From Centralized Workflow Specification to Distributed Workflow Execution

Peter Muth[1], Dirk Wodtke[1], Jeanine Weissenfels[1],
Angelika Kotz Dittrich[2], Gerhard Weikum[1]

[1] University of the Saarland
Department of Computer Science
P.O. Box 151150
D–66041 Saarbrücken, Germany
E–Mail: {muth, wodtke, weissenfels, weikum}@cs.uni–sb.de
WWW: http://www–dbs.cs.uni–sb.de/~mentor

[2] Union Bank of Switzerland
Bahnhofstrasse 45
CH–8021 Zürich, Switzerland
E–Mail: kotz–dittrich@ubs.ch

Abstract

Current workflow management systems fall short of supporting large-scale distributed, enterprise-wide applications. We present a scalable, rigorously founded approach to enterprise-wide workflow management, based on the distributed execution of state and activity charts. By exploiting the formal semantics of state and activity charts, we develop an algorithm for transforming a centralized state and activity chart into a provably equivalent partitioned one, suitable for distributed execution. A synchronization scheme is developed that guarantees an execution equivalent to a non-distributed one. This basic solution is further refined in order to reduce communication overhead and exploit parallelism between partitions whenever possible. The developed synchronization schemes are compared in terms of the number and size of synchronization messages.

1 Introduction

Workflow management is a rapidly growing research and development area of very high practical relevance (Georgakopoulos et al., 1995; Mohan, 1994; Vossen and Becker, 1996; Workflow Management Coalition, 1995; Sheth, 1996). Typical examples of (semi–automated) workflows are the processing of a credit request in a bank, the editorial handling and refereeing process for papers in an electronic journal, or the medical treatment of patients in a hospital. Presently, virtually no major enterprise executes its mission-critical business processes under the support of a workflow management system. Because of their monolithic and centralized architecture, most of the systems that are currently on the market cannot cope with the requirements that large–scale workflow applications pose (Gawlick, 1994). The workloads that are to be anticipated for the future are in the order of several thousands of involved human actors and of ten thousands to more than hundred thousands of workflow executions in parallel (Kamath et al., 1996).

1.1 Problem Statement

The target of this paper are large scale workflow applications which involve several business units, create a very large number of concurrent workflow instances, and thus impose a high load on the workflow engine. For mission critical workflows, high availability and failure-resilience is required.

The specification of a workflow is usually done via high–level graphical interfaces, e.g., by drawing nodes and arcs. This specification must be mapped into an internal representation that serves as the basis for execution. In many workflow management systems, the underlying internal representation uses an ad hoc model and thus lacks capabilities for formal correctness reasoning. In contrast, general–purpose specification formalisms for dynamic systems such as Petri nets, state charts, temporal logic, or process algebras, which are pursued in various research projects and a few products, come with a rich theory and thus provide an excellent basis for formal proofs. For the work in this paper we have adopted the method of state and activity charts by Harel et al. (Harel, 1987a; Harel, 1987b; Harel, 1988; Harel et al., 1990; Harel and Naamad, 1995), which is perceived by practitioners as more intuitive and easier to learn than Petri nets yet has an equally rigorous semantics. In particular, state chart specifications are amenable to model checking (McMillan, 1993; Helbig and Kelb, 1994), so that critical workflow properties that are expressible in temporal logic can be formally verified; an example would be that a credit request must be rejected if it turns out that the customer has insufficient collaterals.

Formal reasoning about specifications implicitly assumes a centralized execution model; there is no notion of distribution, interoperating workflow engines, and so on. So tools for formal reasoning must have access to the complete workflow specification in a uniform representation, regardless of whether the workflow may in reality span different autonomous business units of an enterprise or even different enterprises. In fact, however, distributed, decentralized execution of workflows (Alonso et al., 1995) is a mandatory requirement for complex, large–scale applications for two reasons.

- Such applications may involve a very large number of concurrent workflow instances which impose a high load on the workflow engine. Consequently, scalability and availability considerations dictate that the overall workflow processing must be distributed across multiple workflow engines that run on different servers, and this workload partitioning may itself require the partitioning of individual workflows.
- Whenever a workflow spans multiple business units that operate in a largely autonomous manner, it may be required that those parts of a workflow that are under the responsibility of a certain unit are managed on a server of that unit. Thus, the partitioning and distribution of a workflow may fall out naturally from the organizational decentralization.

A distributed workflow execution requires synchronization between the underlying workflow engines. As large scale workflows will involve several business units, within or across enterprises, minimizing communication costs is an important issue. In wide area networks, the latency for establishing a connection is a major performance factor. Therefore, the goal is not only to reduce the overall message size, but also to reduce the number of synchronization messages. Failure–resilience and high availability are further requirements of utmost importance. In this setting, the integration of standard middleware components (Bernstein and Newcomer, 1997) such as TP monitors seems to be a well–suited approach (Dayal et al., 1993).

1.2 Contribution of the Paper

Partitioning a centralized workflow specification in order to enable distributed execution must preserve the original execution semantics. For large scale workflows, this is only feasible if the original specification is based on a formal model with rigorous semantics, and the partitioning is carried out automatically. For the well known specification method of state and activity charts we have developed a partitioning algorithm that transforms the original state chart into *orthogonal components*. The semantics of the original and the orthogonalized state chart is provably equivalent. For distributed execution, the orthogonal components are assigned to a number of workflow servers such that each server executes a *partition* of the original workflow containing at least one orthogonal component.

There exists an inherent cost in the distributed execution of large scale workflows in a real–life setting, namely, the necessary amount of communication for the correct synchronization. This paper combines the distributed execution with a comprehensive model for the synchronization of a set of more or less autonomous workflow engines. We proceed in three major steps:

(1) We first develop a basic solution for the synchronization of distributed workflow engines where the communication is based on the primitives provided by a TP monitor.

(2) We improve this solution by restricting communication between the workflow engines to the cases where the progress in one workflow engine potentially influences other workflow engines. This scheme guarantees the correct workflow execution and, in addition, exploits the potential parallelism of activities as defined in the workflow specification.

(3) To evaluate the feasibility of our approaches, we analyze the communication costs and compare them by means of an example workflow.

The work reported here is embedded in the Mentor project (Weissenfels et al., 1996; Wodtke et al., 1996) on "middleware for enterprise–wide workflow management". This project uses a state chart based workflow engine and integrates our own "glueing" software together with the TP monitor Tuxedo (Tuxedo System 5, 1994; Primatesta, 1994) as the backbone of the execution environment. The idea of transforming a centralized state chart specification into a behaviorally equivalent partitioned state chart specification that is amenable to distributed execution has been introduced in (Wodtke, 1997; Wodtke and Weikum, 1997).

The outline of the paper is as follows. Section 2 discusses related work. Section 3 gives a brief overview of our system architecture for distributed execution of enterprise–wide workflows. In Section 4, we describe our specification method of state and activity charts and introduce our running example. In Section 5, a transformation method for partitioning a given state chart based workflow specification into subworkflows is described, and its correctness is shown. In Section 6, different alternatives for synchronizing the distributed workflow execution are presented. Finally, in Section 7 we quantify the communication costs for the different synchronization schemes presented in Section 6. Section 8 concludes the paper.

2 Related Work

A wide range of products and research projects offer support for workflow management. Some examples of commercial products are FlowMark, Lotus Notes, Staffware, etc. (see (Georgakopoulos et al., 1995; Jablonski and Bussler, 1996; Mohan, 1994) for an overview). The products have been primarily developed for use within an office environment and are not geared towards department–spanning or even enterprise–wide workflows. Typically, all relevant data on an ongoing workflow is kept on a single central server. The most critical deficiencies of these products are to be seen in a lack of scalability and fault tolerance; for example, communication between activities is often based on e–mail, which is way behind the strong guarantees that one would expect in online transaction processing.

There is a plethora of related research on the aspect of workflow specification and business process modelling. This work ranges from using (and possibly extending) standard specification methods such as Petri net variants to specifically designed languages. Work in the first category includes (Kappel and Schrefl, 1991; Ellis and Nutt, 1993; Oberweis et al., 1994); the language approach is pursued, for example, in (Bernstein et al., 1995; Forst et al., 1995) and in the Workflow Process Definition Language (WPDL) of the Workflow Management Coalition (Workflow Management Coalition, 1995). In addition, some approaches such as (Kappel et al., 1995; Dayal et al., 1991; Rusinkiewicz and Sheth, 1994) are based on Event–Condition–Action rules (ECA rules), as used in active database systems (Widom et al., 1995), for describing the control flow between activities. We are not aware of any other project that is based on state transition machines like the state chart method.

Less work exists on the distributed and scalable execution of workflows. One of the first and most advanced research projects that are middleware–centered and address especially reliability issues is the ConTract project (Wächter and Reuter, 1992; Schwenkreis, 1993; Reuter and Schwenkreis, 1995). The focus of this project has been to extend transaction-oriented runtime mechanisms for fault-tolerant workflow execution in a distributed environment. Specification issues, on the other hand, have been of secondary concern in this project. Other notable research projects with similar objectives are the DOM project (Georgakopoulos and Hornick, 1994) which is primarily oriented towards applications in the telecommunications industry and views workflows as a set of control flow dependencies between transactional steps, and the MOBILE project (Jablonski, 1994) which aims to develop a comprehensive and modular architecture of the execution environment. A distributed enactment service for workflows is also studied in the METEOR project (Sheth et al., 1996; Das et al., 1997). The architecture is based on CORBA (OMG, CORBA, 1995b) and intends to use object services like the Object Transaction Service (OTS), the Concurrency Control Service and the Persistence Service once they are available (OMG, CORBAservices, 1995a).

Very similar to our approach with respect to availability and scalability issues, is the Exotica project (Alonso et al., 1995; Alonso et al., 1996) which aims to enhance FlowMark by addressing these issues. An approach has been discussed where parts of the workflow specification are distributed across a set of workflow engines such that the workflow can be executed in a distributed fashion, based on IBM's MQ Series (Alonso et al., 1995; Blakeley et al., 1995). However, synchronization issues are not considered at a formal level.

Another approach which aims at the distribution of the workflow specification is based on "intelligent" information carriers (INCAs) such as smart cards (Barbara et al., 1996). During the workflow execution an INCA which contains the context of a workflow instance is routed from one processing station to the next according to the control flow specification that is stored on the INCA and in the processing stations. Although, similarly to our approach, no processing station has complete knowledge of all steps that comprise the workflow, the INCA approach is different in that it does not allow for parallel processing of activities since each INCA implicitly serves as an exclusive token that can solely be processed at a single location at each time.

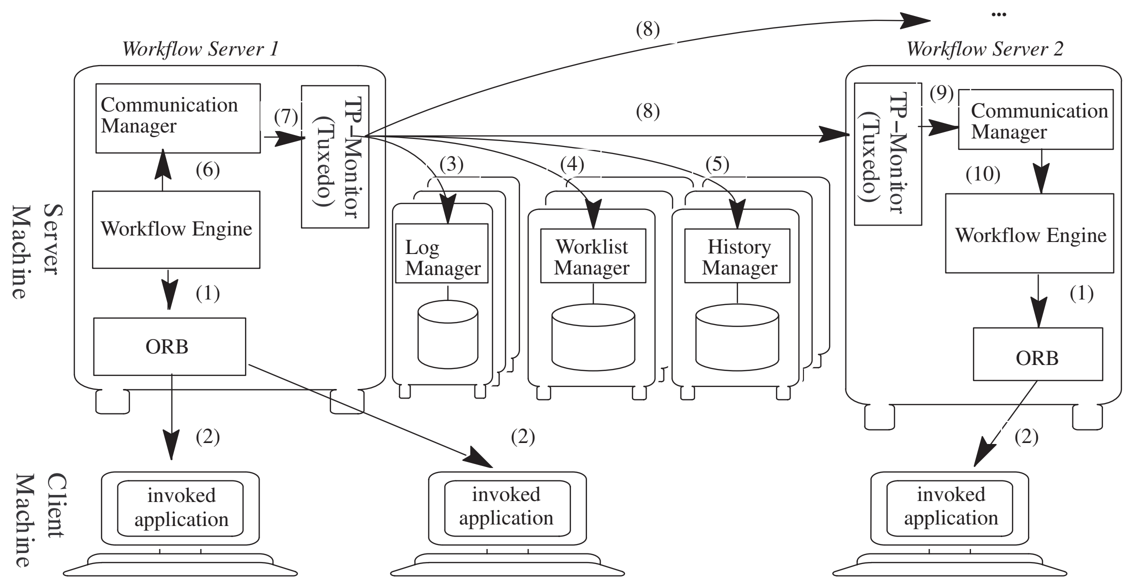

Fig. 1 : Components of the Mentor execution environment and their interaction

3 System Architecture

We now introduce the Mentor architecture for enterprise–wide workflow management (Weissenfels et al., 1996; Wodtke et al., 1996), which is in line with the reference model of the Workflow Management Coalition (Workflow Management Coalition, 1995). It is based on a *client–server model,* as shown in Fig. 1. The workflow itself is orchestrated by appropriately configured *servers*, while the invoked applications of a workflow's various activities are run at *client* sites (where the applications may in turn issue requests to other servers, regardless of whether an application is invoked within a workflow or not). The *workflow engines* and a set of key components, i.e., a *log manager*, a *worklist manager*, a *history manager*, and a *communication manager* are run on the servers. The architecture relies mostly on *standard middleware components* and conceives additional components only where the standard components fall short of functionality or scalability. Particularly, a TP monitor (Gray and Reuter, 1993; Bernstein and Newcomer, 1997), which provides transactional services and persistent message queues, is integrated for fault tolerance reasons. We use the TP monitor Tuxedo (Tuxedo System 5, 1994; Primatesta, 1994). Updates to the local system configuration of workflow engines are stored in transactional resource managers, under the control of the TP monitor. An object request broker (ORB) (Mowbray and Zahavi, 1995; OMG, CORBA, 1995b) is integrated to cope with the potential heterogeneity of the invoked applications that belong to the workflow. Furthermore, the architecture is an *open and modular architecture* where – if needed – further components can be added and components can easily be replaced by alternative implementations. Particularly, the architecture allows the coexistence of workflow servers with different workflow engines and the coexistence of multiple worklist managers each with its own worklist policy.

We will now describe the flow of control between all system components of the Mentor execution environment. The following numbers refer to the numbers that are annotated to the arcs in Fig. 1.

(1) Each workflow engine executes its corresponding partition of a workflow, and can thus invoke a certain subset of activities directly. This is performed by calling an *Object Request Broker (ORB)*. We use Orbix (IONA Technologies, 1995) as our CORBA compliant ORB (OMG, CORBA, 1995b).

(2) The ORB maps the calling parameters derived by the workflow engine on the actual calling parameters of the application and invokes the application.

(3) The *log manager* keeps track of every change of the system configuration of the local workflow engine. In the case of a system crash, this allows the workflow engine to recover to the latest system configuration before the crash happened.

(4) Whenever an activity becomes ready for execution the according information is sent to the *worklist manager* as a new work item. The worklist manager assigns work items to one of the actors who are able to fill the corresponding role. The assignment of work items is based on information about role resolution policies, vacation periods, etc. which is stored in the worklist database.

(5) Similarly to the log manager, the *history manager* is in charge of bookkeeping of workflow executions. However, the objective of the history manager is to store information for monitoring purposes and to allow for both online status inquiries and long-term evaluation.

The following operations implement the posting of changes in the system configuration of local workflow engines to other workflow engines if the change is relevant for them. This ensures that the global system configuration consisting of all local system configurations is kept consistent:

(6) Whenever a system configuration in a local workflow engine changes, i.e., the corresponding transaction is ready to commit, the update is propagated to the local communication manager.

(7) The communication manager determines the workflow engines that need to receive the update, prepares a synchronization message representing the update, and invokes the TP monitor. The task of the TP monitor is twofold. At first, it stores the synchronization message together with a list of recipients in a persistent message queue. Secondly, it combines the updates of the system configuration of the local workflow engine and the insertion of the synchronization message into the message queue into a single atomic transaction. This ensures that each committed update of the system configuration is eventually propagated to the receiving workflow engines.

(8) The system configuration information is propagated by means of the communication primitives provided by the TP monitor.

(9) At the recipient site, the TP monitor reads the received message from the local message queue, which is also stored on stable storage. Reading the message is performed inside the transaction that also tracks the local state progress. This ensures that the message is processed exactly once. The corresponding system configuration information is then propagated to the local communication manager.

(10) The communication manager forwards the received system configuration information to its local workflow engine. Based on the received data, the local system configuration is updated. The corresponding transaction forms an atomic unit with the transaction that reads the synchronization message from the queue of incoming messages. This guarantees that the update to the system configuration eventually reaches the local workflow engine.

4 Centralized Workflow Specification

In this section, we will describe the specification of distributed workflows based on the formalism of state and activity charts (Harel, 1987a; Harel, 1987b; Harel et al., 1990; Harel and Naamad, 1995). State charts are well established in software engineering (Harel and Gery, 1997), especially for specifying reactive systems. The benefit of employing a formally based specification method is that there exists a precise operational semantics. Consequently, the semantics of our state and activity chart based workflow specification is independent of the implementation of the workflow engine. This is crucial, for example, for the correctness reasoning of workflow specifications since it allows using standard verification tools (e.g., symbolic model checking) to prove that a workflow specification is a correct model of the corresponding business process (Wodtke, 1997).

Our design procedure for workflow specifications is organized in two steps: In the first step, a centralized workflow specification consisting of at least one state chart and one activity chart is developed. Specifications that are given in another language (e.g., in a scripting language such as FlowMark's FDL (IBM Corp., 1994)) can be converted into state and activity charts. Basically, this implies a decoupling of the specification environment from the execution environment so that state and activity charts can serve as a canonical internal workflow representation. In the second step, the workflow specification is partitioned into a set of subworkflows such that all activities and corresponding states of one business unit constitute one partition. The challenge is to guarantee that the partitioning preserves the behavior of the original workflow specification.

Example Workflow – Credit Processing

Throughout the paper, we will use an example from the area of credit processing in a bank. It considers credit requests from companies, which is a very important and sensitive part of a bank's business. An application for a credit involves, among other things, the checking of the company's current credit balance, the company's credit rating and a risk evaluation. The activities that are required for the decision making are thus spread over the different divisions and sites. In the sequel, a very simplified version of this workflow will be used for the description of state and activity charts

Overview of the State Chart Specification Method

State and activity charts were originally developed for reactive systems (e.g., embedded control systems in automobiles) and have been quite successful in this area. They comprise two dual views of a specification.

Activities reflect the functional decomposition of a system and denote the "active" components of a specification; they correspond directly to the activities of a workflow. An *activity chart* specifies the data flow between activities, in the form of a directed graph with data items as arc annotations. An example for an activity chart is given in the left part of Fig. 2. It defines the activities *ENCR*, *CCW*, *RSK*, *DEC,* and *ERR,* which constitute our credit processing workflow, and the data flow between the activities. In activity *ENCR* (enter credit request) the companys credit request is entered into a credit database. The company information *CO_INF* constitutes the data flow from *ENCR* to activities *CCW, RSK* and *ERR*. Activity *CCW* checks the company's credit worthiness by looking up the appropriate account and balance sheet data of the company. Activity *RSK* evaluates the potential risk that is associated with the requested credit. *RSK* takes into account the bank's overall involvement in the affected branch of industry (e.g., the total of all

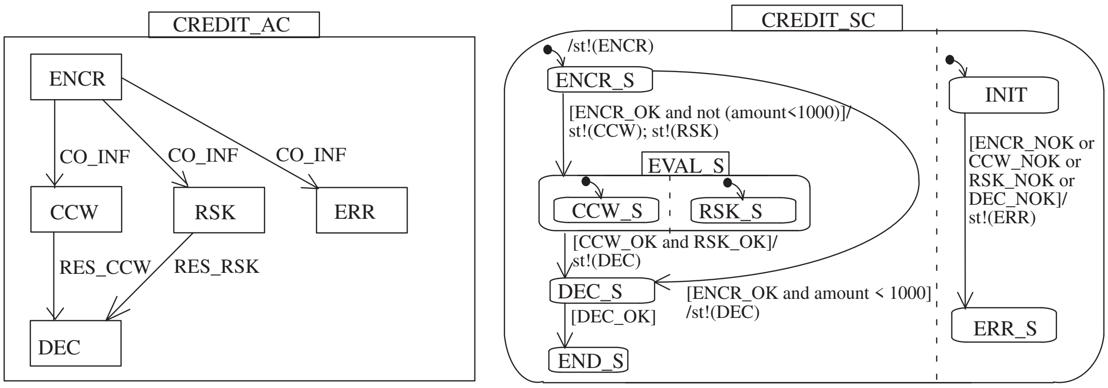

Fig. 2 : Activity chart and state chart of the workflow "credit processing"

credits already granted to computer companies) and the requested currency. Finally, the decision activity *DEC* records the decision that is eventually made on the credit request (i.e., approval or rejection); this would typically be an intellectual decision step based on the results *RES_CCW* and *RES_RSK* of the *CCW* and *RSK* activities. These four activities are complemented by a fifth activity *ERR,* which may be invoked as a generic error handler, for example, to stop the processing of the workflow when the credit requestor goes out of business.

State charts reflect the behavior of a system in that they specify the control flow between activities. A state chart is essentially a finite state machine with a distinguished initial state and transitions driven by Event–Condition–Action rules (ECA rules). Each transition arc between states is annotated with an ECA triple. A transition from state *X* to state *Y* fires if the specified event *E* occurs and the specified condition *C* holds. The effect is that state *X* is left, state *Y* is entered, and the specified action *A* is executed. Conditions and actions are expressed in terms of variables, for example, those that are specified for the data flow in the corresponding activity chart; conditions and events may also refer to states by means of special predicates like *IN(s)* which becomes true if state *s* is currently entered. In addition, an action A can explicitly start an activity, expressed by *st!(activity)*, and can generate an event *E* or set a condition *C*. ECA rules of this kind are notated in the form *E[C]/A*. Each of the three components may be empty. Every state change in a state chart execution is viewed as a single step; thus, state changes induce a discrete time dimension.

Two important additional features of state charts are *nested states* and *orthogonal components*. Nesting of states means that a state can itself contain an entire state chart. The semantics is that upon entering the higher–level state, the initial state of the embedded lower–level state chart is automatically entered, and upon leaving the higher–level state all embedded lower–level states are left. The possibility of nesting states is especially useful for the refinement of specifications during the design process and for incorporating existing (sub)workflows. Orthogonal components denote the parallel execution of two state charts that are embedded in the same higher–level state (where the entire state chart can be viewed as a single top–level state). Both components enter their initial state simultaneously, and the transitions in the two components proceed in parallel, subject to the preconditions for a transition to fire.

An example of a state chart is given in the right part of Fig. 2 for the credit processing workflow. For each activity of the activity chart there exists a state with the same name, extended by the suffix *_S*. Furthermore, the workflow consists of three major sequential building blocks, *ENCR_S*, *EVAL_S*, and *DEC_S* where *EVAL_S* consists of two orthogonal components, *CCW_S* and *RSK_S*. Hence, the activities *CCW* and *RSK* are executed in parallel. For each activity, its outcome in terms of an error variable is available. For example, variable *ENCR_OK* is set to *true* after termination of activity *ENCR* if no error occured. Otherwise the variable *ENCR_NOK* is set to *true* and an error handling activity is started. Initialization and error handling of the workflow are specified in an orthogonal component.

5 Partitioning of Workflow Specifications

In this section, we present our method for the partitioning of workflow specifications. As we will see, partitioning the state chart is the most challenging part. The partitioning transforms a state chart into a behaviorally equivalent state chart that is directly amenable to distributed execution in that each distributable portion of the state chart forms an orthogonal component. The first step towards a partitioned specification is thus called "orthogonalization". The outcome of this step is another state chart that can easily be partitioned. The proof that this transformation is indeed feasible in that it preserves the original state chart's semantics is cast into showing that the resulting "orthogonalized" state chart is a homomorphic image of the original specification. Note that we do not address the issue of "streamlining" or optimizing business processes by altering the invocation order of activities. We are solely concerned with preparing workflow specifications for distributed execution.

Partitioning of state and activity charts

The partitioning of a workflow specification consists of two elementary transformations: the partitioning of the activity chart and of the state chart. We assume that for each activity of the activity chart there exists an assignment to the corresponding execution role and thus an assignment to a department or business unit of the enterprise. Therefore each activity can be assigned to a workflow server of the corresponding department or business unit. Consequently, the partitioning of the activity chart falls out in a natural way in that all activities with an identical assignment form a partition of the activity chart.

While this is straightforward, the partitioning of the state chart requires more than simply assigning states to partitions. State charts can be state trees of arbitrary depth where parent states may have assignments that differ from the assignments of their children states. Furthermore, transitions may interconnect states at different levels of the state tree and/or with different assignments. In order to partition a state chart without changing its behavior, we pursue an approach which is different from the partitioning of the activity chart and is organized in three major steps:

(1) assignment of states to activities

In this step each state is assigned to an activity of the activity chart. For each state, this assignment allows to derive to which execution role and department or business unit it belongs. In the case of nested state charts where a state can contain an entire state chart the higher level state will correspond to a higher–level activity chart with embedded subactivities.

(2) orthogonalization of the state chart

For each state of the state chart an orthogonal component is generated. Each generated orthogonal component emulates the corresponding original state *S* by means of two states, *in_S* and *out_S*, which are interconnected by transitions in both directions. These transitions correspond to the transitions that lead to or originate from state *S*. The transition labels of these transitions are extended by conjunctive terms that refer to the source states of the original transitions. Figure 3 illustrates the orthogonalization for the state and activity chart

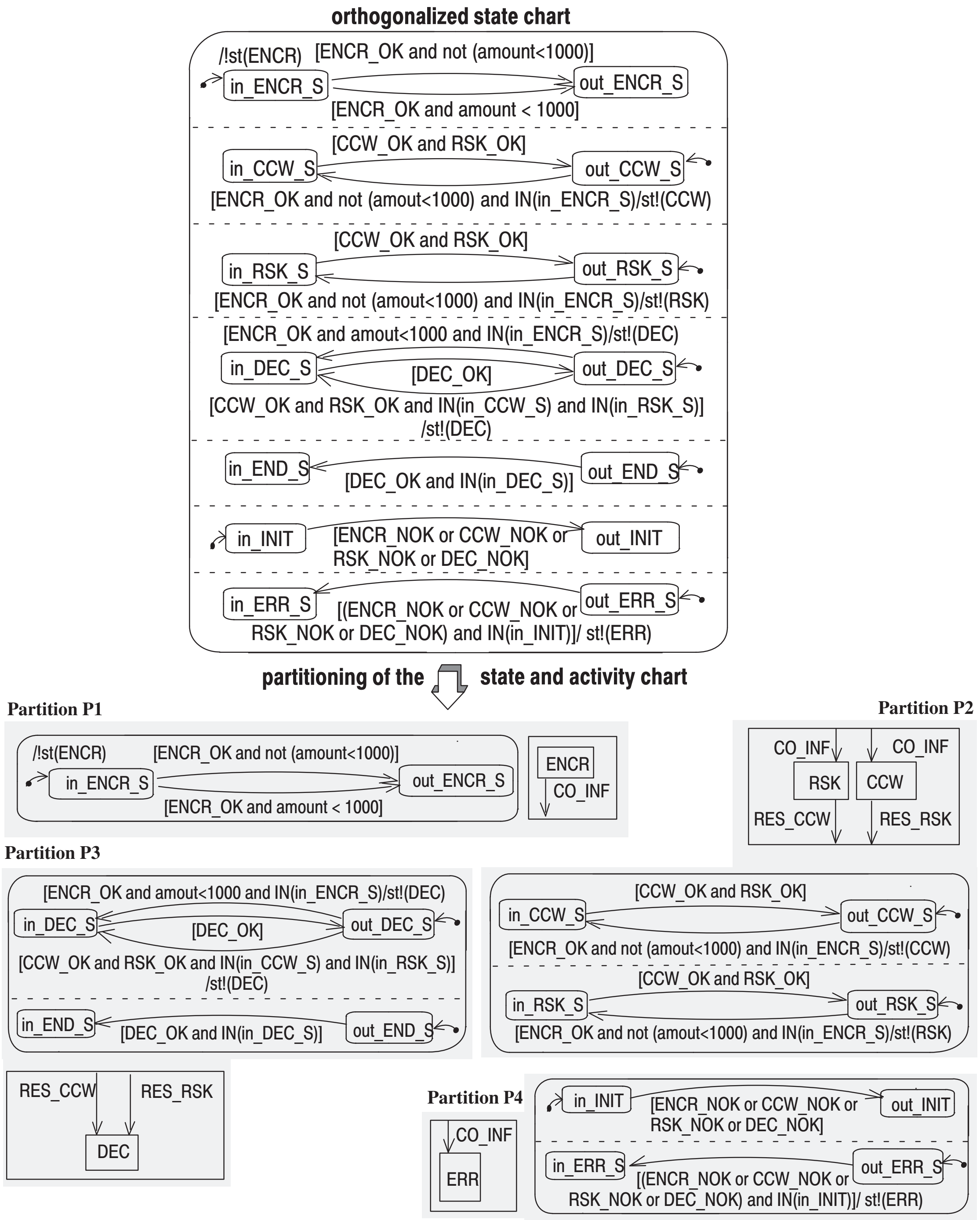

Fig. 3 : Partitioning of the workflow "credit processing"

of Fig. 2. For example, for the transition that interconnects states *ENCR_S* and *EVAL_S* the following transitions are generated: a transition from *in_ENCR_S* to *out_ENCR_S*, a transition from *out_CCW_S* to *in_CCW_S*, and a transition from *out_RSK_S* to *in_RSK_S*. The first transition has the same condition component in its transition label as the original transition, *ENCR_OK and not(amount<1000)*. The other two transitions have the condition *ENCR_OK and not(amount<1000)* extended by a conjunctive term *IN*(*in_ENCR_S*) as their condition components and the start instructions for the corresponding activities as action components of their transition labels. This extension of condition components guarantees that in any case in the orthogonalized state chart all transitions which originate from an original transition fire at the same time. Consequently, in the example the activities *CCW* and *RSK* are started if and only if state *ENCR_S* is left, and the credit amount *amount* is not less than 1000. Note that the partitions of Fig. 3 have been generated by our partitioning algorithm. This is the reason for the two transitions from state *in_ENCR_S* to state *out_ENCR_S* which could be merged into a single one with just *ENCR_OK* as condition component.

(3) assignment of partitions to workflow servers
In this step, the orthogonal components that have been generated in step (2) are grouped to state chart partitions. The criterion for grouping a set of orthogonal components to form a state chart partition is the assignment of their underlying original states to the same department or business unit of the enterprise.

Note that this partitioning procedure can cope with nested activities without special treatment. For the example of Fig. 3, we assume that the activities *CCW* and *RSK* belong to the same partition, whereas the remaining activities belong to different partitions. The data flow between activities belonging to different partitions is indicated by data flow connectors that start or end at the borders of the boxes that represent partitions. If a partition contains several orthogonal components which have formed a coherent part in the original state chart, it is possible to merge these components to rebuild a part of the original state chart. Consider for example partition *P4* in Fig. 3. State *in_INIT* and *out_ERR_S* are left if one of the activities *ENCR, CCW, RSK* and *DEC* fails, i.e., the corresponding "*_NOK*" variable becomes *true*. The transition from *out_ERR_S* to *in_ERR_S* contains the term *IN(in_INIT)*. If we change the source of this transition to the state *in_INIT*, and remove the now unnecessary states *out_INIT* and *out_ERR_S*, we get back a part of the original state chart, namely the orthogonal component on the right of the state chart in Fig. 2. An algorithm for this merge is given in (Wodtke, 1997). For the sake of simplicity, we do not consider such merges in the rest of the paper.

Correctness of the partitioning

The feasibility of the described transformation depends on the assumption that the semantics of the original state chart is preserved. What is needed is a formal correctness proof that this is indeed the case. We will concentrate on the correctness proof of the orthogonalization (step (2)), which is the most critical step with regard to possible changes of the behavior of the state chart.

First, we will refer to a formally defined operational semantics of state charts, which is a simplified version of the semantics given in (Harel and Naamad, 1995). It is tailored to our context of workflow specification (see (Wodtke, 1997) for full details). To this end, we define the set *SC* of *system configurations*. The system configuration describes the set of currently entered states and the context, i.e., the current values of conditions, events, and variables that are part of the state chart. For the execution of a state chart, we define a step operator, *step*, which maps a system configuration to its successor system configuration.

Fig. 4 illustrates the interdependence between the operational semantics of the original state chart and the operational semantics of its orthogonalized representation. We view both state charts as algebras with identical signatures. The carrier sets are the sets of system configurations, and the only operator is the step function. In order to distinguish between both carrier sets and operators, the carrier set and the operator of the algebra of the orthogonalized state chart (i.e., algebra 2) are denoted by *SC'* and *step'*, respectively. The interdependence between the two algebras is described by the mapping h_{SC}, which maps each system configuration of algebra 1 onto a system configuration of algebra 2. For the contexts this mapping is the identity mapping. With regard to the state configuration, h_{SC} maps each state *Z* that is currently entered onto the corre-

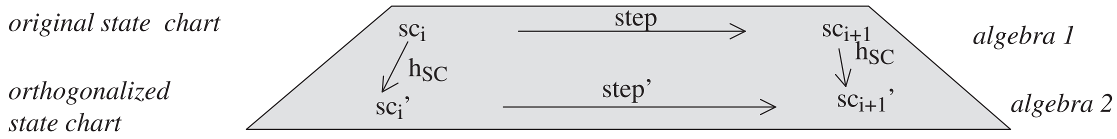

Fig. 4 : Interrelationship between original state chart and orthogonalized state chart

sponding state *in_Z* of the orthogonalized state chart and each state *Z* that is currently not entered onto the corresponding state *out_Z* of the orthogonalized state chart.

If the diagram given in Fig. 4 commutes then algebra 2 of the orthogonalized state chart is a homomorphic image of algebra 1 of the original state chart (Lang, 1993). In this case, we say that the orthogonalized state chart is *behaviorally equivalent* to the original state chart, i.e., has the same properties as the original state chart. For example, the activities are started at the same time whenever both state charts are executed under identical external input. The following theorem states the above formally:

Theorem 1

The mapping h_{SC}, which maps each system configuration of an arbitrary state chart *S* onto system configurations of the orthogonalized state chart *S'* is a homomorphism. That is, for each step *i*, $i \geq 0$ and for each system configuration sc_i of *S* the following holds:

$$h_{SC}\ (step\ (sc_i)) = step'\ (h_{SC}\ (sc_i)),$$

with *step* being the step operator of *S*, *step'* being the step operator of *S'*, and both step operators being defined by the operational semantics of state charts.

A detailed presentation of the formal model and the proof can be found in (Wodtke, 1997) and (Wodtke and Weikum, 1997).

6 Synchronization of Workflow Engines

In this section, we discuss how the workflow engines that execute partitions of a workflow are synchronized. More specifically, we investigate where the exchange of system configuration information has to take place, and which part of the system configuration needs to be communicated. The goal is to minimize synchronization costs in terms of the number of synchronization messages exchanged and in terms of their sizes, while guaranteeing a workflow execution equivalent to a non distributed one (i.e., equivalent to the execution of the orthogonalized state chart before partitioning). If the original state chart contains orthogonal components, it should be possible in a distributed execution to exploit parallelism between them. This has to be considered in the design of the synchronization mechanism. Our synchronization mechanism addresses only the exchange of data between partitions of the same workflow instance. We do not consider the concurrency control between different workflow instances of the same or different type; this would pose a completely different set of issues.

As defined in Section 4, the execution of state charts is performed in steps. A step maps the set of states currently reached by the participating partitions and the current context into the set of states after the step is performed and a new context, depending on the transitions that fire in the step and on the actions performed. The new system configuration is immediately available to determine all transitions and actions that make up the next step. For the distributed execution of a partitioned state chart, the straightforward solution to the synchronization problem is to always communicate the new system configuration of each partition to all other partitions after performing a step. The according messages are called *synchronization messages*. The next step can be performed by a partition after synchronization messages from all other partitions have been received, showing that all other partitions have also completed the previous step. We denote this scheme *strict synchronization*. With strict synchronization, each partition has perfect knowledge about the states reached in other partitions and the according context. In addition, all partitions perform a step at the same time. After the execution of a workflow is finished, all partitions have performed the same number of steps, just as in the non-distributed execution.

Definition 1: (Strict Synchronization)

Two partitions *S* and *S'* are *strictly synchronized* if they exchange their complete system configuration *sc* of *S* and *sc'* of *S'* before performing each step.

The execution of a partitioned state chart is *strictly synchronized* if all pairs of partitions are strictly synchronized.

It is obvious that strict synchronization imposes unnecessary overhead on the communication of partitions. Only a small fraction of the amount of data received by a partition is relevant for the next step. Even worse, waiting for all partitions to finish their current step before a partition can proceed with the next step limits the potential for exploiting parallelism. We now discuss how to reduce the number of synchronization messages and their sizes, compared to strict synchronization.

6.1 Reducing the Size of Synchronization Messages

We proceed in two steps. First, for each pair of partitions, it is determined which part of the system configuration is relevant for synchronizing them. Only the relevant information needs to be sent. Secondly, it is sufficient to send only those parts of the relevant system configuration that have changed during the last step.

A variable is denoted *relevant* for a partition if it is part of a condition in that partition, or read in an action in that partition. A variable is denoted *written* by a partition if it is modified in an action in that partition. We do not consider events here. Events in the specific sense of state charts are valid for a single step only. Therefore, they have to be immediately communicated to all partitions where they occur in an *E[C]/A* rule. They are not subject of further optimization. Note that according to our experience, events are rarely used in the specification of workflows. Our credit request example of Fig. 2 does not contain any events at all.

Information about entering or leaving states might also be relevant for other partitions. This is the case if *state conditions IN(s)* or *OUT(s)* occur in the condition part of the *E[C]/A* rules of a partition.

A static analysis of each partition allows us to derive all variables that are relevant for the partition. These are the only variables that need to be received in synchronization messages. The same is done for relevant state conditions. Note that we can not detect the exchange of data accessed by invoked applications if these applications do not notify the workflow engine about theses accesses. Such data can not be relevant for control flow, as it is not available to the workflow engine, even in the centralized case. In fact, we do not want the workflow management system to route the application data through the workflow engine for performance and possibly also security reasons.

We denote the relevant variables together with relevant state conditions *relevant system configuration* of a partition. The relevant system configuration of a partition reduced to those variables and state conditions that are written by another partition or belong to states of that partition is called *synchronization data* for this pair of partitions. This can be formally stated as follows:

Definition 2: (Relevant Variables and State Conditions, Relevant System Configuration, Written Variables)

A variable *v* or a state condition *IN(s)* or *OUT(s)* is *relevant* for a partition *S* if it occurs in a condition *C* or if it is read by an action *A* of an *E[C]/A* rule assigned to a transition in *S*.

The *relevant system configuration* of partition *S, RSC(S),* is defined as the set of all relevant variables and state conditions in *S*.

A variable *v* is *written* in a partition *S* if it is modified in the action part *A* of an *E[C]/A* rule assigned to a transition in *S*. Let *WSC(S)* denote the set of all written variables of partition *S* and all state conditions *IN(s)* or *OUT(s)* of all states *s* of partition *S*.

Definition 3: (Synchronization Data)

The *synchronization data SD(S,R)* from partition *S* to partition *R* is defined as follows:

$$SD(S,R) = RSC(R) \cap WSC(S)$$

Determining synchronization data can be done statically before the actual workflow execution. In order to send only updated data, we have to determine updates to synchronization data after each step that was actually performed. This can easily be done at runtime with almost no overhead. We thus obtain the following definition for an optimized strict synchronization, denoted *incremental synchronization*.

Definition 4: (Incremental Synchronization)

Two partitions *S* and *S'* are *incrementally synchronized* if they exchange their updated synchronization data *SD(S,S')* and *SD(S',S)* before performing each step. If there are no updated synchronization data after performing a step, an empty message is sent.

The execution of a partitioned state chart is *incrementally synchronized* if all of its partitions are incrementally synchronized.

It is easy to see that incremental synchronization is equivalent to strict synchronization in terms of the resulting workflow execution. As for all pairs of partitions, all updates to relevant variables are sent after each step, including information about the currently entered or non-entered states if relevant, the receiving partition has exactly the same system configuration as if strict synchronization were used. By sending empty synchronization messages in case there are no updates to synchronization data, it is guaranteed that steps are performed synchronously in the participating partitions.

6.2 Reducing the Number of Synchronization Messages

In the previous section, only the size of synchronization messages has been reduced. For *n* partitions, there are still *n(n–1)* synchronization messages that have to be exchanged after each step.

These messages implement a tight coupling of the execution of all partitions, as all partitions perform their steps synchronously. No partition is allowed to do the next step until all other partitions have finished the previous one. This limits potential parallelism between partitions. In this section, we derive a new synchronization scheme denoted *weak synchronization*. It is based on incremental synchronization but sends less messages and allows partitions to perform multiple steps without synchronizing themselves with other partitions. Our scheme is designed such that the synchronization of one pair of partitions is done independently of other pairs.

We start by giving a framework for the exchange of synchronization data. The framework is defined in terms of send points and receiving windows. Synchronization data is sent at send points and received inside of receiving windows. Send points and receiving windows can be automatically derived from the orthogonalized and partitioned state chart. We then discuss how the assignment of receiving windows to send points is derived. The assignment is crucial as it finally determines the execution semantics of weakly synchronized partitions. A correct assignment guarantees that the execution is equivalent to a non-distributed execution of the original state chart.

6.2.1 Receiving Windows and Send Points

The basic idea of weak synchronization is to send synchronization data only if the receiving partition makes use of it in the next step. This leads to a state dependent definition of relevant variables, relevant state conditions, and the relevant system configuration.

Definition 5: (Relevant Variables and Relevant State Conditions in a State, Relevant System Configuration in a State)

A variable *v* or a state condition *IN(t)* or *OUT(t)* is *relevant* for a partition *S* in a state *s* if it occurs in a condition *C* or if it is read by an action *A* of an *E[C]/A* rule assigned to a transition from state *s* in *S*.

The *relevant system configuration* of partition *S* in state *s*, *RSC(S,s),* is defined as the set of all relevant variables and state conditions for *S* in *s*.

Synchronization data is also state-dependent now.

Definition 6: (Synchronization Data)

The *synchronization data SD(S,R,s)* from partition *S* to partition *R* in state *s* of *R* is defined as follows:

$SD(S,R,s) = RSC(R,s) \cap WSC(S)$

Without sending synchronization messages after each step, the set of simultaneously entered states in different partitions need not be the same as in the non-distributed execution. After receiving synchronization data, a partition can process several subsequent steps until further synchronization data from another partition is required. The synchronization data must reach the receiver during the processing of these steps. If it is received before, the execution might be based on the wrong, i.e., 'future' synchronization data. If it is received too late, the execution might be based on outdated data, which is also wrong. We address this problem by defining *receiving windows*.

A receiving window determines the set of states where a receiving partition is ready to receive an update to the relevant system configuration, sent by the sending partition at a particular *send point*. In both the start and the finish state of a receiving window, there is synchronization data required from the sending partition. For the intermediate states of a receiving window, the opposite is true, i.e., their relevant system configuration does neither contain a variable written by the sending partition nor a state condition containing a state of the sending partition. Hence, the relevant system configuration of the receiving partition in the finish state of the receiving window can be received in one of the intermediate states or, at the latest, in the finish state of the receiving window. For successive receiving windows, the finish state of the first receiving window is the start state of the second receiving window. Initial states also qualify as start states of receiving windows.

Definition 7: (Receiving Window)

A *receiving window* $RW = (s,IS,s',S)$ of partition R with respect to partition S is defined by a start state s and a finish state s' in R, $s \neq s'$, connected through a set of transitions and an according set of intermediate states IS in R, $s,s' \notin IS$. In addition, the following must hold:

(1) There exists *synchronization data* $SD(S,R,s)$ from partition S to partition R at state s, or s is an initial state.

(2) There exists *synchronization data* $SD(S,R,s')$ from partition S to partition R at state s'.

(3) For all states s'' in IS, it holds: There exists no *synchronization data* $SD(S,R,s'')$ from partition S to partition R at state s''.

Send points are the counterparts of receiving windows. They define at which state a sending partition sends synchronization data.

Definition 8: (Send Point)

A *send point* $SP = (s,R)$ of partition S with respect to partition R is defined by a state s of partition S for which one of the following conditions holds:

(1) There exists at least one variable v relevant for Partition R such that v is modified by an action A of an *E[C]/A* rule assigned to a transition into state s in S (v can be updated immediately before reaching s), or

(2) state s is contained in a state condition in partition R.

For a complete specification of weak synchronization, we need an assignment of receiving windows to send points, i.e., we need a way to specify whether synchronization data sent at a send point has to be received in a synchronization window, or not.

Definition 9: (Assigning Receiving Windows to Send Points)

The set $RWS(SP)$, with $SP=(s,R)$, denotes the set of receiving windows of partition R *assigned* to send point SP.

We are now ready to define the notion of *weak synchronization,* based on the notion of receiving windows and send points.

Definition 10: (Weak Synchronization)

Two partitions S and S' are *weakly synchronized* if all of the following three conditions hold:

(1) S and S' send synchronization data to S' and S, resp., only at send points.

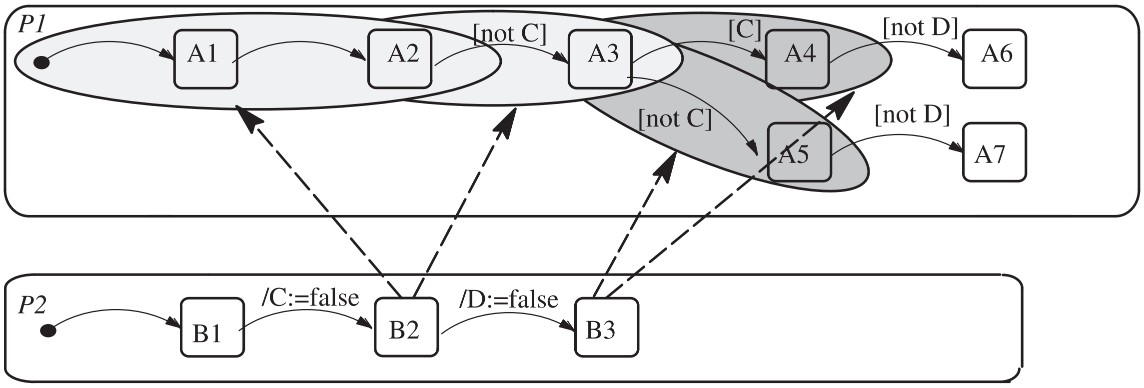

Fig. 5: Assignment of receiving windows to send points

(2) The part of the system configuration sent at a send point *(s,S')* or *(s',S)* contains all synchronization data from the sending to the receiving partition in state *s* or *s'*, resp, that were updated since they have last been send to *S'* or *S,* resp.

(3) Synchronization data sent at a send point is made part of the system configuration of the receiver if the receiver is an intermediate state or in the finish state of one of the receiving windows assigned to the send point.

(4) If a partition reaches the finish state of a receiving window without receiving synchronization data from at least one send point assigned to the receiving window, it does not perform the next step until it receives synchronization data from at least one send point assigned to it.

The execution of a partitioned state chart is *weakly synchronized* if all pairs of partitions are *weakly synchronized*.

Multiple receiving windows can be assigned to a single send point. This is the case if the part of the system configuration sent at a send point is relevant for the evaluation of more than a single condition or read by more than a single action in the receiving partition, and the according transitions belong to different states. These states might be entered successively by the receiving partition, or alternatively. As an example, consider Fig. 5. It shows two state charts representing two partitions *P1* and *P2*. Receiving windows are illustrated by ellipses. A receiving window consists of all states inside of an ellipsis. Assignments of receiving windows of *P1* to send points in *P2* are indicated by dotted arcs between the state representing the send point and the ellipses representing receiving window(s) assigned to the send point. The send point at state *B2* has two receiving windows: *(init, {A1}, A2, P2)* and *(A2, ∅, A3, P2)*. The start state of the first receiving window is the initial state of the state chart of *P1* (the default values of variables *C* and *D* are the synchronization data here). These receiving windows are entered successively, whereas the receiving windows *(A3, ∅, A4,P2)* and *(A3, ∅, A5,P2)* of the send point at state *B3* are entered alternatively .

A receiving window can also be assigned to multiple send points. This is the case if the actual send point depends on the execution of the workflow, as shown in Fig. 6. Depending on whether state *B2* or *B3* is entered, the update to variable *D* is sent at different send points. In the example, the send points are reached alternatively. Assigning a single receiving window to send points that are successively reached is also possible. In this case, in the finish state of the receiving window the receiving partition will only wait for the first send point to be reached. If synchronization data from the first send point is received, according to Def. 10, further waiting for subse-

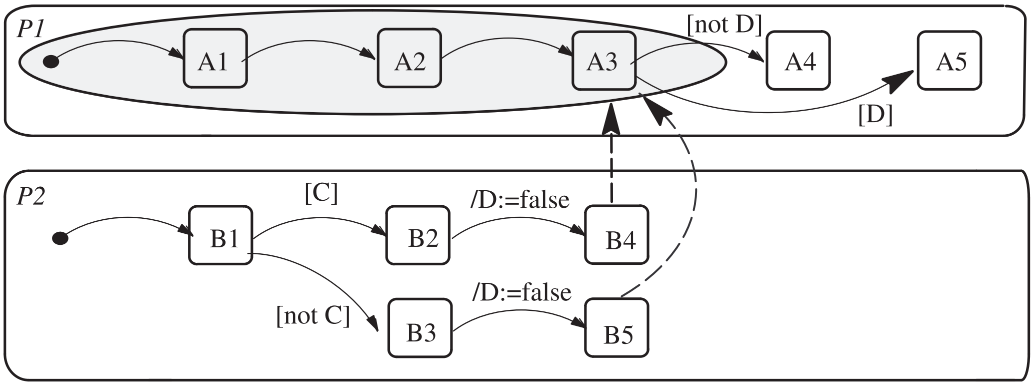

Fig. 6: Send points sharing a receiving window

quent send points will not occur. However, if subsequent send points are reached before the receiving partition leaves the synchronization window, the corresponding synchronization data is used to update the system configuration. In the example of Fig. 6, assume, the receiving window *(init, {A1, A2}, A3, P2)* in Fig. 6 is also assigned to the send point represented by the initial state of the state chart in partition *P2*. In this case, the execution in *P1* will not wait in state *A3* for the execution of the state chart in *P2* to reach state *B4* or *B5*, because the send point at the initial state of *P2* has already been passed. However, according to the state chart semantics there is no need to wait anyway, as *P1* and *P2* perform a step at the same time, and thus state *A3* in partition *P1* is not left before either state *B4* or state *B5* is entered in partition *P2*. Hence, in weak synchronization mode, the state chart semantics has to be guaranteed by a correct assignment of receiving windows to send points. Using the above assignment of the receiving window to two send points, this is not guaranteed as the execution of *P1* then depends on the time it takes *P2* to reach state *B4* or *B5*. If state *B4* or *B5* is reached after state *A3* is left in partition *P1*, the update of variable *D* in partition *P2* is not recognized in partition *P1*. We address the problem of determining a valid assignment of receiving windows to send points with respect to the state chart semantics in the next section.

6.2.2 Determining the Assignment of Receiving Windows to Send Points

The key problem for weak synchronization is how to determine the assignment of receiving windows to send points. We start by determining the *potential* receiving windows for each send point. This can be done by a static analysis of the partitioned state chart. Consider all pairs of send points and receiving windows. Whenever there is a non-empty intersection between the synchronization data to be sent at the send point and the relevant system configuration of the receiving partition at the finish state of the receiving window, the receiving window is a potential receiving window for the considered send point.

Note that for orthogonal components in the original state chart that do not have control flow dependencies, no send point belonging to one component has potential receiving windows belonging to the other. Hence, there is no need to exchange synchronization data. The method of weak synchronization allows to fully exploit parallelism here, as an execution equivalent to the execution of the original state chart is guaranteed without a strict synchronization of steps.

We continue by distinguishing two cases. In the first case, which we expect to be the application standard case, we consider all send points which have all their potential receiving windows in the same component of the original state chart (i.e., not in different orthogonal components). These send points and their assignment of receiving windows can be determined automatically. In the second, more involved and less application relevant case, at least one of the potential receiving windows of a send point belongs to a component in the original state chart that is orthogonal to the component of the send point. We will see that in this case, an assignment of synchronization windows to send points that results in an execution equivalent to the execution of the original state chart can not be derived automatically. A manual assignment by the workflow designer is necessary. If such an assignment is not provided, we have to resort to incremental synchronization which implements the original execution semantics of state charts.

We will now discuss the above two cases in detail.

1.) A send point and all its potential receiving windows belong to a single component of the original state chart

In this case, the assignment of receiving windows to send points is trivial:

For each send point, assign all its potential receiving windows to it.

This assignment ensures that the distributed execution is equivalent to the execution of the original state chart. Additional specifications by the workflow designer are not required.

Proof Sketch:

The proof sketch is based on the observation that there is no parallelism in the execution of partitions that belong to a single component in the original state chart. There, only a single state (including its substates) is entered at a time, and the orthogonalization algorithm as discussed in Section 5 ensures that this is also the case for the corresponding 'in_'–states in the orthogonalized state chart.

Assume, the distributed execution of a single component in the original state chart has reached a certain state, and up to this point, the execution was equivalent to a non-distributed one. Now assume, in a partition S involved in this execution a transition form state s to state s' takes place, whereas in the corresponding non-distributed execution, the transition is from state s to state s'' (For the sake of simplicity, we disregard the mapping of states in the original state chart to corresponding "in_" and "out_" states in the orthogonalized and partitioned state chart here). The system configuration of both executions in state s must be different. By Def. 5, the relevant system configuration in state s contains all variables and state conditions used in at least one condition or read by at least one action of the transitions from s to other states. By Def. *7,* s is the finish state of a receiving window. An erroneous system configuration used in state s is either caused by wrong synchronization data sent at a previous send point, or caused by using an outdated system configuration due to a send point that was not reached. The latter case is impossible as up to the time the finish state of the receiving window is reached, the execution was equivalent to a non-distributed execution. Receiving wrong synchronization data is also impossible. This could only be the case for synchronization data from a send point that was entered too early, and the corresponding updates to the system configuration should not be considered when leaving state s. This is in contradiction to our assumption that up to entering state s, the execution was equivalent to a non-distributed execution with only a single state entered at a time.

2.) A send point and the finish state of at least one of its potential receiving windows belong to states in orthogonal components of the original state chart

In this case, we are faced with control flow dependencies between orthogonal components in the original state chart, and a correct assignment of receiving windows to send points can not be derived automatically. Only if the structure of the orthogonal components is simple, i.e., they contain no loops, no complex branches etc., a manual assignment seems feasible. For example, this is the case in Fig. 6, where it can be easily seen that the step semantics of state charts enforces that either states *A3* and *B4* or states *A3* and *B5* are entered at the same time. We feel that these cases are the only ones where control flow dependencies between orthogonal components can be reasonably used anyway, as this requires knowledge about concurrently entered states during the execution of orthogonal components. There is a further reason why control flow dependencies between orthogonal components should be rare. As orthogonal components implement parallelism, well designed orthogonal components will be independent of each other. Otherwise, the potential parallelism is limited by the need to exchange data at distinguished time points.

The best way to solve the problem of control flow dependencies between orthogonal components is to avoid them in the original specification of workflows. Otherwise, we propose to use incremental synchronization when executing such components. Only in obviously simple cases a manual assignment of receiving windows to send points should be used. We allow switching between weak synchronization and incremental synchronization dynamically during execution. Whenever two partitions enter orthogonal components of the original state chart, it is checked whether a manual assignment of send points to receiving windows for the considered orthogonal components is provided. If not, the system automatically uses incremental synchronization until the orthogonal components are left. This allows a workflow to be executed without any explicit assignment, resulting in an execution according to the original state chart semantics.

7 Communication Costs

In this section, we will quantify the resulting communication costs by determining the number as well as the size of the necessary synchronization messages. Our consideration discriminates between strict, incremental and weak synchronization.

To quantify the communication costs for the execution of a particular workflow, we need to analyze the corresponding partitioned state chart. Let *n* denote the number of partitions, and *step* the number steps executed during workflow execution. Note that *step* is execution dependent. The number and size of synchronization messages for strict, incremental and weak synchronization can then be derived as follows:

(1) **Strict Synchronization**

After each step, each partition sends messages to all other $n-1$ partitions. The total number of synchronization messages send during the execution of the workflow execution can be computed by:

$$number\_of\_msgs = n(n-1) * step$$

The size of synchronization messages sent from partition *i* can be computed by counting the number of state conditions and variables in partition *i*. Let z_i be the number of state conditions of partition *i*, v_i be the number of variables of partition *i*. Therefore, the message size is:

$$size\_of\_msg_i = z_i + v_i$$

Note that the *size_of_msg* gives the number of data items rather than the number of bytes.

(2) **Incremental Synchronization**

Here, only the part of the system configuration which has changed during the last step is sent. The number of messages is not reduced since empty messages have to be sent unless updates occurred in the last step. Let z_{ij} represent the number of state changes and let v_{ij} represent the number of changed variables in partition i during step j. Therefore, the message size is:

$$size\_of\_msg_{ij} = z_{ij} + v_{ij}$$

(3) **Weak Synchronization**

The number of messages sent in weak synchronization mode depends only on the number of send points passed during the execution of a partition. Let s_i denote this number for partition i. Therefore, the number of messages is:

$$number\_of\_msgs = \sum_{i=1}^{n} s_i$$

Let z_{ijk} denote the number of state changes and let v_{ijk} denote the number of changed variables in partition i which are sent from partition i to partition k at send point j. The size of a synchronization message sent in send point j from partition i to partition k is:

$$size\_of\_msg_{ijk} = z_{ijk} + v_{ijk}$$

We will now determine the communication costs for the partitioned example workflow of Fig. 3 for the strict, incremental and weak synchronization. The static analysis of the partitioned specification with partitions P1, P2, P3 and P4 shows that P1 consists of 2 states, has 4 local variables (including one state condition for each "*in_*"state, and two variables "*_OK*" and "*_NOK*" for the outcome of each activity) and reads no external variables. For partitions P2 (P3 and P4 in parentheses) the according numbers are 4 (4, 4) states, 6 (4, 2) local variables, 3 (7, 4) external variables. As the actual number of steps performed in each partition is execution dependent, we consider, as a worst case scenario, the longest execution path which is 3 steps long. When the workflow is executed in strict synchronization mode, the total number of synchronization messages is 36 with 144 synchronization data items. The incremental synchronization method also requires 36 messages but with only 14 data items.

The analysis of the weak synchronization method will now be investigated in more detail. Send points, receiving windows and their assignments for partitions *P1, P2*, and *P3* are shown in tables 1, 2 and 3 below. Partition *P4* never sends synchronization data to other partitions. The first column represents the send points, the second represents the receiving windows. We assume that the initial values of variables and state conditions of each partition need not be communicated, and corresponding send points are not given in the table. Send points are assigned to all receiving windows in the same row. The given assignment results in an execution semantics equivalent to the semantics of the original state chart. The third column of each table shows the synchronization data that is sent at a send point. The total number of messages sent with weak synchronization equals the total number of rows in the tables, as in the example, each send point is passed only once. For the same reason, the number of data items sent can be derived by counting the synchronization data items in the third column of the tables. We obtain a number of 8 synchronization messages with a total number of 14 data items.

Send Points of P1	Receiving Windows of P2, P3, P4	Synchronization Data
(in_ENCR_S, P2)	(init, Ø, out_CCW_S, P1), (init, Ø, out_RSK_S, P1)	in_ENCR_S, ENCR_OK, amount
(in_ENCR_S, P3)	(init, Ø, out_DEC_S, P1)	in_ENCR_S, ENCR_OK, amount
(in_ENCR_S, P4)	(init, Ø, in_INIT, P1), (init, Ø, out_ERR_S, P1)	ENCR_NOK

Table 1: Send Points of Partition *P1* and Receiving Windows

Send Points of P2	Receiving Windows of P3, P4	Synchronization Data
(in_RSK_S, P3)	(init, Ø, out_DEC_S, P2)	RSK_OK, in_RSK_S
(in_CCW_S, P3)	(init, Ø, out_DEC_S, P2)	CCW_OK, in_CCW_S
(in_RSK_S, P4)	(init, Ø, in_INIT, P2), (init, Ø, out_ERR_S, P2)	RSK_NOK
(in_CCW_S, P4)	(init, Ø, in_INIT, P2), (init, Ø, out_ERR_S, P2)	CCW_NOK

Table 2: Send Points of Partition *P2* and Receiving Windows

Send Points of P3	Receiving Windows of P4	Synchronization Data
(in_DEC_S, P4)	(init, Ø, in_INIT, P3), (init, Ø, out_ERR_S, P3)	DEC_NOK

Table 3: Send Points of Partition *P3* and Receiving Windows

In the example, all start states of all receiving windows are the initial states of the corresponding partition. This is no longer the case if orthogonal components assigned to the same partition are merged as briefly discussed in Section 5. For the sake of simplicity, we have not considered this in our example. Details can be found in (Wodtke, 1997).

8 Conclusions

Enterprise-wide workflows will soon play a crucial role in managing business processes of large institutions. Providing rigorous foundations for the design and implementation of distributed workflow management systems has therefore become a major research challenge. The verification of mission critical properties of workflows must be possible. This requires a rigorous formal semantics of workflow specifications. We have approached this problem by using state and activity charts as our specification method, as this provides a formally founded basis to reason about execution semantics, for example, by means of symbolic model checking. Since most workflows are specified in a centralized manner without considering a distributed execution, a further problem to solve is the partitioning of the workflow specification, such that the semantics of the original specification is preserved. We have presented a provably correct partitioning method for state and activity charts, thus enabling a distributed execution according to the original semantics.

A synchronization scheme has been developed which guarantees the correct synchronization between the workflow engines executing the partitions of a workflow. The scheme is further improved in terms of the number and size of synchronization messages that have to be exchanged between the partitions. Further optimizations are possible by taking into account knowledge about typical executions. If some synchronization data is rarely used in a receiving partition because in most executions, the corresponding receiving window is not entered, it is more efficient to explicitly request the data if necessary instead of sending it in each execution. This is a subject of future work.

The distributed execution of enterprise-wide workflows should also consider the issue of fault tolerance. The state configuration including the current workflow context has to be maintained persistently in order to allow the continuation of workflows after site failures. Updating the system configuration after each step of the execution and sending the corresponding synchronization messages to other partitions has to be performed as an atomic unit. An *exactly once* semantics for delivering synchronization messages is required, in presence of communication or site failures. In our prototype, this is implemented by using a TP Monitor for providing reliable messages queues and transactional services, and a database system for maintaining the local system configuration.

We consider our analysis of communication costs for different synchronization schemes to be a first approach for reasoning about performance of enterprise-wide workflows. Future steps in this direction should include additional resources such as disks and memory. The final goal is to come up with analytical results for determining the configuration of an enterprise-wide workflow management system and predicting its performance and reliability under a given load of concurrently executing workflows.

References

Alonso, G., Agrawal, D., El Abbadi, A., Kamath, M., Günthör, R., and Mohan, C. (1996). Advanced Transaction Models in Workflow system configurations, IEEE International Conference on Data Engineering, New Orleans

Alonso, G., Mohan, C., Günthör, R., Agrawal, D., El Abbadi, A., and Kamath, M. (1995). Exotica/FMQM: A Persistent Message–Based Architecture for Distributed Workflow Management, Proc. IFIP WG8.1 Working Conference on Information Systems for Decentralized Organizations, Trondheim.

Barbara, D., Mehrotra, S., and Rusinkiewicz, M. (1996). INCAs: Managing Dynamic Workflows in Distributed Environments, Journal of Database Management, Special Issue on Multidatabasees, 7 (1)

Bernstein, A., Dellacros, C., Malone, T.W., and Quimby, J. (1995). Software Tools for a Process Handbook, Bulletin of the Technical Committee on Data Engineering, Vol. 18, No. 1

Bernstein, P.A. and Newcomer, E. (1997). Principles of Transaction Processing, Morgan Kaufmann Publishers

Blakeley, B., Harris, H. and Lewis, J.R.T. (1995). Messaging and Queuing Using the MQI: Concepts and Analysis, Design and Development, McGraw–Hill

Das, S., Kochut, K., Miller, J., Sheth, A., and Worah, D. (1997). ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for *METEOR*$_2$, Technical Report UGA–CS–TR–97–001, University of Georgia

Dayal, U., Garcia-Molina, H., Hsu, M., Kao, B., and Shan, M.-C. (1993). Third Generation TP Monitors. A Database Challenge, ACM SIGMOD Conference

Dayal, U., Hsu, M., and Ladin, R. (1991). A Transactional Model for Long–Running Activities, VLDB Conference

Ellis, C.A., and Nutt, G.J. (1993). Modeling and Enactment of Workflow Systems, Invited Paper, 14th International Conference on Application and Theory of Petri Nets

Forst, A., Kühn, E., and Bukhres, O. (1995). General Purpose Work Flow Languages, Distributed and Parallel Databases, Vol. 3, Nr. 2

Gawlick, D. (1994). High Performance TP-Monitors - Do We Still Need to Develop Them?, IEEE Bulletin of the Technical Committee on Data Engineering, 17 (1): 16 - 21

Georgakopoulos, D. and Hornick, M.F. (1994). A Framework for Enforceable Specification of Extended Transaction Models and Transactional Workflows, International Journal of Intelligent and Cooperative Information Systems, 3(3)

Georgakopoulos, D., Hornick, M., and Sheth, A. (1995). An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure, Distributed and Parallel Databases, 3(2)

Gray, J. and Reuter, A. (1993). Transaction Processing: Concepts and Techniques, Morgan Kaufmann

Harel, D. (1987a). State Charts: A Visual Formalism for Complex Systems, Science of Computer Programming Vol.8, 1987, pp. 231-274

Harel, D. (1987b). On the Formal Semantics of State Charts. Symposium on Logics in Computer Science, Ithaca, New York

Harel, D. (1988). On Visual Formalisms, Communications of the ACM, 31(5)

Harel, D. et al. (1990). STATEMATE: A Working Environment for the Development of Complex Reactive Systems, IEEE Transactions on Software Engineering, 16(4)

Harel, D. and Gery, E. (1997). Executable Object Modeling with Statecharts, IEEE Computer, 30(7)

Harel, D. and Naamad, A. (1995). The STATEMATE Semantics of State Charts, Technical Report, i-Logix Inc.

Helbig, J. and Kelb, P. (1994). An OBDD–Representation of state charts, Proc. European Design and Test Conference

IBM Corp. (1994). FlowMark Rel. 1, Programming Guide and Modelling Guide

IONA Technologies Ltd. (1995). Orbix Programming Guide

Jablonski, S. (1994). MOBILE: A Modular Workflow Model and Architecture, Proc. of the 4th Int. Working Conference on Dynamic Modelling and Information Systems, Nordwijkerhout

Jablonski, S. and Bussler, C. (1996). Workflow Management, Modeling Concepts, Architecture, and Implementation, International Thomson Computer Press

Kamath, M., Alonso, G., Günthör, R. and Mohan, C. (1996). Providing High Availability in Very Large Workflow Management Systems, International Conference on Extending Database Technology, Avignon

Kappel, G., Lang, P., Rausch–Schott, S. and Retschitzegger, W. (1995). Workflow Management Based on Objects, Rules, and Roles, Bulletin of the IEEE Technical Committee on Data Engineering, Vol. 18, No. 1

Kappel, G. and Schrefl, M. (1991). Object/Behavior Diagrams, IEEE International Conference on Data Engineering, Kobe

Lang, S. (1993). Algebra, Third Edition, Addison-Wesley

McMillan, K.L. (1993). Symbolic Model Checking, Kluwer

Mohan, C. (1994). Tutorial: Advanced Transaction Models - Survey and Critique, Presented at ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota

Mowbray, T. J. and Zahavi**,** R. (1995). The Essential Corba**,** John Wiley and Sons

Oberweis, A., Scherrer, G. and Stucky, W. (1994). INCOME/STAR: Methodology and Tools for the Development of Distributed Information Systems, Information Systems Vol.19 No.8

OMG, CORBAservices (1995a). Common Object Services Specification, Technical Report, Object Management Group

OMG, (1995b). The Common Object Request Broker: Architecture and Specification, Rev. 2.0, Technical Report, Object Management Group

Primatesta, F. (1994). TUXEDO, An Open Approach to OLTP. Prentice Hall

Reuter, A. and Schwenkreis, F. (1995). ConTracts – A Low–Level Mechanism for Building General–Purpose Workflow Management Systems, IEEE Data Engineering Bulletin Vol.18 No.1

Rusinkiewicz, M. and Sheth, A. (1994). Specification and Execution of Transactional Workflows, in: W. Kim (Editor), Modern Database Systems: The Object Model, Interoperability, and Beyond, ACM Press

Schwenkreis, F. (1993). APRICOTS – A Prototype Implementation of a ConTract System – Management of the Control Flow and the Communication System, 12th Symposium on Reliable Distributed Systems

Sheth, A. (ed.) (1996). Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems, Athens, GA., May 1996, http://lsdis.cs.uga.edu/activities/NSF-workflow/proc_cover.html

Sheth, A., Kochut, K., Miller, J., Worah, D., Das, S., and Lin, C. (1996). Supporting State–wide Immunization Tracking using Multi–Paradigm Workflow Technology, VLDB Conference

TUXEDO System 5 (1994). System Documentation. Novell, 1994

Vossen, G. (ed.) and Becker, J. (ed.) (1996). Busines Process Modelling and Workflow Managment, Models, Methods, and Tools, International Thomson Publishing, Bonn (in German)

Wächter, H. and Reuter, A. (1992). The ConTract Model, in: A.K. Elmagarmid (Editor), Database Transaction Models for Advanced Applications, Morgan Kaufmann

Weissenfels, J., Wodtke, D., Weikum, G., and Kotz Dittrich, A. (1996). The MENTOR Architecture for Enterprise-wide Workflow Management. In: [Sheth 96]

Widom, J., Ceri, S., and Dayal, U. (Editors) (1995). Active Database Systems, Morgan Kaufmann

Wodtke, D. (1997). Foundation and Architecture of Workflow Management Systems, Ph.D.Thesis, DISDBIS, Infix Verlag, 1997

Wodtke, D. and Weikum, G. (1997). A Formal Foundation for Distributed Workflow Execution Based on State Charts, In: Proceedings of the International Conference on Database Theory, Springer LNCS 1186, Delphi, Greece

Wodtke, D., Weissenfels, J., Weikum, G., and Kotz Dittrich, A. (1996). The MENTOR Project: Steps Towards Enterprise-Wide Workflow Management. IEEE International Conference on Data Engineering, New Orleans

Workflow Management Coalition (1995). http://www.aiai.ed.ac.uk/WfMC/