Integrating Organisational and Transactional Aspects of Cooperative Activities*

Frans J. Faase[1], Susan J. Even[1], Rolf A. de By[2], Peter M. G. Apers[1]

[1] University of Twente, Enschede, The Netherlands
E-mail: {faase,seven,apers}@cs.utwente.nl
[2] International Institute for Aerospace Survey & Earth Sciences ITC, Enschede, The Netherlands
E-mail: deby@itc.nl

Abstract. This paper introduces the specification language CoCoA. The features of CoCoA are designed for the specification of both organisational and transactional aspects of cooperative activities, based on the CoAct cooperative transaction model. The novelty of the language lies in its ability to deal with a broad spectrum of cooperative applications, ranging from cooperative document authoring to workflow applications.

1 Introduction

CoCoA is a specification language for cooperative activities [20]. The novelty of the language lies in its ability to deal with a broad spectrum of cooperative applications, ranging from cooperative document authoring to workflow applications. CoCoA is unique in that it deals with both organisational and transactional aspects of cooperation in a single language, but without coupling them as is done in *transactional workflows* [27], which assign transactional properties to the organisational steps of a workflow. In CoCoA, the organisational aspects of a cooperative activity are specified by means of a procedure definition mechanism, which is based on a formal state transition model. Transactional aspects are specified by means of execution order rules. Termination constraints link the state of the execution order rules to the transitions in the procedure definition. The language features in CoCoA are used to extend an existing database schema for cooperative work.

CoCoA has a rich set of primitives for specifying the organisational aspects of a cooperative activity. A procedure definition specifies which operations are enabled at each stage of an activity by means of *steps*. The CoCoA concept of a step is much broader than that found in traditional workflow systems. A single step can deal with more than one user, and each user can be involved in more than one step at the same time. A step can allow a user to execute many different operations, without prescribing a fixed order. Also, CoCoA allows

* This research was supported by the Esprit BRA project TransCoop (8012). TransCoop was funded by the Commission of the European Communities. The partners in the TransCoop project were GMD (Germany), Universiteit Twente (The Netherlands), and VTT (Finland).

the specification of the dynamic instantiation of a single step with different parameters. This makes *CoCoA* suitable for specifying free forms of cooperation, such as those found in cooperative document authoring, while still being able to specify the more restricted forms of cooperation found in workflow applications. Details of the procedure definition mechanism are given in Section 3.

CoCoA specifies dynamic consistency requirements on data operation invocations. To do this, *execution order rules* are used to specify the allowed orderings of invocations by means of an extended form of regular expression with operation invocation patterns. Each execution order rule applies to a parameterised subset of the invoked data operations. Details of the execution order rules mechanism of the language are given in Section 4.1.

The underlying transaction model of *CoCoA* is the CoAct transaction model [20], which is based on the idea of exchanging partial results between the users involved in a cooperative activity. In addition to a centralised database, each user has a workspace, in which private copies of the data reside. Users can exchange partial results with each other, or via the central database. After the completion of the cooperative activity, which can be considered as a long-lived transaction, the result of the activity is found in the shared central database. In the CoAct model, the operations performed on the data are exchanged, instead of the data itself. CoAct uses a merge algorithm [24], which exploits the commutativity properties of database operations, to allow the users in the cooperative activity to work in parallel. We refer to [13] for a discussion on the need for operation-based merging in the CAMERA system. Because operation-based merging permits a larger number of histories to be merged than state-based merging, more work can be done in parallel.

Both the specification language *CoCoA* [7] and the transaction model CoAct were designed during the ESPRIT TransCoop project. *CoCoA* is based on the object-oriented, functional database specification language TM [2, 3]. The semantics of *CoCoA* has been defined by mappings to the language LOTOS/TM [6], which is based on TM and the process-algebraic language LOTOS [5], both of which have well-defined semantics. A tool set for *CoCoA* has been implemented within the TransCoop project; it includes a graphical scenario editor, a simulation environment (based on the TM Abstract Machine [8]), and a compiler to a run-time environment, which consists of a CoAct transaction manager and a cooperation manager running on top of the VODAK object-oriented database system [10]. This environment is being studied in the context of the SEPIA cooperative document authoring system [22]. Proving the correctness of commutativity relationships is the subject of further research.

The different aspects of the *CoCoA* language are illustrated in this paper by means of a cooperative document authoring (CDA) example, to which workflow aspects have been added. For the sake of compactness, the details of this example are presented as the various aspects of the language are explained. Section 2 gives an introduction to the CDA example, and shows the general parts of a *CoCoA* specification by means of the example. Section 3 explains how the procedure of a scenario is specified in *CoCoA*. The subsequent sections deal with the trans-

actional consistency rules. Section 4.1 explains the execution order rules, and Section 4.2 discusses the commutativity rules that are needed for the CoAct transaction model. Section 5 looks at the integration of the organisational and transactional consistency rules. Section 6 outlines how properties of the organisational aspects can be verified. Our conclusions are given in Section 8.

2 An Example CDA Scenario

The example cooperative scenario that we use throughout this paper describes an editor who, with the help of some co-authors, must write a document that is reviewed by a referee. The document consists of a number of chapters with text, which can be spell-checked and annotated. The example has been constructed such that it demonstrates sequencing, parallelism, choice, repeated activation and dynamic step activation. It also illustrates a break-point condition that enforces that all annotations are processed, and a termination condition that guarantees that the final version of the document is spell-checked.

The organisation of the cooperative scenario consists of three steps: a *preparation step* (in which the editor writes a title page and an introduction), a *writing step* (in which the editor assigns the writing tasks to groups of authors that perform the actual writing), and a *review step* (in which the referee reviews the document).

Figure 1 provides the first part of the example scenario specification. Only the *signatures* of the data(base) operations are required, as they are defined in a separate database schema. Chapters can be added, edited, removed and spell-checked; annotations can be added to or removed from a chapter text. Type definitions also originate from the database schema, and only their names need to be mentioned.

CoCoA allows the specification of *user roles*. The actual users are determined at execution time (i.e., when the scenario is instantiated). A user role is identified by means of a user type in *CoCoA*, and is assigned a workspace type. Workspace types restrict the data operations that can be recorded in the private workspace of a user. In our example, only one workspace type is defined, and it allows all data operations. Three user types are defined: referee, editor, and author.

Users can exhibit other activities besides data operations. Through so-called *communications*, a user may initiate a state transition in the scenario. State transitions influence the set of allowed data operations of other users in the scenario. (By including a 'system' user, issuing certain communications at regular intervals, reactive applications can be specified.) In Figure 1 only the names and parameter types of the communications are provided. They are used in the procedure definition, as illustrated in the next section.

The underlying CoAct transaction model uses history merging as the principle for its operation. Each user workspace maintains a history of data operations that have been performed since the start of the scenario. A user can import or export data, and this exchange is achieved by re-executing a relevant sequence of data operations from one workspace in another workspace. To allow the user

```
scenario write_document
data types  chapter, text, annotation

database operations
  addChapter(chapter)  remChapter(chapter)
  editChapter(chapter, text)  spellCheck(chapter)
  addAnnotation(chapter, annotation)  remAnnotation(chapter, annotation)

workspace types
  cda = { addChapter, editChapter, remChapter, spellCheck,
          addAnnotation, remAnnotation }

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

communications
  introWritten(),
  startTask(chapter, ℙ author), completeTask(chapter),
  readyWriting(), documentOkay(), reviseDocument(), abortWriting()

data exchange operations
  Annotations(c : chapter)
  = select addAnnotation(c,_), remAnnotation(c, _)
  Chapter(c : chapter)
  = select addChapter(c), editChapter(c,_), remChapter(c)
```

Fig. 1. Interface specification of scenario

this data exchange, the specifier needs to indicate those data operations from the history that are relevant to a particular piece of data. The `select` construct can be used for this. Two data exchange operations are defined in Figure 1. One allows exchange of annotations, the other allows the exchange of text changes. Selection clauses use invocation patterns of the data operations, in which the symbol '_' is used to indicate a *don't-care* value of a parameter.

When exchanging data, consistency needs to be preserved, which means that additional, logically dependent data operations should also be selected from the workspace history. The rules for selecting these operations are described in Section 4.2. They are implemented in CoAct's merge algorithm, which takes two histories and tries to combine them into a consistent one.

3 Organisation of a Cooperative Activity

3.1 Procedure Definition

A *CoCoA* procedure serves to define the organisation of activities within a scenario. It lists a number of steps and a number of transition rules. The latter de-

fine how the former are chained together, and define, so to say, the coarse-grain control flow of the procedure. Each step definition defines its *entry*, *interrupt*, *signal* and *exit interaction points*, at which interaction events with other steps can take place. This form of interaction is mandatory: the initiators of such an event forces the receivers to follow. The allowed interactions are defined in the transition rules. Entry interactions activate the step; exit interactions deactivate it. There may be several of each of them. Interrupts are received while the step is active; signals are sent when it is active.

As an illustration, consider the partial specification provided in Figure 2. It defines a `preparation` step and a `writing` step, amongst others. The first has an exit (interaction point) `done`, the second has an entry point `start`. The two interaction points are made to coincide in the second transition rule, i.e., the second line with the '**on** ... **do**' syntax. This example shows a standard sequence of two steps, but more elaborate control flow can be built. As an aside, we mention that these definitions can be carried out in a graphic interface, which makes it less cumbersome.

The procedure itself also has entry and exit interaction points, and these are declared between square brackets on the header line. They signal start and end of the procedure, and are used to declare interaction with the procedure's step interaction points. In addition, the procedure header also the defines the different user roles of users in the scenario.

In some transactional workflow techniques, step-like structures serve also to define transactional boundaries. This is not the case in CoCoA, where steps only help to organise the work in smaller units of activity.

3.2 Inside Steps

A step definition defines which data operations, data exchange operations, and communications can be enabled for the users of the scenario inside the step. This is free-form usage: enabled operations can be invoked any number of times, and in any given order. The enabling takes place only when the required communication takes place, as defined in the step. There exists no explicit disabling in the language: when a step terminates all permissions issued from it are automatically withdrawn. Only by invoking (enabled) communications, can one or more other steps be activated or deactivated.

Data operations and data exchange operations are enabled inside a step for a specific user role, using the following construct:

on ⟨`intpoint`⟩ **enable** ⟨`userrole`⟩ : ⟨`operationlist`⟩ **endon**

This indicates that whenever the interaction at `intpoint` occurs, the user in the role of `userrole` is allowed to perform the listed operations, (at least) up to the point where the step terminates. Figure 3 shows examples in the context of the `preparation` step. In these examples, a literal argument value for an operation indicates that the user is allowed to invoke an operation only with that value.

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

  step preparation[in start  out done] ...

  step writing[in start  out done]
  begin
    parallel(ch : chapter)
      step task[in start(ℙ author)  out compl] ...
    endpar

    on start enable
       when ed issues startTask(c, tas) iff
                 (tas subset authors) do task(c).start(tas),
       when ed issues readyWriting() do done
    endon
  end

  step review[in start  out accept, reject, revise] ...

  on start do preparation.start
  on preparation.done do writing.start
  on writing.done do review.start
  on review.accept do done
  on review.revise do writing.start
  on review.reject do cancel
end
```

Fig. 2. Procedural specification of a scenario. Ellipses indicate omitted text.

The enabling of communications (also illustrated in the figure) requires slightly more involved syntax:

when ⟨userrole⟩ **issues** ⟨communication⟩ **do** ⟨intpoint⟩

This construct enables the user in the given user role to submit the indicated communication. The **do**-part identifies which interaction will occur.

The enabling of communications can sometimes be conditional. In such cases, a statement of the form **iff** ⟨condition⟩ is added to the communication enabling statement. Figure 2 has an example that indicates that a writing task should only be started if the involved authors are known to the overall procedure.

3.3 Dynamic Step Activation

A special form of step definition is *dynamic step activation*, of which the `task` step inside the `writing` step of Figure 2 is an example. It defines an *a priori*

```
step preparation[in start  out done]
begin
  on start enable
    ed : addChapter("title"), editChapter("title", _),
         addChapter("intro"), editChapter("intro", _),
         export Chapter("title") to document,
         export Chapter("intro") to document
    when ed issues introWritten() do done
  endon
end
```

Fig. 3. The enabling of operations and communications inside a step

unlimited number of similar tasks, which can be active in parallel. To identify them, they should be parameterised with appropriate parameters, either users or data sources, for instance. In the case of our example, the chapter serves as the identification. In this example, the editor can only issue a `startTask` communication if the prospective set of authors `tas` is a subset of the set of known `authors`. The set `tas` is transferred to the `task` step via an additional parameter associated with the interaction point `start`.

3.4 Informal Interaction Point Semantics

Interaction points identify the interaction possibilities between steps, and between a step and its substeps. If an interaction takes place, the involved steps coincide at the interaction point. We assume synchronous communication, and thus abstract away from the asynchronous communication characteristics of a possible implementation. An interaction at an entry point brings a step to life; an interaction at an exit point terminates a step. Interactions cannot be ignored, i.e., they are mandatory. A step can be defined to have several entry and exit interaction points. In addition, there can be interrupt interaction points, at which interrupts will be received and handled by the step only if it is active. These interrupts can be subject to synchronisation with interactions at points internal to the step. An active step can also submit interactions, known as signals (at signal interaction points), but the step will remain active after doing so. There is a natural relationship between all these types of interaction and the primitives of a process specification language like LOTOS [5], and we refer to [7] for a detailed discussion.

4 Transaction Consistency Rules of a Cooperative Activity

Whenever two users want to exchange information, they will perceive it as data-based exchange: the receiving user obtains a new version of the entity of inter-

est. The preservation of *data consistency* in a workspace, however, should not be data-based, but rather operation-based, as this has a far bigger potential for conflict-resolution. To this end, each workspace maintains a history of invoked operations. When an entity is selected for data export, the system will determine the *relevant* (not necessarily contiguous) *operation subsequence* of the workspace's history, and export this subsequence. Then, an attempt is made to merge this subsequence with the history of the receiving workspace.

The definition of workspace consistency is based on these ideas, and takes shape through two types of rules: *execution order rules* and *history merge rules.* The first type restricts the allowed sequences of operations in a workspace; the second type defines what constitutes the relevant operation subsequence, and allows to identify potential conflict situations.

4.1 Execution Order Rules

The *execution order rules* restrict the order of invoked data operations in the workspace history. A history contains both data operations executed by its owner, and imported data operations from other workspaces. The ordering constraints are expressed through extended regular expressions, the elements of which are data operation patterns. These patterns may include values and variables for the operation parameters. A history is *order correct with respect to an execution order rule*, if and only if it is a prefix of one of the filtered histories described by the rule's regular expression. A *filtered history* is obtained from the real history by removing all data operation invocations that do not match any pattern in the rule. This includes proper treatment of parameter instantiations. A history is *order correct* if it is order correct for all execution order rules.

The following execution order rule is defined for the given editing example:

```
data operation order
  chapter_rule :
    forall c : chapter
    order addChapter(c) "edited";
          (editChapter(c,_) "edited" | spellCheck(c))*;
          remChapter(c)
```

The rule states that a chapter can only be edited, spell-checked, or removed after it has been added to the document, and that a chapter cannot be edited or spell-checked after it is removed. This rule does not restrict how often a chapter can be edited or spell-checked. The string `"edited"` following the `addChapter(c)` and `editChapter(c,_)` operation patterns indicates that the chapter is in the edited state, directly after these data operations are carried out. Section 5 explains how these state tags are used to integrate the transactional aspects and the organisational aspects of a CoCoA scenario.

The following order rule places restrictions on the occurrence of operations on annotations:

```
  annotation_rule :
    forall c : chapter, an : annotation
```

```
order addChapter(c);
      (addAnnotation(c,an) "added"; remAnnotation(c,an))*;
      remChapter(c)
```

The rule states that annotations can only be made to a chapter after it has been added to the document, and that all annotations have to be removed before a chapter is removed.

In [9], regular expressions are used to specify the external behaviour of objects in an object database. This approach can be compared to the execution order rules of *CoCoA* when the database itself is considered as a single, complex object. Nodine [17] describes *transaction groups* as a formal notation for the specification of cooperative transactions. An LR(0) grammar is used to describe a transaction group's correctness criteria in terms of valid histories. Neither of these approaches deals with organisational aspects or history merging. Furthermore, they do not consider constraints that depend on the state tags of the execution order rules.

4.2 History Merge Rules

Consistent operation history merging is implemented via the merging algorithm of the CoAct transaction model, which needs information about the commutativity of data operations. The first requirement for performing a consistent merge is that the relevant operation subsequence is correctly determined. In our example, we cannot select an `editChapter` operation without its corresponding `addChapter` operation. The notion of *backward commutativity*, as defined in [25], is used to determine which operations depend on each other, but it is the specifier who has to indicate which pairs of operations backward commute. Given an initial set of operations selected by the user who invokes a data exchange operation, the first part of the merge algorithm calculates the minimal transitive closure of this set with respect to the defined backward commutativity conflicts.

The second part of the merging algorithm determines how the extended set of selected operations can be merged with receiving history such that a consistent result is produced. The notion of *forward commutativity*, also defined in [25], is used to detect conflicts between operations of the relevant operation subsequence and operations in the receiving history. If no conflicts are present, the two histories can be merged. In case there are conflicts, the user is given the option to either choose a smaller set of operations to be merged, or to undo operations present in the receiving history. For details about the history merging algorithm of CoAct, we refer to [24].

In *CoCoA*, commutativity relations are specified by enumerating pairs of conflicting operation patterns. For each pair, a predicate expression over the parameters of the operations identifies when a forward and/or backward commutativity conflict exists. When the conditions for forward and backward commutativity are the same, which is often the case, syntax allows to provide the predicative only once. Below, an example for the `editChapter` and `spellCheck` operations is given. Because these operations do not, in general, commute (neither forward nor backward), we specify their non-commutativity using history merge rules, as follows:

```
history rules
  forall c : chapter
  non-commutative editChapter(c,_) and editChapter(c,_),
  non-commutative editChapter(c,_) and spellCheck(c),
  non-commutative spellCheck(c) and spellCheck(c)
```

In the CDA example, we have assumed that there are no commutativity conflicts between the data operations that edit the text of a chapter and the data operations that add or remove its annotations. However, to capture the application semantics that an edit operation incorporates prior annotations that were removed since the last edit, a backwards commutativity rule can be specified to enforce that `remAnnotation` operations should always be exchanged with a subsequent `editChapter` operation. This application semantics requirement can be specified using the following history rule:

```
  forall c : chapter
  non-commutative remAnnotation(c,_) and editChapter(c,_)
  backward true  forward false
```

Because the backward commutativity relationship of CoAct is symmetric, the above conflict also implies that `remAnnotation` operations depend on previously issued `editChapter` operations. This unwanted side effect could be avoided, if the CoAct transaction model supported an asymmetric relationship, such as the *right backwards commutativity* relationship introduced in [26].

Relationship with the Execution Order Rules Both the execution order rules and the history merge rules are based on the semantics of the operations. For this reason, it is not surprising that they enforce overlapping constraints. The execution order rule `chapter_rule`, for example, specifies that the operation `addChapter` can only occur once in each history for each chapter. This allows the situation where two users issue this operation independently for the same chapter. Any attempt to merge these two operations into a single history will fail because of `chapter_rule`. Such a merge also fails if the following forward commutativity conflict is specified:

```
  forall c : chapter
  non-commutative addChapter(c) and addChapter(c)
  backward false  forward true
```

The reverse, however, is not true. Specifying the above conflict does not state that a single user can execute the operation `addChapter` more than once for the same chapter. In case an execution order rule enforces that an operation pattern can only occur once in a history, then this, in a certain sense, implies a forward commutativity conflict.

In our example, the `chapter_rule` enforces that the `addChapter` operation for a certain chapter always occurs before any of the other operations on that chapter. This means a user cannot import an `editChapter` operation without importing also the related `addChapter` operation. To ensure that the `addChapter` operation is included, a backward commutativity conflict has to be specified.

Again, we could conclude that an execution order rule implies a certain commutativity conflict.

Unfortunately, not all commutativity conflicts are 'implied' by execution order rules. The conflicts between the `editChapter` and `spellCheck` operations, for example, are not implied by any of the given execution order rules. Because it lies in the intention that these operations can be executed in any order, it is not possible to add execution order rules that would imply the existing conflicts.

It is also clear that not all order constraints as specified by the execution order rules could be enforced by the commutativity rules. A merging algorithm based on execution order rules seems to be an interesting research direction.

5 Integrating Organisation and Transaction Consistency

In the previous sections, we explained how organisational and transactional aspects of a cooperative activity are specified in isolation. Both the operation/communication enabling statements and ordering constraints are descriptive, not prescriptive: they express what *may* happen, not what *must* happen. *Termination constraints* are used for the latter purpose. They can be added to the communication enabling statements in the form of a predicate. Such a predicate is allowed to query rule states, i.e. check the current state value of a rule. If the enabling concerns an exit point, we have defined a condition that must be satisfied before the step can finish. This approach allows the formulation of break-point and termination conditions.

To express the condition that the editor can only terminate the writing step when all the chapters in the shared workspace are spell-checked, the following condition is added:

```
step write[in start, revise  out done]
begin  ....
  on start enable  .....
    when ed issues readyWriting()
    iff forall ch : chapter
          |(query document on chapter_rule(ch) <> "edited")
    do done
  endon
 step
```

Here, the predicate checks the status of the execution order rule with respect to all chapters. If the condition is not met, the `readyWriting` communication is not enabled, and the step cannot exit. The parameter list attached to the rule name serves to uniquely identify the rule instantiation, and for `chapter_rule` we need one such parameter. In case the current state of the execution order rule is not tagged with a string, the query expression returns an empty string value.

If we would like to state that a task for writing can be completed only if all the annotations are removed from the chapter (and hopefully processed as well), this can be done by adding the following condition to the transition:

```
when a issues completeTask(ch)
iff forall an : annotation
     |(query document on annotation_rule(ch, an) <> "added")
do compl
```

6 Properties of Scenarios

A major reason to use a formal specification technique is to find design flaws at an early stage of the application development process. This section discusses checking run-time properties at design time. The run-time properties can be divided into generic properties, such as termination, and user-defined ones, like post-state requirements. An example is the question of whether all chapters will be spell-checked at the end of the scenario. To a certain degree the generic properties can be checked by means of static analysis of the scenario specification, by analysing how steps are activated through transitions, ignoring the conditions on the execution order rules, which may be set on the various communication operations.

A more accurate analysis can be performed by generating the state space of the organisation of the scenario. In practical examples, generating a complete state space may be both impossible and undesirable. There are, however, sometimes possibilities for finitely representing an infinite state space. Other, more pragmatic approaches may yield still interesting analysis results by limiting the maximum number of step instantiations and data items to a fixed number. If those limits are set appropriately, one may hope to generalise the results to arbitrary numbers.

To perform such analyses, we have to provide the formal semantics of the language first. Section 6.1 below indicates how procedure specifications are mapped to a state transition system. This mapping has been automated in one of the tools developed by the TRANSCOOP project.

When organisational characteristics are analysed in isolation, the blocking conditions of termination and break points are ignored. It may be useful to check if such conditions can be met, using the properties of the execution order rules. These rules are (tagged) regular expressions that define a language, i.e. a set of sentences, and some of the blocking conditions can be rephrased as quantified logical expressions over this set. In a similar way, some of the user-defined statements about the result of a scenario can be investigated. We discuss this in greater detail in Section 6.2.

6.1 Formalisation of the Organisational Aspects

The organisation of a scenario can be represented as a state transition system. Such a system is defined by a set of state representations, and a state transition function. In our case, the first represents the status of the scenario instance, and the second represents the behaviour of the defined procedure, i.e., how instances change from one to the other.

A state of a scenario instance S consists of four parts, U, A, E and T:

- U: The assignment of users to user parameters, i.e., the set of actual users and their roles. These parameters are assigned a value when the scenario is instantiated, and this assignment does not change during the execution of the scenario instance.
- A: The set of steps currently active in the scenario. Each active step is represented as a structure a, of which the first component is a list of step names identifying the hierarchical position of the step in the procedure definition, and the second component is a (possibly empty) list of values for the parallel clause parameters that enclose the step.
- E: The set of enabled interaction points that have operation enabling statements attached. Each point is represented as a structure with four components. The first two components are the same as for a step. The third component is the name of the interaction point, and the fourth component is a list of values for the point parameters.
- T: The terminations state set. This is a possibly empty set of names of exit points of the scenario. This set is empty if and only if it represents a running scenario instance.

From any scenario state, its set of enabled operations can be inferred. The initial state of a scenario instance is determined from the scenario's entry points. Communications semantically are state transitions, and each communication is defined precisely once in the procedure definition. This allows us to construct a complete state transition function from the communications. An entry signal is interpreted as an insertion of the step into the set A of active steps. An exit signal from a step is interpreted as removal of the step from the set A of active steps. At the same time, any steps that were activated from that now deactivated step are automatically also deactivated, i.e., they are removed from A. Interrupt signals do not affect the set A, and therefore steps that are not active cannot become active because of them. Bellow we give a pseudo-code algorithm which, for a given state S and signal s, calculates the effects on the A and E components of state S:

```
P := points_of_communication(s);
while not_empty(P)
{
    for all p in interrupt_points_in(P)
    {   if step_of(p) in A
          {  E := E ∪ p; }}
    for all p in entry_points_in(P)
    {   A := A ∪ step_of(p);
        E := E ∪ p; }
    for all p in exit_points_in(P)
    {   A := A - step_and_sub_steps_of(p);
        E := E - points_of(step_and_sub_steps_of(p)); }
    P := points_reached_by_transitions_from(P);
}
```

Once the state transition model is established, it can be used to perform a state space analysis, to find out if the scenario terminates, and whether there are no

blocking states. It is also useful for checking whether the specified behaviour matches the behaviour the specifier had in mind. As the mapping does not take into account the state of the workspaces, it does not match with the run-time behaviour. The next section describes how reasoning about the execution order rules can give a more accurate analysis of run-time behaviour.

6.2 Including Transactional Aspects

The mapping described in the previous section does not take into account transactional aspects. This means that scenario instances may not terminate when the above analysis would conclude that they likely do. Also, questions like whether the last version of the chapters are always spell-checked, cannot be answered.

To make such additional claims, we can analyse the conditions under which the communications occur, and compare these with the execution order rules. In our example, the communication `readyWriting` inside the `writing` step has a condition attached to it, which queries the `chapter_rule`. An analysis of this rule reveals that this condition is violated by the `addChapter` and `editChapter` operation, and made valid again by the `spellCheck` and `remChapter` operations. By querying which operations are enabled at the various paths reaching the state in which the `readyWriting` communication is enabled, we can verify that the condition can be met.

Likewise, if we would want to verify if indeed all chapters are spell-checked when the scenario terminates successfully, we have to check if this condition is valid when the scenario terminates at the `done` exit. Analysis shows this condition is enforced when the `writing` step is deactivated through the `readyWriting` communication, and that step `review` is activated next, which can lead to a successful termination of the scenario. We know that in the reviewer is only allowed to make annotations in the `review` step, which means that the operations `addChapter` and `editChapter` are not enabled. From this, we may conclude that all the chapters will be spell-checked when the scenario terminates at the `done` exit.

7 Related Work

Several extended transaction models have been designed to deal with the relaxed correctness requirements found in cooperative systems. See, for example, the open nested transaction model [29, 10], transaction groups [17], and the ConTract model [19, 21]. Although these transaction models support long-lived, multi-participant transactions, they are limited in their support for the organisational aspects of cooperation. In contrast, workflow applications typically offer plenty of support for the organisation of activities. During the past decade, several workflow systems have been extended with transactional abilities. See, for example, the TriGS system [12], and the Exotica system [1]. However, the extension of workflow systems with transactional properties is not sufficient to model cooperation: the tight coupling of the control-flow and transactional behaviour

in these systems limits the types of behaviour that can be modelled. In addition to workflow, there are other domains for which cooperative applications have been developed, such as software engineering and cooperative document authoring [11, 18]. Although these applications have good support for various forms of cooperation, they typically do not offer transaction support.

The transactional features of *CoCoA* are based on the CoAct transaction model [20]. The CoAct model assumes that each scenario instance is an ACID transaction. All data operations are considered to be ACID subtransactions within a scenario execution. In the CoAct model, there are no 'transactional' conflicts between operations executed in different workspaces; conflicts can only arise during an attempt to merge operations from different workspaces [24]. For this reason, the CoAct model does not describe a *transactional workflow*, as defined in [27]. The transaction model by which the data operations are executed in CoAct is the *open nested transaction* model [28], which is a *semantic transaction model* [27]. The semantics of the CoAct model has played an important role in the design of *CoCoA*.

The history merging mechanism provided by the CoAct transaction model has allowed us to take advantage of high-level system primitives in the definition and implementation of the *CoCoA* language. In related work, the ASSET transaction framework [4] has identified a number of system-level primitives to support the definition of application-specific transaction models. It is shown how to use the primitives to model the relaxed correctness requirements found in cooperating transactions and workflows. It is assumed that the primitives are used in the code generated by a compiler; the issue of mapping a high-level specification language to these primitives is not addressed. In contrast, in the TransCoop Project, we have concentrated on the development of high-level language features to describe cooperation in the context of a specific cooperative transaction model.

The ASSET primitives have recently been used in [15] to synthesise delegation and the resulting history rewriting found in advanced transaction models. Our work also involves delegation, but we take a more fine-grained view with respect to the operations that are delegated. Delegations (exchanges) in the TransCoop model are made by users (specified in terms of user roles) rather than by transactions as they are in ASSET. User-directed delegation provides more flexibility for the support of cooperative activities: multiple users can participate in a transaction, and a transaction boundary is not tied to a particular user. Another difference between our work and the ASSET approach is that delegation in the TransCoop model is semantics-based, whereas in [15] delegation is described in terms of a *generic update operation* on database objects.

During the first year of the TransCoop Project, we investigated the use of the process algebraic specification language LOTOS [5] to specify cooperative activities [6]. A drawback of this approach was the fact that distinct scenario concepts were all described by events. This lack of a linguistic distinction between different concepts proved difficult for the typical scenario specifier to understand and use. It was for this reason that we opted to design the more conceptual lan-

guage CoCoA. Other formal techniques have been used to specify the semantics of workflows: Aalst [23] shows how Petri-nets can be used to verify properties of workflows, whereas [16] is an example of a communication constraint formalism based on Propositional Temporal Logic [14]. With these formalisms, we see the potential for the same drawbacks that we experienced with LOTOS. It is also the case that some semantics issues are abstracted away in these approaches. For example, [23] abstracts away from the computations done by the tasks in a workflow.

The MENTOR Project [31, 30] looks at the specification and execution of distributed workflows. The formalism is based on state and activity charts, which share similarities with the step definition facilities of CoCoA. A notable difference is that an activity in MENTOR is 'an arbitrary piece of C code' [31]. The main contribution of [30] is a method for the behaviour-preserving transformation of a centralised workflow specification into a distributed workflow specification. The MENTOR transformation assumes that the centralised state and activity chart is defined in such a way that *orthogonal business units* can be identified by the partitioning. The main distinction between the MENTOR approach and the model underlying the CoCoA language is that CoCoA addresses cooperation inside what would be a *single* business unit in the MENTOR approach. In contrast to our work, the formal model described in [30] does not address the kinds of complex values found in object-oriented databases. In a distributed MENTOR workflow, variable updates done in different activities must be detected and collected (apparently as part of the C code) for communication to other activities to keep shared data consistent [31].

8 Conclusions

In this paper we describe CoCoA, a specification language for data-intensive cooperative applications; the language is based on the CoAct cooperative transaction model. Compared to other advanced transaction model work, CoCoA takes an alternative approach in combining organisational aspects and transactional aspects of cooperative activities.

During the design of CoCoA, special attention has been paid in defining the semantics of the language through mappings to two well-defined formal languages: the process-algebraic language LOTOS, and the database specification language TM. Within the ESPRIT TransCoop project, prototype implementations of a cooperation manager that implements the organisational aspects of CoCoA specifications, and of the CoAct transaction model have been made. These were tested with a demonstrator application based on the SEPIA cooperative hypertext authoring system.

Language features for cooperative decisions could be improved upon. CoCoA lacks an 'and'-join possibility in the organisational rules of the language. The reason for this lies in the fact that different semantics can be assigned to 'and'-joins, all of them adding additional state variables. [7] suggests an extension of

the language which allows groups of users to execute a communication according a specified protocol.

The usefulness of execution order rules as a specification mechanism needs further investigation. The current implementation of the CoAct transaction model does not check execution order rules. Imposing execution order rules as a post-condition on the result histories of the current merging algorithm seems to be too restrictive an approach. Alternative merging algorithms which do take into account the execution order rules during the merge would allow more histories to be merged. On the other hand, it is not clear whether the proposed execution order rules are powerful enough to specify all possible ordering constraints.

We have observed that operation-based merging does not always result in the most intuitive semantics from the perspective of the end-users. For example, when a user wants to import a certain data item, which happens to be an older version than the one the user has, the merging algorithm is such that nothing is changed in the user's workspace, because there is nothing new to be imported. This is logically correct, but confusing to the users who are not always able to keep track of object versions. Going back to an older version can only be achieved by undoing operations.

A language to support the specification of cooperative systems must provide many diverse features. Within the TransCoop project, we designed *CoCoA* in such a way that it could be used to specify information needed by the CoAct cooperative transaction model, as well as information helpful in structuring the activities of a cooperative scenario. The combination of features in *CoCoA* has been interesting to study. The experience we have gained in using the language to describe a real system has revealed that some features are easier to use than others (as described above). The ability to specify different aspects of a cooperative system in an orthogonal manner is pleasing to us. However, the interactions of these orthogonal components requires more study. This also holds for the interaction of the merge algorithm and the execution rules of the transaction model.

References

1. G. Alonso, D. Agrawal, A. El-Abbadi, M. Kamath, R. Günthör, and C. Mohan. Advanced transaction models in workflow contexts. In *Proceedings of the 12th International Conference on Data Engineering*, pages 574–583, New Orleans, Louisiana, March 1996. IEEE Computer Society Press.
2. René Bal, Herman Balsters, Rolf A. de By, Alexander Bosschaart, Jan Flokstra, Maurice van Keulen, Jacek Skowronek, and Bart Termorshuizen. The TM Manual, version 2.0, revision f. Technical Report IMPRESS/UT-TECH-T79-001-R2, Universiteit Twente, The Netherlands, Enschede, The Netherlands, February 1996.
3. H. Balsters, R. A. de By, and R. Zicari. Typed sets as a basis for object-oriented database schemas. In Oscar M. Nierstrasz, editor, *Proceedings of the Seventh European Conference on Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 161–184, Kaiserslautern, Germany, 1993. Springer-Verlag.

4. A. Biliris, S. Dar, N. Gehani, H. V. Jagadish, and K. Ramamritham. ASSET: A System for Supporting Extended Transactions. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 44–54, Minneapolis, Minnesota, May 1994.
5. Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14:25–59, 1987.
6. Susan J. Even, Frans J. Faase, and Rolf A. de By. Language features for cooperation in an object-oriented database environment. *International Journal of Cooperative Information Systems, Special Issue on Formal Methods*, 5(4):469–500, December 1996.
7. Frans J. Faase, Susan J. Even, and Rolf A. de By. An Introduction to CoCoA. Technical Report INF-96-10, University of Twente, Enschede, The Netherlands, September 1996.
8. Jan Flokstra and Reinier Boon. The TM Abstract Machine (TAM). Internal working document, University of Twente, Enschede, The Netherlands, February 1996.
9. Nicoletta De Francesco and Gigliola Vaglini. Concurrent Behavior: A Construct to Specify the External Behavior of Objects in Object Databases. *Distributed and Parallel Databases*, 2(1):33–58, January 1994.
10. GMD-IPSI. VODAK V4.0 User Manual. Arbeitspapiere der GMD 910, Technical Report, GMD, April 1995.
11. Philip M. Johnson. Experiences with EGRET: An exploratory group work environment. *Collaborative Computing*, 1(1), January 1994.
12. G. Kappel, B. Pröll, S. Rausch-Schott, and W. Retschitzegger. TriGS$_{flow}$—Active Object-oriented Workflow Management. In *Proceedings of the 28th International Conference on System Sciences*, 1995.
13. Ernst Lippe and Norbert van Oosterom. Operation-based merging. In *Proceedings of the Fifth Symposium on Software Development Environments*, volume 17 of *ACM SIGSOFT Software Engineering Notes*, pages 78–77, Tyson's Corner, Virginia, December 1992.
14. Z. Manna and A. Pnueli. Verification of temporal programs: the temporal framework. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*. Academic Press, New York, 1981.
15. C. P. Martin and K. Ramamritham. Delegation: Efficiently Rewriting History. In *Proceedings of the Thirteenth International Conference on Data Engineering*, Birmingham, U.K., April 1997.
16. A. H. H. Ngu, R. Meersman, and H. Weigand. Specification and verification of communication constraints for interoperable transactions. *International Journal of Cooperative Information Systems*, 3(1), 1994.
17. H. M. Nodine, S. Ramaswamy, and S. B. Zdonik. A cooperative transaction model for design databases. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 3, pages 53–85. Morgan Kaufmann Publishers, Inc., 1992.
18. Atul Prakash and Hyong Sop Shim. DistView: Support for building efficient collaborative applications using replicated objects. In *Proceedings of the Fifth Conference on Computer-Supported Cooperative Work*, Chapel Hill, North Carolina, October 1994.
19. Andreas Reuter and Helmut Wächter. The ConTract Model. *IEEE Data Engineering Bulletin*, 14(1):39–43, March 1991.

20. Marek Rusinkiewicz, Wolfgang Klas, Thomas Tesch, Jürgen Wäsch, and Peter Muth. Towards a Cooperative Transaction Model—The Cooperative Activity Model. In *Proceedings of the 21st VLDB Conference*, Zurich, Switzerland, September 1995.
21. F. Schwenkreis. APRICOTS—A Prototype Implementation of a ConTract System—Management of the Control Flow and the Communication System. In *Proceedings of the 12th Symposium on Reliable Distributed Systems*, Princeton, New Jersey, 1993. IEEE Computer Society Press.
22. N. Streitz, J. Haake, J. Hannemann, W. Schuler A. Lemke, H. Schuett, and M. Thuering. SEPIA: A Cooperative Hypermedia Authoring Environment. In *Proceedings of the ACM Conference on Hypertext*, pages 11–22, Milano, Italy, 1992.
23. W. M. P. van der Aalst. Verification of workflow nets. In P. Azema and G. Balbo, editors, *Application and Theory of Petri Nets*, Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.
24. Jürgen Wäsch and Wolfgang Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering: Interoperability on Nontraditional Database Systems (RIDE-NDS'96)*, pages 76–85, February 1996.
25. W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, 1988.
26. William E. Weihl. The impact of recovery on concurrency control. *Journal of Computer and System Sciences*, 47:157–184, 1993.
27. Gerhard Weikum. Extending transaction management to capture more consistency with better performance. In *Proceedings of the 9th French Database Conference*, Toulouse, France, September 1993. Invited Paper.
28. Gerhard Weikum and Hans-Jörg Scheck. Multi-level transactions and open nested transactions. *IEEE Data Engineering Bulletin*, 14(1), 1991.
29. Gerhard Weikum and Hans-Jörg Scheck. Concepts and applications of multi-level transactions and open nested transactions. In Ahmed K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, chapter 13. Morgan Kaufmann Publishers, Inc., 1992.
30. Dirk Wodtke and Gerhard Weikum. A formal foundation for distributed workflow execution based on state charts. In *Proceedings of the Sixth International Conference on Database Theory (ICDT'97)*, volume 1186 of *Lecture Notes in Computer Science*, Delphi, Greece, January 1997. Springer-Verlag.
31. Dirk Wodtke, Jeanine Weissenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The MENTOR Project: Steps towards Enterprise-Wide Workflow Management. In *Proceedings of the 12th International Conference on Data Engineering*, February 1996.