MODELLING CONCURRENT PROCESS COORDINATION IN WORKFLOW SPECIFICATIONS

Alistair P. Barros
Distributed Systems Technology Centre
The University of Queensland
Brisbane Qld 4072
*Australia**
abarros@dstc.edu.au

Arthur H.M. ter Hofstede
Cooperative Information Systems Research Centre
Queensland University of Technology
GPO Box 2434, Brisbane Qld 4001
Australia
arthur@icis.qut.edu.au

Keywords: workflow, conceptual modelling, formalisation, semantics

Abstract The provision of business services, in practice, exhibits the processes of several organisations, run in parallel for their local stakeholds, with points of synchronisation serving to coordinate the differently executing parts. Current workflow tools restrict business processing coordination to *single* workflows. This means different processing parts are unnaturally coupled into the same unit of modularity, thereby restricting workflow resusability. In this paper, we demonstrate three extensions which serve to coordinate the execution of concurrently executing and interdependent workflows: the adaption of synchronous and asynchronous messaging for inter-workflow communication, the abort of inter-related execution paths for coordinated exception handling and the exclusive, i.e. atomic, execution of nested processes in the context concurrently executing processes. These extensions are applied at the conceptual level

*Part of this work has been supported by CITEC, a business unit of the Queensland Government's Department of Public Works and Housing.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35500-9_30

E. D. Falkenberg et al. (eds.), *Information System Concepts: An Integrated Discipline Emerging*

of specification, illustrated through highlights from an industrial case study and assigned a formal semantics.

1. INTRODUCTION

Through workflow management, a number of concepts emerging from the last twenty years of dynamic and process modelling research, are exploited to capture models of business processes such that those models are directly interpreted and executed. Constructs like task sequencing, choice, iteration and parallelisation, combined with mechanisms of data flow and organisational resource models, are now common place in a large number of (production) workflow tools (see workflow technology survey in (Makey, 1996)).

Currently, workflow tools restrict the coordination of business processes to single workflows. While this seems adequate for organisations in fairly early phases of workflow use or where their business processing fits into relatively isolated and single workflows, in practice, single workflow coordination is restrictive. This is because the provision of business services such as insurance claims, property transfers and legal court matters, involves not just customers and service providers, but a whole spectrum of stakeholders (typically external agencies), whose processing occurs in parallel with points of synchronisation "threading" together the workflow.

Of course, constructs like multiple task triggers (e.g. AND splits) and synchronisers (e.g. AND and OR joins) allow the processing of different parties to run in parallel and to be synchronised "downstream". Furthermore, the nested feature of workflow tasks allows entire workflows to be encapsulated into tasks, allowing modular specifications to be built up and reused. Under this construction, however, the workflow is still relatively sequential and one part serves to coordinate the workflow as a whole. For service provision involving truly autonomous and separate parts, for instance domains utilising autonomous agents in E-commerce services, the workflow components may be observed to be inflexibly constructed and quasi-autonomous. What is required is the interoperability of truly autonomous workflows where synchronisation can occur across the different workflows, as opposed to occurring in a single (coordinator) workflow.

Concurrent interaction in conceptual modelling has been has been well-researched when one considers, say, the work done with Petri nets (Aalst, 1996). Petri nets have provided an *expressive* and a *formal* base for technique development, and much of the adaptations of higher order Petri nets have yielded relatively *comprehensive* and *suitable* constructs

for commercial domains, e.g. (Ramackers, 1994; Kappel and Schrefl, 1990). On the other hand, two related areas of omission are exception handling (Eder and Liebhart, 1998) and rollback recovery (Alonso et al., 1996) both of which are vital for workflow suitability (Barros and Hofstede, 1998). Currently in techniques, the ramifications of concurrent processing for exception handling are not well understood.

In this paper, we demonstrate how the coordination of a number of workflow components can captured at the conceptual level. In particular, we adapt *messaging* which allows the different components to communicate in an asynchronous and synchronous manner. Our proposal focussed on process control aspects. As shown in (Barros and Hofstede, 1997) the integration with a data modelling technique allows the typing of message tokens which captures the document/data structure they represent. Moreover, data model query languages like RIDL and LISA-D (Hofstede et al., 1993) can be used in detailed task specifications of the workflow to manipulate the passed information - thereby preserving full conceptualisation of the specification.

To complete the workflow treatment, we illustrate how exception handling can be incorporated in the parallel context. For this, we introduce *aborts* which are applied to a related set of processes running concurrently and we indicate how such a construct fits into rollback recovery.

Through the combination of the messaging and parallel aborts, some quite powerful workflow processing situations can be captured - like sending out requests for information to a number of stakeholders, continuing processing, receiving any responses and if any exception arises, aborting processing.

The introduction of such powerful constructs in the face of concurrent workflow execution, particularly the presence of multiple instances of the same type of process and process decompositions, can complicate correct workflow execution. We show how a formal semantics can be assigned for the constructs using a formal language from the Process Algebra family, Algebra of Communicating Processes (ACP) (Baeten and Weijland, 1990) which has had significant applications (Baeten, 1990).

The organisation of this paper is as follows. In section 2, we describe the modelling extensions and their contribution to concurrent workflow coordination. Examples are drawn from an industrial case study, the tenure administration of Road Closures documented in (Barros et al., 1997). In section 3, the formal semantics is presented based on a full formalisation defined in (Barros and Hofstede, 1997). In section 4, the paper is concluded and future research directions are described.

2. CONCEPTUAL WORKFLOW MODELLING: A PRACTICAL ILLUSTRATION

In this section, we describe the workflow extensions (section 2.2) of abstract messaging and aborts to concurrent workflow coordination, within a practical setting. In particular, the proposed constructs are used to illustrate the business processing "patterns" of: remote information polling, deferred exception handling and remote object validation.

The extensions are applied to a conceptual process modelling technique Task Structures (Hofstede and Nieuwland, 1993). Task Structures provide a number of constructs which are characteristic of process-centric workflow specifications (section 2.1). Importantly, Task Structures are integrated with a multi-level, data modelling technique CDM (Creasy and Proper, 1996). This allows for a (schema) typing of information manipulated by the workflow at the conceptual level. The manipulation capturing *detailed* process specifications, is described in a conceptual specification language, LISA-D (Hofstede et al., 1993).

2.1. BASIC TASK STRUCTURES

As depicted in Figure 1, the coordinative mechanism in Task Structures is a trigger (solid arrow) and the chief end of triggering is the execution of a process, also referred to as a task (rounded box). A process may be defined in terms of other processes known as its subprocesses. The execution of a process decomposition follows the triggering structure, starting with its initially executing elements (indicated by bent arrows at the top of the elements). An initially executing element needs to be indicated because iteration, i.e. a trigger looping back to a preceeding element, introduces uncertainty as to what this is.

Decisions (circles) model moments of choice, where each outgoing trigger is associated with a condition. A decision's outcome, i.e. executed trigger, is the one which is truth satisfied. Like guarded commands of (Dijkstra, 1975), decision conditions may overlap, allowing nondeterministic choices to be modelled. Some decisions terminate, meaning that one of their outcomes results in no further processing, thereby terminating the execution path.

Synchronisers (triangle) allow a junction of incoming and/or outgoing triggers. The former models the synchronisation of execution paths (running in parallel), prior to further processing continuing. The latter models a set of execution paths to be started in parallel (this would typically be used in relation to a decision, as multiple output triggers of a task also capture parallel initiation of other task objects).

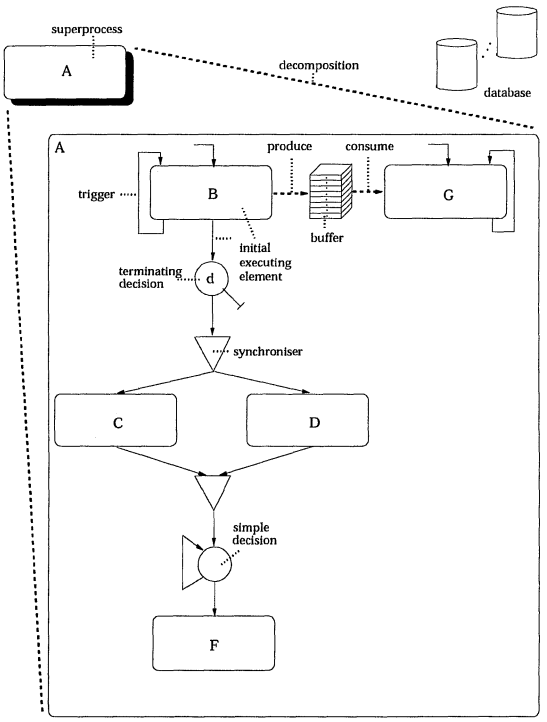

Figure 1 Basic Task Structure concepts

In line with multi-database workflow environments, we allow a set of databases to be associated with a workflow model. Database schemas are captured through CDM data models and detailed specifications of processes (preconditions, actions and postconditions) and decisions (conditions) are expressed through LISA-D. In LISA-D statements, therefore, the databases that they apply to should be qualified. Through abstraction, specialisation and generalisation hierarchies, different types of multi-database configurations can be captured. As we will see through messaging extensions below, this modelling flexibility can also be applied to capture document flow and manipulation.

In addition to the synchroniser, Task Structures allow concurrent parts of a workflow to be coordinated through buffers. By using buffers, e.g. in-trays, pigeon holes and e-mail boxes, different though data dependent processes can run independently, where data produced by one process, is consumed by another, via a buffer. Two special operations, *produce* and *consume*, are available for buffer manipulation and buffers have

a *protocol* which defines their order, e.g. a First-In, Last-Out (FILO) or an arbitrarily ordered queue. When a process produces data into a buffer, the value is "appended" to the buffer as part of its termination. When a process consumes from a buffer, the value of buffer's "top" location is removed and allocated to the process as part of its initiation.

To assist in the expressive power of specifications, variables may be defined in decompositions and accessed by processes and decisions in their decomposition hierarchies. Values produced in, or consumed from, buffers are stored in variables (having the same name as buffers).

2.2. EXTENSIONS TO TASK STRUCTURES

Abstract messaging Our application of messaging at the workflow level, namely *abstract* messaging, permits communication between processes running in separate decomposition contexts. In contrast to buffering, messaging involves direct process to process communication.

Like in traditional messaging systems, messaging at the abstract level is characterised by two modes. A *synchronous* mode, also available in object-oriented techniques, e.g. UML (Fowler, 1997), involves a suspension of execution related to a messaging dependency on some other process. An *asynchronous* mode involves no such suspension of execution. As illustrated in Figure 2, these modes are used in three fundamental configurations to capture different message processing requirements.

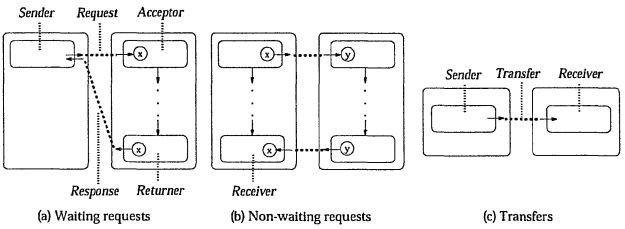

Figure 2 Messaging configurations

Through a *waiting request* configuration (a), a message is passed to a remote source, however further execution of the process is suspended until a return message is received. Hence, we say the configuration embodies a request. The required construct for the initiating request is clearly a synchronous *sender*.

In another decomposition, the request is received by an *acceptor*. As apparent, receiving is inherently synchronous, i.e. the process is blocked from execution until receipt of the message. To return a *response* for the request, we also require a *returner* which simply performs an asyn-

chronous send. For the construction of responses, we allow any general processing to occur prior to the return. This means that the acceptor and the returner should be *correlated*, to ensure that their relationship is preserved and not mixed up with other message constructs in the decomposition.

The configuration of *non-waiting requests* (b) arises through liberating the suspension of execution associated with the request. In this case, the request is issued through an asynchronous sender. A *receiver* which is correlated with this sender is therefore required to receive the request. We describe this pair as a *send-receive* pair, and for reasons which we have just described, a corresponding accept-return pair contributes to the full configuration.

Finally, the configuration of (c) involves one-way communication only, between an (asynchronous) sender and a receiver. Since no other dependency occurs in the communication, we describe this configuration as a *transfer*.

To illustrate the application of messaging for workflows, consider Figures 3 and 4. Together, these illustrate a scenario which we refer to as *remote information polling*. One part of the process model in Figure 3 involves the determination of information from a remote site using a waiting request; processing from the sender's side, only, is illustrated. A message label (Parcel Info) is used to indicate the request and transfer communication between processes. Thus, to specify the receiving side in this example, the same message label would be have to be assigned to an acceptor. The other part of Figure 3 involves the transfer of messages (Notify Stakeholders) based on information gathered from the preceeding search.

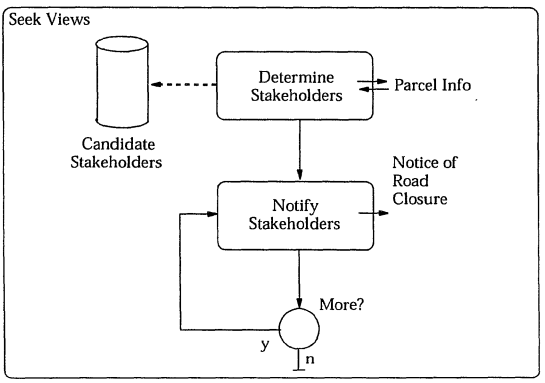

Figure 3 Information polling example: request sending

Figure 4 depicts the receiving of messages, anticipated as a result of the requests which have been issued. A receiver accepts incoming correspondences which are each split up and examined. Note, the cyclic trigger for message receiving, since multiple responses should arrive. In effect, each receipt generates a separate and possibly parallel execution path for subsequent processing. Also note, the parallel processing between the splitting and examination processes via a buffer.

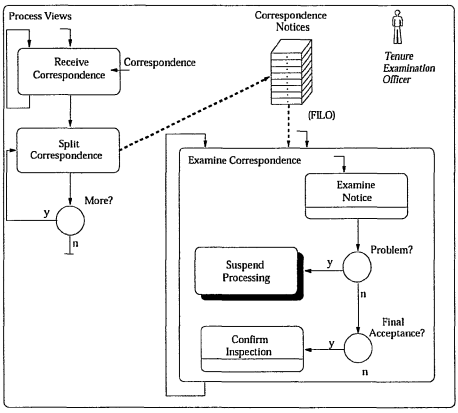

Figure 4 Information polling example: response receiving

Aborts A second extension arises from the existence of parallel (and semantically related execution paths) in a workflow. We have just seen an example of this situation in Figure 4, where process instances are generated continuously through cyclic triggers. An important question is: when does such a process (decomposition) terminate? In general, while constructs are available to initiate multiple execution paths, e.g. through multiple initially executing elements and multiple triggers from a process or synchroniser, no associated construct is available which issues their termination. This can lead to a practice where such a design is avoided or termination is left to programming detail, thereby compromising the conceptualisation of the design.

For this type of situation, we introduce a particular type of trigger, an *abort*, whose function is to terminate related execution paths. To control which parts of a workflow are terminated, we restrict the effect of aborts to decompositions. In general, when an abort is executed, all the processing in a decomposition is terminated and an *abort handler* is executed.

In Figure 5, the use of aborts is combined with messaging to illustrate an efficiency in business processing which we refer to as *deferred exception handling*. In the example, under normal circumstances, customers are required to confirm acceptance and make payments before further processing can continue. An efficiency is introduced by allowing the processing to go ahead on the understanding that customers typically make these provisions, and if the provisions are not made, processing is aborted. Two examples of such deferred exception handling are evident in the figure.

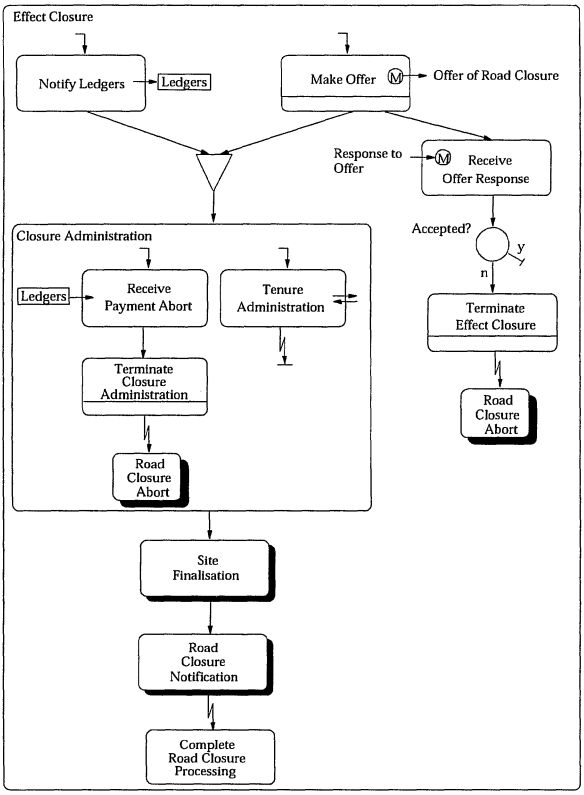

Figure 5 Deferred exception handling example using aborts

Firstly, Make Offer issues an Offer of Road Closure and triggers two processes, one which allows the main processing to continue ahead and the other which waits for the Response to (the) Offer. If the offer is not Accepted?, the whole of the Effect Closure is aborted. The abort (zig-zag arrow) is activated by Terminate Effect Closure, and as a result, the

abort handler, Road Closure Abort, is run. This can be used for post-abort processing, e.g. notifying stakeholders about the abort.

Secondly, within Closure Administration the new tenure information is prepared remotely, with another process used to Receive Payment Abort; this might have occurred because payment was not made for whatever reason. There is an assumption that if the Tenure Administration completes prior to any payment abort, then remaining processing continues. Hence, an abort is triggered after Tenure Administration to ensure that no part of the decomposition "hangs" around after its execution. For the same sort of reason, we include an abort of Effect Closure at the end.

Another application of aborts and messaging is a commonly encountered scenario which we refer to as *remote object validation.* Here, validation checks through remote searches, are run in parallel, where the determination of some outcomes might not warrant further processing to continue. Figure 6 illustrates this by incorporating messaging and aborts into a composite decision construct.

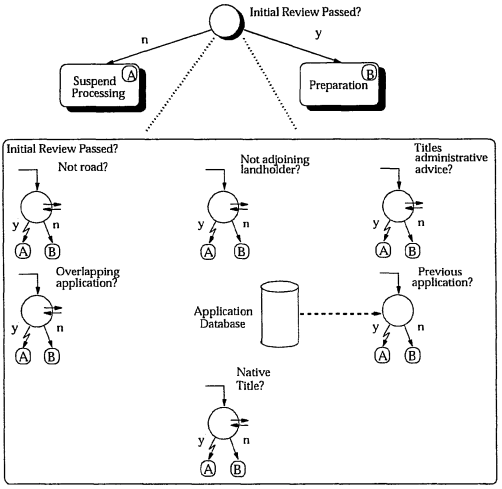

Figure 6 Object validation example: parallel searches in complex decisions

The example involves the validation of a Road Closures Application object to determine whether further processing is warranted. The whole validation is modelled as a complex decision whereby its two outcomes are determined from its internal structure. Each check is modelled as a simple decision running in parallel (hence the initial element symbol). Decisions with synchronous messaging capture the remote searches; note

messaging on a decision is a short-hand for a messaging process followed by a decision.

For the requirement that any one decision failing is sufficient to determine a complex decision outcome, we use the abort construct. The triggering of an abort results in the complex decision being terminated (i.e. all its execution paths), and the relevant external outcome (A) being executed. For non-aborting outcomes, we do not want multiple executions of the related complex decision outcome (B), for each of the internal, exiting outcomes, to be executed. Rather, each such internal outcome has the effect of an exit, without terminating the complex decision. The exception is, of course, the last outcome which should indeed trigger the external outcome.

Figure 7 illustrates a modification to the example where prioritisation in the decision processing is introduced. This is useful since object validation in workflow applications can involve large and expensive remote searches which somehow should be prioritised.

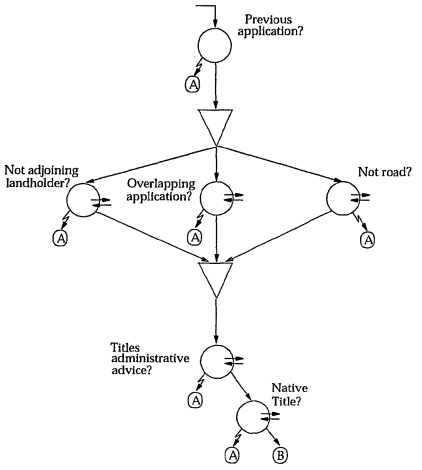

Figure 7 Object validation example: prioritised searches in complex decisions

Combining buffering, messaging and aborts Yet another illustration of buffering, messaging and aborts for concurrent workflow processing arises in the example of a Change of Address service, illustrated in Figures 8 and 9. The intention of this workflow is to provide people on the Web with a flexible mechanism of notifying particular parties (e.g. banks, insurance companies, civil service departments, doctors etc.) of their address changes.

The first part, illustrated in Figure 8 shows the specification of parties by a person through one or more invocations of Change of Address entry/update. Each invocation generates an execution path which determines whether personal authentication is required from a trusted agent and accordingly aborting the service or effecting notifications (based on the supplied parties). Effecting notifications involves a further parallelism - for each notification sent, a corresponding receiver waits for the response from the party.

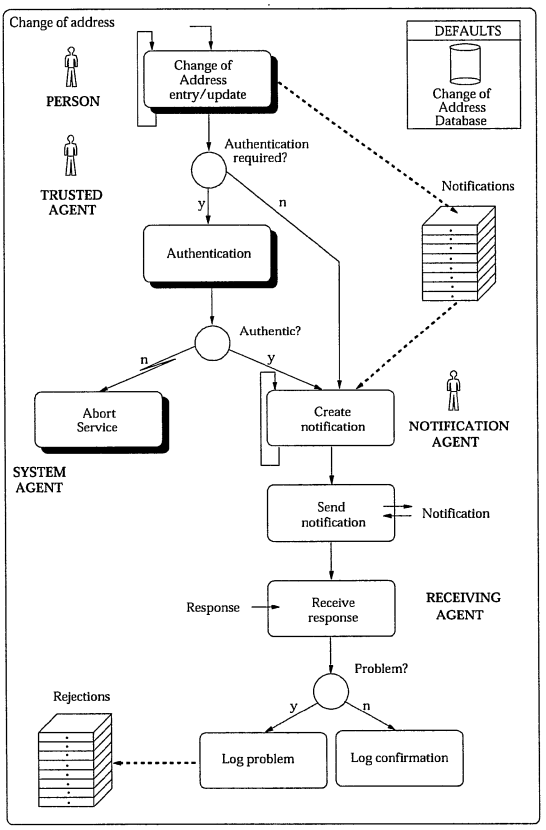

Figure 8 Another information polling example - part 1

The second part, illustrated in Figure 9 shows the specification allowing people to deal with rejections in parallel and converting them into notification, proper. The third part, also in the same figure, caters for service termination. This has to be done explicitly since the flexibility of the workflow activates new invocations of all its parts (specified through

cyclic triggers on the initial processes). Such flexibility caters for people using the same service over a "long" duration (which is typically required when people recollect how they have to notify for their address changes).

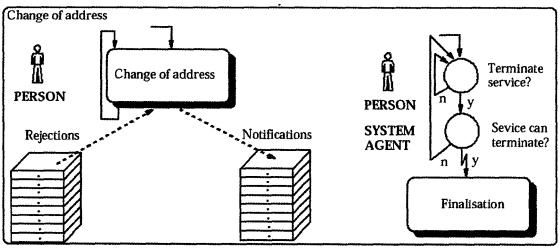

Figure 9 Another information polling example - part 2 and 3

3. FORMAL SEMANTICS

In this section, we demonstrate how the advanced workflow concepts discussed in the previous section can be assigned a formal semantics. Naturally, in the context of concurrent and distributed systems, such as workflows, it is imperative that their specifications are unambiguous and allow for formal reasoning. Hence, such specifications need a formal semantics, which can be achieved through a mapping to a formal system. In this case we have chosen the Process Algebra as our formal domain. Although the name Process Algebra suggests a single algebra to describe processes, it actually refers to a whole family of algebras based on the same principles. Traditionally, only the family member used is presented. As an algebraic theory, Process Algebra belongs to the same family as CSP (Hoare, 1978) and CCS (Milner, 1989) Our translation is based on a particular Process Algebra, the Algebra of Communicating Processes with the empty action (which we will abbreviate to ACP_ε). An in-depth description may be found in (Baeten and Weijland, 1990) with a useful set of applications described in (Baeten, 1990).

It should be noted right from the start though, that this section will not provide the complete formal semantics of the presented workflow specification language (for this, the reader is referred to (Barros and Hofstede, 1997)). Rather, focus is on the formalization of the process control aspects of the three workflow extensions presented in the previous section including three illustrating (abstract) examples, showing how the formal semantics can be manipulated. Data environments, e.g. databases, buffers and messages, are not considered in the formalization. Apart from its importance for e.g. equivalence considerations, the

formalization serves to demonstrate that the presented workflow specification language is more than just another drawing convention.

3.1. PROCESS MODEL TRANSLATION

In assigning a formal semantics to workflow specifications, basically each concept will correspond to a process variable with a corresponding ACP_ε equation. Typically, for a workflow object x the corresponding process variable is denoted E_x. For each aborting processes z we also need to introduce a process variable T_z, as such objects need to be initiated differently. As we do not aim at a comprehensive treatment of the formalization, the syntax will be discussed "on the fly" and only as far as needed for a proper understanding of the semantics.

Executing a task t means executing its body, given by its name $\mathsf{Name}(t)$, followed by triggering in parallel its output task objects:

$$E_t = \mathsf{Name}(t) \cdot \mathbf{trigrest}(t)$$

The abbreviation $\mathbf{trigrest}(t)$ provides the possible triggering paths following the execution of t. As previously mentioned, the translation of triggering and aborting process elements is distinguished. Synchronizers have to be treated differently too. Every process element x which is to trigger synchroniser s, starts an atomic action $\sigma_{x,s}$ after termination. A synchroniser is started when all such atomic actions, related to its input process elements, communicate, noting that communication is not necessarily binary since more than two process elements may be input to a synchroniser. The details of the formal semantics of synchronizers will not be discussed further in the context of this paper, we refer the reader to (Hofstede and Nieuwland, 1993).

$$\mathbf{trigrest}(t) = \left(\Big\|_{\substack{x \in \mathcal{X} \setminus (\mathcal{Y} \cup \mathcal{Z}), \\ \mathsf{Trig}(t,x)}} E_x \Big\|_{\substack{z \in \mathcal{Z} \setminus \mathcal{Y}, \\ \mathsf{Trig}(t,z)}} T_z \Big\|_{\substack{y \in \mathcal{Y}, \\ \mathsf{Trig}(t,y)}} \sigma_{t,y} \right)$$

In the above definition, $\mathcal{X}$ is the set of all process elements, $\mathcal{Y}$ is the set of all synchronizers, and $\mathcal{Z}$ is the set of all aborting processes. The relation Trig defines the trigger relations between workflow objects. If t does not invoke one of the three possible types of execution paths, the relevant merge will be defined over the empty set. This exception is dealt with by defining the merge over the empty set to be the empty action ε as this is the neutral element for parallel composition.

Tasks that have names that do not have an associated decomposition correspond to atomic actions. For names that have an associated decomposition the corresponding equation states that its execution corresponds

to starting all its initial items in parallel. Naturally, the usual distinction between aborting and nonaborting processes needs to be made, but we also need to add the brackets for the aborting context (this will be explained in the next subsection):

$$V = \ll \Bigg(\mathop{\Big\|}_{\substack{\mathsf{Init}(x) = V \\ x \notin \mathcal{Z}}} E_x \quad \mathop{\Big\|}_{\substack{\mathsf{Init}(x) = V \\ x \in \mathcal{Z}}} T_x \Bigg) \gg$$

The partial function Init applied to a process element x yields, if defined, the name of the supertask of x. This function is only defined for those process elements that are initial item.

The corresponding equation for a decision d has to reflect the choice for the possible workflow objects involved:

$$E_d = \sum_{\substack{d \,\mathsf{Trig}\, y, \\ y \in \mathcal{Y}}} \sigma_{d,s} + \sum_{\substack{d \,\mathsf{Trig}\, z, \\ z \in \mathcal{Z}}} T_z + \sum_{\substack{d \,\mathsf{Trig}\, x, \\ x \notin \mathcal{Y} \cup \mathcal{Z}}} E_x + \zeta_d$$

where $\zeta_d = \varepsilon$ if d is a terminating decision and $\zeta_d = \delta$ otherwise.

3.1.1 Translation of abort processes. An entire decomposition's execution can be aborted when a process element, specifically a process or decision, in the decomposition is executed. The only processing which follows is an abort handler associated with the aborting element. We saw two significant applications of this, namely messaging interrupts (Figure 5) and complex decision processing (Figure 6).

To provide a general treatment for aborts, we introduce the abort operator which is axiomatised below.

$$\ll \varepsilon \gg \;=\; \varepsilon \tag{I1}$$

$$\ll a \cdot X \gg \;=\; a \cdot \ll X \gg \text{ if } a \notin Q \tag{I2}$$

$$\ll X + Y \gg \;=\; \ll X \gg + \ll Y \gg \tag{I3}$$

$$\ll \langle X \rangle \looparrowright \langle Y \rangle \cdot Z \gg \;=\; XY \tag{I4}$$

For the translation of aborting process elements, we introduce in I4, an abort expression of the form $\langle X \rangle \looparrowright \langle Y \rangle$, where X denotes the aborting element and Y denotes the abort handler. Since aborting elements and abort handlers may be entire decompositions, X and Y are delimited. The effect of the abort operator is achieved by ignoring everything after the abort handler in an abort context. (The abort context is also removed since the abort expression is removed). The set of all abort expressions is given by Q, and such expressions should be treated as atomic actions

in terms of the ACP_ε axioms. Through axiom I2, we can see that atomic actions can be shifted outside abort contexts provided they are not abort expressions.

Since an abort expression already contains the equation denotation for an aborting element, its translation requires an alternate equation denotation to represent the whole equation. This circumvents circularity in the translation. So for an aborting process $z \in \mathcal{Z}$, we have:

$$T_z = \langle E_z \rangle \looparrowright \langle E_{\mathsf{Abt}(z)} \rangle$$

where $\mathsf{Abt}(z)$ yields the abort handler of z.

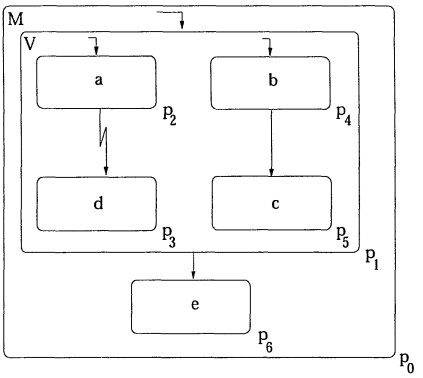

Figure 10 Aborting process example

Example 3.1

In the process model of Figure 10, a decomposition V contains two subprocesses, p_2 and p_4, started in parallel, one of which, p_2, aborts processing, triggering the abort handler p_3 in the decomposition.

The previous translations applied to the process model yield the following equations:

$$\begin{array}{lcllcl} E_{p_0} & = & M & E_{p_2} & = & a \\ M & = & E_{p_1} & E_{p_3} & = & d \\ E_{p_1} & = & V \cdot E_{p_6} & E_{p_4} & = & b \cdot E_{p_5} \\ E_{p_6} & = & e & E_{p_5} & = & c \\ V & = & \ll T_{p_2} \parallel E_{p_4} \gg & T_{p_2} & = & \langle E_{p_2} \rangle \looparrowright \langle E_{p_3} \rangle \end{array}$$

The abort processing axioms can then be applied to reduce the equation for the process model:

$$E_{p_0} = \ll E_{p_2} \parallel E_{p_4} \gg \cdot e$$

$$
\begin{aligned}
&= \ll \langle a \rangle \looparrowright \langle d \rangle \parallel bc \gg \cdot e \\
&= \ll \langle a \rangle \looparrowright \langle d \rangle \lfloor\!\lfloor bc + bc \rfloor\!\rfloor \langle a \rangle \looparrowright \langle d \rangle \gg \cdot e \\
&= (\ll \langle a \rangle \looparrowright \langle d \rangle\, bc \gg + \ll b(c \parallel \langle a \rangle \looparrowright \langle d \rangle) \gg) \cdot e \\
&= (ad + b(\ll c\, \langle a \rangle \looparrowright \langle d \rangle \gg + \ll \langle a \rangle \looparrowright \langle d \rangle \cdot c \gg)) \cdot e \\
&= (ad + b(cad + ad))e \\
&= ade + b(cade + ade)
\end{aligned}
$$

This shows that after executing the abort task named a the tasks named b and c are not executed anymore.

3.1.2 Translation of messaging. All up, we have described three essential types of messaging configurations which may be identified in specifications involving direct (i.e. unbuffered) inter-process messaging. These are *waiting requests* involving a synchronous sender and an accept-return pair (two way communication), *non-waiting requests* involving send-receive and accept-return pairs (two way communication), and *transfers* involving an asynchronous sender and a receiver (one way communication). For communication, processes require the same messaging scopes, and inherent in this is the compatibility of parameter passing. Moreover, accept-return and send-receive pairs require a correlation.

As, from a formal perspective, non-waiting requests and transfers are simpler than waiting requests, just representing variations on a similar theme, only the (simplified) formalization of waiting requests is presented here. The approach is simplified in that 1) individualization of processes (and hence of their messages) is (virtually) not considered, and 2) information passing and how this affects the global state of the system is ignored (again, the reader is referred to (Barros and Hofstede, 1997) for a full treatment of these issues). Our strategy has been adapted from the approach undertaken in a formalisation of messaging for Jacobson's Objectory (Hubbers and Hofstede, 1997).

Synchronous senders cater for situations like *requests* for information or the provision of notifications, where *responses* or acknowledgements (to the sender) are required. Characteristic of this type of communication is the suspension of execution which follows the message sending, until a return message is received.

Intuitively, the translation of synchronous senders $w \in \mathcal{W}$ contains a send and a receive part. The send part requires a messaging scope $\mathsf{MesgScope}(w)$ "within" which, the message occurs, together with parameters $\mathsf{SendPar}(w)$ containing queries from which any associated information can be constructed. The wait part requires parameters $\mathsf{RecPar}(w)$

which will contain any returned data.

$$\begin{aligned} E_w \quad = \quad & \mathtt{send}(\mathsf{MesgScope}(w); \mathsf{SendPar}(w)) \cdot \\ & \mathtt{wait}(\mathsf{MesgScope}(w); \mathsf{RecPar}(w)) \cdot \\ & \mathbf{trigrest}(w) \end{aligned}$$

We allow a number of message parameters for generality, e.g. for multiple documents or the passing of process control data accompanying a document.

As it is possible that multiple instances of the same task are active at the same time, one needs to be able to distinguish between messages sent by these different instances. To this end, a rename operator ρ^{α} (inspired by the work in (Vaandrager, 1990), where it was used for the formal semantics of the parallel object oriented language POOL) is used that individualises send related statements. This leads to expressions such as:

$$\rho^{\alpha}(\mathtt{send}(c; q_1, \ldots, q_n) \cdot X) \quad = \quad \mathtt{send}(c; q_1, \ldots, q_n)@\alpha \cdot \rho^{\alpha}(X)$$

In this equation, α represents a process id, which is incorporated in the send action. This allows other actions to recognize which particular process instance has sent the message. In the context of this paper though, we will not go into further details in relation to process individualization.

Of course, a request cannot be materialised unless the sent message is received by an acceptor a. Its translation involves an `accept` action with its parameters corresponding to the sent message:

$$E_a = \mathtt{accept}(\mathsf{MesgScope}(a); \mathsf{RecPar}(a))$$

A `request` (pending) is generated when individualised `send` and `accept` actions communicate:

$$\begin{aligned} & \mathtt{send}(c; q_1, \ldots, q_n)@\alpha \mid \mathtt{accept}(c; s_1, \ldots, s_n)@\beta = \\ & \mathtt{request}(\langle \alpha, \beta, \rangle; q_1{:}\, s_1, \ldots, q_n{:}\, s_n) \end{aligned}$$

Effectively, the sent and received parameters are associated within a specific messaging context denoted by the pair $\langle \alpha, \beta \rangle$. This enables messaging configuration construction, where α identifies the sender of the request and β identifies the acceptor. The communication function γ restricts `request` actions generated to those which involve compatible message scopes c, i.e. both `send` and `accept` actions have the same messaging scope c. Final materialisation of a request occurs when a state operator passes through the `request` action.

Corresponding to an acceptor is a returner r, which is instrumental in providing a response to requests. It has the following translation:

$$E_r = \mathtt{return}(\mathsf{MesgScope}(\mathsf{Acceptor}(r)); \mathsf{SendPar}(r))$$

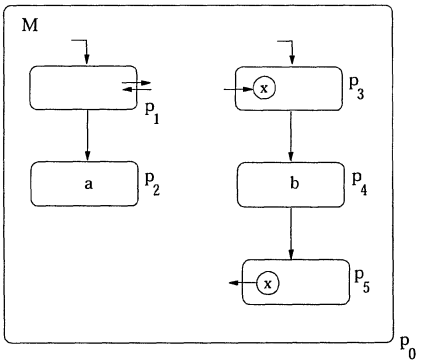

Figure 11 Example of a waiting request configuration

where Acceptor(r) yields the corresponding acceptor of r. A `response` is generated when a `return` communicates with a `wait`:

$$\texttt{return}(c; t_1, \ldots, t_m)@\beta \mid \texttt{wait}(c; r_1, \ldots, r_m)@\alpha = \texttt{response}(\langle \alpha, \beta \rangle; r_1 : t_1, \ldots, r_m : t_m)$$

The state operator restricts `response` actions to those where the `return` and `wait` are part of the same messaging configuration.

Finally, it should be remarked that as communications actions are used, the encapsulation operator needs to be applied. Hence, the corresponding equation for the main task becomes:

$$E_{p_0} = \preccurlyeq \partial_H(\mathsf{Name}(E_{p_0})) \succcurlyeq$$

where H represents the set containing all communication actions.

Example 3.2

The process model of Figure 11 depicts a waiting request configuration. The synchronous sender is p_1 and the corresponding accept-return pair consists of p_3 and p_5, with p_3 the acceptor and p_5 the returner. For communication to proceed, the messaging scope is defined as $\mathsf{MesgScope}(p_1) = \mathsf{MesgScope}(p_3) = m$ *and we assume no information passing.*

Ignoring abort and system contexts (as no aborts or monitors occur), the previous translations applied to the process model yield the following equations:

$$\begin{array}{lcllcl} E_{p_0} & = & M & M & = & \partial_H(E_{p_1} \parallel E_{p_3}) \\ E_{p_1} & = & \texttt{send}(m) \cdot \texttt{wait}(m) \cdot E_{p_2} & E_{p_4} & = & b \cdot E_{p_5} \\ E_{p_2} & = & a & E_{p_5} & = & \texttt{return}(m) \\ E_{p_3} & = & \texttt{accept}(m) \cdot E_{p_4} & & & \end{array}$$

The reduction for E_{p_0} is:

$$
\begin{aligned}
E_{p_0} &= \partial_H(E_{p_1} \parallel E_{p_3}) \\
&= \partial_H(\texttt{send}(m) \cdot \texttt{wait}(m) \cdot E_{p_2} \parallel \texttt{accept}(m) \cdot E_{p_4}) \\
&= \partial_H((\texttt{send}(m) \cdot \texttt{wait}(m) \cdot a \mid \texttt{accept}(m) \cdot b \cdot \texttt{return}(m))) \\
&= \partial_H(\texttt{send}(m) \mid \texttt{accept}(m))(\texttt{wait}(m) \cdot a \parallel b \cdot \texttt{return}(m)) \\
&= \texttt{request}(m) \cdot \partial_H(b \cdot \texttt{return}(m) \between \texttt{wait}(m) \cdot a) \\
&= \texttt{request}(m) \cdot b \cdot \partial_H(\texttt{return}(m) \parallel \texttt{wait}(m) \cdot a) \\
&= \texttt{request}(m) \cdot b \cdot \partial_H((\texttt{return}(m) \mid \texttt{wait}(m)) \cdot a) \\
&= \texttt{request}(m) \cdot b \cdot \texttt{response}(m) \cdot a
\end{aligned}
$$

4. CONCLUSION

In this paper, we argued that more sophisticated forms of inter-process communication are necessary for increased workflow automation. In particular, two extensions which preserve the parallelisation of execution paths were proposed: asynchronous and synchronous messaging across execution paths and aborts of related execution paths. Illustrations from an industrial case study showed that such requirements exist in practice despite the shortcomings of existing WFMS products. By combining these advanced workflow constructs with classical workflow constructs it is possible to capture complex but common themes of business processing.

To precisely impart the ramifications for workflow execution, we showed how a formal semantics can be assigned. In the short term, we see that it is viable to simulate increasingly complex aspects of business processing. In the longer term, specific workflow specification languages, among others as supported by current workflow tools, can be targeted and augmented with such features. The formal semantics guides such extensions.

As the formal semantics was defined in terms of Process Algebra, formal theory developed for Process Algebra, e.g. in the context of process equivalence (bisimulation), can be exploited when reasoning about workflow specifications. Further research will focus, among others, on the formalization of other needed concepts in workflow specifications, such as temporal constraints, actor resource models, and compensation.

References

Aalst, W. v. d. (1996). Three Good reasons for Using a Petri-net-based Workflow Management System. In Navathe, S. and Wakayama, T., editors, *Proceedings of the International Working Conference on In-*

formation and Process Integration in Enterprises (IPIC'96), pages 179–201, Cambridge, Massachusetts.

Alonso, G., Agrawal, D., El Abadi, E., Kammath, M., Günthör, R., and Mohan, C. (1996). Advanced transaction models in workflow concepts. In *Proceedings of the 12th International Conference on Data Engineering*, New Orleans, Lousiana.

Baeten, J., editor (1990). *Applications of Process Algebra.* Cambridge University Press, Cambridge, United Kingdom.

Baeten, J. and Weijland, W. (1990). *Process Algebra.* Cambridge University Press, Cambridge, United Kingdom.

Barros, A. and Hofstede, A. t. (1997). Formal Semantics of Coordinative Workflow Specifications. Technical Report 420, Department of Computer Science & Electrical Engineering, University of Queensland, Brisbane, Australia.

Barros, A. and Hofstede, A. t. (1998). Towards the construction of workflow-suitable conceptual modelling techniques. *Information Systems J.*, 8(4):313–337.

Barros, A., Hofstede, A. t., and Proper, H. (1997). Towards Real-Scale Business Transaction Workflow Modelling. In Olivé, A. and Pastor, J., editors, *Proceedings of the Ninth International Conference CAiSE'97 on Advanced Information Systems Engineering*, volume 1250 of *Lecture Notes in Computer Science*, pages 437–450, Barcelona, Spain. Springer Verlag.

Creasy, P. and Proper, H. (1996). A Generic Model for 3-Dimensional Conceptual Modelling. *Data & Knowledge Engineering*, 20(2):119–162.

Dijkstra, E. (1975). Guarded Commands, Nondeterminacy, and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457.

Eder, J. and Liebhart, W. (1998). Contributions to Exception Handling in Workflow Management. In Bukhres, O., Eder, J., and Salza, S., editors, *Proceedings EDBT Workshop on Workflow Management Systems*, pages 3–10, Valencia, Spain.

Fowler, M. (1997). *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, Reading, Massachusetts.

Hoare, C. (1978). Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677.

Hofstede, A. t. and Nieuwland, E. (1993). Task structure semantics through process algebra. *Software Engineering Journal*, 8(1):14–20.

Hofstede, A. t., Proper, H., and Weide, T. v. d. (1993). Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523.

Hubbers, J. and Hofstede, A. t. (1997). Formalization of Communication and Behaviour in Object-Oriented Analysis. *Data & Knowledge Engineering*, 23(2):147–184.

Kappel, G. and Schrefl, M. (1990). Using an object-oriented diagram technique for the design of information systems. In Sol, H. and van Hee, K., editors, *Proceedings of the International Working Conference on Dynamic Modelling of Information Systems*, Noordwijkerhout, The Netherlands.

Makey, P., editor (1996). *Workflow: Integrating the Enterprise.* Report of the Butler Group.

Milner, R. (1989). *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey.

Ramackers, G. (1994). *Integrated Object Modelling, an Executable Specification Framework for Business Analysis and Information System Design.* PhD thesis, University of Leiden, Leiden, The Netherlands.

Vaandrager, F. (1990). Process Algebra Semantics of POOL. In Baeten, J., editor, *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*, pages 173–236. Cambridge University Press, Cambridge, United Kingdom.