ...
endproc
process e2[b,c](i,j,a_k)
   b !j ?k [ not(k in a_k) ];
       (e2[b,c](i,j, a_k union {k})
```

[15] TM allows the definition of arbitrary type expressions for this. Although, for a more readable output, it might be better to generate sorts (and sort definitions) for these.

```
            |||
        c !k !i; stop)
  []
   c !j ?k [ not(k in a_k) ];
      (e2[b,c](i,j, a_k union {k})
            |||
        b !j !k; stop)
endproc
...
```

The third combination, that deals with the case where there are both defining occurrences which are found with all alternatives, and some that are not found with all alternatives, makes use of local defined LOTOS/TM processes, where the defined parameters that occur with all alternatives are dealt with first. We illustrate this with the following example:

```
forall i : int, j : int, k : int
order  a(k,i); b(k,j); c(k,i,j) | b(k,j); a(k,i); c(k,i,j)
```

The code that should be generated for this execution rule is:

```
e1[a, b, c]({})
where
  process e1[a, b, c](a_k) =
     a ?k ?i [not (k in a_k)];
          (e1[a, b, c]({k} union a_k)
              |||
           (e2[a, b, c](k, {i}, {})
              |[b]|
            e3[b, c])(k, i, a_j))
      []
     b ?k ?j [not(k in a_k)];
          (e1[a, b, c]({k} union a_k)
              |||
           (e2[a, b, c](k, {} {j})
               |[a]|
            e4[a, c])(k, a_i, j))
  endproc
  process e2[a, b, c](k, a_i, a_j) =
     a !k ?i [not(i in a_i)];
          (e2[a, b, c](k, {i} union a_i, a_j)
              |[b]|
           e3[b, c](k, i, a_j))
    []
     b !k ?j; [not(j in a_j)];
         (e2[a, b, c](k, a_i, {j} union a_i)
```

```
            |[a]|
        e4[a, c](a_i, j))
endproc
...
```

4.2 Organisational view analysis tool

The organisational view analysis tool allows the specification designer the ability to simulate operation enabling as the organisational part (**procedure** definition of the steps) of a CoCoA specification is symbolically executed. The simulation identifies which operations are enabled by the active steps. The tool is intended to help the designer recognise errors in a CoCoA specification, such as an actor not being allowed to export work to the shared workspace. The simulation done by the organisational view analysis tool does not address what the operations are actually doing to the workspaces; it is only a simulation of operation enabling.

In [FaEB96], example LOTOS/TM representations of several CoCoA specifications are given. In this document, a different mapping technique is described, which emphasises TM rather than LOTOS. Here, a TM data structure (defined by module `OrgView` in subsequent specification code) is used to keep track of all active steps of the scenario, and all operations enabled by each of the active steps. Generic methods are defined for step activation, step deactivation, and operation enabling. A generic method is also defined for performing transitions.

The TM data structure is manipulated by a LOTOS/TM process, which offers the enabled operations as events. Generic events are specified for data and data exchange operations. These events correspond to operation invocation *patterns*, rather than operation invocations. Any operation invocation that "matches" the invocation pattern will be allowed to happen when the event is offered. Communication operations are represented as scenario-specific LOTOS/TM events (rather than as generic events) to allow them to receive their parameter values dynamically. Communication operations affect the control flow of the scenario, and their actual parameter values are important in simulating the organisational view.

In a CoCoA specification, communication operations may be conditioned on termination and breakpoint constraints by using an **iff** clause with a **when** construct. Termination and breakpoint constraints are specified in the execution rules part of a specification. The organisational view simulation does not test these constraints, because it does not manage the workspace histories that need to be queried for this information. Simulation of this aspect of a CoCoA specification is instead addressed by the integrated views analysis tool. (See Section 4.4.)

4.2.1 Maintaining active steps in TM

Our algorithm for mapping the organisational view to LOTOS/TM will generate a TM data structure to maintain the set of active steps, and a set of enabled operations for each active

```
Sort StepPath
   type <path : L <step_name: string, params: [| none, chapter: string |] >>
retrieval methods
   contents(out L <step_name: string,
                   params: [| none, chapter: string |] >) = path
   not_substep_of(in s: StepPath, out bool) = not (path sublist contents[s])
end StepPath
```

Figure 19: TM sort specification of step paths.

step. As a first phase of the mapping, we describe how to annotate each point expression (i.e., point name plus type parameters) in a CoCoA specification with a *step path*, which uniquely identifies the point's context. This naming convention serves to "flatten" the points. Transitions are then defined between the flattened points. Depending on the kinds of points that are connected by a transition, it may activate or deactivate steps. Each **enable** construct in a CoCoA specification is considered as a special kind of transition that enables operations, but does not activate more points. Enabled operations are represented by *operation patterns*.

Flattening the step names Based on the static semantics checks identified in Section 3.3, we know certain structural properties of the transitions in a scenario. For example, a transition may only mention the points declared in its enclosing step, or in the substeps of its enclosing step. Using this knowledge, we can "flatten" the references to point names in the scenario, such that they are globally identified by what we call a *step path*. A step path is defined as a list of step names, which shows the nesting structure of the step boxes. Because a step inside a parallel clause takes one or more implicit value parameters, which identify the parallel clause activation, a list entry for a step inside a parallel clause needs to include values to identify the activation. From a CoCoA specification, we have static knowledge of all TM types that are used to identify parallel clauses. Based on these types, we define a variant type that includes them.

Figure 19 includes a TM definition of a step path type. A step path is defined to be a list of records. The `step_name` field is a string representing the name of the step. The variant type used for the `params` field that is specific to the CDA scenario example in Appendix C. For steps immediately nested inside a parallel clause, this field stores the identification of the parallel clause. Otherwise, the value '`[|none|]`' is stored. Two methods are defined on sort `StepPath`: method `contents` returns the `path` attribute of the **self** value; and method `not_substep_of` checks whether the **self** value is a sublist of the step path given as a parameter. Because all steps are uniquely named, the sublist check determines whether the **self** path is a prefix of the argument path. If it is, then the argument path is a substep of the **self** path.

As an example, the step path '`writing.task("intro")`' is represented by the following `StepPath` sort value:

```
   StepPath(<path = [<step_name = "writing",
```



```
Sort OpPattern
   type <actor: string,
         kind: [| data, import, export, comm |],
         op_name: string,
         args: L [| ANY, value: string |] >
end OpPattern
```

Figure 20: TM sort specification of operation patterns.

```
                params = [| none |]
                         as [| none, chapter: string |] >,
               <step_name = "task",
                params = [| chapter = "intro" |]
                         as [| none, chapter: string |] > ]>)
```

Because the TM `StepPath` sort is used to represent the names of the active steps, the dynamic (run-time) identification of the chapter poses no problems.

Operation patterns Now we define a TM sort for *operation patterns*, which are used to represent the operations that are enabled. Figure 20 defines a TM sort for the patterns. The structure of this type follows the LOTOS/TM event offers that are described in [FaEB96]. Before explaining this type definition, we first review the LOTOS/TM event offers. The following CoCoA enable operation:

```
on start enable
   ed : addChapter(ed,"title",_)
```

was represented in [FaEB96] by the following LOTOS/TM event offer:

```
oper !ed !"data" !"addChapter" !ed !"title" ?text: string
```

where '`ed`' has a specific value. We use a slightly different representation of operation patterns here.

The sort definition in Figure 20 includes record fields for the first three event attributes (the actor, the kind of operation, and the operation name), the remaining attributes are given as a list of values of type '`[|ANY,value:string|]`'. The `ANY` variant case is used for anonymous placeholders in the operation pattern. The above `addChapter` operation is represented as a TM `OpPattern` value as follows:

```
OpPattern(<actor = ed,
           kind = [| data |] as [| data, import, export, comm |],
           op_name = "addChapter",
           args = [ [| value = ed |] as [| ANY, value: string |],
                    [| value = "title" |] as [| ANY, value: string |],
                    [| ANY |] as [| ANY, value: string |] ] >)
```

```
module OrgView
(* definitions of sorts StepPath, OpPattern ... *)
module section
attributes
   active_steps: P <step_name: StepPath, opns: P OpPattern>
module constraints
   key_active_steps: active_steps key step_name
module retrieval methods
   enabled_for_actor(in actor: string, out P OpPattern) =
      collect o for o in unnest(collect s.opns for s in active_steps)
      iff (o.actor = actor)
module update methods
   activate(in set_of_steps: P StepPath) =
      self except (active_steps =
         active_steps union (collect <step_name = s,
                                      opns = emptyset(OpPattern)>
                             for s in set_of_steps
                             iff (not (exists a in active_steps
                                       | (a.step_name = s))) ) )
   deactivate(in set_of_steps: P StepPath) =
      self except (active_steps =
         collect a for a in active_steps
         iff (forall s in set_of_steps | (not_substep_of[s](a.step_name))) )
   enable(in set_of_step_enables: P <step_name: StepPath, ops: P OpPattern>) =
      self except (active_steps =
         replace <step_name = a.step_name,
                  opns = a.opns union
                         unnest(collect s.ops for s in set_of_step_enables
                                iff (s.step_name = a.step_name))>
         for a in active_steps
         iff (exists s in set_of_step_enables | (s.step_name = a.step_name)))
end OrgView
```

Figure 21: TM specification of active steps.

Note that the `OpPattern` value represents a pattern for the allowed operations, not an operation invocation. The '`ed`' value is also referred to in this definition.

Modelling the set of active steps Figure 21 defines methods for maintaining the set of active steps in the scenario. Attribute `active_steps` of the module holds a set of step paths, each coupled with the operation patterns which are enabled by the step. The `enabled_for_actor` method queries the set of active steps to retrieve the set of all operation patterns that are enabled for a specific actor. Three methods change the state of the scenario, with respect to the set of active steps: `activate` makes a given set of steps active, each with an empty set of enabled operations (the method has no effect when a step is already active); `deactivate` removes a given set of steps from the set of steps, including any active substeps; and for each of a given set of (*step name*, *operations*) pairs, `enable` makes the operations active for the step.

```
Sort Point
   type <step_name: StepPath,
         point_name: string,
         kind: [| in_point, out_point, signal_point, interrupt_point |],
         params: [| none, authors: P string |] >
end Point

Sort Transition
   type <from: Point,
         to: [| transfer: Point, enable: P OpPattern |] >
constraints
   from_not_to: case to of
                   enable = sp : true
                   transfer = p :
                      not(from.step_name = p.step_name
                         and from.point_name = p.point_name)
                endcase
   obeys_direction:
      not(from.kind is interrupt_point)
      and case to of
            enable = p : not(from.kind is out_point)
            transfer = p :
               not(p.kind is signal_point)
                  and (from.kind is in_point implies p.kind is in_point)
                     and (p.kind is out_point implies from.kind is out_point)
          endcase
end Transition
```

Figure 22: TM specification of points and transitions.

Modelling transitions At the activation of a step, a number of operations are enabled. These include data operations and data exchange operations, neither of which alter the control flow of the scenario. Communication operations are also enabled, and these do alter the control flow of the scenario. Communications are also responsible for providing the input parameters to transitions that pass values. Because of the dynamic nature of their parameter values, communications will be modelled by methods. These methods are described in the next section. In this section, we define a generic method for invoking transitions and enabling operations.

Figure 22 defines TM sorts for points and transitions. A `Point` record includes fields for the (flattened) step name in which it is declared, the kind of the point, and any parameters to the point. We use a variant type for the parameters, with cases for each possible kind of parameter. This variant type is dependent on the source CoCoA specification, which in this case is the CDA scenario in Appendix C.

Transitions and enable constructs are both modelled by the `Transition` sort, whose '`to`' field is a variant with cases for both constructs. Constraints are defined on transitions to guarantee that the points are properly connected. These constraints echo the requirements


```
module OrgView
(* definitions of sorts Point, Transition, ... *)
module section
attributes
   ref: string,
   ed: string,
   authors: P string,
   transitions: P Transition,
   active_steps: P <step_name: StepPath, opns: P OpPattern>
module update methods
   do_transitions(in set_of_points: P Point) =
      if count set_of_points = 0 then self
      else do_transitions[
               enable[
                  deactivate[
                     activate[self]
                        (collect p.step_name for p in set_of_points
                         iff (p.kind is in_point))
                  ](collect p.step_name for p in set_of_points
                    iff (p.kind is out_point))
               ](collect <step_name = t.from.step_name,
                          ops = t.to on enable>
                 for t in transitions
                 iff (t.from in set_of_points and t.to is enable))
          ](collect t.to on transfer for t in transitions
            iff (t.from in set_of_points and t.to is transfer))
      endif
end OrgView
```

Figure 23: TM definition of a method for transition invocation.

discussed in Section 3.3. It is possible to define additional constraints on the `Transition` sort to ensure that step nesting requirements (see Section 3.3) are respected by the flattened points. Most nesting requirements can be expressed as checks on the step paths of the two points of a transfer transition. For example, either (a) the two step paths should be the same length, and all step names but the last should be equal (indicating the transition is between steps in the same enclosing step box); or (b) the step paths should differ in length by one, and the shorter path should be a sublist of the longer one (indicating a transition between a point declared in an enclosing step box, and a point declared in one of its substeps). These two checks describe nearly all of the scoping restrictions. However, some (albeit unnatural) transitions between steps in different parallel clause activations can be specified in COCOA, which cannot be easily described by a constraint on the step paths of the two flattened points. We omit a nesting constraint on the transition sort, and instead, we assume the TM code generated for the source COCOA specification respects the nesting restrictions.

Figure 23 defines a generic TM method for "invoking" the transitions from a given set of points. For an input set of points, the `do_transitions` method activates the steps associated with `in_points`, deactivates the steps associated with `out_points` (along with their

active substeps), enables the operation patterns associated with `enable` transitions, and finally, recursively invokes itself on the set of points that appear as targets in `transfer` transitions.

The value of the `transitions` attribute of the TM module `OrgView` needs to be "filled in" based on a source COCOA specification. As an example, suppose we have the following (flattened) transition, taken from the CDA scenario:

```
on prepare.start do prepare.write_proposal.start
```

It will be represented by the TM value of type `Transition` that is shown in Figure 24. If parallel clauses are defined in the COCOA specification, or if value parameters are attached to points, the set of transitions will be infinite. As a second example, consider the following enable construct from the CDA scenario:

```
step writing[in start, out done]
  parallel(CH : chapter)
    step task[in start(ℙ author), out compl]
    begin
      on start(SA) enable
        for a in SA allow
          a : editChapter(a, CH, _)
```

The set of `editChapter` operations that can be enabled by the flattened interaction point '`writing.task(CH).start(SA)`'(assuming particular values for `CH` and `SA`) is modelled by the set of transitions specified in Figure 25. An infinite set of transitions is described, one for each possible chapter identification used by the parallel clause (the `CH` value), and each possible set of authors (the `SA` value). We make use of predicated sets in the TM specification. It's obvious from this specification that TM's definition mechanisms are quite powerful. (Remember that the TM code is a formal specification, not an implementation.)

4.2.2 Simulation of operation enabling in LOTOS/TM

In this section, we define a LOTOS/TM process that uses the TM data structures introduced in the previous section. The process offers the enabled operations repetitively, until a communication operation is chosen. The communication influences the control flow of the scenario, such that a different set of steps and operations becomes active. The activation and deactivation of steps, and the respective enabling and (implicit) disabling of operations is modelled by a LOTOS/TM process.

Operation enabling using events Figure 26 specifies a LOTOS/TM process that offers enabled operations as events. The process takes two parameters: a set of authors (provided as a parameter to the scenario definition), and the set of active steps. The process is defined generically for data and data exchange operations, as seen in the following code fragment:


```
Transition(
   <from = Point(
               <step_name =
                   StepPath(<path = [<step_name = "prepare",
                                      params = [| none |]
                                               as [| none,
                                                     chapter: string |] >]
                                >),
                point_name = "start",
                kind = [| in_point |] as [| in_point,
                                            out_point,
                                            signal_point,
                                            interrupt_point |],
                params = [| none |]
                         as [| none, authors: P string |] >),
     to = [| transfer =
               Point(
                  <step_name =
                     StepPath(<path = [<step_name = "prepare",
                                        params = [| none |]
                                                 as [| none,
                                                       chapter: string |] >,
                                       <step_name = "write_proposal",
                                        params = [| none |]
                                                 as [| none,
                                                       chapter: string |] >]
                                   >),
                   point_name = "start",
                   kind = [| in_point |] as [| in_point,
                                               out_point,
                                               signal_point,
                                               interrupt_point |],
                   params = [| none |]
                            as [| none, authors: P string |] >) |]
          as [| transfer: Point, enable: P OpPattern |]
   >)
```

Figure 24: TM representation of the flattened COCOA transition: '**on** `prepare.start` **do** `prepare.write_proposal.start`'.

```
unnest (collect (collect
    Transition(
       <from = Point(
                   <step_name =
                       StepPath(<path = [<step_name = "writing",
                                          params = [| none |]
                                                   as [| none,
                                                         chapter: string |] >,
                                         <step_name = "task",
                                          params = [| chapter = CH |]
                                                   as [| none,
                                                         chapter: string |] >]
                                    >),
                    point_name = "start",
                    kind = [| in_point |] as [| in_point,
                                                out_point,
                                                signal_point,
                                                interrupt_point |],
                    params = [| authors = SA |]
                             as [| none, authors: P string |] >),
         to = [| enable =
                (collect OpPattern(
                        <actor = a,
                         kind = [| data |] as [| data, import, export, comm |],
                         op_name = "editChapter",
                         args = [ [| value = a |] as [| ANY, value: string |],
                                  [| value = CH |] as [| ANY, value: string |],
                                  [| ANY |] as [| ANY, value: string |] ]
                        >)
                 for a in SA) |]
              as [| transfer: Point, enable: P OpPattern |]
       >) for CH in { c : string | true })
    for SA in { sas : P string | sas subset authors })
```

Figure 25: TM representation of the enabling construct on page 67 as a set of transitions.

```
process enable[oper](authors: P string, active_steps: OrgView) :=
choice a:string in authors
[] (choice e:OpPattern in enabled_for_actor[active_steps](a)
    [] ([e.kind is data or e.kind is import or e.kind is export] ->
            oper !a !e.kind !e.op_name !e.args
            ; enable[oper](authors, active_steps)
        []
        [e.kind is comm and e.op_name = "startTask"] ->
            oper !a !e.kind !e.op_name ?CH: string ?SA: P string
                   [SA subset authors] ;
            enable[oper]
                  (authors,
                   do_transitions[active_steps]({ Point(
                        <step_name = StepPath(
                              <path = [<step_name = "writing",
                                        params = [| none |]
                                                 as [| none,
                                                       chapter: string |] >,
                                        <step_name = "task",
                                         params = [| chapter = CH |]
                                                  as [| none,
                                                        chapter: string |] >
                                    ]>),
                         point_name = "start",
                         kind = [| in_point |] as [| in_point,
                                                     out_point,
                                                     signal_point,
                                                     interrupt_point |],
                         params = [| authors = SA |]
                                  as [| none, authors: P string |] >) })
                                   )
        []  . . . (* choices for other communication operations *) . . . ) )
endproc
```

Figure 26: LOTOS/TM process that offers enabled operations as events.

```
process enable[oper](authors: ℙ string, active_steps: OrgView) :=
choice a:string in authors
[] (choice e:OpPattern in enabled_for_actor[active_steps](a)
    [] ([e.kind is data or e.kind is import or e.kind is export] →
            oper !a !e.kind !e.op_name !e.args
            ; enable[oper](authors, active_steps)
        [] ··· ) )
endproc
```

For each actor in the set of authors, an event is offered for each operation that is enabled for the actor. The events are based on the operation patterns, rather than actual operation invocations. In the code fragment, a guard on the event checks whether the operation pattern is a data or data exchange operation. If so, the event is offered on gate `oper`, using the parameter value pattern list as the last attribute. Compare this event to the event (and its operation pattern) shown on Page 63. We could accommodate this more complex kind of event by using a complex cases construct to decompose the `args` attribute of the operation pattern.

In the simulation of the organisational view, we are only concerned about what operations are enabled; we do not apply data and data exchange operations to the workspaces. For this reason, it is sufficient to offer operation patterns as events. The more complete form of operation event that appears on Page 63 will be used in the integrated views analysis tool, which also takes execution rules into account. (See Section 4.4.)

Communications as LOTOS/TM events The only parts of the TM specification shown in Section 4.2.1 that are *dependent* on a CoCoA specification are the types of parameter values that are used in step paths and points. (Refer to Figures 19 and 22.) Communications are also dependent on the CoCoA specification. We cannot describe communications in a generic way, because the actual parameter values to a communication must be embedded in the point representation that becomes activated as a result of the communication. Furthermore, some value parameters to the LOTOS/TM event may be made parameters to a step name in the step path, and others may be used as parameters to the point itself. To allow us to describe parameter passing such as this, each communication will be mapped to a scenario-specific LOTOS/TM event, which "catches" all of its parameter values. Once the event takes place, the generic `do_transitions` method is invoked on the point that becomes activated by the communication. A TM representation of this point is constructed using the parameter values.

The LOTOS/TM event representation of the following communication operation (from the writing step of the CDA scenario):

```
when ed issues startTask(CH,SA)
   iff SA subset authors do writing.task(CH).start(SA)
```

is shown in Figure 26. On a synchronisation attempt, the event receives values for the `CH` and `SA` parameters, and checks whether the `SA` value is a subset of the set of authors. If so, the event happens. The process is then recursively instantiated with the set of active steps that results from performing the transitions enabled by the `writing.task(CH).start(SA)` point. This is specified by applying the `do_transitions` method to the active steps, with the TM representation of the point given as a parameter.

4.3 Transactional view analysis tool

The transactional view analysis tool allows the specifier to study the effects of the TRANSCOOP Transaction Model on the cooperative scenario by simulating its history administration. This simulation will be a symbolic simulation, meaning that data operations are only considered at an abstract level. This, in turn, will be reflected by accounting for each data operation invocation by adding an item to the workspace history at hand. The rationale for this abstraction stems from the observation that in quite a few circumstances of collaboration, the actual data operations may be complex, interactive procedures that may easily distract from the purpose of transaction simulation itself.

In this section, we describe the representation of aspects of the transactional view in TM for simulation purposes. It is easy to describe histories (operation invocation sequences) by a TM data structure. Likewise, the merge algorithm can also be expressed in terms of methods on this data structure. The following will be described in TM:

- Commutativity relations ($\oplus$ and $\diamond$)
- Workspace histories (with execution rule constraints)
- Queries over a workspace history to select operations for exchange
- Methods for determining dependencies between operations in a workspace history
- Methods for merging the operations selected from a source workspace history with the operations in a destination workspace history

The above seems to specify a simulation of the transaction model, especially if we would use the TM Abstract Machine to simulate database computations. But note that we only consider the data operations at an abstract level (i.e., their signatures), without semantics. We do not model their application to the workspaces in the transaction view analysis, so as not to duplicate the run-time environment. Furthermore, we do not model compensating actions in the transaction view analysis, and we do not model the merging algorithm in the case of conflicts. We can fix this by assuming that compensation simply removes the conflicting operations from a history. At the moment, however, it is not clear from the cooperative transaction model whether compensating actions should also be stored in the workspace histories, so we do not model them.

The TM data structures we define for the transaction view analysis tool serve as a formal specification of the workspace data structures. The TM specifications can be given as input to the TM Abstract Machine for simulation of the dynamic semantics of the transactional view of a COCOA specification. Before we begin introducing data structures, we look at a concrete example that illustrates the idea of operation invocations as events. The example also illustrates some issues in data modelling. This same example will be used again later, when we discuss the commutativity analysis tool in Section 4.5.


```
Sort Table
   type <ENTRIES: P <name: int, item: string>>
   constraints
      keyname: ENTRIES key name
   retrieval methods
      defined(out P int) = collect p.name for p in ENTRIES
      size(out int) = count ENTRIES
      lookup(in n: int, out string) =
          (unique for p in ENTRIES iff (p.name = n)).item
   update methods
      insert(in n: int, s: string) =
         self except (ENTRIES = ENTRIES union {<name = n, item = s>})
      delete(in n: int) =
         self except (ENTRIES = {p in ENTRIES | (p.name <> n)})
      modify(in n: int, s: string) =
         self except (ENTRIES = replace <name = n, item = s>
                                for p in ENTRIES iff (p.name = n))
end Table
```

Figure 27: TM sort specification of a table.

4.3.1 Badrinath's Table

As an example, we look at the table data structure described in [BaRa92]. The table stores (`name`, `item`) pairs, where the name of each entry is unique. Five operations are specified for the table: *insert*, *delete*, *modify*, *size*, and *lookup*. The informal semantics of the operations is described as follows. The operation *insert* inserts a new pair (`name`, `item`) in the table. If the `name` is already present, it returns *Failure*, otherwise it returns *Success*. The *delete* operation deletes the pair with the given `name` from the table. If the `name` is not present, it returns *Failure*, otherwise it returns *Success*. The *modify* operation modifies the value of the item associated with the given `name`. If the `name` is not present, it returns *Failure*, otherwise it returns *Success*. The *size* operation returns the number of entries in the table. *Lookup* returns the value of the item associated with the given `name`, if it exists in the table. If no such item exists, the result returned is the string `"not_found"`.

The TM specification of a table is given in Figure 27. In TM, each table entry is modelled as a record with `name` and `item` components. The uniqueness requirement is modelled as a TM constraint on the set of entries:

constraints
`keyname: ENTRIES` **key** `name`

Here, `keyname` identifies the constraint.

Now let's compare the operational semantics explained above (given in [BaRa92]) with the TM specification in the figure. It's not possible to directly represent the operational semantics

in TM, because a TM update method cannot be defined to return both an updated value *and* status information. The TM methods are defined for "correct" input arguments; the result of applying a method to "incorrect" input arguments is undefined. Two of the methods for sort `Table` (*insert* and *lookup*) can give rise to undefined results. The *insert* operation results in a constraint violation if the `name` of the entry to be inserted is already in the table (and a different value is used for the `item`). (In such a case, unioning a singleton set containing the new entry with the set `ENTRIES` would result in a new set of records for which `name` is no longer unique. This violates constraint `keyname`.) The *lookup* operation is not well-defined when an entry for the given `name` argument does not exist in the table. In this case, the **unique** construct is not defined, since it is applied to a set that does not contain exactly one element. The two remaining retrieval methods (*defined* and *size*) are well-defined for all input arguments. The two remaining update methods (*delete* and *modify*) are also well-defined for all input arguments; they simply "do nothing" to the table when an entry for the given `name` does not exist.

In order to return an indication of *Success*/*Failure* for the above update methods, we include this information as part of an LOTOS/TM process that encapsulates a `Table` value. Such a process is specified in Figure 28. The process offers the `Table` methods as events. The functionalities of the update methods are enhanced by the event offers. The event offers for the update methods include as parameters the input arguments to the method invocation, plus a *Success*/*Failure* indication. Whenever "incorrect" input arguments are provided, *Failure* is indicated and the table is left unchanged in the recursive instantiation of the process. The TM retrieval method *defined* on sort `Table`, which returns the set of `name` fields that are stored in the table, is used to aid in the definition of the LOTOS/TM process.

The process offers events for allowed method invocations on the `Table` value. The event offers are dependent on the *current* value of the `Table` parameter. We can observe from the specification of the two `insert` event offers that if the given `name` is already defined in the table, *Failure* is indicated. Otherwise, if the `name` is not defined in the table, *Success* is indicated and the `insert` update method is applied to the `Table` value. Analogously, the behaviours at the `modify` and `delete` gates also follow the operational semantics suggested earlier. However, note that there is a subtle distinction between a method invocation event that returns *Failure*, and a method invocation that violates database constraints. Earlier, we pointed out that the methods *modify* and *delete* are well-defined for all input arguments; they do nothing to the table if an entry for the `name` argument does not exist. Irrespective of this, the events in the LOTOS/TM process check whether an entry for the name exists. The corresponding method invocation events return *Failure* if it does not, regardless of the fact that the database constraints would not be violated if the method were applied. This subtle distinction is important for defining certain state-independent relations over methods, and also for method re-execution. (See Section 4.3.2.)

The selection predicate of the `lookup` event offer requires some explanation. First of all, we remark that for this event offer, as for the other event offers that involve method invocations, both event parameters (`n` and `s`) must be bound to values *before* the methods in the selection predicate can be applied. The selection predicate for the *Success*ful version of the event uses a conditional expression to avoid application of the *lookup* method to a `name` parameter that is undefined. In such a case, *false* is returned by the **else** branch as the result of the conditional, and the *lookup* method in the **then** branch is not applied to the non-existent entry.

```
process encapsulated_table[insert,delete,modify,size,lookup]
                          (t: Table): noexit
:=
    insert ?n: int ?s: string !Failure [ n in defined[t] ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    insert ?n: int ?s: string !Success [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](insert[t](n,s))
[]
    delete ?n: int !Failure [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    delete ?n: int !Success [ n in defined[t] ]
    ; encapsulated_table[insert,delete,modify,size,lookup](delete[t](n))
[]
    modify ?n: int ?s: string !Failure [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    modify ?n: int ?s: string !Success [ n in defined[t] ]
    ; encapsulated_table[insert,delete,modify,size,lookup](modify[t](n,s))
[]
    size !size[t]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    lookup ?n: int !"not_found" [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    lookup ?n: int ?s: string
         [ if n in defined[t] then s = lookup[t](n) else false endif ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
endproc
```

Figure 28: LOTOS/TM specification of a process that encapsulates a `Table` value.

```
process encapsulated_table[insert,delete,modify,size,lookup]
                          (t: Table): noexit
:=
    insert ?n: int ?s: string [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](insert[t](n,s))
[]
    delete ?n: int [ n in defined[t] ]
    ; encapsulated_table[insert,delete,modify,size,lookup](delete[t](n))
[]
    modify ?n: int ?s: string [ n in defined[t] ]
    ; encapsulated_table[insert,delete,modify,size,lookup](modify[t](n,s))
[]
    size !size[t]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    lookup ?n: int !"not_found" [ not(n in defined[t]) ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
[]
    lookup ?n: int ?s: string
         [ if n in defined[t] then s = lookup[t](n) else false endif ]
    ; encapsulated_table[insert,delete,modify,size,lookup](t)
endproc
```

Figure 29: A LOTOS/TM process that encapsulates a `Table` value, but only offers successful method invocations as events.

The example serves to illustrate several things we would like to be able to prove about a LOTOS/TM specification. For example, we may want to prove that the method invocations that appear in the process definition do not violate the database constraints. We might also want to prove algebraic properties about sequences of operations on a `Table`. For example, we may want to show that a (successful) *insert* operation, followed by a (successful) *lookup* operation indeed returns the value of the `item` associated with the entry that was inserted.

As an alternative to the previous LOTOS/TM process, we can define another process, which only allows successful method invocations to take place. This process is shown in Figure 29. The process offers successful method invocations on the `Table` as events.

4.3.2 Representing history rules in TM

In this section, we look at the representation of history rules in TM. History rules are used to describe semantic dependency relations between method invocations. COCOA offers the specification designer the ability to define **noncommutative** relations between operation invocations. The relations are state-independent, but they can depend on actual parameter values to the operations.

Figure 30 gives an example commutativity table from [BaRa92] for the table example we looked at earlier. For our purposes, we shall refine this table to express commutativity in a negative

OPERATION REQUESTED	OPERATION EXECUTED				
	insert	delete	lookup	size	modify
insert	yes-dp	yes-dp	yes-dp	no	yes-dp
delete	yes-dp	yes-dp	yes-dp	no	yes-dp
lookup	yes-dp	yes-dp	yes	yes	yes-dp
size	no	no	yes	yes	yes
modify	yes-dp	yes-dp	yes-dp	yes	yes-dp

Figure 30: Commutativity table from [BaRa92]. (The notation "yes-dp" stands for "yes, if the parameters are different.")

way (i.e., non-commutativity). Moreover, we restrict our consideration to update methods. In COCOA, only the update method invocations (i.e., state transformations) are stored in an actor's workspace history.

Figure 31 illustrates the specification of a non-commutativity relation as a TM method, defined on sort `Opn`. Sort `Opn` is used to model operation invocations. The sort specifies a variant type, with a tag for each operation name. The record value stored for an operation tag holds the operation invocation's actual parameters. The `non_commutative` method takes a second operation invocation as a parameter. The method checks whether the `name` field is the same for the two operation invocations. If it is, the operation invocations do not commute. Notice that the `non_commutative` method does not depend on the value of the `item` field. This observation will be important in the integrated views simulation. (See Section 4.4.)

When the `non_commutative` method is used in later sections, we assume its `Opn` argument is an operation invocation that occurs *before* the **self** argument in a workspace history. It is worthwhile to point out that we can use a similar TM specification to define other, non-symmetric relations between operation invocations.

4.3.3 Workspace histories

A workspace history needs to record the sequence of operations invoked on a workspace. A set of workspace histories will be maintained, one for each workspace. A TM module definition for a set of workspace histories is given in Figure 32. The definition is dependent on (but modular to) the `Opn` sort, defined in the previous section. The `History` sort defines a history `LOG` as a list of records, each with an operation field '`op`', and an action identification field '`aid`'. An action identification uniquely identifies an operation invocation. The constraint `unique_aid` ensures that the `aid` component of each element in the `LOG` list uniquely identifies the element.

The module section of the code declares three persistent attributes: `maxaid`, the maximum action identification tag used so far; `wsp_names`, a set of workspace names; and `histories`, a set of records, each with a workspace name field '`wid`', and a '`history`' field. Constraints are defined on the module's attributes to ensure that for each workspace name, a workspace history is defined, and for each workspace history, all of its action identifications are in the

```
Sort Opn
   type [| insert: <name: int, item: string>,
           modify: <name: int, item: string>,
           delete: <name: int> |]
   retrieval methods
      non_commutative(in o2: Opn, out bool) =
         case self of
           insert = p1 : (case o2 of
                             insert = p2 : p1.name = p2.name
                             modify = p2 : p1.name = p2.name
                             delete = p2 : p1.name = p2.name
                          endcase)
           modify = p1 : (case o2 of
                             insert = p2 : p1.name = p2.name
                             modify = p2 : p1.name = p2.name
                             delete = p2 : p1.name = p2.name
                           endcase)
           delete = p1 : (case o2 of
                             insert = p2 : p1.name = p2.name
                             modify = p2 : p1.name = p2.name
                             delete = p2 : p1.name = p2.name
                           endcase)
         endcase
end Opn
```

Figure 31: Specification of a non-commutativity relation as a TM method.

```
module Workspaces
(* definition of sort Opn... *)
Sort History
   type <LOG: L <op: Opn, aid: int>>
   constraints
      unique_aid: count (collect g.aid for g in LOG) =
                  count (collect g.aid for g in unlist(LOG))
   update methods
      doop(in o: Opn, a: int) =
          self except (LOG = LOG concat [<op = o, aid = a>])
end History
module section
attributes
   maxaid : int,
   wsp_names : P string,
   histories : P <wid: string, history: History>
module constraints
   posmax : maxaid >= 0
   history_defined : forall w in wsp_names
                        | (exists h in histories | (h.wid = w))
   unique_wid : histories key wid
   histories_okay : forall h in histories
                       | (h.wid in wsp_names and
                         (forall a in (collect g.aid for g in h.history.LOG)
                             | (a >= 0 and a <= maxaid)))
module update methods
   do_op(in w:string, o: Opn) =
        self except
        (maxaid = maxaid+1,
         histories = replace <wid = g.wid,
                              history = doop[g.history](o,maxaid+1)>
                     for g in histories iff (g.wid = w))
end Workspaces
```

Figure 32: Specification of workspace histories in TM.

range of used values. Because of data exchange, the same action identification, operation invocation combination can appear in more than one workspace history. We have chosen not to use a class definition for operation invocations, because although the same invocation can appear in different histories, it is actually "re-executed" when it is put in another history, meaning that the invocation is not "shared" by the histories that include it, as would be suggested by using objects instead.

The specification defines two methods for adding an operation to a workspace history, one on the `History` sort, and another on the module. First, an unused action identification is chosen for the operation invocation, and then together they are stored in a new record that is concatenated to the end of the `LOG` list for the workspace. Additional methods on workspace histories are defined in the next section.

Constraints for execution rules Method `do_op` of the specification in Figure 32 simply concatenates a new operation onto the end of a workspace history (using method `doop`). The methods can be modified to accommodate the execution rule checks by using the TM specification in Appendix A. The set of execution rules needs to be added to the `Workspaces` module, as another attribute:

module section

```
...
Exec_rules: ℙ exec_rule
```

But before this can be done, unification of the representations of operation invocations in the two specifications is needed: the specification in the appendix is very general; whereas the specification here uses operation signatures that are specific to the scenario definition, because of the commutativity relation. Because the commutativity relation can not be made generic (it depends on actual parameter values), the specification in the appendix should be made specific to the operations in the scenario. We do not show how to do this here. Once this is done, we can invoke the method `correct_hist` (from the appendix) on the hypothetical history that is obtained by adding the new operation. If the method returns true, the new operation invocation is added to the workspace history, otherwise the workspace history is left unchanged.

In addition to adding the test on the execution rules to the `do_op` method, we can add a constraint to the module specification to ensure that each workspace history value obeys the execution rules. The constraint would take the following form:

module constraints

```
histories_obey_execution_rules:
   forall h in histories | (correct_hist[self](h.history))
```

Note that this constraint definition has consequences for importing a subhistory into a given workspace history; it must hold for the history that results from the data exchange operation.

4.3.4 Methods on workspace histories

In this section, we define several methods on workspace histories. To give an idea of what methods are needed in order to simulate aspects of the transaction model, we first show a brief example of data exchange. TM method definitions follow the example. As a matter of convenience, the word "operation" is used to abbreviate "operation invocation" in the description of the example, and elsewhere in the text when it does not cause ambiguities.

An example Suppose we are given the following two histories, where $\mathcal{H}_1$ is the source history, and $\mathcal{H}_2$ is the destination history:

$$\mathcal{H}_1 = a_1; a_2; b_1; a_3; b_2; b_3; a_4; a_5$$
$$\mathcal{H}_2 = c_1; c_2; c_3; c_4$$

Suppose operations b_1, b_2, and b_3 have been selected from $\mathcal{H}_1$ for exchange. The b operations are said to be non-commutative (semantically dependent) on each other, so they must be exchanged as a unit. Based on the data exchange protocol, we know that the b operations are semantically independent from the a operations, and they can be moved "before" the a operations in the workspace history. Thus, the following history is equivalent to $\mathcal{H}_1$:

$$\mathcal{H}_1' = b_1; b_2; b_3; a_1; a_2; a_3; a_4; a_5$$

In the two histories $\mathcal{H}_1$ and $\mathcal{H}_1'$, each b operation is said to *commute backward* with the a operations that it is moved before.

We want to be able to construct the following history as the result of the data exchange:

$$\mathcal{H}_3 = c_1; c_2; c_3; c_4; b_1; b_2; b_3$$

That is, we want to put the exchanged operations at the end of the destination history. If we know that each of the b operations is semantically independent from each of the c operations, we know that $\mathcal{H}_3$ is equivalent to the following history:

$$\mathcal{H}_3' = b_1; b_2; b_3; c_1; c_2; c_3; c_4$$

We will show how to derive history $\mathcal{H}_3'$ momentarily. Working backwards, we know that workspace histories $\mathcal{H}_1$ and $\mathcal{H}_2$ have both started on the same workspace state. This is an assumption made by the transaction model, and likewise also by COCOA. Thus, we can execute the b operations at the front of $\mathcal{H}_3'$, when it is empty, without constraint violations, before the c operations are executed. To show how to derive $\mathcal{H}_3$, we need to know that each of the b operations *commutes forward* with each of the c operations, and can thus be moved "after" the c operations in the workspace history. Additionally, we should know that all preconditions of the c operations still hold if the b operations are done first.

Methods From the previous example, we see the need for two methods on workspace histories: a method for selecting the operation invocations from a history on which a given set


```
Sort History
(* ... *)
retrieval methods
   depends_on(in aids: P int, out L <op:Opn, aid:int>) =
       (filterdeps[AH](tagged union dependson[AH](tagged))
            where (tagged = index[AH](aids)))
       where (AH = AnnHistory(<LOG = convert[self](1)>))
   convert(in i: int, out L <label: int, op: Opn, aid: int>) =
       if count (LOG) = 0
       then emptylist(<label:int, op:Opn, aid:int>)
       else (([<label = i, op = hd.op, aid = hd.aid>]
                 concat (convert[History(<LOG = tl>)](i+1)))
            where (hd = head (LOG), tl = tail (LOG)))
       endif
end History
Sort AnnHistory
type <LOG: L <label: int, op: Opn, aid: int>>
retrieval methods
   index(in aids: P int, out P <label: int, op:Opn>) =
       unlist(collect <label = g.label, op = g.op>
              for g in LOG iff (g.aid in aids))
   dependson(in As: P <label: int, op:Opn>, out P <label: int, op:Opn>) =
       if count(S) = 0 then S
       else S union dependson[self](S) endif
       where (S = unlist(collect <label = g.label, op = g.op>
                         for g in LOG
                         iff (exists p in As |
                                 (g.label < p.label
                                  and non_commutative[p.op](g.op)))))
   filterdeps(in deps : P <label: int, op:Opn>, out L <op: Opn, aid: int>) =
        collect <op = g.op, aid = g.aid> for g in LOG
        iff (exists d in deps | (d.label = g.label))
end AnnHistory
```

Figure 33: Methods to select dependent operation invocations from a workspace history.

of operation invocations depend, and a method for checking whether a given set of operation invocations conflicts with the operation invocations in a destination history.

Figure 33 defines TM methods for calculating the dependent operations in a workspace history. An auxiliary sort `AnnHistory` is used, which numbers the records in a history list. The `aid` fields of the records in a history do not order the records. Because ordering is important for calculating the set of dependent operations, numbers are added to the records before the calculation is made. Method `convert` of sort `History` attaches a numeric `label` field to each record in the `LOG` list. The label corresponds to the record's position in the list, starting with value one. The output type of this method is the type of the `LOG` component of an `AnnHistory` value. The actual calculation of the dependent subhistory is done by the methods defined on sort `AnnHistory`. Method `depends_on` of sort `History` labels each operation invocation in the history (using the `convert` method), calculates the indices of the selected operations in the converted history (using method `index` of sort `AnnHistory`), calculates the set of depended-on operations and unions this to the original set (using method `dependson` of sort `AnnHistory`), and finally removes the labels from the depended-on operations in the annotated history (using method `filterdeps` of sort `AnnHistory`).

The methods defined on sort `AnnHistory` work as follows. For a given set of action identifications, method `index` returns the set of `label-op` records associated with these actions in the annotated history. For a given set of `label-op` records, method `filterdeps` projects the corresponding `op-aid` records from the annotated history, effectively removing the labels. For each `label-op` record in a given set of `label-op` records, method `dependson` collects the set of operations in the annotated history that happened before it and do not commute with it. Method `dependson` is defined recursively: once the set of depended-on operations is collected from the annotated history for the argument set of records, the method is again applied to the depended-on set. The method converges because each recursive application only considers operations that happened earlier in the history, and so on. The results of all of the recursive method invocations are unioned together. This gets rid of any duplicates.

Figure 34 defines two additional operations on workspace histories: `partition_subH` and `import_subH`. The method `partition_subH` partitions the operations of a source subhistory into three classes: operations that are already in the destination history; operations that conflict with operations in the destination history; and operations that can be safely added to the destination history. An operation is already in the destination workspace history if an entry with the same `aid` is found. An operation conflicts with an operation in the destination workspace history if they do not commute. An operation is safe if it commutes with all the operations in the destination workspace history. The `import_subH` method first invokes the `partition_subH` method to partition the subhistory. If there are no conflicting operations, the safe operations are concatenated to the end of the destination workspace history's `LOG`, otherwise the destination workspace history is not changed.

Figure 35 gives the specifications of module methods for the workspace operations. These methods are parameterised with a workspace name. The indicated workspace history is chosen from the set of histories, and the appropriate sort method is applied to it.


```
Sort History
   type <LOG: L <op: Opn, aid: int>>
   retrieval methods
      partition_subH(in subH: L <op:Opn, aid:int>,
                     out <already in: L <op:Opn, aid:int>,
                          safe: L <op:Opn, aid:int>,
                          conflicts: L <op:Opn, aid:int>>) =
         ((<already in = collect p for p in subH iff (p.aid in found),
            safe = collect p for p in impH
                   iff (not (exists b in destH |
                                (non_commutative[p.op](b.op)))),
            conflicts = collect p for p in impH
                        iff (exists b in destH |
                                (non_commutative[p.op](b.op)))>)
           where (impH = collect p for p in subH
                         iff (not (p.aid in found)),
                  destH = collect p for p in LOG
                          iff (not (p.aid in found))))
          where (found = unlist(collect p.aid for p in subH
                                iff (exists b in LOG | (b.aid = p.aid))))
   update methods
      import_subH(in subH: L <op:Opn, aid:int>) =
          if count(ops.conflicts) > 0 then self
          else self except (LOG = LOG concat ops.safe) endif
          where (ops = partition_subH[self](subH))
end History
```

Figure 34: Methods to partition and import subhistories.

```
module Workspaces
(* definitions of sorts Opn, History, and AnnHistory *)
module section
(* ... *)
module retrieval methods
   depends(in w:string, aids: P int, out L <op:Opn, aid:int>) =
      depends_on[(unique for g in histories
                  iff (g.wid = w)).history](aids)
   partition_impH(in w:string, impH: L <op:Opn, aid:int>,
                  out <alreadyin: L <op:Opn, aid:int>,
                       safe: L <op:Opn, aid:int>,
                       conflicts: L <op:Opn, aid:int>>) =
      partition_subH[(unique for g in histories
                     iff (g.wid = w)).history](impH)
module update methods
   import(in w: string, subH: L <op:Opn, aid:int>) =
      self except
      (histories = replace <wid = g.wid,
                            history = import_subH[g.history](subH)>
                   for g in histories iff (g.wid = w))
end Workspaces
```

Figure 35: Module methods for importing subhistories.

4.3.5 Selecting operations for exchange

The CoCoA data exchange protocol mechanism offers the specifier a **select** construct for chosing a set of operations for import or export from a workspace history. These are the operations that comprise the subhistory parameter (called an *exchange history* in [KTWP96]) to the `import` method defined in the previous section.

As an example, consider the following data exchange operation:

data exchange operations
`get_modifys(n : int) = select modify(n,_)`

To express this select statement in TM, we make use of two methods: `is_modify` (defined on sort `Opn`), which checks whether an operation invocation meets the selection criterion, and `get_modifys` (defined on the `Workspaces` module), which filters the selected operations from the history. These methods are defined in Figure 36. Observe that method `get_modifys` also computes the depended-on operations in the history and returns them with the selected `modify(n_)` operations.

4.3.6 Discussion

The specifications given in this section formally define commutativity relations, workspace histories, and methods on workspace histories in TM. The specifications can, to a large extent, be automatically generated; only the `Opn` sort definition and the **select** statement queries

```
module Workspaces
Sort Opn
(* ... *)
      is_modify(in n: int, out bool) =
         case self of
           insert = p : false
           modify = p : p.name = n
           delete = p : false
         endcase
end Opn
module section
(* ... *)
module retrieval methods
   get_modifys(in w:string, n:int, out L <op:Opn, aid:int>) =
         depends_on[H](collect h.aid for h in unlist(H.LOG)
                       iff (is_modify[h.op](n)))
         where (H = (unique for g in histories iff (g.wid = w)).history)
end Workspaces
```

Figure 36: Methods for selecting the `modify(n,_)` operations from a history.

are dependent on the data operations specified in the COCOA specification. The specifications can be executed using the TM Abstract Machine (see Section 5.2).

For the most part, the specifications are easy to read, once the reader is acquainted with TM. We realise that the type constructors in TM are sometimes cumbersome to use, but we believe the ability to express constraints and complex queries in predicate logic makes up for this shortcoming.

It is worth pointing out that other, non-symmetric relations between operation invocations can be defined as TM methods, in an analogous way to the `non_commutative` method on sort `Opn` (defined in Section 4.3.2). This would allow weaker relations such as $\oplus$ and $\diamond$ to be defined as TM methods [WäKl96].

Unresolved issues

- How do we know that an operation *can* be executed in the destination workspace, *after* other operations (with which it commutes)? (This is where the import or export wants to put it.) Can we assume that commutativity implies that the preconditions are not violated? Perhaps, if it is state independent. But we need to prove this, since preconditions are not expressed in the commutativity relations.
- If an operation is already in the destination history, do we know that all of the operations on which it depends are also in the destination history? Assume these operations are also being imported. By definition of "on which it depends", they are before this operation in the source history. Thus, they should be before it in the destination history, and hence already in the destination history. We should prove that this situation cannot arise.

- In order for a data exchange to take place, it may sometimes be necessary to reorder the operations in the resulting, merged history in order to comply with the execution ordering rules. This aspect of the transactional view simulation requires further investigation.

4.4 Integrated views analysis tool

The integrated views analysis tool will offer the specifier a simulation that combines the organisational view and the transactional view simulations. Our intention is to simulate the scenario definition with respect to everything but the actual execution of the data operations. To do this, we need to combine the specifications of the two views to obtain an integrated LOTOS/TM specification. Its general structure will be that of the LOTOS/TM process defined for the organisational view. (See Page 70.) The new process will offer operation invocation patterns as events. Some modifications of the patterns are needed in comparison to those used in the organisational view analysis. These are explained below.

- Data operation patterns are less generic than those in the organisational view analysis tool. Non-arbitrary parameters in the patterns (i.e., parameters that are not '_') must acquire values in the integrated views simulation, because the values of these parameters are needed for testing the commutativity of the operation invocations in a workspace history.
- Communication operations are offered as events. The events take values for all actual parameters, since the values influence the control flow of the scenario. Communications are essentially the same as for the organisational view, but now preconditions on the communications (which can query the termination and breakpoint markers in the workspace histories) need to be tested. The preconditions can be modelled as selection predicates on the events.
- Data exchange operations require values for both arbitrary and non-arbitrary parameters in operation patterns for the integrated views simulation. The values are necessary because of the data exchange protocol exercised by the transactional view. For example, a destination workspace cannot be arbitrarily identified if a merge operation is to be done on the workspace history.

A feature of the integrated views analysis simulation not seen in the other verification tools is the storage of termination and breakpoint constraint markers in the workspace histories. This kind of marker can appear in the COCOA execution rules, following a data operation pattern. The simulation will test termination and breakpoint constraints as part of communications.

Our implementation strategy for the integrated views analysis tool will be to offer the operation invocation patterns described above as events of a LOTOS/TM process. The TM data structures generated to manage execution rules, workspace histories, and active steps will be

passed around and manipulated by the LOTOS/TM process. The events offered by the process will be defined so that synchronisation can only take place (between the LOTOS/TM process and the invoker of the operation) for operations when they are allowed by both the organisational view and the transactional view. Selection predicates on the event offers will be used to check (using the TM data structures) the requirements imposed by both views.

The integrated views analysis tool depends on the organisational and transactional view analysis tools; its design will be be worked out in complete detail after these other tools are implemented.

4.5 Commutativity analysis tool

This component of the Verification Toolbox involves the definition of a semi-automatic, interactive analysis tool for checking whether operations (specified as TM methods) commute. The history rules of a CoCoA specification define **noncommutative** rules for data operations. These can include state-independent, parameter-dependent expressions over the operation parameters. We want to express these relations in a form that is appropriate for use with the TM Proof Tool [Spel95]. Section 4.5.1 looks at the TM Proof Tool. Section 4.5.2 looks at the formulation of semantic dependency relations.

4.5.1 The TM Proof Tool

The TM Proof Tool ([Spel95]) infers properties from the semantics of a specification, using a higher-order logic (HOL) based theorem prover. The tool is built on top of Isabelle, which is a generic theorem prover based on the technique of natural deduction [Paul94]. Isabelle/HOL provides a basic theory of booleans, integers, sums, tuples, sets, and lists, embedded in a simply-typed lambda calculus. The FM language with subtyping (the formal model underneath TM [BaFo91]) is shown to be reducible to this calculus. Additional information is added to the theory for the "input" specification. To be able to exploit the semantics of a specification for the purpose of theorem proving, an external Schema Translator is needed to first compute the semantics of the input specification. Since this has not been automated yet, the translation needs to be carried out by hand.

As an example, we show how parts of the TM specification in Figure 27 (Page 73) are translated into a form that can be input to the TM Proof Tool. The translated representation of the specification is shown in Figure 37. The `types` section declares synonyms for the types of the classes and sorts underneath the schema. The `consts` section declares the constraints and methods; the `defs` section lists their definitions in basic HOL.

Isabelle/HOL comes with a fully automated proof strategy, based on term-rewriting and natural deduction. The proof strategy is applied to prove properties about a given specification. For example, when we want to prove that two `insert` methods on a `Table` value commute, the following proof goal can be entered:


```
Spec = FM + Integ +

types
table = "(int*string) set" (** types **)
consts
keyname :: "table=>bool" (** constraints **)

Insert :: "[table,int,string]=>table" (** methods **)
Delete::  "[table,int]=>table"
Modify::  "[table,int,string]=>table"
defs
C1 "keyname self == ! x:self. ! y:self. fst x = fst y --> x = y"
M1 "Insert self n s == self Un {(n,s)}"
M2 "Delete self n   == collect p for p in self iff (fst p ~= n)"
M3 "Modify self n s == replace (n,s) for p in self iff (fst p = n)"
end
```

Figure 37: Translation of the TM specification of Badrinath's Table for use with Isabelle/HOL.

```
goalw Spec.thy [M1]
      "insert(insert(self,n1,s1),n2,s2)
           = insert(insert(self,n2,s2),n1,s1)";
```

`M1` is an axiom for method `insert`. It is supplied as a parameter to the proof goal and is used to unfold the method body, thus yielding a normal HOL proof-term. In this example, the goal solves by rewriting, using standard properties of the set and tuple operators found in the definition `M1`.

It is also possible to use the TM Proof Tool to show that the application of an update method to a TM value (module, class, object, sort) does not violate the constraints specified in the schema. The user may define additional assumptions, such as preconditions on the parameter values, to be used in the proof derivation. In cases where the proof goal cannot be solved, it is sometimes possible to use Isabelle in a "backwards" way to derive minimal preconditions that must be satisfied in order to prove the goal.

4.5.2 Formulation of commutativity checks

In Section 4.3.2, we observed that relations between operation invocations other than **non-commutative** could be defined as TM methods. The relations assumed in the TRANSCOOP transaction model (called $\oplus$ and $\diamond$ in [RKTW95], and *backward* and *forward commutativity* in [KTWP96]) are slightly different than the **noncommutative** relation. It would be convenient to be be able to use the TM Proof Tool to investigate these other relations. But in order to do this, we must be able to state precisely what these relations mean with respect to the TM database schema, and state precisely what is to be proved about them. This possibility will be studied during the implementation phase of the TSE design.

Research on advanced transaction models that uses commutativity relations sometimes assumes that preconditions are defined/known for the operations. Preconditions are used in the specification of *partial* operations, which are undefined for some input values. In TM terminology, the result of a method application is said to be undefined if the precondition does not hold for the input database state. (The method's application might violate the database constraints.) Preconditions are used in [Weih88] to define notions of *backward* and *forward commutativity*. Similar definitions are used in the CoAct model [KTWP96]. Both commutativity relations are symmetric. We now look briefly at what this means in terms of the TM Proof Tool.

TM does not provide a mechanism to declare preconditions for methods. Thus, we need to define preconditions as additional assumptions when specifying a theorem to prove. It should be possible to use the TM Proof Tool to check whether these preconditions actually prevent database constraint violations. Informally, when preconditions are used in combination with commutativity, we need to prove claims about two operations such as: "doing the latter operation first (i.e., moving it backwards) does not violate the precondition of the former operation, and does not change its result." This requires further investigation.

5 The Simulation Environment

The Simulation Environment will be a key player in supporting all forms of simulation offered by the Verification Toolbox. The Simulation Environment includes the LOTOS/TM Simulator and the TM Abstract Machine (the TAM [FlBo96]) as components. The LOTOS/TM Simulator will be based on SMILE (the LOTOS Simulation Environment [Eert94]) and the TAM, both of which have been developed at UT. In the mappings of the Verification Toolbox, TM is used not only to model the database schema and types of a scenario, but also the meta-data needed to manage the scenario during the simulations. The simulations offered in the Verification Toolbox ignore, for the most part, the actual application of data operations to the workspaces. The data operation invocations abstract away from some of the operation parameters. Our use of the TAM will be primarily to support the management of the scenario meta-data. However, it is worth noting that it would also be possible to offer a complete simulation of a scenario definition, including the application of data operations to real data.

5.1 The SMILE Simulator

SMILE is an acronym for 'SyMbolic Interactive LOTOS Execution.' The SMILE simulator is designed for analysis of the dynamic semantics of specifications written in Full LOTOS [Eert94,Eert93]. SMILE allows one to "walk through" the events offered by a defined behaviour specification, make selections and thereby cause gate events to occur. The specifier is also able to view the unfolded tree of behaviour traces, highlight events that have already happened, and backtrack to previous choice points. The specifier can zero-in on smaller parts of a specification and analyse their behaviour in isolation. Some reachability analysis is possible, by selecting a so-called *target* event and then instructing the simulator to *goto* the target within a specified number of unfolding steps. This can result in the message "Target Not Reached" displayed in a window. The simulator recognises "deadlock" in a behaviour specification when no events are offered at a particular stage in the simulation. This is indicated by a STOP node in the simulation tree path. The simulator is able to detect recursion in the unfolding, and the user can view the matching nodes that have already been unfolded. It is possible to save a path or behaviour tree (from the simulation window) to a file in ascii form; it is also possible to generate an Extended Finite State Machine (EFSM) representation of the behaviour specification. The EFSM is essentially a flattened representation of the original LOTOS specification.

5.2 The TM Abstract Machine

The TM Abstract Machine offers a specifier the ability to populate a TM database schema and then apply methods to the database [FlBo96]. The database constraints are checked to ensure the database state is consistent, as defined by the specifier. *When* (module, class, object, and sort) constraints are tested can be varied by setting a mode flag to `immediate`, `deferred`, or `off`. This is useful for experimenting with method compositions within larger expressions. (For example, when performing two update methods in succession, database constraints may be temporarily violated before the second method is applied.) The TAM user

can evaluate complex query and update expressions in an *ad hoc* manner, from the command line. All levels of method definition (sort, object, class, module) can be applied in this way by providing an explicit **self** value to the invocation. This capability will come in useful for interfacing the TAM with SMILE.

5.3 The LOTOS/TM Simulator (Surgery on SMILE)

The SMILE simulator, as it is currently designed and implemented, interacts with a tool for analysis of the ADT parts of a Full LOTOS specification [CaSa92]. It is these parts of the SMILE simulator that we must replace with analogous interactions with the TAM. Because execution in SMILE is based on syntactic rewriting, a cryptic internal representation called CR (for Common Representation) is used for all (Full LOTOS) behaviour and value expressions. The SMILE implementation manipulates the CR representations. To evaluate a value expression that is in CR format, the interface between the LOTOS/TM version of SMILE and the TAM must translate between the CR representation of a LOTOS/TM value expression used by SMILE, and the ascii/string representation of the value expression used by the TAM. As part of the translation, free variables in the LOTOS/TM expression are first bound to values by the LOTOS/TM environment. These values are also in CR format. The ground value expression (obtained by instantiating the free variables) is then converted to an ascii string for evaluation by the TAM.

Values play an integral part in LOTOS/TM: they influence the offering of events when used in guards and selection predicates, and they play an important role in event synchronisation. In the LOTOS/TM simulator, we will invoke the TAM for evaluation of guards and selection predicates, and equality comparisons of values during synchronisation attempts.

It will take a substantial amount to work to implement the combined LOTOS/TM Simulation Environment. UT will consider this as one of its major results of the TRANSCOOP project. We are currently studying the precise difficulties we may encounter when combining the already existing tools, and it is clear that this work is far from trivial. We do not discuss specific implementation issues in this report, and refer the reader to later studies in the context of this work package.

6 Conclusions

In this document, we have described the various components of the TRANSCOOP Specification Environment (TSE). These include a Graphical Specification Editor, a Parser/Type-checker, a Verification Toolbox for specification analysis, and a Simulation Environment that includes a LOTOS/TM Simulator and a TM Abstract Machine. The Verification Toolbox relies on executable semantics descriptions as a means to verification. It includes components that map different aspects of a COCOA specification (the execution rules, the organisational view, the transactional view, the history rules, and an integrated scenario view) to LOTOS/TM and TM specifications. These mappings will be implemented as back-end code generators of the Parser/Type-checker.

In our descriptions of the TSE components, we have made an effort to use formal specifications (LOTOS/TM and TM code fragments) as an alternative to English (or to some other formal notation). Our motivation behind this is twofold: (1) it enables precise definitions of the data structures and data operations needed to manage various aspects of a scenario (e.g., the information that is maintained in the organisational view simulation), and (2) the definitions can later be executed using the Simulation Environment.

Some components of the tool set have been described in more detail than others. (Bear in mind that the COCOA language is still fairly young.) In preparing this document, we have concentrated our efforts on describing the specifications to be generated by the Execution Rules Analysis Tool, the Organisational View Analysis Tool, and the Transactional View Analysis Tool.

Although we do not specify facilities in the Verification Toolbox for automated proofs of properties about a specification (this must be done mechanically by the specifier using simulation), we are, nevertheless, excited about the analysis possibilities to be offered by the TSE design. We predict that after an initial implementation of the tool set is available, we will be in a position to define and implement more advanced specification analysis strategies, or at least provide guidelines for the specifier to do so.

In our future work, we imagine more active use of the TM Proof Tool in the analysis of a specification. For example, the specifier could define constraints on the TM components generated by the verification mappings, and test whether the method definitions that define the dynamic semantics of the scenario preserve these constraints. Furthermore, we might investigate algebraic properties of the methods in a workspace history. The idea would be to construct workspace histories as method composition terms, without evaluating the methods (much as they are in the Transactional View Analysis Tool), but then to actually be able to apply them to the database as complex methods. These ideas are certainly feasible given the language facilities we have, but further exploration is required.

7 References

[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.

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

[Bent94] H. van Benthem, "Safety, Sets & TM," Universiteit Twente, 23 February 1994.

[BrWo92] A. Brueggemann-Klein & D. Wood, "Deterministic Regular Languages," *STACS 1992, LNCS 577* (1992), 173–184.

[BLPV95] R. A. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen & J. Wäsch, "Specification of the prototype architecture," TRANSCOOP Deliverable III.1 (TC/REP/VTT/D3-1/952), April 10, 1995.

[BLPV96] R. A. de By, A. Lehtola, O. Pihlajamaa, J. Veijalainen & J. Wäsch, "Specification of the demonstrator," TRANSCOOP Deliverable III.2, TC/REP/VTT/D3-2/952, April 1996.

[CaSa92] M. Caneve & E. Salvatori, eds., "The LITE User Manual," The LOTOSPHERE Consortium, Lotosphere Deliverable Lo/WP2/N0034/V08, ESPRIT 2304, March 30, 1992.

[Eert94] H. Eertink, *Simulation Techniques for the Validation of LOTOS Specifications*, PhD Dissertation, Universiteit Twente, Enschede, The Netherlands, 1994.

[Eert93] H. Eertink, "SMILE User Manual, release 4.0," Universiteit Twente, 24 December 1993.

[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, November 1995, (accepted for publication in the *International Journal of Cooperative Information Systems*, Special Issue on Formal Methods).

[FaEB96] F. J. Faase, S. J. Even & R. A. de By, "Deliverable IV.3," Universiteit Twente, Enschede, The Netherlands, 21 February 1996, Report TC/REP/UT/D4-3/033, ESPRIT Project TransCoop (LTR 8012).

[FlKS94] J. Flokstra, M. van Keulen & J. Skowronek, "The IMPRESS DDT: A Database Design Toolbox based on a Formal Specification Language," in *Proceedings of ACM-SIGMOD 1994 International Conference on Management of Data, Minneapolis, MN, May 24–27, 1994*, R. T. Snodgrass & M. Winslett, eds., ACM Press, New York, NY, 1994, 506, (also appeared as SIGMOD RECORD, 23, 2, June, 1994).

[FlBo96] J. Flokstra & R. Boon, "The TM Abstract Machine (TAM)," Universiteit Twente, Enschede, The Netherlands, 20 February 1996, Internal Working Document.

[KTWP96] J. Klingemann, T. Tesch, J. Wäsch, J. Puustjärvi & J. Veijalainen, "Definition of the TRANSCOOP Cooperative Transaction Model," April 1996, GMD, Project TransCoop, ESPRIT LTR 8012, Deliverable TC/REP/GMD/D5-2/511.

[Meta] MetaCase, "`http://www.jsp.fi/metacase/`," MetaCase Consulting, Finland.

[Paul94] L. C. Paulson, *Isabelle: A Generic Theorem Prover*, Lecture Notes in Computer Science #828, Springer-Verlag, Berlin, 1994.

[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.

[WäKl96] J. Wäsch & W. Klas, "History merging as a mechanism for concurrency control in cooperative environments," *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 An algorithm for checking execution rules

Here we include a formal TM specification of the algorithm discussed in Section 4.1. The TM specification defines a schema for storing execution rules, and a retrieval method that checks if a given history is correct for the stored execution rules. The algorithm to check a given history against an execution rule consists of the following three steps:

1. Construct a subset of parameter value combinations to test, based on all possible combinations that are found in the given history.
2. For each of these combinations, construct a subhistory that contains only those operations that have the given parameter values.
3. For each of these subhistories, check the subhistory against the execution rule, with the associated combination of parameter values.

The histories (the initial history and all the subhistories) are represented by the class `hist`, which uses the sort `oper` to represent the data operations in the history:

```
module Check_Exec_Rule
  Sort oper
     type < oper_name : string, params : L string >
  end oper

  Class hist
     attributes
        oper_list : L oper

     object retrieval methods
       filter(in t_ps : t_params, op : order_expr, out hist) =
         self except (oper_list = collect h for h in oper_list
                                  iff match[op](h, t_ps))
     object update methods
       tail_ = self except(oper_list = tail(oper_list))
  end hist
```

The retrieval method `filter` of the class `hist` is used in the second step of the algorithm. It returns a subhistory for a given combination of values `t_ps`, and a given ordering expression `order_expr`.

The sort `t_params` is used to represent a combination of values for the **forall**-parameters. It uses the sort `t_value` to represent a single parameter and its value. This sort makes use of a variant type to represent the parameter value, which can either be a string (the `value` alternative), or an indication for any other value (the `any_other` alternative).

```
Sort t_value
   type [| any_other, value : string |]
end t_value

Sort t_params
   type P < param : string, value : t_value >
   retrieval methods
     assoc(in p : string, out P string) =
       unnest(
         collect
           case p_v.value of
             any_other : emptyset(string)
             value = v : { v }
           endcase
         for p_v in self
         iff (p_v.param = p))
end t_params
```

The retrieval method `assoc` returns the value (if any) of the parameter `param` in the set of parameters.

The sort `oe_param` is used to represent a parameter in a data operation invocation pattern in an order expression, which can be a string value, a **forall**-parameter, or `any` (the underscore in the syntax of order expressions). The sort `oe_param_list` is used for the list of parameters of a data operation invocation pattern.

```
Sort oe_param
   type [| any, value : string, param : string |]
end oe_param

Sort oe_param_list
   type L oe_param
   retrieval methods
     assoc(in pn : string, i : int, out P int) =
       if count(self) = 0
       then emptyset(int)
       else   case head(self) of
                param = p : if p = pn then { i }
                            else emptyset(int) endif
              else emptyset(int)
              endcase
            union assoc[tail(self)](pn, i+1)
       endif
```

The retrieval method `assoc` is used to determine the list of positions at which a certain **forall**-parameter (with name `pn`) is used in a data operation invocation pattern (in self). As such, it is used in the first step of the algorithm to calculate the values that are used for an operation in the history, according to a given data operation invocation pattern.

```
    match(in o_ps : L string, t_ps : t_params, out bool) =
      if count(self) = 0
      then true
      else if case head(self) of
              any : true
              value = v : v = head(o_ps)
              param = p : exists t_p in t_ps
                          | (      t_p.param = p
                             and case t_p.value of
                                   any_other : false
                                   value = v : v = head(o_ps)
                                 endcase)
            endcase
           then match[tail(self)](tail(o_ps), t_ps)
           else false
           endif
      endif
end oe_param_list
```

The retrieval method `match` is used to check if the actual parameters of a data operation in the history (`o_ps`) match with those in a data operation invocation pattern (self) according to a given set of **forall**-parameters (`t_ps`). This method is used by method `match` (part of the second step of the algorithm) and method `accept` (part of the third step of the algorithm) of class `order_expr`.

The class `order_expr` is used to represent the ordering expressions.

```
Class order_expr
  attributes
      e : [| oper   : < oper_name : string,
                        params    : oe_param_list >,
             seq    : < left  : order_expr,
                        right : order_expr >,
             opt    : order_expr,
             times  : order_expr,
             plus   : order_expr,
             set    : P order_expr,
             choice : P order_expr |]
  object retrieval methods
    collect_possible_param_values(in H : hist, ps : P string,
                                  out P t_params) =
      case e of
        oper = o:
          unlist(
            collect
              t_params(
                collect < param = p,
                          value =
                            if count(pv) = 1
```

```
                        then t_value( [| any_other |]
                             as [| any_other, value : string |])
                        else t_value( [| value = unique in(pv) |]
                             as [| any_other, value : string |])
                        endif
                        where( pv = collect h.params at i
                                    for i in assoc[o.params](p, 1)
                             )
                    >
              for p in ps)
          for h in H.oper_list
          iff (h.oper_name = o.oper_name))
      seq = sq :
              collect_possible_param_values[sq.left](H, ps)
        union collect_possible_param_values[sq.right](H, ps)
      opt = o :
        collect_possible_param_values[o](H, ps)
      times = t :
        collect_possible_param_values[t](H, ps)
      plus = p :
        collect_possible_param_values[p](H, ps)
      set = st :
        unnest(collect collect_possible_param_values[v](H, ps)
               for v in st)
      choice = ch :
        unnest(collect collect_possible_param_values[v](H, ps)
               for v in ch)
    endcase
```

The retrieval method `collect_possible_param_values` returns the value combinations which are used in an ordering expression (self) for a given history (`H`) and set of **forall**-parameters (`ps`). This method contains the core algorithm of the first step.

```
match(in op : oper, t_ps : t_params, out bool) =
  case e of
    oper = o:
          o.oper_name = op.oper_name
      and match[o.params](op.params, t_ps)
    seq = sq : match[sq.left](op, t_ps) or match[sq.right](op, t_ps)
    opt = o  : match[o](op, t_ps)
    times = t : match[t](op, t_ps)
    plus = p  : match[p](op, t_ps)
    set = st  : exists v in st | match[v](op, t_ps)
    choice = ch : exists v in ch | match[v](op, t_ps)
  endcase
```

The retrieval method `match` returns whether a given data operation (`op`) matches a given value combination (`t_ps`) according to the data operation invocation patterns found in the ordering expression (self). It is called by method `filter` of class `hist`, which is part of the implementation of the second step.

```
accept(in H : hist, t_ps : t_params, cont : L order_expr, o : oid,
       out bool) =
  if count(H.oper_list) = 0
  then true
  else case self.e of
         oper = op:
           if   (head(H.oper_list).oper_name = op.oper_name and
                  match[op.params](head(H.oper_list).params, t_ps))
           then if count(cont) = 0
                then count(H.oper_list) = 1
                else accept[head(cont)](tail_[H],
                                        t_ps, tail(cont), o)
                endif
           else false
           endif
         seq = sq:
           accept[sq.left](H, t_ps, [sq.right] concat cont, o)
         opt = op:
           if accept[op](H, t_ps, cont, o) then true
           else if (count(cont) = 0) then false
                else accept[head(cont)](H, t_ps, tail(cont), o)
                endif
           endif
         times = t:
           if accept[t](H, t_ps, [t] concat cont, o)
           then true
           else if (count(cont) = 0) then false
                else accept[head(cont)](H, t_ps, tail(cont), o)
                endif
           endif
         plus = p:
           accept[p](H, t_ps, [order_expr(inc( o ),
                     <e= [| times = p |]
                         as [|oper:<oper_name:string,
                                    params:oe_param_list>,
                              seq:<left:order_expr,
                                   right:order_expr>,
                              opt:order_expr,
                              times:order_expr,
                              plus:order_expr,
                              set:P order_expr,
                              choice:P order_expr|]
                     >) ] concat cont,
                     inc(o))
         set = s:
           if count(s) = 0
           then if count(cont) = 0
                then false
                else accept[head(cont)](H, t_ps, tail(cont), o)
                endif
```

```
                else exists v in s
                     | accept[v](H, t_ps,
                                  [order_expr(inc( o ),
                                  <e= [| set = s minus {v} |]
                                      as [|oper:<oper_name:string,
                                                 params:oe_param_list>,
                                           seq:<left:order_expr,
                                                right:order_expr>,
                                           opt:order_expr,
                                           times:order_expr,
                                           plus:order_expr,
                                           set:P order_expr,
                                           choice:P order_expr|]
                                  >) ]
                                  concat cont, inc(o))
                  endif
                choice = ch:
                  exists v in ch | accept[v](H, t_ps, cont, o)
              endcase
        endif

end order_expr
```

The method `accept` checks whether a given history (`hist`) is correct according to an ordering expression (self) followed by a list of ordering expressions (`cont`) (which are viewed as being glued together with the sequence operator), and a given value combination (`t_ps`).[16]

The class `exec_rule` is used to represent the execution rules. Each object of this class has two attributes: a set of parameters used in the **forall**-clause, and an ordering expression, which is an object of the class `order_expr`.

```
Class exec_rule
   attributes
      fa_params  : P string,
      rule       : order_expr
   object retrieval methods
     all_comb(in p_t_p : P t_params, out P t_params) =
       collect
         t_params(
           collect
             < param = p,
               value =
                 if count(pv) = 1
                 then t_value( [| value = unique in(pv) |]
                               as [| any_other, value : string |])
                 else t_value( [| any_other |]
                               as [| any_other, value : string |])
```

[16]The argument `o` is needed to create additional objects of the class `order_expr`, if needed.

```
                endif
                where( pv = unnest(collect assoc[t_p](p)
                                   for t_p in s_t_p) )
            >
        for p in self.fa_params)
     for s_t_p in { s_t_p : P t_params | s_t_p subset p_t_p }
     iff if count(s_t_p) > 0
        then forall p in self.fa_params
             | (count(
                 unnest(
                   collect assoc[t_p](p)
                   for t_p in s_t_p)) <= 1)
        else false
        endif
```

The retrieval method `all_comb` returns the set of all possible combined value combinations for given a set of value combinations (`p_t_p`). (The self argument is not used.) This method is part of the first step of the complete algorithm.

```
    correct_hist(in H : hist, o : oid, out bool) =
      forall
        t_ps in all_comb[self](
                collect_possible_param_values[rule](H, self.fa_params))
        | accept[rule](filter[H](t_ps, rule),
                       t_ps, emptylist(order_expr), o)

end exec_rule
```

The retrieval method `correct_hist` determines whether a given history (`H`) is correct for an execution rule (self). It implements the complete algorithm for a single execution rule. The first step of the algorithm is specified by method `collect_possible_param_values` of class `order_expr`, and method `all_comb` of this class. The second step of the algorithm is specified by method `filter` of class `hist`, which uses method `match` of class `order_expr` to check whether an operation occurs somewhere in the ordering expression of the execution rule. The third step of the algorithm is specified by method `accept` of class `order_expr`. This method checks whether a given subhistory is accepted by the ordering expression, taking into account the values of the **forall**-parameters.

The module section of the TM schema specification contains the module attribute `Exec_rules`, which contains the set of execution rules. It defines the module retrieval method `correct_hist` to check if a given history is correct for all the execution rules in the database.

```
module section
  attributes
    Exec_rules : P exec_rule
```

```
  module retrieval methods

    correct_hist(in H : hist, out bool) =
      forall e_r in Exec_rules
      | correct_hist[e_r](H, lastid)

end Check_Exec_Rule
```

We define an additional module update method to illustrate how to initialise the TM database with the following execution rule:

forall `i : int; j : int`
order `[a(i)] ; (b(j) ; c(i,j))`

The TM specification of this execution rule is included in the `init` method below[17]:

```
  module update methods
    init = self except ( Exec_rules = {
      exec_rule(inc(lastid),
      < fa_params = { "i", "j" },
        rule = order_expr(inc( lastid ),
               <e=  ([| seq =
                < left =  order_expr(inc( lastid ),
               <e=  [| opt =  order_expr(inc( lastid ),
               <e=  ([| oper = <oper_name =  "a" ,
                params = oe_param_list(
                 [ oe_param([| param =  "i"  |]
                as [| any, value : string,
                param : string |])  ] )> |])
                 as [| oper:<oper_name:string,
                             params:oe_param_list>,
                       seq: <left:order_expr, right:order_expr>,
                       opt:order_expr, times:order_expr,
                       plus:order_expr, set:P order_expr,
                       choice:P order_expr |]
                  >)    |]
                  as [| oper:<oper_name:string,
                               params:oe_param_list>,
                         seq:<left:order_expr, right:order_expr>,
                         opt:order_expr, times:order_expr,
                         plus:order_expr, set:P order_expr,
                         choice:P order_expr|]>),
           right =
               order_expr(inc( lastid ),
              <e=  ([| seq =
```

[17] This method definition is rather lengthly because of the many type casting expressions (following `as`) that are required by TM when variant types are used.

```
                < left = order_expr(inc( lastid ),
                        <e=  ([| oper = <oper_name =  "b" ,
                                         params = oe_param_list(
                             [ oe_param([| param =  "j"  |]
                                        as [| any, value : string,
                                              param : string |])  ]
                             )> |])
                   as [| oper:<oper_name:string,
                               params:oe_param_list>,
                         seq:<left:order_expr,
                              right:order_expr>,
                         opt:order_expr, times:order_expr,
                         plus:order_expr, set:P order_expr,
                         choice:P order_expr |] >),
                  right = order_expr(inc( lastid ),
                          <e=  ([| oper = <oper_name =  "c" ,
                                   params = oe_param_list(
                                   ([ oe_param([| param =  "i"  |]
                                               as [| any,
                                                     value : string,
                                                     param : string |]) ,
                                    oe_param([| param =  "j"  |]
                                             as [| any, value : string,
                                                   param : string |]) ]) )> |])
                   as [| oper:<oper_name:string,
                               params:oe_param_list>,
                         seq:<left:order_expr,right:order_expr>,
                         opt:order_expr, times:order_expr,
                         plus:order_expr, set:P order_expr,
                         choice:P order_expr |] >)    > |])
            as [| oper:<oper_name:string, params:oe_param_list>,
                  seq:<left:order_expr,right:order_expr>,
                  opt:order_expr, times:order_expr,
                  plus:order_expr, set:P order_expr,
                  choice:P order_expr|]>)    > |])
                  as [| oper:<oper_name:string,params:oe_param_list>,
                        seq:<left:order_expr,right:order_expr>,
                        opt:order_expr, times:order_expr,
                        plus:order_expr, set:P order_expr,
                        choice:P order_expr |] >)
      >)
    })
end Check_Exec_Rule
```

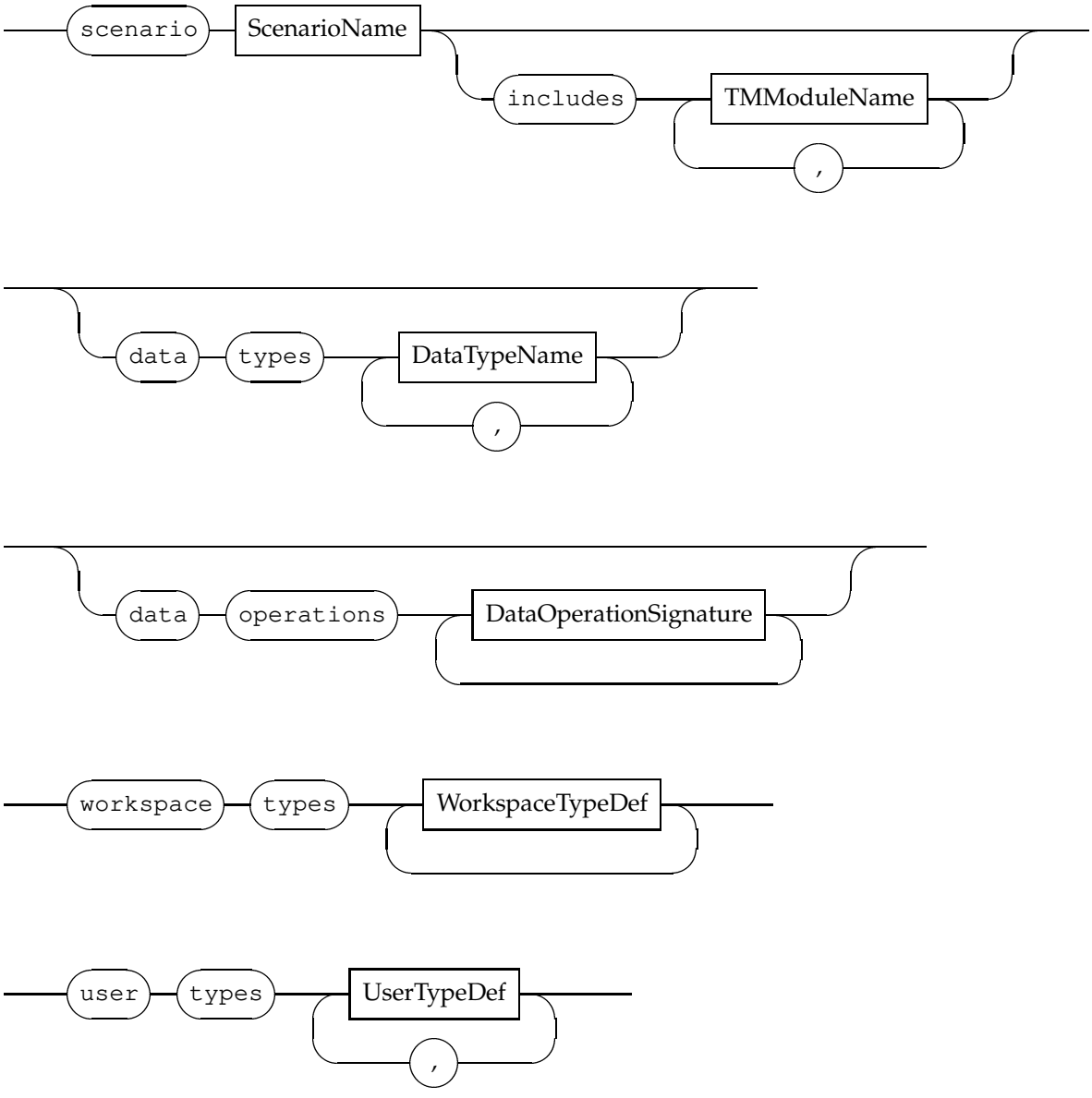
B CoCoA Syntax

This Appendix provides the grammar of CoCoA using rail diagrams. A CoCoA specification can be thought to exist of three parts. The first part gives the general definitions, which are used by the other two parts. The second part gives the specification of the organisational aspects of the scenario, and the third part gives the transactional aspects of the scenario.

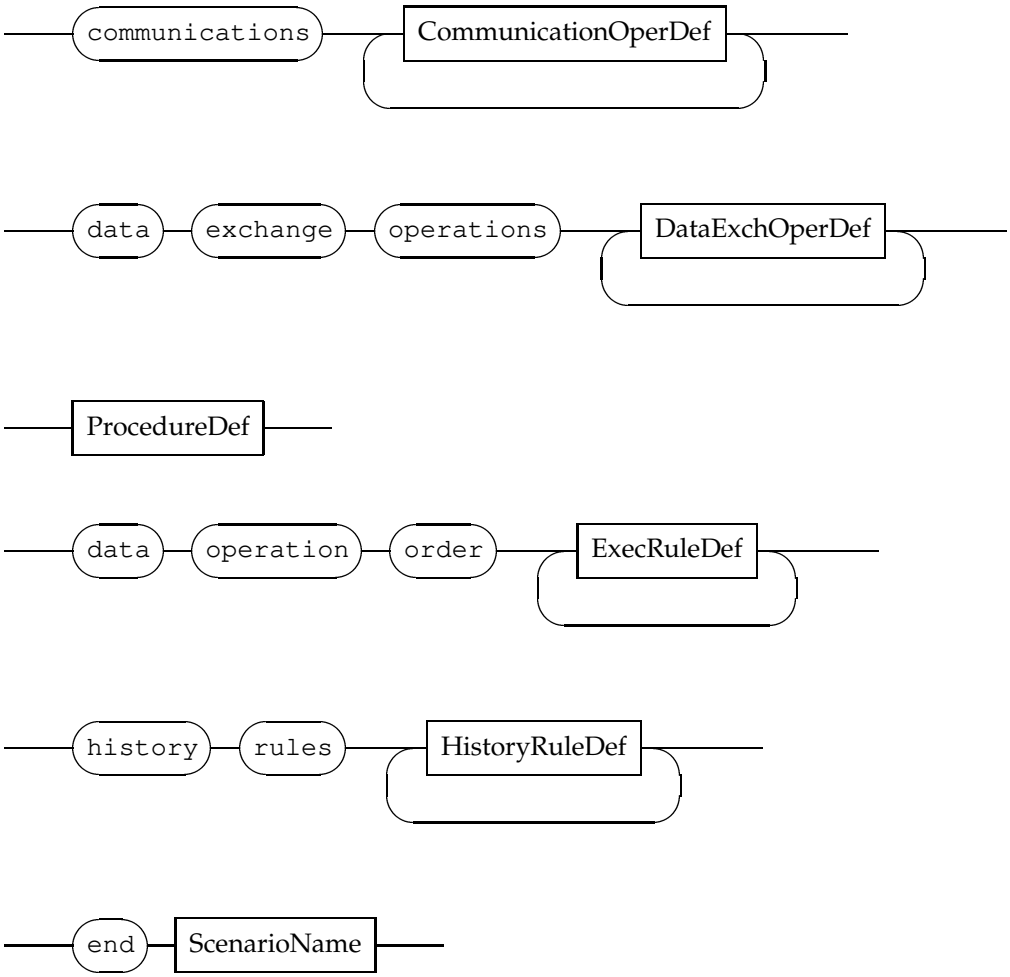
B.1 General part

The grammar rule for a complete CoCoA scenario is:[18]

CoCoA

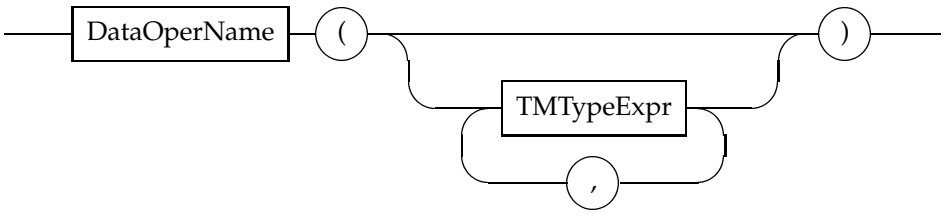


[18] All the lines should be read as following each other.



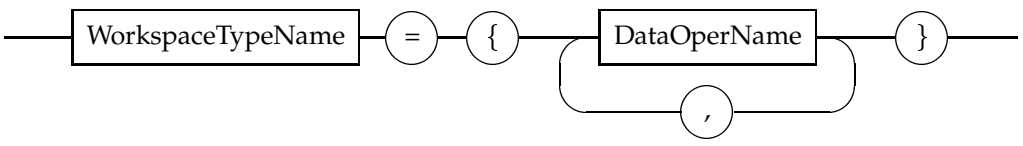
A data operation is defined by its name and input parameter types:

DataOperationSignature



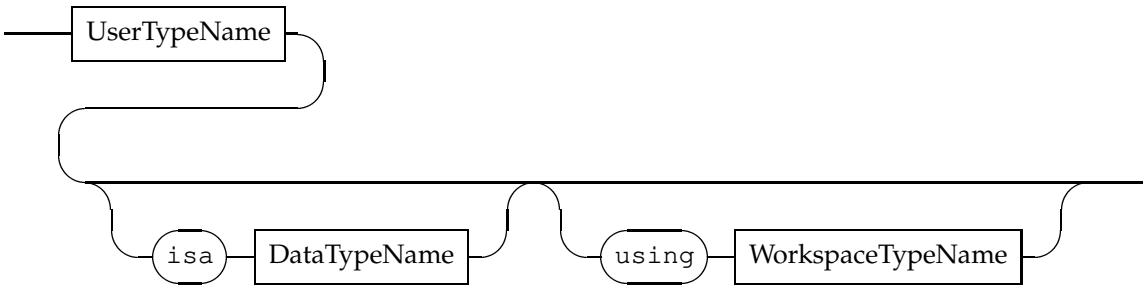
A workspace type is defined by enumerating the data operations it can contain:

WorkspaceTypeDef



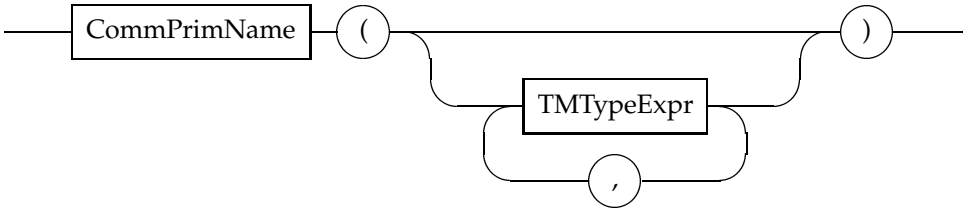
A user type is given by its name. It can have an optional data type, if it is used as an argument to a data operation. It can have an optional workspace type, which limits the data operations the user can do.

UserTypeDef



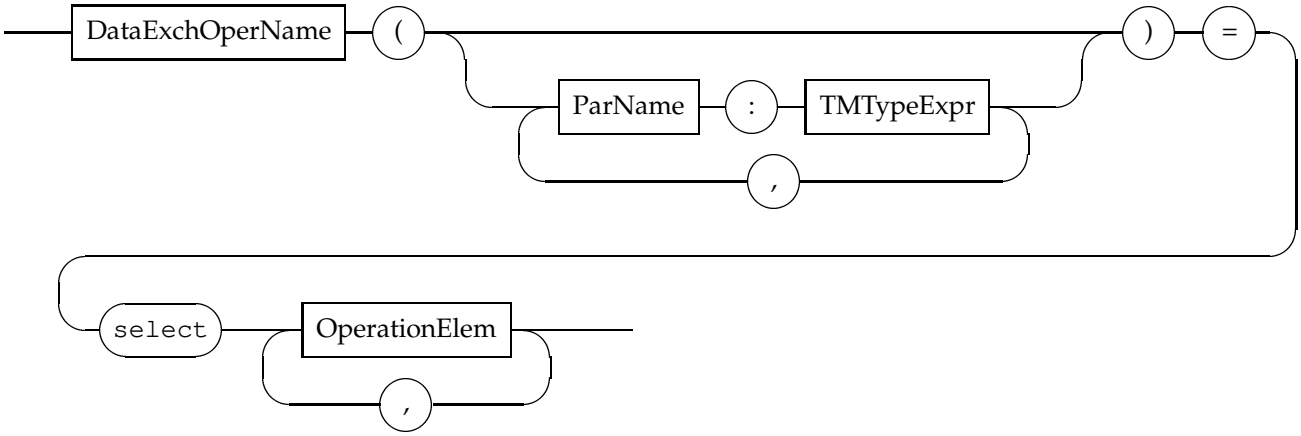
A communication primitive is defined by its name and input parameter types:

CommunicationPrimDef

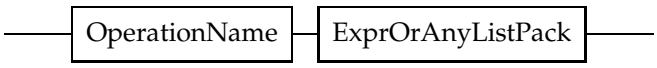


A data exchange operation is defined as a select operation on a workspace:

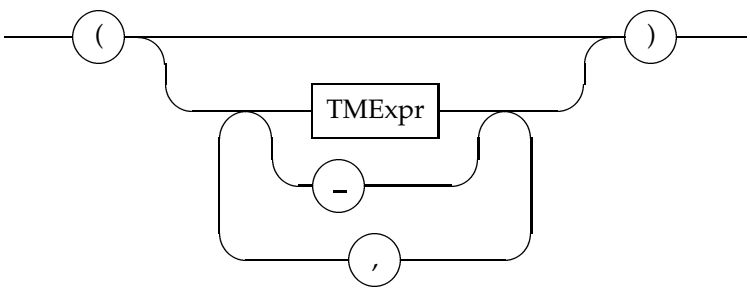
DataExchOperDef



OperationElem



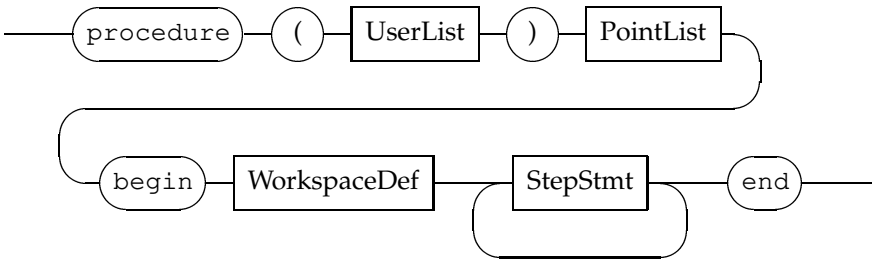
ExprOrAnyListPack



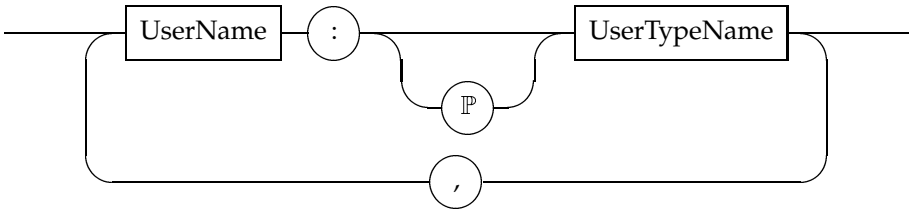
B.2 Organisational aspects

The organisational aspects of the scenario are given in the procedure definition, which also specifies the parameters for the users, and the interaction points of the top-level step:

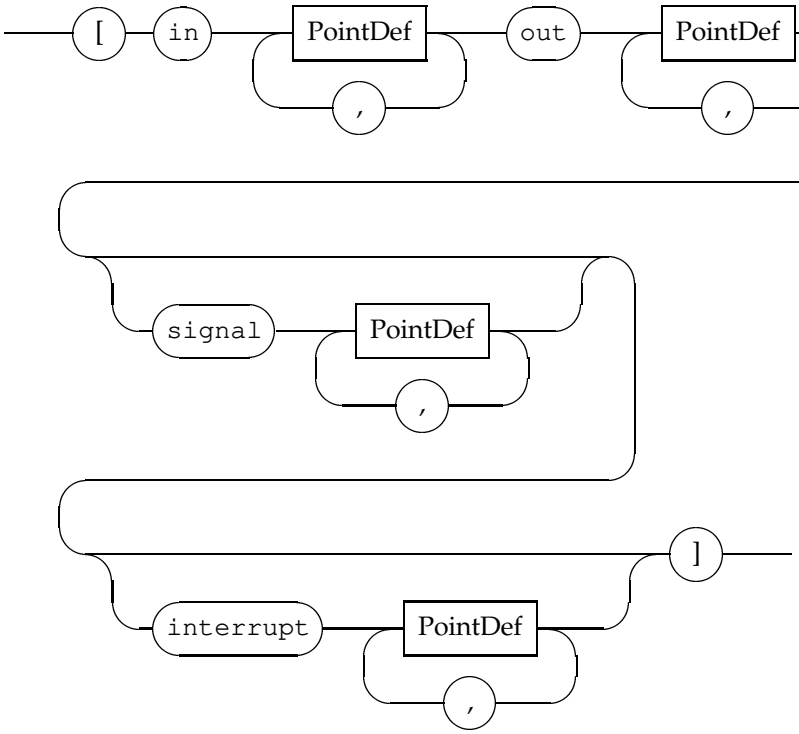
ProcedureDef



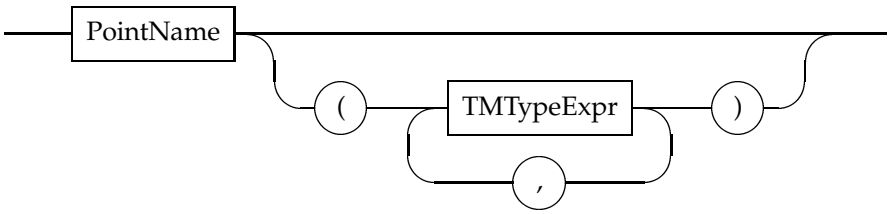
UserList



PointList



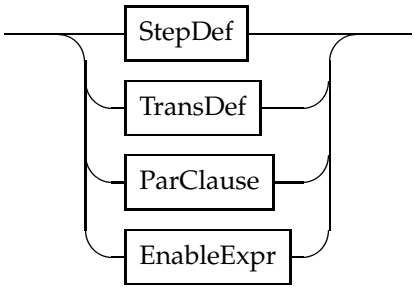
PointDef



WorkspaceDef

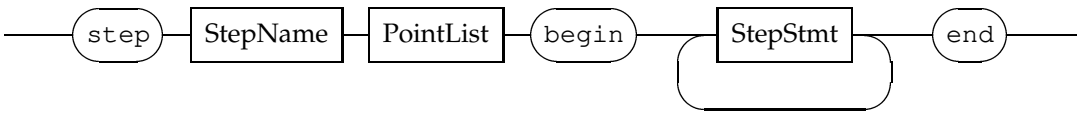
The body of the procedure consists of a list of step statements, each of which is either a step definition, a transition definition, a parallel clause or an enabling expression:

StepStmt



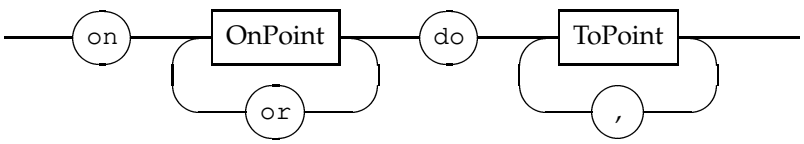
The definition of step consists of a name with an enumeration of interaction points followed by a number of step statements:

StepDef

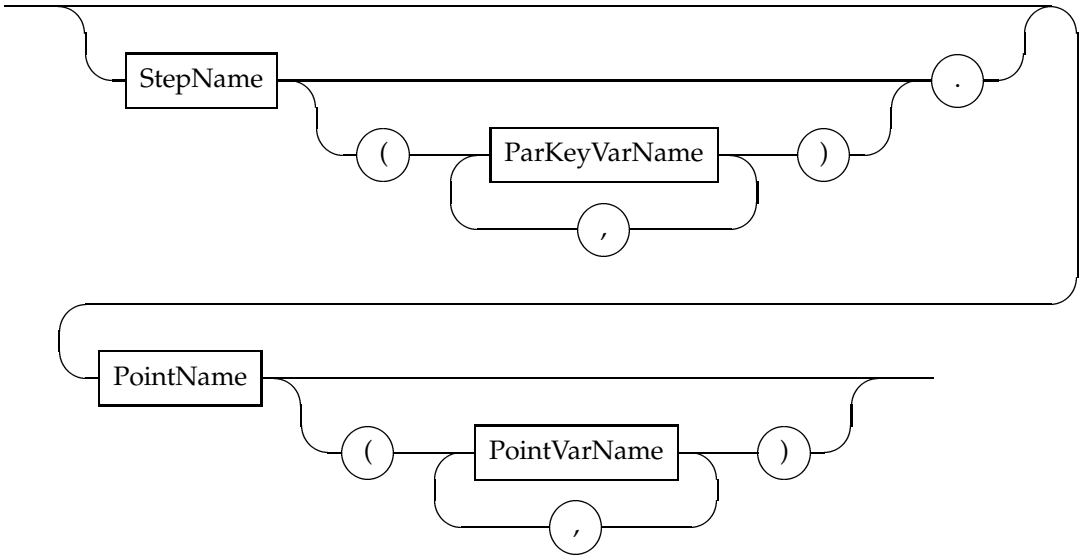


A transition is defined by an expression specifying on which points it starts, and a list of points at which it ends:

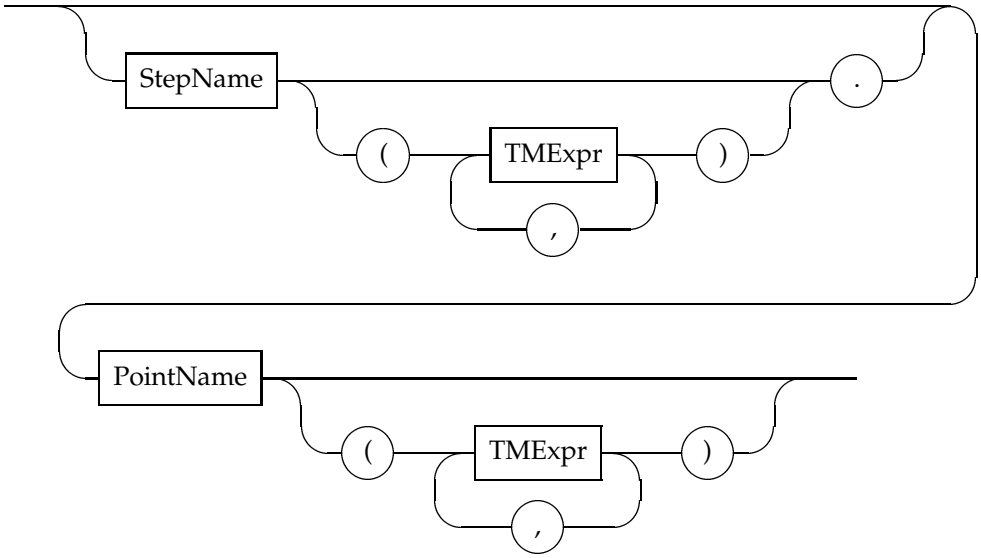
TransDef



OnPoint

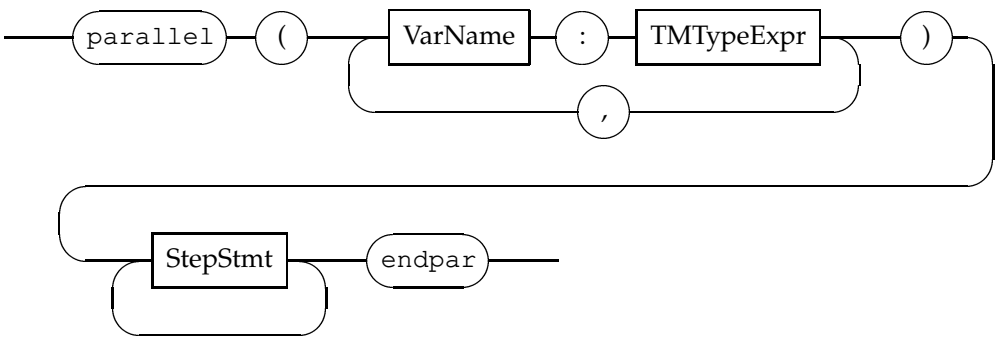


ToPoint



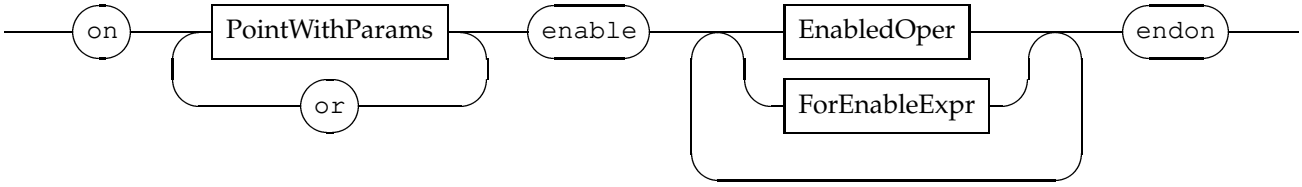
A parallel clause has a list of parameters, by which the instances are identified, and contains a list of step statements that define the instances:

ParClause

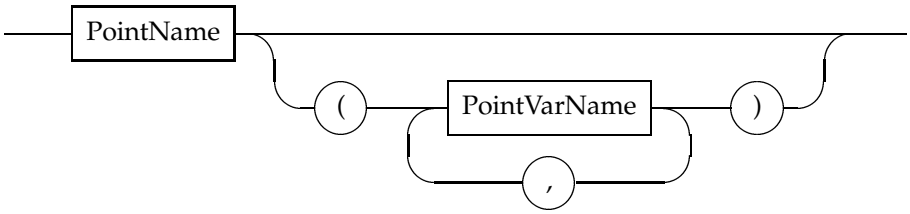


An enabling expression states which operations are enabled after an incoming transition has occured. Each enabling expression can enable a number of data operation, data exchange operations, and communications:

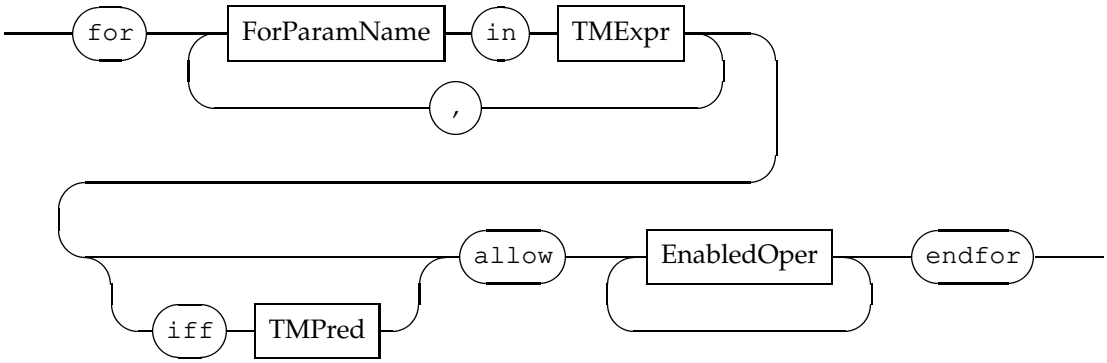
EnableExpr



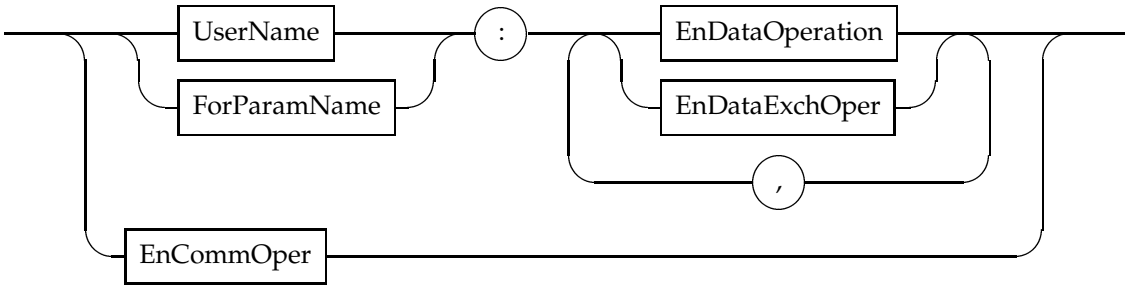
PointWithParams



ForEnableExpr

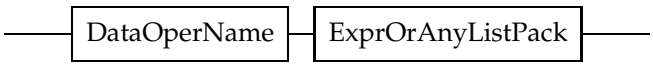


EnabledOper



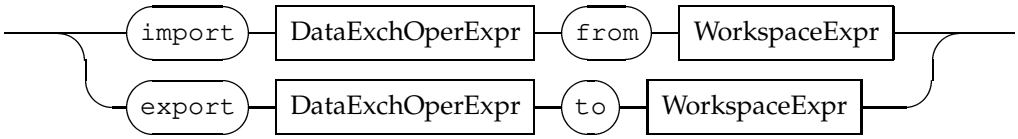
The enabling of a data operation is done by giving its name, and specifying the restrictions on the values of the parameters:

EnDataOperation

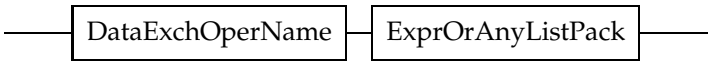


An data exchange operation can be used for importing and exporting operations between the workspace of the user and any of the other workspaces. The enabling of a data exchange operation is done by giving its name, specifying the restrictions on the values of the parameters, and stating the name of the workspace:

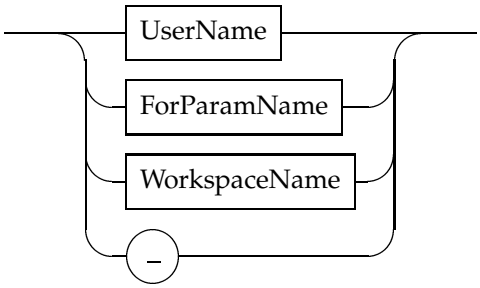
EnDataExchOper



DataExchOperExpr

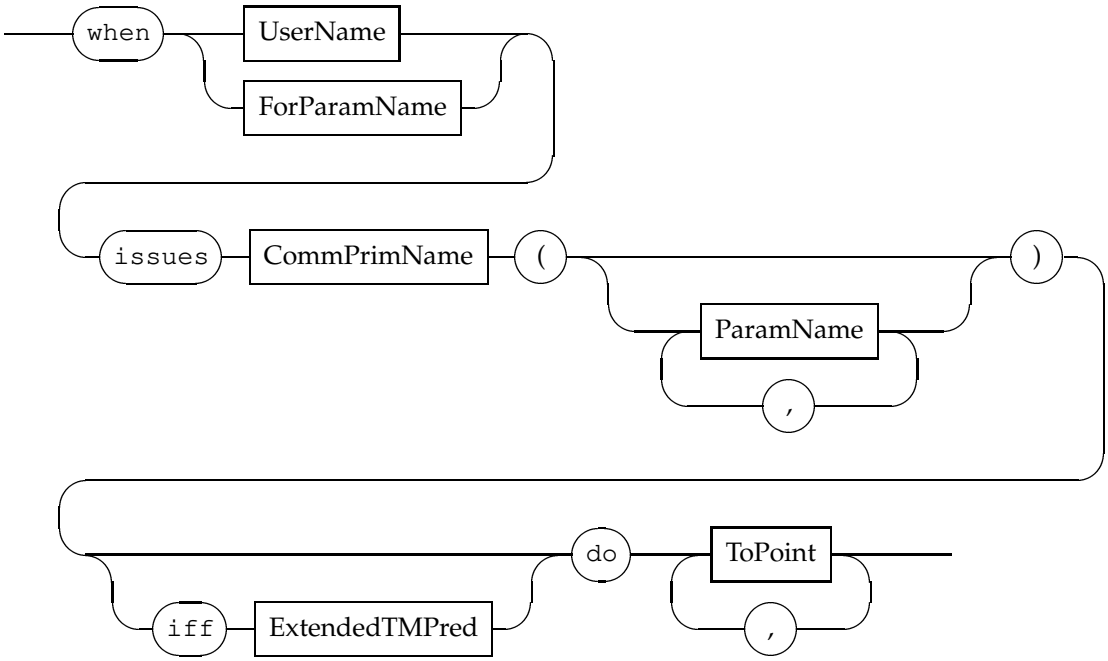


WorkspaceExpr



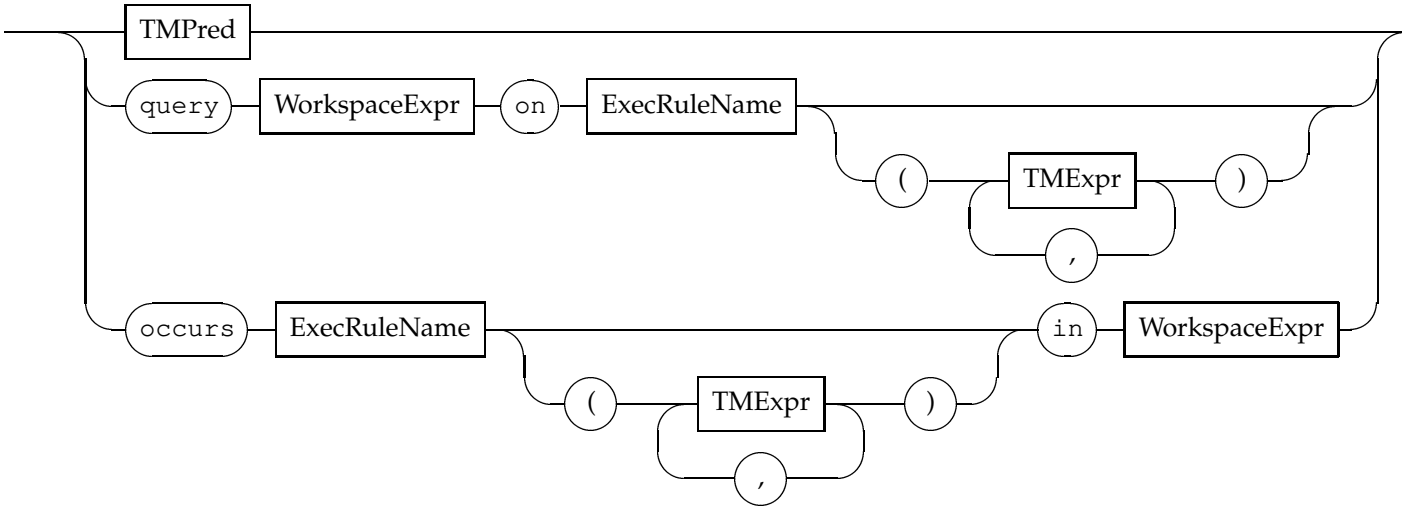
Communication operations are invoked by the users of the scenario, by giving the name of the operation and the values for the parameters. It is also possible to specify an additional condition that has to be met, in order for a transition on a given point to occur:

EnCommOper



The extended TM predicates contain two extra clauses not found in the TM predicates:

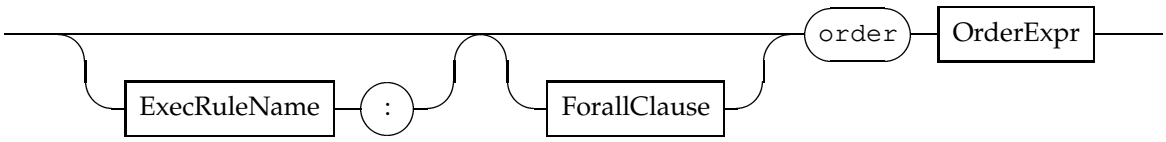
ExtendedTMPred



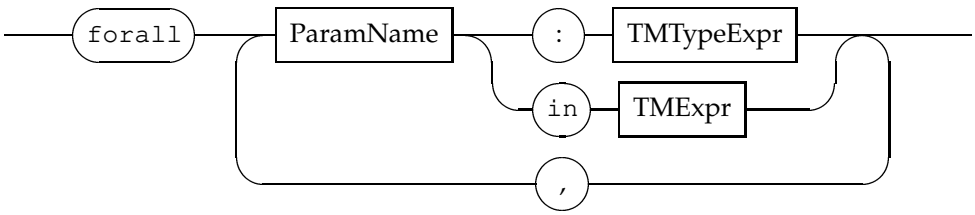
B.3 Transactional aspects

The transactional view defines the execution rules, and the history rules. We first give the grammar of the execution rules and the ordering expressions which are used in these rules:

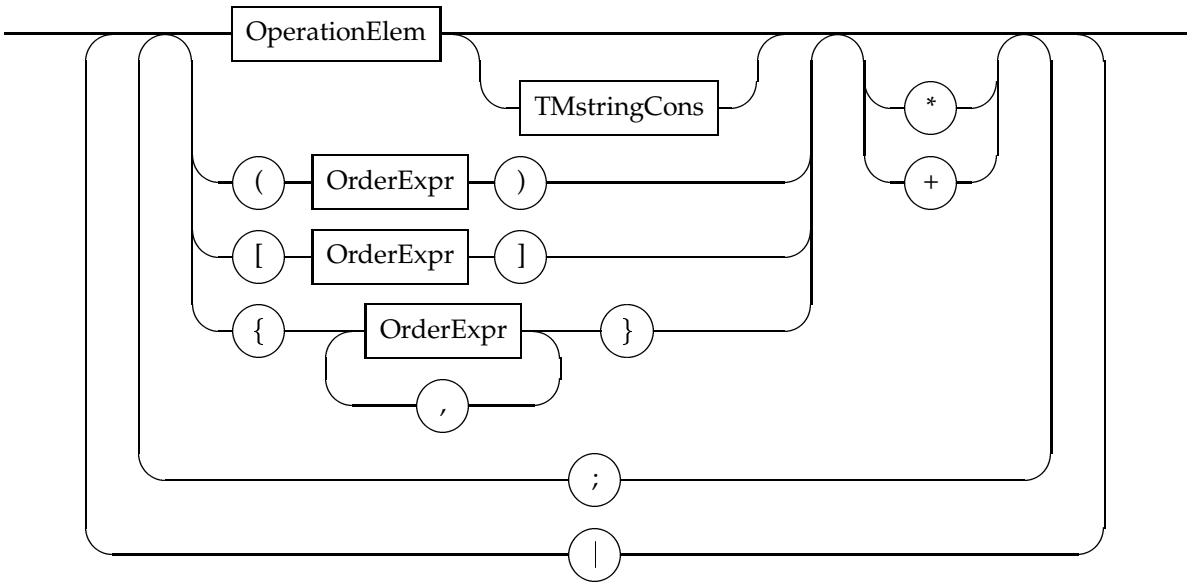
ExecRuleDef



ForAllClause

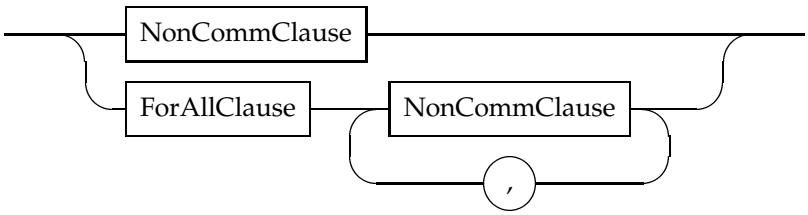


OrderExpr

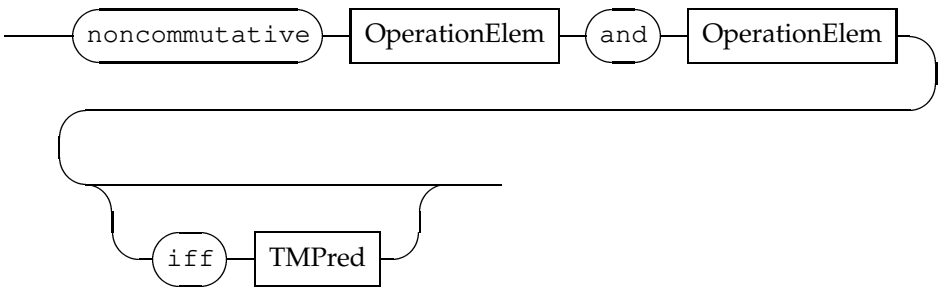


We give the grammar of the history rules below; these rules define the commutativity relationships between the operators:

HistoryRule



NonCommClause



B.4 TM grammar symbols

In the above grammar we have used a number of grammar symbols that refer to TM grammar [BBBB95]. These are the grammar symbols starting with the letters 'TM', and they refer to the TM symbols without these letters, except for TMExpr, which is denoted by X in the TM grammar.

C Example CoCoA Specification

This appendix gives a revised CoCoA specification of the CDA scenario of [FaEB96].

```
scenario write_document
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
  referee,
  actor,
  editor isa actor using cda,
  author isa actor using cda

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

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

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

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

```
begin

  shared workspace document : 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 or 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]
  begin
```

```
    parallel(ch : chapter)

      step task[in start(ℙ author)  out compl]
      begin
        on start(task_authors) enable
          for a in task_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 forall an : annotation
                  | occurs annotation_rule(ch, an) in document
                    implies query document
                                on annotation_rule(ch, an) = "processed"
            do compl
          endfor
          for a1 in task_authors, a2 in task_authors  allow
            a1 : import Chapter(ch) from a2,
                 import Annotation(ch) from a2
          endfor
        endon
      end

    endpar

    on start enable
       ed : import Chapter(_) from _,
            export Chapter(_) to document
            //....
       when ed issues startTask(c, a_s)
          iff a_s subset authors do task(c).start(a_s)
       when ed issues completeWriting() do done
    endon

  end

  step complete[in start  out done]
  begin
    on start enable
      ed : import Chapter(_) from _,
           export Chapter(_) to document
           //....
      when ed issues documentReady do done
    endon
```

```
  end

  on start 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 a1 : author, a2 : author, c : chapter, t1 : text, t2 : text
 noncommutative editChapter(a1,c,t1) and editChapter(a2,c,t2)

 forall a1 : author, a2 : author, c : chapter, t : text, an : annotation
 noncommutative remAnnotation(a1,c,an) and editChapter(a2,c,t),
 noncommutative editChapter(a1,c,t) and addAnnotation(a2,c,an)

end write_document
```