Formal Semantics of Coordinative Workflow Specifications*

A.P. Barros[1] and A.H.M. ter Hofstede[2]

[1] Department of Computer Science and Electrical Engineering,
The University of Queensland,
Brisbane Qld 4072, Australia,
e-mail barros@cs.uq.oz.au

[2] Cooperative Information Systems Research Centre,
Queensland University of Technology,
GPO Box 2434, Brisbane Qld 4001, Australia,
e-mail arthur@icis.qut.edu.ay

Keywords: Conceptual Modelling, Transactional Workflows,
Business Process Re-engineering, Formal Semantics, Process Algebra

Abstract

To advance the coordinative business processing cognition of conceptual workflow specifications, several fundamental extensions of execution dependency, over and above process to process triggering, are recognised. These include: information passing through buffers and messaging, abort handling, and the application of decomposition for decision processing. The addition of these concepts, however, can complicate the overall meaning of a workflow's execution, given the highly concurrent processing nature of workflow domains. To demonstrate such a workflow's *formal semantics*, we apply these extensions to a conceptual workflow specification technique, and we translate its execution constructs into Process Algebra equations.

1 Introduction

In recent times, workflow technology [Mak96],[Kob97] has been deployed to provide coordinative and collaborative support for business processing. Without such support, organisations require *detailed* quality management procedures so that human and computerised tasks combine to deliver business objectives. Towards a greater automation of this end, workflow management systems (WFMS) enable the specification, scheduling and execution of processes within

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

a workflow, be they manual or computerised, from different applications, databases and other types of information resources.

Business processing is diverse and it is well recognised that there is not one type of workflow, but several types [McC92]. In the widely cited classification of [GHS95], production and administrative workflows require a greater *coordination* of tasks, each typically the operational responsibility of an individual, where a precise execution structure is involved. The processing of insurance claims, bank loans and purchase requisitions are indicative examples. At the other extreme, ad-hoc workflows, cater for a *collaboration* in group-based activities, where the tasks and their execution dependencies are not as easily mechanised. The preparation of documents for group-related goals, e.g. for business submissions (tenders) and dissemination (forums), compels the need for this type of workflow, and in its fullest form, for computer supported cooperative work (CSCW) [WW93, Rod91].

Given this diversity, it is not surprising that a large number of workflow specification techniques and languages, embodying different paradigms, have been proposed. In a *process-centric* paradigm, as adopted in the most widely used WFMS product, FlowMark (IBM) [LR94], tasks form the focal points of workflow model cognition with execution constraints expressed both within tasks, e.g. pre-and post-conditions, and across tasks, typically control flows. In sum, an execution order involving sequence, iteration, choice, parallelism and synchronisation captures workflow coordination. A *document-centric* paradigm, adopted in Lotus Notes (IBM) and LinkWorks (DEC) [Lin95] routes documents between actors. A *state-centric* paradigm, adopted in InConcert (XSoft) [InC97], objectifies processes where workflow coordination is implicit in the process state lifecycles. A *speech-centric* paradigm permits collaborative aspects of workflows to be captured through speech-acts between actors [FL80]. A pioneer of this approach, Action Workflow (Action Technologies) [MWFF92], propagates a workflow through a series of performer-requester acts, where in contrast to the other paradigms, the communication intention behind a task - preparation, negotiation, performance and negotiation - serves to structure the workflow.

A crucial step for the implementation of a workflow, as in traditional IS development, is its *conceptual* design. At the conceptual level [Gri82], an essential understanding of a workflow is required to be imparted, independent of implementation detail. The benefit of validating conceptual specifications and detecting errors, prior to the much greater cost of doing so at the implementation level [Dav90], is nowadays accepted without question. Indeed, in recognition that workflow specifications can be large and complex, accommodating the requirements of many stakeholders, support for conceptual design and validation through *enactment*, is regarded as pivotal in the mosaic of WFMS functionality. This is now endorsed by the Workflow Management Coalition[1] through the process definition standard (in Interface 1).

Yet the commercial reality remains, that the quality of workflow conceptual modelling is remarkably weak [Moh96, CCPP95]. In our view, a major part of the problem is the partial cognition of business processing embodied in WFMS products, which allows only a partial workflow conceptualisation. Consequently, a dependence on implementation-level constructs is required to capture more fully, and validate, workflow requirements. This can be seen as an incarnation of the *waterfall*, prevalent in IS specifications [KG94]. Consider the following observations.

[1] Refer to http://www.aiai.ed.ac.uk/WfMC/index.html for more details.

Firstly, document flow is integral to business process execution - documents are received by, transiently stored in, passed around, and sent out from organisations, and this influences the execution order of business processes. Like triggering, document flow - or *messaging*, to use a more generic notion - is a form of process invocation. In FlowMark, document flow amounts to variable passing (through data connectors), where variables are defined in an implementation-level, scripting language. In Lotus Notes, which is currently being integrated with FlowMark, and LinkWorks, redundant information is required in programming variables, as documents cannot be directly manipulated; they are external (image) files which are referenced through pointers. Moreover, document-centric tools typically coflate business processing to single-document workflows. While InConcert does cater for multiple document workflows, its state-centricity makes it difficult to validate the global triggering structure of the workflow. This is not made any easier without the support of scenarios [RBP+91] which have proven invaluable in object-oriented analysis. In general, the synchronous and asynchronous modes of messaging available in general purpose conceptual modelling techniques, notably Rationale's Universal Modelling Language[2], are strikingly absent.

Secondly, workflows are never expected to execute from start to finish in a problem-free fashion. Rather, exceptions arise which warrant a partial or complete backout of a workflow, e.g. a customer requests termination of a business service or an external supplier is no longer able to provide certain parts for an order. Through an adaption of *transactional* characteristics for workflows (see e.g. [RS94] for general discussion), a rollback recovery of a workflow is possible. This involves running the reverse of the execution order, using rollbacks or compensations in place of uncommitted and committed tasks respectively. In a well-integrated environment, event-reactive workflow specifications at the conceptual level [CCPP95] can be used to drive exception handling mechanisms like rollback recovery through interface with WFMS service enactment facilities. However, as apparent in FlowMark, rollback recovery only addresses part of workflow termination. This is because certain workflow constructs can lead to the concurrent execution of different parts of the same workflow, and the termination of concurrent parts, we observe, is left open in currently deployed exception handling strategies.

Thirdly, the provision of decision (choice) constructs in workflow specifications reflects more of a programming than business behaviour. In practice, business decisions can be complicated involving a triggering structure of subdecisions, with access to data from remote resources and databases needed to evaluate decisions rules. Searches involving large numbers of decisions are typical in a validation of applications for "everyday" business services, e.g. in insurance claims and bank loans. Abstractions extended to decisions, in our view, can be used to explicitly capture *business* decision processing, and with it, to further improve the modularisation of workflow specifications.

Clearly, the inclusion of messaging in a triggering structure, abort handling and decision abstractions significantly improve the business cognition of workflows and therefore warrant a full conceptualisation. Any such lacking compromises a workflow's product-independent validation and verification. At the same time, the amalgamation of a complex set of concepts into a technique can compound the *meaning* of a technique. As we have remarked, a workflow's execution is highly concurrent having not just different parts of a workflow executing in parallel, but having multiple instances of the same workflow type and multiple instances of the

[2]Refer to http://www.rational.com/uml for more details.

same types of workflow elements across and within workflow instances. Not only are databases mutable by nature, but multiple instances of the same message types are bound to occur. Being able to identify different instances of workflow elements and their processing states is important to obtain the correct execution.

The purpose of this paper is to demonstrate the *formal semantics* for conceptual workflow specifications using the aforementioned extensions. Rather than claim a "silver bullet" which nowadays is disregarded for specification techniques [BS87, Bro87, ML83], our efforts are aimed at coordinative workflows, i.e. the combined concerns of administrative and production workflows. In [BHP97a], we have synthesised the suitability of techniques for such workflow domains in a set of general principles. Given their formal foundation and expressive power, Petri Nets, e.g. [Pet81, Rei85], by and large, have been used to develop a formal semantics for dynamic modelling techniques involving the communication of parallel processes. Despite higher order Petri Net adaptions, e.g. [Gen87, Jen91] however, such techniques remain cumbersome and unwieldy for even basic applications. Our starting point, instead, has been to extend a technique which is simple and characteristic enough of those deployed in process-centric workflow specifications, Task Structures [HN93].

A major advantage of Task Structures is its integration with a data modelling technique - i.e. an Object-Role model (ORM) variant - in the method Hydra [Hof93]. This makes it possible to use schema definitions to type data populations in databases and messages. This together with the provision of a highly-expressive, specification language LISA-D [HPW93] preserves the conceptualisation of workflow specifications, down to detailed specifications. Another major advantage, is the assignment of a formal semantics of Task Structures; in its case using the Process Algebra of [BW90]. The result of our extensions to Task Structures is placed in a method which we have called Aquino [BHP97b].

The organisation of this paper is as follows. In section 2, we informally introduce the process modelling component of Aquino. In section 3, a formal syntax is provided. It serves the paper's goal of a formal semantics assignment, and so the syntax is not claimed to be complete. In section 4, the formal semantics is defined by mapping process modelling constructs into Process Algebra equations. In section 5, the paper is concluded, recapping on the major technical issues and the future research that we propose to undertake. Appendix A contains the non-standard mathematic notation used in the paper. Appendix B contains the graphical notation for process modelling.

2 Informal introduction of Aquino process modelling

In this section, we introduce our proposals for workflow specification extensions using Aquino's process modelling component. By using illustrations from a real-world case study involving the governmental land administration of road closures [BH96], we describe the modelling concepts and features within a practical context. Less emphasis is placed on the detailed, LISA-D level of the specifications. Figure 1 contains an example of a process model.

From this depiction, three different types of elements which undergo execution, are apparent. These are: processes (rounded boxes), decisions (circles) and synchronisers (triangles). Processes perform actions like accessing databases (Application Entry) or sending and/or receiving

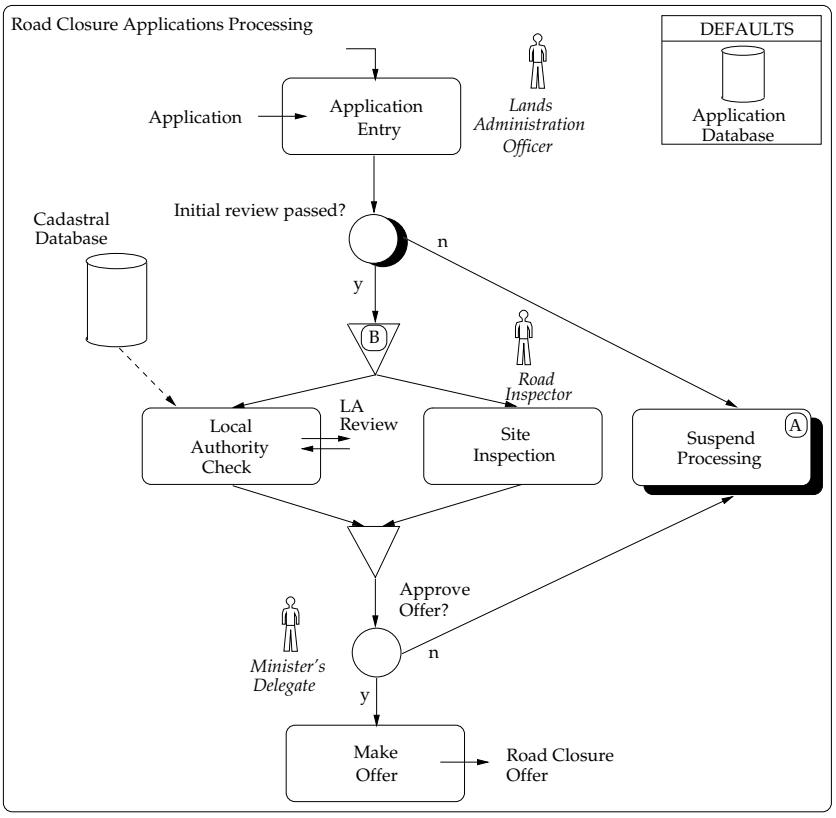

Figure 1: An example of a process model

messages (Application Entry, Local Authority Check, Make Offer). Decisions (Approve Offer?) allow a (non-deterministic) choice of execution paths to be followed depending the outcome, i.e. truth satisfaction of rules, of the decision. Synchronisers allow a junction of execution paths, whether divergent or convergent. The first synchroniser in Figure 1 results in two processes being executed; at the same implicit time. The second requires the two execution paths to be complete prior to further processing.

One or more elements are initially executed (bent arrow on top of element) while some elements - processes and decisions - are abstractions (shadowed symbols) which are decomposed further. Decision decompositions are discussed in more detail below. Processes and decisions may require a human actor in a role (human symbol) meaning that execution is either partially automated or entirely manual. In this paper, the focus is on fully automated processing only, and so, actors will be omitted from process models hereafter.

Processes and decisions access data from object stores, buffers (stack symbols) and through messaging. Object stores (cylinders), typically databases, are centrally accessible, persistent repositories of data. Within an integrated modelling context, the typing of an object store is achieved by binding it to an object model. Abstractable object modelling techniques such as the Extended Entity Relational Modelling Technique of [BCN92] and the Conceptual Data Modelling Kernel [CP96], enable an object *type* binding and with it, abstracted definitions,

including those involving generalisation and specialisation hierarchies. Figure 2 illustrates an example of an object type and its decomposition (rounded boxes are nested object types, ovals are object types, double-lined ovals indicate object types which occur elsewhere, boxes are roles, arrowed lines are uniqueness constraints on roles, and dots are mandatory role constraints). As we can see, nested abstractions (Party) are possible. Importantly, distributed, multi-database architectures, can be integrated into workflow specifications at the conceptual level, since abstraction hierarchies allow the capture of global and local schema mapping.

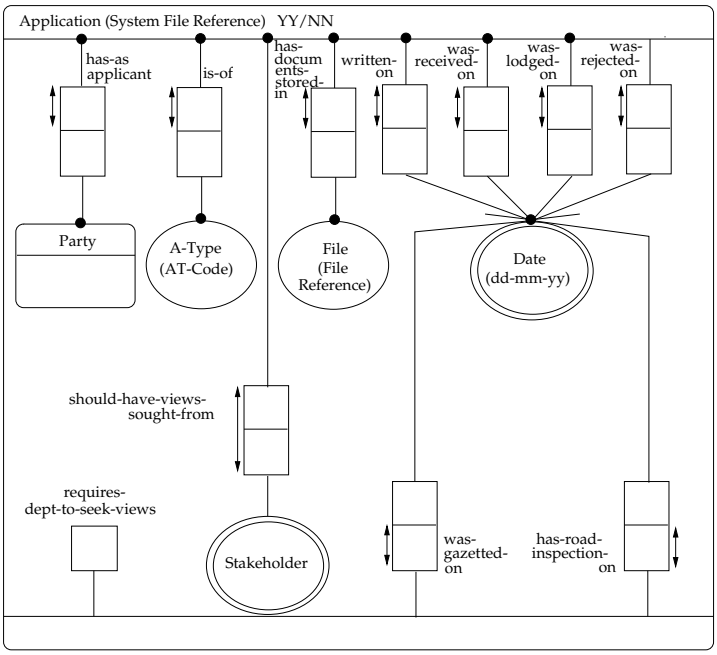

Figure 2: An example of an abstract object type

Through LISA-D manipulation of object models, detailed specifications of processes, i.e. pre- and post-conditions and detailed action statements, and decisions, i.e. decision rules, may be evaluated. For distributed transaction processing, we LISA-D has been extended to allow qualification of a set of databases in its database access statements. Access to an object store may be explicitly specified (arrowed broken line), though for clarity we allow, a default database (Application Database) to be indicated. This means that any process or decision in that process model accessing a database, accesses the default database, unless otherwise specified.

Buffers are transient, storage repositories of data, located within process decompositions and accessible only by processes defined in the same decomposition level. A buffer contains an ordered sequence of untyped values, e.g. a FIFO queue, which may be *consumed* and *produced* asynchronously by processes within the same decomposition level as the buffer. Such a mechanism is well-applied in the areas of process control and system design for reducing processing "bottlenecks". Figure 3 illustrates an example of buffer interaction involving producers (Store Notice) and consumers (Process Notice).

When a process consumes from a buffer (broken arrowed line from buffer), the "top" value of the buffer is removed and assigned to the process prior, to the process being initiated, i.e. prior

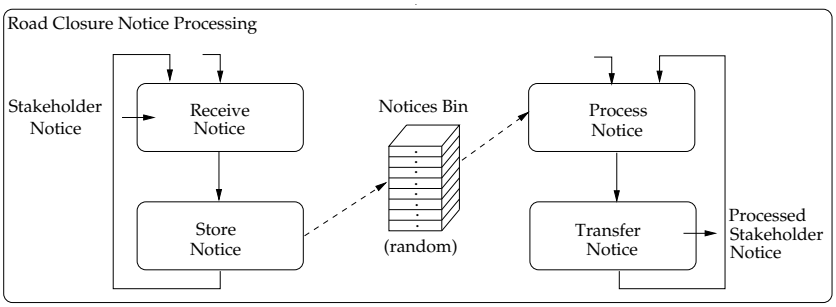

Figure 3: An example of buffer interaction

to its precondition being evaluated. We cater for different orders of data storage in a buffer. A buffer is therefore assigned a protocol (random) from which the "top" value to be consumed is determined. When a process produces a value into a buffer (broken arrowed line to buffer), the value is "appended" in the buffer after the termination of the process.

Following Task Structures, variables may be defined within decomposition levels to increase the expressive power of specifications. In this regard, buffers are essentially variables, and so, consumption involves assigning a consumed value to a variable - of the same name as the buffer and in the same decomposition level - and updating the buffer variable, to reflect the buffer state after consumption. Buffer production is simply updating the buffer variable so that the value of the process's variable becomes its "top" value.

Through *messaging*, we extend information passing to occur directly between processes across remote locations, i.e. decompositions, much like remote procedure calls. Associated with send and receive message actions is some data, typically a document, required to be transmitted. If a sender in one decomposition level has the same message label as a receiver in another decomposition level, communication can occur. As with databases, an integrated modelling context permits the typing of any data which is part of a message. Therefore, documents which are being passed between processes may be typed (abstractly) and access to their contents, once "within" a process, is amenable through the LISA-D functionality.

Messaging is characterised by two modes. A *synchronous* mode involves a suspension of execution related to a messaging dependency on some other process. An *asynchronous* mode involves no such suspension of execution. The different modes permit different messaging situations to be captured. We define three different messaging *configurations* - senders and receivers and their processing interactions - to capture different messaging requirements. These are illustrated in Figure 4.

Through a *waiting request* configuration (a), information or a notification can be passed to a remote source, however further information or a notification is required to be passed back for further processing to proceed. Hence, we say the configuration embodies a request; moreover followed by a wait at the sender end. The required construct for the request is clearly a synchronous sender (Local Authority Check). In another decomposition, a receiver is defined to accept the request. As apparent, a receiver (Application Entry) is inherently synchronous, i.e. its body is blocked from execution until receipt of the message. To return a response for the request, we also require an asynchronous sender; asynchronous because this type of sender (Make Offer) serves a simple transfer function. For the construction of responses, we allow any

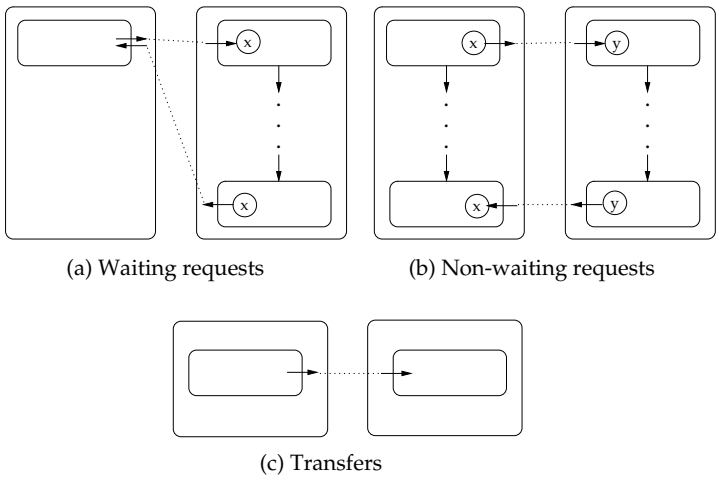

Figure 4: Types of messaging configurations

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. We describe such a pair which pertains to a request as an *accept-return* pair.

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*.

Note, data may or may not be passed through a message. Thus, we generalise into the notion of message, the concept of a signal, unlike its separation in, typically, object-oriented techniques, e.g. Jacobsen's Objectory [JCJO92] and Universal Modelling Language. Therefore, messages embodying requests and responses may simply be invocations to continue processing.

An advanced application of a non-waiting request involves deferred exception handling. This is a useful streamlining strategy where processing is allowed to go ahead after a request has been issued even though the response is not yet known. Of course, waiving synchronicity in favour of asynchronicity in this fashion is only useful if any response is unlikely to warrant undoing the elapsed processing.

Deferred exception handling is illustrated in Figure 5. Here two processes (Prepare Notification and Payment Preparation) send out notifications but continue processing without requiring confirmation of notification. Associated with each are processes which await any associated notifications. In either case, if a notification is received, an *abort* is issued ("zig-zag" arrow) which triggers an *abort handler*. In doing so, other processes in the decomposition are terminated. Within the decomposed process (Closure Administration), execution paths started from its initially executing processes both result in aborts being executed (one with a null abort han-

dler). This means that the first abort reached will terminate the rest of the decomposition. Also notice that the process model as a whole is terminated an abort (through Abort Preparation or Complete Road Closure Processing). In general, when an aborting process is introduced in a decomposition, care must be taken to ensure that any active processes (still active in this case so that notifications may be received), are terminated; otherwise the decomposition will not terminate.

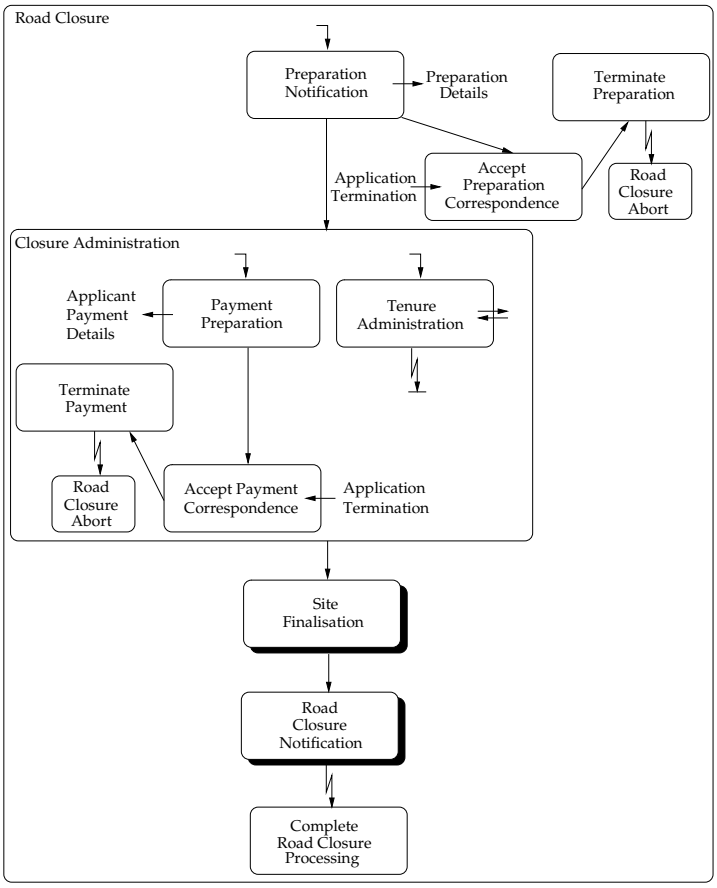

Figure 5: Deferred exception handling using messaging and aborts

Through this example, we have introduced the abort construct as an exception handling mechanism. Aborts are not restricted to non-waiting request configurations only, but are proposed as a general way of terminating execution in a decomposition.

The example of Figure 1 alludes to an advanced decision construct in Aquino. This follows the observation that decision processing in business transactions can be quite complex, involving not just the evaluation of one decision's outcomes, but the collective consideration of a (large) number of outcomes. We, therefore, allow such decision processing to be abstracted into complex decisions. Figure 6 illustrates the decomposition of a complex decision (Initial Review Passed? from Figure 1) where a number of simple decisions are undertaken in parallel. As apparent, most of these decisions require information from remote sources. For this purpose, we have used messaging on a decision as a "shorthand" for a messaging process

preceding the decision.

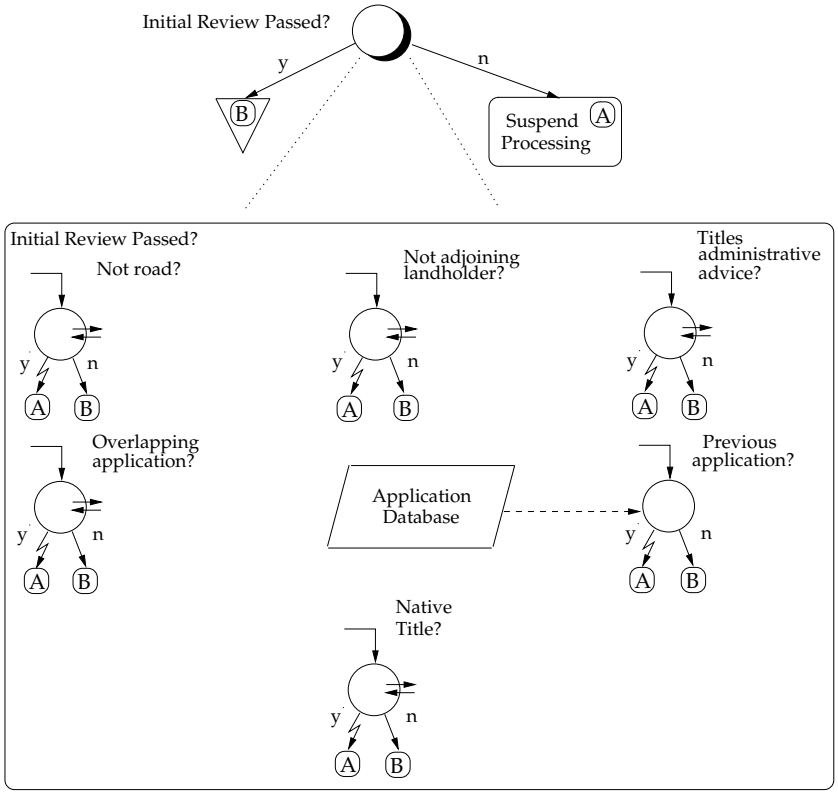

Figure 6: An example of a complex decision

It can be seen that labels have been introduced on the outcomes of the sub-decisions. This allows outcomes within a complex decision to be correlated to the complex decision's different triggering paths. In other words, the evaluation of a complex decision does not involve rules on the "outside", but rather, rules of outcomes within its internal execution structure. The outcome of some decisions is sufficient for determining the "outcome" of the complex decision. For this, the abort construct can be used as an outcome to stop processing in the complex decision and to run the associated element (similarly labelled). These are known as *aborting* decisions, which in general, may also be used in process decompositions as well. The outcome of other decisions do not have any effect on the remaining execution of the decision, meaning that while this outcome is determined, remaining outcomes should also be evaluated. Ultimately, either an abort-related action or some other action is run.

The following is an example of a decision rule specification associated with one of the outcomes (aborting outcome of Previous Application?) in Figure 6. It serves to illustrate LISA-D's expressive power for detailed specification (LISA-D expressions are built from *paths* through

object models using object type names and role names):

Application(has-as-parent-block Parcel **CONTAINING** Lot
elementary-surveyed-unit-of Parcel
has-road-area-related-to Current-App
AND ALSO
received-on Date $<$ Date marks-receipt-of Current-App
AND ALSO
received-on Date $\geq$ Date marks-receipt-of Current-App $-$ 2 years)

In addition to simple decision processing, terminating decision outcomes are permitted in complex decisions. This allows the execution path to exit without further processing (like a null action in an if-then-else programming language statement). If a subdecision terminates and it is the last execution path, the entire complex decision will terminate, i.e. it will not trigger anything else. Unlike simple decisions, complex decisions themselves are not allowed to terminate.

In the above example, any decision yielding an affirmative outcome is a sufficient determinant for executing an exception handler, otherwise normal processing continues. Figure 7 illustrates a modification to the example where prioritisation in the decision processing is introduced. This is useful since most of the decisions involve remote searches; in general, this illustrates how long and potentially expensive searches can be cascaded for decisions.

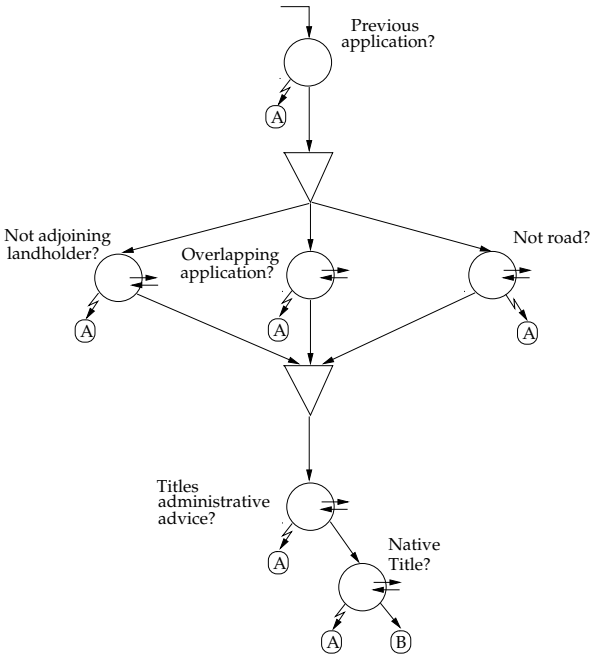

Figure 7: An application of complex decisions - prioritisation in decision processing

In line with the requirement for extending isolation atomicity in workflows, and advanced database transaction models for that matter, we provide *exclusive processes*. An exclusive process has an associated decomposition and its execution is atomic. Figure 8 illustrates an

example of a process which reads and prints a large amount of data. It has been specified as an exclusive process (double-lined process symbol) so that other tasks do not update that data while the process is running[3]. We also provide a nesting of exclusive processes, i.e. where an exclusive process contains exclusive processes in its decomposition hierarchy.

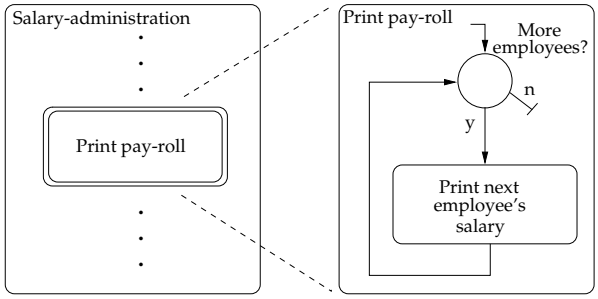

Figure 8: An example of exclusive processing

A further example in Figure 9 of exclusive processing involves a variation of deferred exception handling (described earlier). Here we include with the aborting mechanism, an exclusive process to determine whether incoming responses should result in a termination or corrective action, respectively. In the case of the latter, the exception handler will execute exclusively, to perform updates, say, related to the incoming notification. After this, normal processing will continue. In effect, a (form of an) *interrupt* mechanism, i.e. a temporary abort, is delivered.

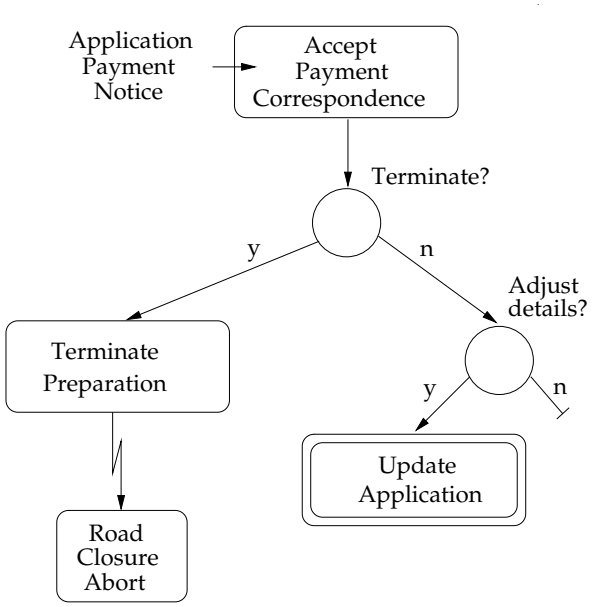

Figure 9: Implementing interrupts using exclusive processing

[3]No particular implementation strategy for concurrency control on data access, e.g. locking, is assumed at the conceptual level of our specification.

3 Formal syntax

Having informally described Aquino's process modelling, we now present its formal syntax. The syntax relates to the essential rather than concrete level of the technique, and is used for the development of the formal semantics. As such the static definitions and constraints presented are not claimed to be complete. Appendix A defines the set-theoretic notation used to express it (as far as it is non-standard).

3.1 General

In its broadest sense, the set $\mathcal{X}$ of process elements is partitioned into the set $\mathcal{P}$ of processes, the set $\mathcal{D}$ of decisions and the set $\mathcal{Y}$ of synchronisers. The triggering between the process elements is defined through the relation $\mathsf{Trig} \subseteq \mathcal{X} \times \mathcal{X}$.

We have described a variety of process and decision constructs. Accordingly, the set of processes is defined to contain inclusively, the set $\mathcal{C}$ of messaging processes, the set $\mathcal{G}$ of exclusive processes and elements of aborting processes from the set $\mathcal{Z} \subseteq \mathcal{X}$. In addition to aborting decisions from this set, the set of decisions includes the set $\mathcal{D}_t$ of terminating decisions, i.e. termination results from one of their outcomes.

For the purposes of identification at the type level, processes and decisions are denoted by names, as determined through the function $\mathsf{Name}\colon \mathcal{P} \cup \mathcal{D} \to \mathtt{V}$. Names are not required for synchronisers since they all have the same functionality. For general reference to name spaces of different process element sets X, we provide the function $\mathsf{Name}(X) = \{\mathsf{Name}(x) \mid x \in X\}$, where X is a general set denoting named process elements. We require the name spaces of processes $\mathsf{Name}(\mathcal{P})$, messaging processes $\mathsf{Name}(\mathcal{C})$ and decisions $\mathsf{Name}(\mathcal{D})$ to be mutually disjoint.

We saw that both processes and decisions may have decompositions. To cater for this, the partial function $\mathsf{Sup}\colon \mathcal{X} \rightarrowtail \mathtt{V}$ is defined to provide the decomposition that the process element belongs to. If $\mathsf{Sup}(x) = v$, this means that processing element x is part of the decomposition of v. For convenience, we define the set of *decomposition names* as those elements which have an associated decomposition:

$$\mathtt{V}_\mathtt{d} \equiv \{v \in \mathtt{V} \mid \exists_{x \in \mathcal{X}} [\mathsf{Sup}(x) = v]\}$$

Ultimately, a process model has a "top", i.e. the *main* process, which is not part of a decomposition. We call this main process p_0:

$$\exists!_{p \in \mathcal{P}} [\mathsf{Sup}(p)\uparrow]$$

For real-world workflow domains involving a *number* of interacting workflows, such a scheme may seem a little artificial. The notion of a single main process is, however, a technicality, introduced to simplify the assignment of formal semantics. For all intensive design purposes, multiple, top-level processes can be started up as initial items within the main process.

The execution of a decomposed process implodes "downwards" until its elementary parts are executed. Within a decomposition level, the initial elements of execution are obtained through

the partial function $\mathsf{Init} \subseteq \mathsf{Sup}$. Thereafter, Trig provides the triggering within a decomposition level. In this scheme, we require that triggers do not cross decomposition levels:

$$x_1 \,\mathsf{Trig}\, x_2 \Rightarrow \mathsf{Sup}(x_1) = \mathsf{Sup}(x_2)$$

At the same time, we allow recursive decomposition where a process is defined in its own decomposition hierarchy.

The execution of processes is atomic. Through exclusive processes, we have extended execution atomicity to entire process decompositions, hence:

$$\mathsf{Name}(\mathcal{G}) \subseteq \mathtt{V_d}$$

To provide greater specification power for execution coordination, we have introduced a form of exception handling through aborts. When an aborting process element is executed, its decomposition's execution is stopped and its associated abort handler, obtained through the function $\mathsf{Abt}\colon \mathcal{Z} \rightarrow \mathcal{X} \setminus (\mathcal{Y} \cup \mathcal{Z})$ is run. As defined, synchronisers and aborting process elements, themselves, are disallowed from being abort handlers to avoid semantic difficulties. Typically, abort handlers are processes, though they may also be decisions. To allow for modular abort handling specifications, they may also be decomposed.

For the detailed specification of processes (described in section 3.2) and decisions (section 3.3), data may be accessed centrally through object stores or via information passing through messages or buffers. The typing of databases is given through the function $\mathsf{Db}\colon \mathcal{J} \rightarrow \mathcal{O}$, where $\mathcal{J}$ is the set of object stores and $\mathcal{O}$ is the set of object types. This ensures that their underlying populations should conform to object model type definitions and constraints. As we will see, the typing of information passed through messages is defined in a query structure associated with message sending.

We have chosen to leave buffers, defined in the set $\mathcal{B}$, untyped, given their transient storage nature. At any point in time, a buffer is assigned values from the set of all values $L \subseteq \Omega$ in a total order $\prec$. The pair $\langle L, \prec \rangle$ is used to denote buffer content. As we have seen, buffers are accessed through consume and produce operations (described in section 3.2). A buffer's ordering of values, e.g. a FILO (first-in, last-out) queue, determines the next value which can be consumed. If we let Υ be the set of all pairs $\langle L, \prec \rangle$, a protocol function $p\colon \Upsilon \rightarrow \Omega$ may be defined to assign to each $\langle L, \prec \rangle$, a value in L. That is to say, $p(\langle L, \prec \rangle) \in L$ provides the next value which can be consumed from a buffer. The protocol of a buffer may then be determined through the function $\mathsf{Protocol}\colon \mathcal{B} \rightarrow \Gamma$, where Γ is the set of all protocol functions.

Syntactically, buffers are variables assigned to decomposition names, as given by the function $\mathsf{Buff}\colon \mathtt{V_d} \rightarrow \wp(\mathit{Var})$. *Var* is a LISA-D syntactic category of variable names. In general, local variables are declared within decompositions of both superprocesses and complex decisions. These are determined through the function $\mathsf{Locvar}\colon \mathtt{V_d} \rightarrow \wp(\mathit{Var})$. Processes and decisions may access variables within their decomposition hierarchies, providing the same sense of variable scoping in programming languages such as ALGOL68 (see e.g. [WMP+76]). To avoid naming conflicts, the names of local variables and buffers in a decomposition should differ:

$$\mathsf{Locvar}(v) \cap \mathsf{Buff}(v) = \varnothing$$

Following standard programming discipline, the creation of a decomposition requires initialisation of its environment. This involves the initialisation of buffers to empty and the initialisation of local variables to values determined by evaluating a transaction. The partial function $\mathsf{Locinit}: \mathtt{V_d} \times \mathit{Var} \rightarrowtail \mathit{Transaction}$ provides the transaction. Locinit can only be used for local variables v in decomposition n if v is a local variable of n:

$$\mathsf{Locinit}(n, v)\downarrow \Rightarrow v \in \mathsf{Locvar}(n)$$

3.2 Processes

All processes, atomic or otherwise, have pre- and post-conditions which define constraints for commencement and completion conditions of process execution, respectively. These are expressed using LISA-D predicates assigned through the functions $\mathsf{Pre}: \mathcal{P} \rightarrow \mathit{Predicate}$ and $\mathsf{Post}: \mathcal{P} \rightarrow \mathit{Predicate}$. LISA-D predicates reference the various types within object models and local variables. The special LISA-D predicate true can be used to indicate that there is no pre- or postcondition. For the purposes of evaluating queries and predicates, we have assumed a basic extension to LISA-D where a given query qualifies the set of databases that it refers to. This permits predicates, queries and transactions to be evaluated in the context of multiple databases.

For buffer interaction, a process must contain a variable with a name which corresponds to (i.e. has the same name as) the buffer name. The functions $\mathsf{Cons}: \mathcal{P} \rightarrow \wp(\mathit{Var})$ and $\mathsf{Prod}: \mathcal{P} \rightarrow \wp(\mathit{Var})$ define the buffer consumption and production relations. The mapping to a powerset of buffer variables follows from the fact that processes may interact with single, multiple or no buffers. Processes may only consume from, and produce for, buffers which are part of the same decomposition:

$$\mathsf{Cons}(p) \cup \mathsf{Prod}(p) \subseteq \mathsf{Buff}(\mathsf{Sup}(p))$$

Atomic processes may undertake transactions or be involved in messaging. Transactions are a sequence of these elementary actions, drawn from the LISA-D syntactic category *Transaction*. Using the following definition for the names of *transactional* processes:

$$\mathtt{V_t} \equiv \mathtt{V} \setminus (\mathtt{V_d} \cup \mathsf{Name}(\mathcal{C}) \cup \mathsf{Name}(\mathcal{D})$$

we can determine their transaction assignment through the function $\mathsf{Trans}: \mathtt{V_t} \rightarrow \mathit{Transaction}$.

Essentially, only three types of messaging processes are required. These are (asynchronous) senders $\mathcal{U}$, (synchronous) receivers $\mathcal{V}$, and synchronous senders $\mathcal{W}$. For convenience, we define the set $\mathcal{C} \equiv \mathcal{U} \cup \mathcal{V} \cup \mathcal{W}$ of all messaging processes. As alluded to in Figure 4, our messaging configurations of waiting requests, non-waiting requests and transfers can be built from these essential constructs. Messaging constructs other than these, e.g. messaging on decisions, can also be reduced to an elementary form.

As we have seen, a messaging configuration conveys the complete interaction messaging requirement. For the messaging to propagate correctly, communication should only occur between those processes which have the same messaging scope. Inherent in this requirement is correct message "handshaking, so that any information passed is correctly received. We further require that where needed, correlated send-receive and accept-return pairs be present.

To cater for correct "handshaking", we *type* the information content associated with messaging passing (noting once again that messaging with a null information content is equivalent to signalling). This is possible by using LISA-D queries to specify the information content, from the syntactic category *Query*. The type is, of course, inherent in the query. So that they can be evaluated at the receiving end to obtain the information content, the queries themselves are parameterised into a message passing action.

To cater for more generalised requirements like transferring multiple documents or passing control information along with a document, we allow a number of queries to be associated with message passing. The parameters of senders are determined through the function $\mathsf{SendPar}: (\mathcal{U} \cup \mathcal{W}) \rightarrow \mathit{Query}^*$. At the "other end", parameters are required for anticipated data in a message receiving action. These are determined through the function $\mathsf{RecPar}: (\mathcal{V} \cup \mathcal{W}) \rightarrow \mathcal{F}^*$. Received parameters are also, essentially variables.

In order to determine which processes can communicate, we define a messaging scope function $\mathsf{MesgScope}: \mathcal{C} \rightarrow \mathcal{M}$. Messaging processes which share the same message label (recall Application and LA Review from Figure 1) are allowed to communicate.

Furthermore, compatibility of parameter passing is required. Within our scheme, this means the number of messaging parameters in a communicating sender and receiver should be the same. More formally, for each $x \in \mathcal{U} \cup \mathcal{W}$ and $y \in \mathcal{V}$:

$$\mathsf{MesgScope}(x) = \mathsf{MesgScope}(y) \Rightarrow |\mathsf{SendPar}(x)| = |\mathsf{RecPar}(y)|$$

The correlation of loosely-coupled messaging pairs is determined through two functions. The function $\mathsf{Sender}: \mathcal{V} \rightarrowtail \mathcal{U}$ determines senders for receivers in send-receive pairs while the function $\mathsf{Acceptor}: \mathcal{U} \rightarrowtail \mathcal{V}$ determines acceptors for returners in accept-return pairs. Loosely-coupled pairs should be in the same level of decomposition:

$$\mathsf{Sup}(\mathsf{Sender}(x)) = \mathsf{Sup}(x) \text{ and } \mathsf{Sup}(\mathsf{Acceptor}(x)) = \mathsf{Sup}(a)$$

Receivers are used in both send-receive and accept-return pairs. The same receiver should therefore not be used in both, i.e. $\mathsf{dom}(\mathsf{Sender}) \cap \mathsf{ran}(\mathsf{Acceptor}) = \varnothing$. We do not define other syntactic constraints on messaging structures, e.g. to force synchronous senders to communicate with receivers which are part of accept-return pairs. As we will see, this will be handled in the formal semantics, i.e. when invalid communication takes places, a deadlock will result.

3.3 Decisions

Simple decisions do not do anything other than allow one of a set of outcomes, i.e. triggers, to be executed. Each outcome is attached a predicate which when evaluated to true can allow execution to proceed. If the predicate of more than one outcome evaluates to true, an arbitary choice of these outcomes is made, i.e. non-determinism is allowed in the choice of decisions. Rules of decisions are recorded by the function:

$$\mathsf{Choice}: ((\mathcal{D} \times \mathcal{X}) \cap \mathsf{Trig}) \cup (\mathcal{D} \times \{\multimap\}) \rightarrow \mathit{Predicate}$$

$\mathsf{Choice}(d, \multimap) = p$ means that d is a terminating decision which may lead to termination if p is fulfilled. Also $\mathsf{Choice}(d, \multimap) \equiv \mathit{false}$ for non-terminating decisions $d \in \mathcal{D} \setminus \mathcal{D}_t$. For aborting decisions, the function $\mathsf{Abt}: \mathcal{D} \times \mathit{Predicate} \rightarrowtail \mathcal{X}$ yields their abort handlers.

The execution of complex decisions has similarities to the execution of process decompositions in so far as an initial set of elements are determined from Init followed by the triggering within the decomposition thereafter, determined through Trig. Any processing element may be used in a complex decision. The partial auxiliary function $\mathsf{ComDec}: \mathcal{X} \rightarrowtail \mathcal{D}$ determines for process elements, the complex decisions they belong to. $\mathsf{ComDec}(x) = d$ means that process element x is part of complex decision d, hence $\mathsf{Sup}(x) = \mathsf{Name}(d) \wedge d \in \mathcal{D}$.

As described, the main purpose of a complex decision should be to determine from a set of decisions, a collective decision outcome. Accordingly, only decisions are allowed as the last elements in execution paths of complex decisions:

$$\mathsf{ComDec}(x)\downarrow \wedge \neg\, \exists_{y \in \mathcal{X}}\, [x\ \mathsf{Trig}\ y] \Rightarrow x \in \mathcal{D}$$

For those decisions that abort the entire decomposition, control should be transfered to an abort handler, given by Abt, associated with, i.e. triggered by, the complex decision:

$$\mathsf{Abt}(d,p)\downarrow \wedge\, \mathsf{ComDec}(d)\downarrow \Rightarrow \mathsf{ComDec}(p)\ \mathsf{Trig}\ \mathsf{Abt}(d,p)$$

Similarly, for those decisions that finish their part of the complex decision, control should be transfered to a processing element, given by the function $\mathsf{Fin}: \mathcal{D} \times \mathit{Predicate} \rightarrowtail \mathcal{X}$, associated with the complex decision:

$$\mathsf{Fin}(d,p)\downarrow \wedge\, \mathsf{ComDec}(d)\downarrow \Rightarrow \mathsf{ComDec}(p)\ \mathsf{Trig}\ \mathsf{Fin}(d,p)$$

Terminating decisions are allowed in complex decisions with the same behaviour. However, we do not allow complex decisions, themselves, to terminate:

$$d \in \mathcal{D}_t \Rightarrow \mathsf{Name}(d) \notin \mathtt{V_d}$$

4 Formal semantics

Having introduced process modelling in Aquino and provided a formal syntax of its essential concepts, it is now possible to define its formal semantics. We do so by translating a process model and its various constructs to equations in Process Algebra. Those operators and axioms defined within Process Algebra which serve our translation are described in section 4.1. Our Process Algebra translation is then presented in section 4.2.

In the presentation, we will use the following notational convention: sans serif font for syntactic functions (e.g. Name); typewriter font for Process Algebra generated actions (e.g. `consume`); and bold font for our abbreviations (e.g. **body**).

4.1 Process Algebra

Although the name Process Algebra suggests a single algebra to describe processes, it actually refers to a whole family of algebras e.g. CSP [Hoa78] and CCS [Mil89] based on the same principles. Our translation is based on a particular Process Algebra, the Algebra of Communicating Processes with the empty action and process creation (which we will abbreviate to

$X+Y=Y+X$	A1	$X+\delta=X$	A6
$(X+Y)+Z=X+(Y+Z)$	A2	$\delta\cdot X=\delta$	A7
$X+X=X$	A3	$X\cdot\varepsilon=X$	A8
$(X+Y)\cdot Z=X\cdot Z+Y\cdot Z$	A4	$\varepsilon\cdot X=X$	A9
$(X\cdot Y)\cdot Z=X\cdot(Y\cdot Z)$	A5		
$X\,\|\|\,Y=X\,\lfloor\!\lfloor Y+Y\,\lfloor\!\lfloor X+X\,\|\,Y+\surd(X)\,\surd(Y)$	C1	$aX\,\|\,bY=(a\,\|\,b)(X\,\|\|\,Y)$	C7
$\varepsilon\,\lfloor\!\lfloor X=\delta$	C2	$(X+Y)\,\|\,Z=X\,\|\,Z+Y\,\|\,Z$	C8
$aX\,\lfloor\!\lfloor Y=a(X\,\|\|\,Y)$	C3	$X\,\|(Y+Z)=X\,\|\,Y+X\,\|\,Z$	C9
$(X+Y)\,\lfloor\!\lfloor Z=X\,\lfloor\!\lfloor Z+Y\,\lfloor\!\lfloor Z$	C4	$a\,\|\,b=\gamma(a,b)$ if $\gamma(a,b)$ defined	C10
$\varepsilon\,\|\,X=\delta$	C5	$a\,\|\,b=\delta$ otherwise	C11
$X\,\|\,\varepsilon=\delta$	C6		
$\lambda_s^m(\gamma)=\gamma$ for $\gamma\in\{\delta,\varepsilon\}$	S1	$\partial_H(a)=a$ if $a\notin H$	D1
$\lambda_s^m(a\cdot X)=\mathsf{action}(a,m,s)\cdot\lambda^m_{\mathsf{effect}(a,m,s)}(X)$	S2	$\partial_H(a)=\delta$ if $a\in H$	D2
$\lambda_s^m(x+y)=\lambda_s^m(x)+\lambda_s^m(y)$	S3	$\partial_H(X+Y)=\partial_H(X)+\partial_H(Y)$	D3
		$\partial_H(X\cdot Y)=\partial_H(X)\cdot\partial_H(Y)$	D4
$E_\varphi(\gamma)=\gamma$ for $\gamma\in\{\delta,\varepsilon\}$	PC1	$\surd(\varepsilon)=\varepsilon$	TE1
$E_\varphi(aX)=a\cdot E_\varphi(X)$ for $a\notin\mathtt{cr}(D)$	PC2	$\surd(a)=\delta$	TE2
$E_\varphi(\mathtt{cr}(d)\cdot X)=\overline{\mathtt{cr}}(d)\cdot E_\varphi(\varphi(d)\,\|\|\,X)$	PC3	$\surd(X+Y)=\surd(X)+\surd(Y)$	TE3
$E_\varphi(X+Y)=E_\varphi(X)+E_\varphi(Y)$	PC4	$\surd(X\cdot Y)=\surd(X)\cdot\surd(Y)$	TE4

Table 1: Algebra of Communicating Processes with the empty action and process creation

$\mathrm{ACP}^{\mathrm{c}}_{\varepsilon}$). An in-depth description may be found in [BW90] with a useful set of applications described in [Bae90]. The axioms of $\mathrm{ACP}^{\mathrm{c}}_{\varepsilon}$ are listed in Table 1.

The units of Process Algebra are atomic actions. The set of all atomic actions is called $\mathcal{A}$. Although they are units of calculation, atomic actions need not be indivisible (see [GW96]). Starting with atomic actions, new processes can be constructed by applying sequential and alternative composition ("$\cdot$" resp. "$+$"). The algebra that results (axioms A1-A5) is called basic process algebra (BPA). As a convention, the names of atomic actions are written in lowercase (e.g. a, b, red_nose_reindeer), while process variables are written in uppercase (e.g. A, B, RUDOLPH). Normally, the $\cdot$ will be omitted unless this results in ambiguity. A special constant δ, *deadlock*, denotes the inaction, or impossibility to proceed (axioms A6-A7).

To add parallelism, an additional operator has to be introduced. This operator, called (free) *merge* and denoted as $\|$, is defined with the aid of an auxiliary operator $\lfloor\!\lfloor$, the *left-merge* (axioms C1-C4).

Another special constant ε, the *empty action*, is used to denote the process that does nothing but terminate successfully (axioms A8-A9). After adding ε, processes may terminate directly.

The *termination operator* $\surd$ determines whether or not this termination option is present for a given process (axioms TE1-TE4). The inclusion of the expression $\surd(X)\,\surd(Y)$ in axiom C1 guarantees that $\varepsilon \,\|\, \varepsilon = \varepsilon$.

The communication merge $|$ allows parallel processes to exchange information, i.e. to communicate (axioms C5-C9). Associated with communication is a *communication function* γ defined over pairs of atomic actions (axioms C10-C11). Specific process specifications will have to define the (partial) communication function γ. This function is both commutative and associative. Axioms C10 and C11 assume that communication is binary, but higher order communication is permitted as well.

The *encapsulation operator* ∂_H prevents the isolated occurrence of atomic actions meant to communicate with other actions. In fact it is a whole family of operators, one for each $H \subseteq \mathcal{A}$ (axioms D1-D4).

The *state operator* λ in Process Algebra is used to describe processes with an independent global state (axioms S1-S3). Informally, the expression $\lambda_s^m(X)$ represents the execution of process X on machine m in state s. The action function action calculates which action has to be performed as a result of executing X in state s on machine m, while the effect function effect calculates the new state.

Finally, as its name suggests, the *process creation* operator E_φ, introduced in [Ber90], enables process creation through the provision of a function when it encounters actions of a particular form (axioms PC1-PC4). In this regard, we can think of the function as an initialisation function $\varphi(d)$ and the action as one which *preempts* creation $\mathtt{cr}(d)$. d represents the data needed to create a new process from the set D of all create-related data and $\mathtt{cr}(D)$ is the set of all create actions. The action $\overline{\mathtt{cr}}(d)$ denotes the fact that creation has taken place and the initialisation runs in parallel with any other execution.

In our description of the various translations, we will refer to the $\mathrm{ACP}_\varepsilon^\mathrm{c}$ notion of *normalised* expressions. These have sequential and alternative composition only.

4.2 Process model translation

At the outset, a major issue in assigning a formal semantics for process model constructs is the occurrence of multiple instances of the same type running *concurrently*. This phenomenon, which we describe as *dynamic duplication*, is illustrated in Figure 10.

It can be seen that the synchroniser serves to *spawn* an instance of p_1, and repeated invocation of the synchroniser can result in more than one p_1 instance running concurrently. As illustrated, each instance has its own data space which should be distinguished somehow, if correct processing of its elements is to occur. Afterall, despite the syntactic constraint that buffer interaction is confined to buffers local to a decomposition level, dynamic duplication raises the issue of *which* buffer instance should instances of p_2 and p_3 interact with. Similarly open is the correspondence of a returning message with one of several waiting requests of the same type.

Our solution to this problem is to use *identifiers* for all processes, where uniqueness exists across decompositions only. This applies to decompositions of superprocesses as well as complex decisions. Hence, all subprocesses in the same decomposition (instance) will have the

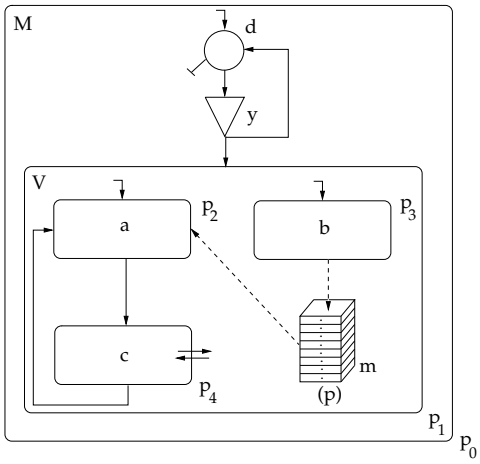

Figure 10: Dynamic duplication in a process model

same identifier. Applied to Figure 10, we can see that processes in duplicate instances of V running concurrently will be distinguished sufficiently. Of course, dynamic duplication can exist inside a decomposition; e.g. duplicate instances of synchronous message senders also have been spawned off in V and they could communicate with the duplicate instances of receivers in another decomposition. For intra-decompositional duplication, unique identifiers would have to be allocated. We have chosen, however, not to cater for this situation, because in our view, practical applications of it are difficult to find.

In order to achieve our *individualisation* in translations, we make use of a *rename* operator, ρ^α, introduced for the formalisation of the object-oriented language POOL [Vaa90]. The function of a given ρ^α, in passing through an expression containing a "target" action a (corresponding to a process), is to rename it with α; in our notational convention, to $a@\alpha$. The axioms for the ρ^α are defined in Table 2. Hereafter, we will assume that action individualisation in an expression will take R2's form unless stated otherwise.

$$\rho^\alpha(\gamma) = \gamma \text{ for } \gamma \in \{\delta, \varepsilon\} \quad \text{(R1)}$$
$$\rho^\alpha(a \cdot X) = a@\alpha \cdot \rho^\alpha(X) \quad \text{(R2)}$$
$$\rho^\alpha(X + Y) = \rho^\alpha(X) + \rho^\alpha(Y) \quad \text{(R3)}$$

Table 2: ρ^α operator

To generate process identifiers, and with it, ρ^α operators, we use process creation. As each process instance is created within a decomposition, the next identifier from an infinite set $\mathcal{ID}$ abstracting natural numbers (i.e. $\mathcal{ID} \equiv \mathbb{N}$), $\alpha \in \mathcal{ID}$, is generated (as we will see shortly, through our adaption of the process creation operator). The newly generated ρ^α can then pass over the expression related to the translation of the decomposition, individualising component actions of its subprocesses.

In using process creation, however, we are immediately confronted by a problem, illustrated through Figure 11, which results from axiom PC3.

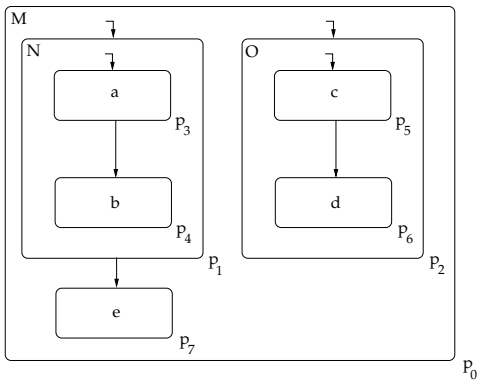

Figure 11: Problem of running process creation in parallel

This requires that actions sequenced *after* a create preemptive action run in *parallel* with the actual creation. Figure 11 illustrates its necessity, otherwise the execution of p_2 would be sequenced before or after the creation of p_1, not run concurrently. However, p_7 will also end up running in parallel with the creation of p_1, therefore allowing it to be interleaved with actions of p_3 and p_4.

So as not to "break" required sequencing, we define a *protected* form of sequential composition $X \odot Y$ which means that X will always be sequenced before Y because X triggers Y. The axioms of protected sequential composition are defined in Table 3.

$$\begin{aligned}
\varepsilon \odot X &= X && \text{(PS1)}\\
X \odot \varepsilon &= X && \text{(PS2)}\\
aX \odot Y &= a(X \odot Y) \text{ if } a \notin \mathtt{cr}(D) && \text{(PS3)}\\
(X+Y) \odot Z &= X \odot Z + Y \odot Z && \text{(PS4)}\\
(X \odot Y) \odot Z &= X \odot (Y \odot Z) && \text{(PS5)}
\end{aligned}$$

Table 3: Protected sequence operator

Using these axioms, we can state, through the following lemma, that protected sequential composition preserves sequential composition as a default (i.e. when no creation precedes the execution path).

Lemma 4.0

If X does not contain any create actions then $X \odot Y = X \cdot Y$.

Proof:

The proof uses induction on the structure of X:

1. $\varepsilon \odot Y = Y = \varepsilon Y$;
2. $aX \odot Y = a(X \odot Y) = a(XY) = (aX)Y$;
3. $(X+Z) \odot Y = X \odot Y + Z \odot Y = XY + ZY = (X+Z)Y$.

We use a variation of the Process Algebra process creation operator, namely $\mathcal{E}^{\alpha}_{\phi}$, where α is the identifier of the next process to be created. As a result of process creation (except, as we will see, that succeeding exclusive processes), α is incremented. The related axioms are defined in Table 4. In particular, axiom PK3 incorporates our adaption, where creates are preempted for decomposition names V within decomposition instance β. As we shall see shortly, the parent decomposition within which the child decomposition is being created is required for the purposes of variable inheritance from the parent to the child. Accordingly, the initialisation function passed on to effect creation contains V, β and, now, the new decomposition instance α; its translation is described below. In doing so, α is incremented to preserve unicity of the next identifier to be passed on. Also note, protected sequential composition is included for the reasons that we have just described. Finally, we have no need to mark creation, hence the absence of $\overline{\mathtt{cr}}(d)$.

$$\begin{aligned}
\mathcal{E}^{\alpha}_{\phi}(\gamma) &= \gamma \text{ for } \gamma \in \{\delta, \varepsilon\} && \text{(PK1)}\\
\mathcal{E}^{\alpha}_{\phi}(aX) &= a \cdot \mathcal{E}^{\alpha}_{\phi}(X) \text{ for } a \notin \mathtt{cr}(D) && \text{(PK2)}\\
\mathcal{E}^{\alpha}_{\phi}((\mathtt{cr}(V,\beta) \odot Y) \cdot Z) &= \mathcal{E}^{\alpha+1}_{\phi}(\varphi(V,\beta,\alpha) \odot Y \parallel Z) && \text{(PK3)}\\
\mathcal{E}^{\alpha}_{\phi}(X+Y) &= \mathcal{E}^{\alpha}_{\phi}(X) + \mathcal{E}^{\alpha}_{\phi}(Y) && \text{(PK4)}
\end{aligned}$$

Table 4: Process creation operator

It is now possible to define a translation for a process model, i.e. the main process p_0 which introduces the first decomposition. The equation E_{p_0} represents the entire system. Central to its translation, therefore, is a create pre-emptive action. Since the main process acts as a bootstrapping agent, i.e. it is not created in any decomposition, we use identifier 0 within its create preemptive action:

$$E_{p_0} = \Delta^{\varnothing,\varnothing}_{I,\varnothing,\varnothing,\varnothing,\mathsf{InitPop}}(\preccurlyeq \partial_H \circ \mathcal{E}^{2}_{\phi}(\ll \mathtt{cr}(\mathsf{Name}(p_0), 0) \gg) \succcurlyeq)$$

The delimiters $\ll$ and $\gg$ are used to denote the contexts of expressions of decompositions containing aborts. As discussed in section 4.2.2, such *abort contexts*, are useful for the purpose of reduction. As we will see, placing these delimiters around contexts which do not have aborts has no effect on reduction, so we can safely assume its generality. Also the delimiters $\preccurlyeq$ and $\succcurlyeq$ denote the outer most parts of process model execution, outside which no other processing can occur. They are used for the reduction of decompositions which are exclusively run, described in section 4.2.3.

The creation of aborting elements which are decompositions presents a problem as far as identifiers go. In the translation of aborting elements, as we will see, process creation has to *distribute* over the abort expressions and expressions of whatever else runs in parallel. But, since we cannot statically determine the number of creates in an abort decomposition, we do not know which identifier to allocate to the succeeding expression. To get around this, we use Gödel numbering. Recall, a Gödel number of a sequence $\langle n_1, \ldots, n_m \rangle$ is $p_1^{n_1} \times p_2^{n_2} \times \ldots \times p_m^{n_m}$, where $p_1, \ldots, p_m$ represent the first m prime numbers. For our purposes, we use the notation $\#\langle n_1, \ldots, n_m \rangle$ to represent the Gödel number of the sequence $\langle n_1, \ldots, n_m \rangle$. If we simply want to create new identifier for a process, we apply the function Succ defined by $\mathsf{Succ}(\#\langle n_1, \ldots, n_m \rangle) \equiv \#\langle n_1, \ldots, n_m + 1 \rangle$. If we need to guarantee unique process identifiers

for distinct processes, we simply access a new "dimension" through the function Next defined by $\mathsf{Next}(\#\langle n_1,\ldots,n_m\rangle) \equiv \#\langle n_1,\ldots,n_m,1\rangle$.

Through the main process translation, we have imposed a system wide encapsulation (note function composition $\circ$ has been used in place of function application). This constrains the communication of atomic actions, i.e. through use of the communication operator, to a particular set H. Its elements are: actions related to synchronisers, namely $\sigma_{x,y}$ for $y \in \mathcal{Y}$ and $\mathsf{Trig}(x,y)$; actions related to messaging, namely $\mathtt{send}, \mathtt{receive}, \mathtt{wait}, \mathtt{accept}$ and $\mathtt{return}$; and actions related to complex decisions $\mathtt{exitdec}$ and $\mathtt{waitdec}$.

We have positioned the state operator to pass through the whole system, allowing process elements to access and update information from databases, variables, buffers and run-time processing state information. Clearly, we have not used the $\mathrm{ACP}_\varepsilon^\mathrm{C}$ state operator, but a state operator of the form $\Delta^{e,c}_{i,o,v,b,d}$. Each of its environments are explained below.

The messaging environment e allows a messaging configuration to be tracked. This involves recording which sender and receiver instances, α and β respectively, are communicating, together with sender s and acceptor a types, so that send-receive and accept-return pairs can be correlated. Collectively, this is denoted by the tuple $\langle \alpha,\beta,s,a\rangle$ which forms a member of e. The complex decision environment c is used to ensure that resultant outcome actions of a complex decision are not triggered by internal outcomes of other complex decisions. For this, the identifier of the complex decision's resultant actions β are recorded with the identifier α of actions inside it; $\langle \beta,\alpha\rangle$. The input environment i contains an infinite sequence of inputs required collectively by processes. The output environment o contains the sequence of output produced collectively by processes. The variable environment $v{:}\mathcal{ID}\times \mathit{Var}\to\Omega$ provides the variable assignments within a particular decomposition. Values for variables are drawn from the set of all possible values Ω. To access these for a particular instance α, we will use the curry $\overline{v}(\alpha)$. The buffer environment $b{:}\mathcal{ID}\times\mathcal{B}\to\Upsilon$ provides the buffers assignments in a particular decomposition. Finally, the database environment $d{:}\,\mathcal{J}\to\mathsf{POP}$ is a function which provides a set of populations for each database.

For system start-up, the output, variable and buffer environments are empty (since no prior processing has occurred). An input stream I must be supplied and the database environment has a set of initial database populations given by $\mathsf{InitPop}$.

The initialisation function which is associated with the creation of a decomposition α, in decomposition β, and having a name V, i.e. $\varphi(V,\beta,\alpha)$, serves to generate a ρ^α for the individualisation of the new decomposition's processes, to initialise its environment and to trigger its initial process elements:

$$\varphi(V,\beta,\alpha) = \rho^\alpha(\mathtt{initproc}(V,\beta)\Big(\mathop{\Big\|}_{\substack{\mathsf{Init}(x)=V\\ x\notin\mathcal{Z}}} E_x \mathop{\Big\|}_{\substack{\mathsf{Init}(x)=V\\ x\in\mathcal{Z}}} T_x\Big))$$

The initialisation of the environment, $\mathtt{initproc}$, involves initialising the local variables and buffers. This occurs when the state operator passes over $\mathtt{initproc}$ (since the state operator "carries" the various environment states):

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{initproc}(V,\beta)@\alpha\cdot X) \;=\; \Delta^{e,c}_{i,o,v',b',d}(X)$$

Inherent in the initialisation of local variables is the inheritance of variables from the higher level of decomposition. This is performed first, followed by the assignment of initial values (so that local assignment is not lost);

$$v' = v \oplus \{ (\alpha, x) : v(\beta, x) \mid (\beta, x) \in \mathsf{dom}(v) \} \oplus \{ (\alpha, x) : \mathbf{T}[\![\mathsf{Locinit}(V, x)]\!](\langle d, \overline{v}(\beta), i, o \rangle) \mid \mathsf{Locinit}(V, x)\downarrow \}$$

The assignment of initial values, performed by a transaction given by Locinit, is evaluated through the semantic function (as formalised in [Hof93]):

$$\mathbf{T}: \mathit{Transaction} \times (\mathsf{DPOP} \times \mathsf{ENV} \times \mathsf{INPUT} \times \mathsf{OUTPUT}) \rightarrow (\mathsf{DPOP} \times \mathsf{ENV} \times \mathsf{INPUT} \times \mathsf{OUTPUT})$$

where: DPOP is an assignment from a set of databases to a set of populations, ENV is a set of variable assignments, INPUT is an infinite sequence of inputs, and OUTPUT is a sequence of outputs. These, of course, are provided by the state operator.

The initialisation of buffers, set to empty, is reflected in the updated buffer environment:

$$b' = \{ (\alpha, x) : \langle \varnothing, \varnothing \rangle \mid x \in \mathsf{Buff}(V) \}$$

For the initial elements which are started in parallel, we distinguish between translations of those elements which undergo normal triggering (E_x) and those which abort (T_x). Section 4.2.2 defines the difference which is present in the translation of aborts.

The translations of specific process elements now follows: general decomposable processes which are not exclusive processes in section 4.2.1; decompositions containing aborting processes in section 4.2.2; decompositions which are exclusive processes in section 4.2.3; atomic processes which perform database operations in section 4.2.4; atomic processes involving messaging in section 4.2.5; synchronisers in section 4.2.6; and finally simple and complex decisions in section 4.2.7.

4.2.1 Translation of decomposed processes

Regardless of their specific functionality, the basic structure of a process remains the same. Prior to the execution of their bodies, afterall, all processes may have preconditions and may consume data from a set of buffers; and after body execution, may have postconditons and may produce data into a set of buffers; finally triggering the next elements in the execution relation.

The translation of a process p with a decomposition which is not exclusively run $\mathsf{Name}(p) \in \mathtt{V_d}$ and $p \notin \mathcal{G}$:

$$E_p = (\nmid \mathsf{Pre}(p) \nmid \cdot \mathtt{consume}(\mathsf{Cons}(p)) \cdot \ll \mathbf{body}(p) \gg) \odot (\nmid \mathsf{Post}(p) \nmid \cdot \mathtt{produce}(\mathsf{Prod}(p)) \cdot \mathbf{trigrest}(p))$$

Following our strategy for process creation, the translation of a process with a decomposition involves create preemption:

$$\mathbf{body}(p) \equiv \mathtt{cr}(\mathsf{Name}(p))$$

Recall from axiom PK3, creation will only be preempted if a cr action includes the identifier of the decomposition in which creation is to occur. Accordingly, we adapt ρ^α individualisation:

$$\rho^\alpha(\ll \mathtt{cr}(V) \gg \odot Y) = \ll \mathtt{cr}(V, \alpha) \gg \odot \rho^\alpha(Y) \qquad \text{(R4)}$$

It is important to note that expressions containing create preemptive actions in protected sequential composition, e.g. $\ll \mathtt{cr}(V,\alpha) \gg \odot Y$, should be considered atomic, otherwise individualisation cannot occur (since these expressions cannot be normalised in a parallel context). These actions are therefore part of $\mathtt{cr}(D)$.

The abbreviation $\mathbf{trigrest}(p)$ provides the possible triggering paths following the execution of p. As previously mentioned, the translation of triggering and aborting process elements is distinguished. So too is that for the execution of synchronisers (which are not initially executing elements) for reasons discussed in section 4.2.6. Accordingly, $\mathbf{trigrest}(p)$ caters inclusively for the triggering of process elements which do not abort and do not trigger synchronisers; aborting processes which are not synchronisers; and synchronisers which may or may not abort:

$$\mathbf{trigrest}(p) = \left(\begin{array}{ccc} \Big\| & E_x & \Big\| \quad T_z \quad \Big\| \ \sigma_{p,y} \\ {\scriptstyle x \in \mathcal{X} \setminus (\mathcal{Y} \cup \mathcal{Z}),} & & {\scriptstyle z \in \mathcal{Z} \setminus \mathcal{Y},} \quad {\scriptstyle y \in \mathcal{Y},} \\ {\scriptstyle \mathsf{Trig}(p,x)} & & {\scriptstyle \mathsf{Trig}(p,z)} \quad {\scriptstyle \mathsf{Trig}(p,y)} \end{array} \right)$$

If p 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 empty action ε as the neutral element for parallel composition. Hence, for example, if no process elements x exist such that $\mathsf{Trig}(p,x)$, that part of the equation will reduce to the empty action. Of course, *no* execution path may be invoked given the fact that some process elements may not trigger any other process elements. If so, the whole of $\mathbf{trigrest}(p)$ reduces to ε.

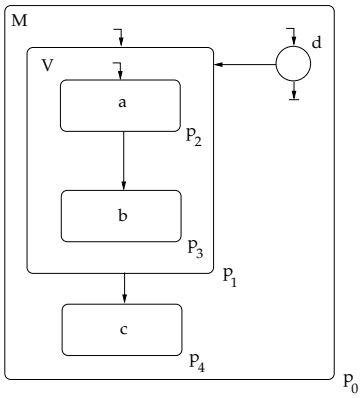

Figure 12: Example of process creation

Example 4.1

In the process model of Figure 12, a process p_1 and a decision d are started in parallel. p_1 triggers p_4 and, as one of its outcomes, d triggers p_1 (opening up the possibility for dynamic duplication). The following set of ACP_ε^C equations is the result of applying the previous translations to the process model of Figure 12. Note, we have omitted the state operator, system and abort contexts for simplicity. Also the translation for decisions is

defined in section 4.2.7:

$$\begin{array}{lcllcl} E_{p_0} & = & \mathcal{E}^2_\phi(\mathtt{cr}(M,0)) & E_d & = & E_{p_1} + \varepsilon \\ \varphi(M,\beta,\alpha) & = & \rho^\alpha(E_{p_1} \parallel E_d) & \varphi(V,\beta,\alpha) & = & \rho^\alpha(E_{p_2}) \\ E_{p_1} & = & \mathtt{cr}(V) \odot E_{p_4} & E_{p_2} & = & a \cdot E_{p_3} \\ E_{p_4} & = & c & E_{p_3} & = & b \end{array}$$

To evaluate pre- and postconditions, the semantic evaluation function

$$\mathbb{P}\colon \mathit{Predicate} \times \mathsf{DPOP} \times \mathsf{ENV} \to \{\mathit{true}, \mathit{false}\}$$

is used. It evaluates predicates from the LISA-D syntactic category *Predicate* in the context of a set of database populations and local variables. In general, for the translation of actions involving predicate evaluation, i.e. pre- and post-conditions and decision rules, we introduce the notion of a *conditional process*. Its relevant axioms are presented in Table 5.

$$\begin{array}{lcll} \lessdot \mathit{true} \gtrdot & = & \varepsilon & \text{(CP1)} \\ \lessdot \mathit{false} \gtrdot & = & \delta & \text{(CP2)} \end{array}$$

Table 5: Conditional processes

The state operator passes over an individualised condition, generating a conditional process:

$$\Delta^{e,c}_{i,o,v,b,d}(\lessdot P@\alpha \gtrdot \cdot X) = \lessdot \mathbb{P}[\![P]\!](d, \overline{v}(\alpha)) \gtrdot \cdot \Delta^{e,c}_{i,o,v,b,d}(X)$$

All processes may undertake buffer consumption and production. Recall, the contents of a buffer is at any point in time, a set of values L with a total order $\prec$. To assist in the translation of buffer consumptions and productions, we define the functions $\rhd$ and $\lhd$ for additions and deletions to buffers.

The deletion of value n from a buffer assignment $\langle V, \prec \rangle$ is given by:

$$\langle V, \prec \rangle \rhd n = \left\langle V \setminus \{n\}, \prec \setminus \{\langle x, y \rangle \mid (x = n \wedge y \in V) \vee (y = n \wedge x \in V)\} \right\rangle$$

The addition of a value n into a buffer assignment is given by:

$$\langle V, \prec \rangle \lhd n = \left\langle V \cup \{n\}, \prec \cup \{\langle v, n \rangle \mid v \in V\} \right\rangle$$

When the state operator encounters a `consume`, it deletes the "top" value from each buffer within a process instance's environment and assigns it to variables with the same names as the buffers:

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{consume}(W)@\alpha \cdot X) \quad = \quad \Delta^{e,c}_{i,o,v',b',d}(X)$$

Recall, the "top" value for a buffer is determined through its protocol, as given by the function $\mathsf{Protocol}$. For convenience, we use the abbreviation $\mathsf{Top}(b, w, \alpha) \triangleq \mathsf{Protocol}(w)(b(\alpha, w))$ to yield

the top value for a buffer w in a particular process α. The update of the target variables resulting from consume may then be defined as:

$$v' = v \oplus \{(\alpha, w) : \mathsf{Top}(b, w, \alpha) \mid w \in W\}$$

while the deletion of the values from buffers may be defined as:

$$b' = b \oplus \{(\alpha, w) : b(\alpha, w) \rhd \mathsf{Top}(b, w, \alpha) \mid w \in W\}$$

By definition, a `consume` from an empty buffer ($b(\alpha, w) = \langle \varnothing, \varnothing \rangle$) will result in deadlock.

When the state operator encounters a `produce` action, it adds to the relevant buffers, values in variables which correspond to it:

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{produce}(W)@\alpha \cdot X) \quad = \quad \Delta^{e,c}_{i,o,v,b',d}(X)$$

where:

$$b' = b \oplus \{(\alpha, w) : b(\alpha, w) \lhd v(\alpha, w) \mid w \in W\}$$

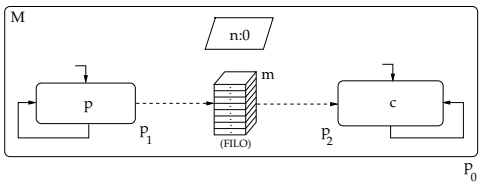

Figure 13: Example of buffering

Example 4.2

The process model of Figure 13 depicts a producer p_1 and a consumer p_2 running in parallel continuously, interacting with a FILO buffer m. Both have pre- and postconditions of true. *The values a_i produced for the buffer m result from the ith iteration of p_1. As p_1 produces a value, it increments variable n.*

To simplify the translation, we ignore the conditional processes of pre- and postconditions. Since only the state operator's variable and buffer environments are relevant for the translation, we use a simplified state operator denotation of the form $\Delta_{v,b}$.

Within this setting, the previous translations are applied to the process model to yield the following ACP^c_ε equations:

$$\begin{array}{lcllcl} E_{p_0} & = & \Delta_{\varnothing,\varnothing} \circ \partial_H \circ \mathcal{E}^2_\Phi(\ll \mathtt{cr}(M,1) \gg) & E_{p_1} & = & p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_1} \\ \varphi(M,\beta,\alpha) & = & \rho^\alpha(\mathtt{initproc}(M,\beta)(E_{p_1} \parallel E_{p_2})) & E_{p_2} & = & \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2} \end{array}$$

The reduction for E_{p_0} is described in three phases. In the first phase, variable n is initialised to 0 (as required), the buffer m is initialised to be empty and the first value a_1 is

produced for m:

$$\begin{aligned}
E_{p_0} &= \Delta_{\varnothing,\varnothing} \circ \partial_H \circ \mathcal{E}^2_\phi(\ll \mathtt{cr}(M,1) \gg) \\
&= \Delta_{\varnothing,\varnothing} \circ \partial_H \circ \mathcal{E}^2_\phi(\varphi(M,1,2)) \\
&= \Delta_{\{(1,n):0\},\{(1,m):\langle\varnothing,\varnothing\rangle\}} \circ \partial_H \circ \rho^1(p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) \\
&= \Delta_{\{(1,n):1\},\{(1,m):\langle\varnothing,\varnothing\rangle\}} \circ \partial_H \circ \rho^1(\mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) \\
&= \Delta_{\{(1,n):1\},\{(1,m):\langle\varnothing,\varnothing\rangle\}} \circ \partial_H \circ \rho^1(\mathtt{produce}(\{m\})(E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot E_{p_2}) + \\
&\qquad \mathtt{consume}(\{m\}) \cdot (c \cdot E_{p_2} \parallel \mathtt{produce}(\{m\}) \cdot E_{p_1})) \\
&= \Delta_{\{(1,n):1\},\{(1,m):\langle\{a_1\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) + \delta
\end{aligned}$$

In the second phase, a value can be consumed from the buffer since it now has a value. Therefore, either consumption or further production can occur:

$$\begin{aligned}
&= \Delta_{\{(1,n):1\},\{(1,m):\langle\{a_1\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) \\
&= \Delta_{\{(1,n):1\},\{(1,m):\langle\{a_1\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(p(\mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) + \\
&\qquad \mathtt{consume}(\{m\}) \cdot (c \cdot E_{p_2} \parallel p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_1})) \\
&= \Delta_{\{(1,n):2\},\{(1,m):\langle\{a_1\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(\mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel \mathtt{consume}(\{m\}) \cdot c \cdot E_{p_2}) + \\
&\quad \Delta_{\{(1,n):1,(1,m):a_1\},\{(1,m):\langle\{a_1\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(c \cdot E_{p_2} \parallel p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_1})
\end{aligned}$$

In the third phase, either further consumption or production can occur:

$$\begin{aligned}
&= \Delta_{\{(1,n):2\},\{(1,m):\langle\{a_1,a_2\},\langle\{a_1,a_2\}\rangle\rangle\}} \circ \partial_H \circ \rho^1(E_{p_1} \parallel \mathtt{consume}(m) \cdot c \cdot E_{p_2}) + \\
&\quad \Delta_{\{(1,n):2,(1,m):a_1\},\{(1,m):\langle\{a_2\},\varnothing\rangle\}} \circ \partial_H \circ \rho^1(\mathtt{produce}(\{m\}) \cdot E_{p_1} \parallel c \cdot E_{p_2}) + \\
&\quad \Delta_{\{(1,n):2,(1,m):a_1\},\{(1,m):\langle\varnothing,\varnothing\rangle\}} \circ \partial_H \circ \rho^1(E_{p_2} \parallel p \cdot \mathtt{produce}(\{m\}) \cdot E_{p_2})
\end{aligned}$$

4.2.2 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 in Table 6. These axioms were inspired by Bergstra's mode transfer operator in [Ber89].

$$\begin{aligned}
\ll \varepsilon \gg &= \varepsilon && \text{(I1)} \\
\ll a \cdot X \gg &= a \cdot \ll X \gg \text{ if } a \notin \mathtt{cr}(D) \cup Q && \text{(I2)} \\
\ll X + Y \gg &= \ll X \gg + \ll Y \gg && \text{(I3)} \\
\ll \langle X \rangle \looparrowright \langle Y \rangle \cdot Z \gg &= XY && \text{(I4)}
\end{aligned}$$

Table 6: Aborts

Through axiom I2, we can see that atomic actions can be shifted outside abort contexts provided they are not a create preemptive action (which recall is considered atomic along with

any action in protected composition with it). 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).

As such, the sequence of triggering in an abort expression should be preserved. For this reason, we consider aborting expressions, contained in the set Q, to be atomic. Obviously, they cannot be used to reduce abort contexts through axiom I2.

The axioms allow us to state through the following lemma, that non-abort contexts can be abort-contextualised without loss of behaviour. This supports the generality of the translation of non-exclusive, decomposable processes in section 4.2.1.

Lemma 4.0

$\ll X \gg = X$ if X does not contain actions from $Q \cup \mathtt{cr}(D)$.

Proof:

By induction on the structure of X.

Table 7 provides axioms for process creation applied to abort contexts and abort expressions.

$$\mathcal{E}^{\alpha}_{\phi}(\ll \mathtt{cr}(V,\beta) \gg \odot Y \cdot Z) = \ll \mathcal{E}^{\mathsf{Succ}(\alpha)}_{\phi}(\varphi(V,\beta,\alpha)) \gg \cdot \mathcal{E}^{\mathsf{Succ}(\alpha)}_{\phi}(Y) \parallel \mathcal{E}^{\mathsf{Next}(\alpha)}_{\phi}(Z) \quad \text{(PK5)}$$

$$\mathcal{E}^{\alpha}_{\phi}(\langle X \rangle \looparrowright \langle Y \rangle \cdot Z) = \left\langle \mathcal{E}^{\alpha}_{\phi}(X) \right\rangle \looparrowright \left\langle \mathcal{E}^{\alpha}_{\phi}(Y) \right\rangle \parallel \mathcal{E}^{\alpha}_{\phi}(Z) \quad \text{(PK6)}$$

Table 7: Process creation for aborts

As a result of encountering a create preemption action in an abort context, the process creation operator enters the abort context with its identifier being the next Gödel number. Since abort expressions are considered atomic, process creation passes through each expression with the same identifier. The expression after that of the abort handler will disappear so it will never run in parallel.

As with other expressions, individualisation applies similarly across aborting expressions:

$$\rho^{\alpha}(\langle X \rangle \looparrowright \langle Y \rangle \cdot Z) = \langle \rho^{\alpha}(X) \rangle \looparrowright \langle \rho^{\alpha}(Y) \rangle \cdot \rho^{\alpha}(Z) \quad \text{(R5)}$$

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:

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

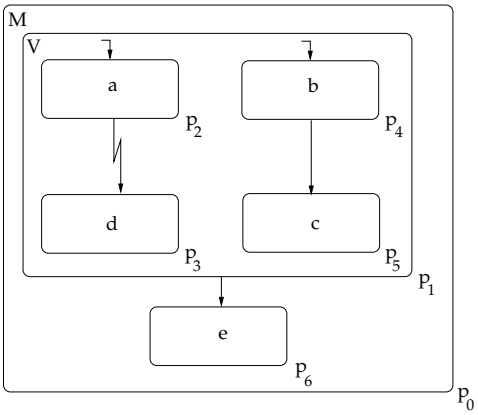

Figure 14: Aborting process example

Example 4.3

In the process model of Figure 14, 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. All processes have true *as pre- and post-conditions.*

In the translation which follows, we ignore the conditional processes for pre- and post-conditions since they have no effect. More generally, to focus the example on the essence of abort processing, the encapsulation and state operators are ignored.

Within this setting, the previous translations are applied to the process model to yeild the following ACP^C_ε equations:

$$\begin{array}{lcllcl}
E_{p_0} & = & \mathcal{E}_\phi^{\#\langle 1\rangle}(\ll \mathtt{cr}(M,0) \gg) & T_{p_2} & = & \langle E_{p_2}\rangle \looparrowright \langle E_{p_3}\rangle \\
\varphi(M,0,\#\langle 1\rangle) & = & \rho^{\#\langle 1\rangle}(E_{p_1}) & E_{p_2} & = & a \\
E_{p_1} & = & \ll \mathtt{cr}(V) \gg \odot E_{p_6} & E_{p_3} & = & d \\
\varphi(V,\#\langle 1\rangle,\#\langle 2\rangle) & = & \rho^{\#\langle 2\rangle}(T_{p_2} \parallel E_{p_4}) & E_{p_4} & = & b \cdot E_{p_5} \\
E_{p_6} & = & e & E_{p_5} & = & c
\end{array}$$

The abort processing axioms can then be included to reduce the equation for the process model, noting that $2 = \#\langle 1\rangle$, $4 = \#\langle 2\rangle$ and $8 = \#\langle 3\rangle$:

$$\begin{array}{rcl}
E_{p_0} & = & \mathcal{E}^2_\phi(\ll \mathtt{cr}(M,0) \gg) \\
& = & (\ll \mathcal{E}^4_\phi \varphi(M,0,2) \gg) \\
& = & (\ll \mathcal{E}^4_\phi \rho^2(\ll \mathtt{cr}(V) \gg \odot e) \gg \\
& = & \ll \mathcal{E}^4_\phi \ll (\mathtt{cr}(V),2) \gg \odot e@2 \gg \\
& = & \ll \ll \mathcal{E}^8_\phi(\rho^4(\langle a\rangle \looparrowright \langle d\rangle \parallel bc)) \gg \cdot e@2 \gg \\
& = & \ll \ll \langle a@4\rangle \looparrowright \langle d@4\rangle \cdot b@4 \cdot c@4 + \\
& & \quad b@4(c@4 \cdot \langle a@4\rangle \looparrowright \langle d@4\rangle + \\
& & \quad\quad \langle a@4\rangle \looparrowright \langle d@4\rangle \cdot c@4) \gg \cdot e@2 \gg \\
& = & \ll (a@4 \cdot d@4 + b@4(c@4 \cdot a@4 \cdot d@4 + a@4 \cdot d@4)) \cdot e@1 \gg \\
& = & a@4 \cdot d@4 \cdot e@1 + b@4(c@4 \cdot a@4 \cdot d@4 \cdot e@1 + a@4 \cdot d@4 \cdot e@1)
\end{array}$$

Ignoring process numbers this is equivalent to $ade + b(cade + ade)$, which shows that after executing the abort task named a the tasks named b and c are not executed anymore.

4.2.3 Translation of exclusive processes

Since the restriction of execution atomicity to elementary processes is unsuitable for more complex workflow requirements and advanced database transaction processing, we have introduced exclusive processes to allow an entire decomposition to run atomically. An interesting application illustrated in Figure 9 involves a message interrupt where the receipt of a message from outside a decomposition, in parallel with other processing in the decomposition, effectively interrupts (but does not abort) existing processing - through the resultant execution of an exclusive process.

For the purposes of denoting isolated execution for exclusive processes we introduce a *monitor* operator, axiomatised in Table 8.

$$
\begin{aligned}
\preccurlyeq \gamma \succcurlyeq &= \gamma \text{ for } \gamma \in \{\delta, \varepsilon\} && \text{(M1)}\\
\preccurlyeq a \cdot X \succcurlyeq &= a \cdot \preccurlyeq X \succcurlyeq \text{ for } a \notin \mathtt{cr}(D) \cup Q && \text{(M2)}\\
\preccurlyeq X + Y \succcurlyeq &= \preccurlyeq X \succcurlyeq + \preccurlyeq Y \succcurlyeq && \text{(M3)}\\
\preccurlyeq [a \cdot Y] \cdot Z \succcurlyeq &= \preccurlyeq a \cdot [Y] \cdot Z \succcurlyeq \text{ for } a \notin \mathtt{cr}(D) \cup Q && \text{(M4)}\\
[X] \cdot Y \between Z &= [X] \cdot (Y \parallel Z) && \text{(M5)}\\
[X + Y] &= [X] + [Y] && \text{(M6)}\\
[a] &= a \text{ for } a \in \mathcal{A} \cup \{\delta, \varepsilon\} && \text{(M7)}\\
[[a \cdot Y] \cdot Z] &= [a \cdot [Y] \cdot Z] \text{ for } a \notin \mathtt{cr}(D) \cup Q && \text{(M8)}
\end{aligned}
$$

Table 8: Monitor operator

Intuitively, if a process has a monitor, the execution of its actions cannot be interleaved with other actions. For the purposes of reducing a monitor, we introduce a system context, delimited by $\preccurlyeq$ and $\succcurlyeq$, outside which we know parallel processing will not occur. This is fixed at system start-up (recall the general process model translation in section 4.2) and axiom M2 permits reduction of a system context without loss of sequence.

For the reduction of monitors, actions can be moved outside monitors without loss of sequence and isolation. Axioms M4 and M8 ensure this; of course, all such reductions do not apply to create preemptive or abort actions.

Following from this, we can see that the translation of exclusive processes $p \in \mathcal{G}$ involves running their bodies and whatever else follows which is part of their execution, within a monitor:

$$E_p = \lessdot \mathsf{Pre}(p) \gtrdot \cdot \mathtt{consume}(\mathsf{Cons}(p)) \cdot [\ll \mathbf{body}(p) \gg \odot \lessdot \mathsf{Post}(p) \gtrdot \cdot \mathtt{produce}(\mathsf{Prod}(p))] \cdot \mathbf{trigrest}(p)$$

As usual, $\mathbf{body}(p)$ will translate to create preemption. Also, we do not need to use protected sequential composition after the monitor since process creation inside the monitor will not

change the order of processing after the monitor. We therefore allow process creation to distribute over the monitor and any subsequent processing:

$$\mathcal{E}^{\alpha}_{\phi}([X].Y) = \left[\mathcal{E}^{\alpha}_{\phi}(X)\right] \cdot \mathcal{E}^{\alpha}_{\phi}(Y) \tag{PK7}$$

The same identifier applies across distribution because the number of decompositions created within the monitor cannot be determined statically. This is unproblematic because any process that runs after the monitor cannot run in parallel with the monitor's process creation (by definition of a monitor).

To allow individualisation within a monitor, we provide an additional axiom for ρ^{α}:

$$\rho^{\alpha}([X] \cdot Y) = [\rho^{\alpha}(X)] \cdot \rho^{\alpha}(Y) \tag{R6}$$

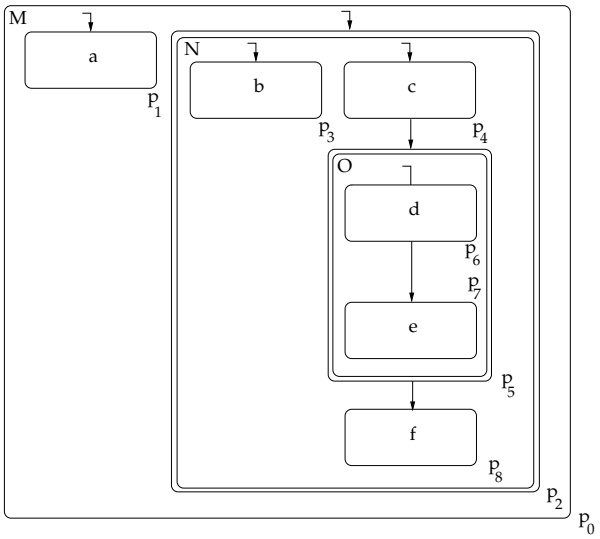

Figure 15: Exclusive process example

Example 4.4

In the process model of Figure 15 two processes are depicted, running in parallel, p_1 which is atomic and p_2 which has a decomposition and is run exclusively. p_2 also runs two processes in parallel, p_3 and p_4. p_4 triggers an exclusive process p_5 which has a decomposition of p_6 triggering p_7. Also p_5 triggers p_8. All processes have true *as pre- and postconditions and no process is involved in buffer interaction.*

In the translation, we ignore the conditional processes for pre-and postconditions since they have no effect. More generally, to focus the example on the essence of exclusive processing, the encapsulation and state operators are ignored, as are process creation and therefore action individualisation, and the abort context (no aborts occur).

Within this setting, the previous translations are applied to the process model to yield the

following $ACP_\varepsilon^\mathcal{C}$ equations:

$$\begin{array}{lcllcl} E_{p_0} & = & \preccurlyeq M \succcurlyeq = \preccurlyeq E_{p_1} \parallel E_{p_2} \succcurlyeq & E_{p_5} & = & [O] \cdot E_{p_8} = [E_{p_6}] \cdot E_{p_8} \\ E_{p_1} & = & a & E_{p_6} & = & d \cdot E_{p_7} \\ E_{p_2} & = & [N] = [E_{p_3} \parallel E_{p_4}] & E_{p_7} & = & e \\ E_{p_3} & = & b & E_{p_8} & = & f \\ E_{p_4} & = & c \cdot E_{p_5} & & & \end{array}$$

The reduction of E_{p_0} is:

$$\begin{array}{rcl} E_{p_0} & = & \preccurlyeq a\,[bc\,[de]\,f + c([de]\,f \parallel b)] \succcurlyeq + \preccurlyeq [bc\,[de]\,f + c([de]\,f \parallel b)]\,a \succcurlyeq \\ & = & a(\preccurlyeq [bc\,[de]\,f] \succcurlyeq + c \ \preccurlyeq [de]\,f \parallel b \succcurlyeq) + \\ & & \preccurlyeq [bc\,[de]\,f]\,a \succcurlyeq + \preccurlyeq [c([de]\,f \parallel b)]\,a \succcurlyeq \\ & = & a(bc \ \preccurlyeq [[de]\,f] \succcurlyeq + c(\preccurlyeq [de]\,(fb + bf) \succcurlyeq + \preccurlyeq b\,[de]\,f \succcurlyeq)) + \\ & & bc \ \preccurlyeq [[de]\,f]\,a \succcurlyeq + \preccurlyeq [c([de]\,(fb + bf) + b\,[de]\,f)]\,a \succcurlyeq \\ & = & a(bcdef + c(de(fb + bf) + bdef)) + bcdefa + c(de(fba + bfa) + bdefa) \end{array}$$

The normalised reduction demonstrates exclusive processing: a occurs either before the actions of N or after, i.e. it is not interleaved with the actions. The actions of O are also not interleaved, i.e. e always follows immediately after d.

4.2.4 Translation of atomic processes

Translation of body for atomic processes $\mathsf{Name}(p) \notin \mathtt{V_d} \wedge \mathtt{p} \notin \mathcal{G} \cup \mathcal{C}$:

$$\mathbf{body}(p) \equiv \lfloor \mathsf{Trans}(p) \rfloor$$

As can be seen, we have used a new type of atomic action, $\lfloor T \rfloor$, which we refer to as a transaction process. When the state operator passes through an expression containing a transaction process it updates its environments through the semantic evaluation of the transaction (introduced in section 4.2):

$$\Delta^{e,c}_{i,o,v,b,d}(\lfloor T \rfloor @ \alpha \cdot X) = \Delta^{e,c}_{R_{\langle 3 \rangle}, R_{\langle 4 \rangle}, R_{\langle 2 \rangle}, b, R_{\langle 1 \rangle}}(X)$$

where $R \equiv \mathbf{T}[\![T]\!](\langle d, \overline{v}\,(\alpha)\,, i, o \rangle)$, and $R_{\langle i \rangle}$ denotes the projection on the i-th element of R.

4.2.5 Translation of unbuffered 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.

We present the translations of messaging configurations in the remaining subsections. Our strategy has been adapted from the approach undertaken in a formalisation of messaging for Jacobson's Objectory [HH97].

Waiting requests 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. We combine both send and wait parts into the one action, `sendwait`, in order to formulate requests from synchronous senders:

$$E_w = \texttt{sendwait}(\mathsf{MesgScope}(w); \mathsf{SendPar}(w); \mathsf{RecPar}(w); w) \cdot \mathbf{trigrest}(s)$$

Recall we allow a number of message parameters for generality, e.g. for multiple documents or the passing of process control data accompanying a document. A further generality is the inclusion of w in the `send` to later enable correlation of send-receive pairs. In this case, it serves no use since the message send and receive are both sourced in w. Component `send` and `wait` actions are generated to follow sequentially and are individualised when ρ^α passes over a `sendwait`:

$$\begin{aligned}\rho^\alpha(\texttt{sendwait}(c; q_1, \ldots, q_n; r_1, \ldots, r_n; w) \cdot X) \quad = \quad & \texttt{send}(c; q_1, \ldots, q_n; w)@\alpha \cdot \\ & \texttt{wait}(w; r_1, \ldots, r_m)@\alpha \cdot \rho^\alpha(X)\end{aligned}$$

Of course, a request cannot be materialised unless the sent message is received by an acceptor $a \in \mathcal{V} \cap \mathsf{ran}(\mathsf{Acceptor})$. Its translation involves an `accept` action with its parameters corresponding to the sent message:

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

The name of the acceptor is also parameterised in the `accept` to later enable correlation of accept-return pairs. A `request` (pending) is generated when individualised `send` and `accept` actions communicate:

$$\texttt{send}(c; q_1, \ldots, q_n; w)@\alpha \mid \texttt{accept}(c; s_1, \ldots, s_n; a)@\beta = \texttt{request}(\langle \alpha, \beta, w, a \rangle; q_1 \colon s_1, \ldots, q_n \colon s_n)$$

Effectively, the sent and received parameters are associated within a specific messaging context denoted by the four-tuple $\langle \alpha, \beta, w, a \rangle$. This enables messaging configuration construction, where α identifies the sender of the request, β identifies the acceptor, and for the purposes that we have just discussed, w and a are correlation parameters for send-receives and accept-returns, respectively. 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 the state operator passes through the `request` action. It performs two functions. Firstly, it updates the messaging environment e allowing the tracking of senders and acceptors. Secondly, because it "carries" the variable and database environments, it effects parameter passing by evaluating the query associated with each sent parameter and assigning the result to the corresponding parameter (i.e. variable) in the acceptor's variable environment:

$$\Delta^{e,c}_{i,o,v,b,d}(\texttt{request}(\langle \alpha, \beta, w, a\rangle; q_1{:}\, s_1, \ldots, q_n{:}\, s_n) \cdot X) = \Delta^{e \oplus \langle \alpha,\beta,w,a\rangle, c}_{i,o,v',b,d}(X)$$

A query's result is determined using the semantic evaluation function $\mathbb{L}$ applied over a database environment using a set of variables, i.e. $\mathbb{L}$: *Query* $\times$ DPOP $\times$ ENV $\rightarrow$ QRes, where QRes is the set of query results. Therefore, the variable update performed by the state operator is:

$$v' = v \oplus \{(\beta, s_i){:}\, \mathbb{L}[\![q_i]\!](d, \overline{v}\,(\alpha)) | 1 \leq i \leq n\}$$

Corresponding to an acceptor is a returner $r \in \mathsf{dom}(\mathsf{Acceptor})$ which is instrumental in providing a response to requests. It has the following translation:

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

A `response` is generated when a `return` communicates with a `wait`:

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

The state operator restricts `response` actions to those where the `return` and `wait` are part of the same messaging configuration, i.e. where $\langle \alpha, \beta, a, w\rangle \in e$. Invalid `response` actions are reduced to deadlock while valid ones are materialised through two functions. Firstly, the state operator removes the messaging configuration (since communication is complete) and secondly, it effects parameter passing between the returner and the originating sender:

$$\Delta^{e,c}_{i,o,v,b,d}(\texttt{response}(\langle \alpha, \beta, w, a\rangle; r_1{:}\, t_1, \ldots, r_m{:}\, t_m) \cdot X) = \begin{cases} \Delta^{e \ominus \langle \alpha,\beta,w,a\rangle, c}_{i,o,v',b,d}(X) & \text{if } \langle \alpha, \beta, a, w\rangle \in e \\ \delta & \text{otherwise} \end{cases}$$

where:

$$v' = v \oplus \{(\alpha, r_i){:}\, \mathbb{L}[\![t_i]\!](d, \overline{v}\,(\beta)) | 1 \leq i \leq m\}$$

Example 4.5

The process model of Figure 16 depicts a waiting request configuration across two decompositions, of processes p_1 and p_2 respectively. The synchronous sender is p_4 in p_1 and the accept-return pair consists of p_6 as acceptor and p_9 as returner. All processes have true *as pre- and postconditions and no process is involved in buffer interaction.*

In the translation, we ignore the conditional processes for pre- and postconditions since they have no effect. For communication to proceed, the messaging scope is defined as $\mathsf{MesgScope}(p_4) = \mathsf{MesgScope}(p_6) = m$ *and we assume no information passing, i.e.* $\mathsf{SendPar}(p_4) = \mathsf{RecPar}(p_4) = \mathsf{RecPar}(p_6) = \mathsf{SendPar}(p_9) = \varnothing$. *Since only the state operator's messaging environment is relevant for the translation, we use a simplified state*

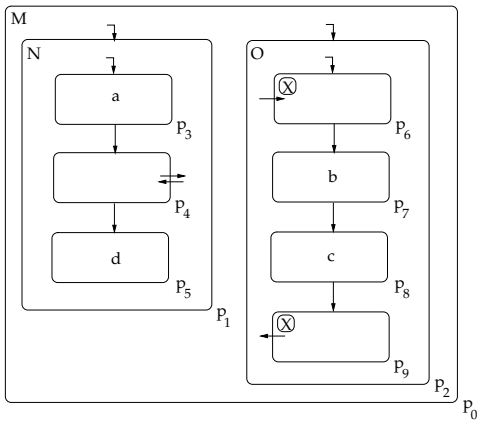

Figure 16: Example of synchronous messaging

operator denotation of the form Δ^e. We modify the state operator's function a little so that when it passes over actions, it strips out their numbers; individualisation does not play a role other than for message communication. Also since no abort exists in the process model, we do not abort contextualise the main process translation.

Within this setting, the previous translations are applied to the process model to yield the following $ACP_\varepsilon^\mathcal{C}$ equations:

$$
\begin{array}{lcllcl}
E_{p_0} & = & \Delta^{\varnothing} \circ \partial_H \circ \mathcal{E}_\phi^2(\mathtt{cr}(M,0)) & E_{p_5} & = & d \\
\varphi(M,\beta,\alpha) & = & \rho^\alpha(E_{p_1} \parallel E_{p_2}) & \varphi(O,\beta,\alpha) & = & \rho^\alpha(E_{p_6}) \\
E_{p_1} & = & \mathtt{cr}(N) & E_{p_6} & = & \mathtt{accept}(m;\varnothing;p_6) \cdot E_{p_7} \\
E_{p_2} & = & \mathtt{cr}(O) & E_{p_7} & = & b \cdot E_{p_8} \\
\varphi(N,\beta,\alpha) & = & \rho^\alpha(E_{p_3}) & E_{p_8} & = & c \cdot E_{p_9} \\
E_{p_3} & = & a \cdot E_{p_4} & E_{p_9} & = & \mathtt{return}(p_6;\varnothing) \\
E_{p_4} & = & \mathtt{sendwait}(m;\varnothing;\varnothing;p_4) \cdot E_{p_5} & & &
\end{array}
$$

The reduction for E_{p_0} is:

$$
\begin{aligned}
E_{p_0} &= \Delta^{\varnothing} \circ \partial_H \circ \mathcal{E}_\phi^2(\mathtt{cr}(M,0)) \\
&= \Delta^{\varnothing} \circ \partial_H \circ \mathcal{E}_\phi^4(\varphi(M,0,2)) \\
&= \Delta^{\varnothing} \circ \partial_H \circ \mathcal{E}_\phi^4(\rho^2(\mathtt{cr}(N) \parallel \mathtt{cr}(O))) \\
&= \Delta^{\varnothing} \circ \partial_H(\mathcal{E}_\phi^8(\rho^4(a \cdot \mathtt{sendwait}(m;\varnothing;\varnothing;p_4) \cdot d)) \parallel \mathcal{E}_\phi^{48}(\mathtt{cr}(O,2))) + \\
&\quad \Delta^{\varnothing} \circ \partial_H(\mathcal{E}_\phi^8(\rho^4(\mathtt{accept}(m;\varnothing;p_6) \cdot bc \cdot \mathtt{return}(p_6;\varnothing))) \parallel \mathcal{E}_\phi^{48}(\mathtt{cr}(N,2))) \\
&= \Delta^{\varnothing} \circ \partial_H(a@4 \cdot \mathtt{send}(m;\varnothing;p_4)@4 \cdot \mathtt{wait}(p_4;\varnothing)@4 \cdot d@4 \parallel \\
&\qquad\qquad \mathtt{accept}(m;\varnothing;p_6)@48 \cdot b@48 \cdot c@48 \cdot \mathtt{return}(p_6;\varnothing)@48) + \\
&\quad \Delta^{\varnothing} \circ \partial_H(\mathtt{accept}(m;\varnothing;p_6)@4 \cdot b@4 \cdot c@4 \cdot \mathtt{return}(p_6;\varnothing)@4 \parallel \\
&\qquad\qquad a@48 \cdot \mathtt{send}(m;\varnothing;p_4)@48 \cdot \mathtt{wait}(p_4;\varnothing)@48 \cdot d@48) \\
&= a \cdot \Delta^{\varnothing} \circ \partial_H((\mathtt{send}(m;\varnothing;p_4)@4 \mid \mathtt{accept}(m;\varnothing;p_6)@48) \cdot \\
&\qquad\qquad \mathtt{wait}(p_4;\varnothing)@4 \cdot d@4 \parallel b@48 \cdot c@48 \cdot \mathtt{return}(p_6;\varnothing)@48) + \\
&\quad a \cdot \Delta^{\varnothing} \circ \partial_H((\mathtt{accept}(m;\varnothing;p_6)@4 \mid \mathtt{send}(m;\varnothing;p_4)@48) \cdot \\
&\qquad\qquad \mathtt{wait}(p_4;\varnothing)@48 \cdot d@48 \parallel b@4 \cdot c@4 \cdot \mathtt{return}(p_6;\varnothing)@4)
\end{aligned}
$$

$$
\begin{aligned}
= \;& a \cdot \Delta^{\varnothing} \circ \partial_H((\mathtt{request}(\langle 4, 48, p_4, p_6\rangle; \varnothing) \cdot \\
& \qquad (\mathtt{wait}(p_4; \varnothing)@4 \cdot d@4 \parallel b@48 \cdot c@48 \cdot \mathtt{return}(p_6; \varnothing)@48))) + \\
& a \cdot \Delta^{\varnothing} \circ \partial_H((\mathtt{request}(\langle 48, 4, p_4, p_6\rangle; \varnothing) \cdot \\
& \qquad (\mathtt{wait}(p_4; \varnothing)@48 \cdot d@48 \parallel b@4 \cdot c@4 \cdot \mathtt{return}(p_6; \varnothing)@4))) \\
= \;& abc \cdot \Delta^{\{\langle 4,48,p_4,p_6\rangle\}} \circ \partial_H((\mathtt{wait}(p_4; \varnothing)@4 \mid \mathtt{return}(p_6; \varnothing)@48) \cdot d@4) + \\
& abc \cdot \Delta^{\{\langle 48,4,p_4,p_6\rangle\}} \circ \partial_H((\mathtt{wait}(p_4; \varnothing)@48 \mid \mathtt{return}(p_6; \varnothing)@4) \cdot d@48) \\
= \;& abc \cdot \Delta^{\{\langle 4,48,p_4,p_6\rangle\}} \circ \partial_H(\mathtt{response}(\langle 4, 48, p_4, p_6\rangle; \varnothing) \cdot d@4) + \\
& abc \cdot \Delta^{\{\langle 48,4,p_4,p_6\rangle\}} \circ \partial_H(\mathtt{response}(\langle 48, 4, p_4, p_6\rangle; \varnothing) \cdot d@48) \\
= \;& abcd
\end{aligned}
$$

Non-waiting requests A variation of the previously described messaging configuration is one where the sender has a separate receiver, i.e. in a send-receive pair. Therefore, for the situations of requests for information or the provision of notification, responses involve communication not with the sender but with the sender's associated receiver, i.e. no wait occurs.

The sender we are dealing with is an asynchronous sender $u \in \mathcal{U}$ and it has the following translation:

$$E_u = \mathtt{send}(\mathsf{MesgScope}(u); \mathsf{SendPar}(u); u) \cdot \mathbf{trigrest}(u)$$

The translation of its receiving counter-part $v \in \mathsf{dom}(\mathsf{Sender})$ is:

$$E_v = \mathtt{receive}(\mathsf{Sender}(v); \mathsf{RecPar}(v)) \cdot \mathbf{trigrest}(v)$$

The rest of the translations for synchronous sending apply similarly and are not repeated here. The only difference is that now a `response` is generated when a `return` communicates with a `receive`:

$$\mathtt{return}(a; t_1, \ldots, t_m)@\beta \mid \mathtt{receive}(w; r_1, \ldots, r_m)@\alpha = \mathtt{response}(\langle \alpha, \beta, w, a\rangle; r_1{:}\, t_1, \ldots, r_m{:}\, t_m)$$

The function of the state operator passing over a `response` remains the same.

Transfers Communication where responses are not required is essentially asynchronous. For this, a messaging configuration of an uncorrelated sender and an uncorrelated receiver are required.

As before, the translation of an uncorrelated sender $u \in \mathcal{U}$ results in a `send` while that of an uncorrelated receiver $v \in \mathcal{V} \setminus (\mathsf{dom}(\mathsf{Sender}) \cup \mathsf{ran}(\mathsf{Acceptor}))$ results in a `receive`. The translation of an uncorrelated sender $u \in \mathcal{U}$ remains the same as that of a correlated sender.

`send` and `receive` actions communicate generating a `transfer`, provided they have messaging scope compatibility:

$$\mathtt{send}(c; q_1, \ldots, q_n; u)@\alpha \mid \mathtt{receive}(c; r_1, \ldots, r_n; v)@\beta = \mathtt{transfer}(\langle \alpha, \beta, v, u\rangle; r_1{:}\, q_1, \ldots, r_n{:}\, q_n)$$

The state operator materialises a `transfer` by simply effecting parameter passing between the sender and the receiver:

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{transfer}(\langle \alpha, \beta, u, v\rangle; r_1{:}\, q_1, \ldots, r_n{:}\, q_n) \cdot X) = \Delta^{e,c}_{i,o,v',b,d}(X)$$

where:

$$v' = v \oplus \{(\beta, r_i) : \mathbb{L}[\![q_i]\!](d, \overline{v}(\alpha)) | 1 \leq i \leq n\}$$

The messaging environment is irrelevant since no tracking of sender and receivers is required subsequently, i.e. the materialisation of the `transfer` completes the configuration.

4.2.6 Translation of synchronisers

The translation of a synchroniser is unusual compared to the general triggering behaviour observed so far. This is because a number of input triggers may be involved, all of which need to be complete, prior to the synchroniser being triggered (otherwise the synchroniser would be triggered "prematurely").

Following its treatment in [HN93], we achieve synchronisation through communication. That is, every process element x which triggers synchroniser y, starts an atomic action $\sigma_{x,y}$ after termination. A synchroniser is started when 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 translation for a synchroniser $y \in \mathcal{Y}$ such that $\{x \in \mathcal{X} \mid \mathsf{Trig}(x, y)\} \neq \varnothing$ is:

$$\Big|_{\mathsf{Trig}(x,y)} \sigma_{x,y} = E_y$$

It is important to note, as defined previously, that the σ's are included into the membership of the encapsulation set H to restrict their communication to each other. Furthermore, unlike the atomic actions related to previously discussed process elements, synchroniser actions are not individualised. This is because ρ^{α} passes over normalised expressions only. This being the case, communication would have already have taken place, defeating the purpose any individualisation.

The execution of a synchroniser either involves an abort or follows normal triggering. The translation of aborting synchronisers $y \in \mathcal{Y} \cap \mathcal{Z}$ is:

$$E_y = \langle \varepsilon \rangle \looparrowright \langle E_{\mathsf{Abt}(y)} \rangle$$

The empty action is used for the body since a synchroniser does not do anything, *per se*. For the same reason, the translation of "normal" synchronisers, i.e. those which are not aborting, $y \in \mathcal{Y} \setminus \mathcal{Z}$ involves invoking the execution paths that follow:

$$E_y = \mathbf{trigrest}(y)$$

Example 4.6

In the process model of Figure 17, two initially executing processes, p_1 and p_2, need to be synchronised, prior to process p_3 being executed. All have true *as pre- and postconditions.*

In the translation, we ignore the conditional processes for pre-and postconditions since they have no effect. More generally, to focus the example on the essence of synchronisation, the encapsulation and state operators are ignored, as are process creation and therefore action individualisation, and the abort context (no aborts occur).

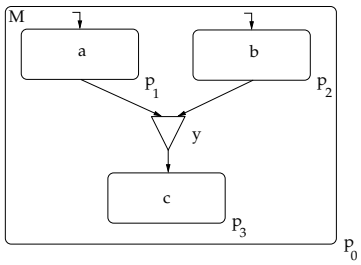

Figure 17: Synchroniser example

Within this setting, the previous translations are applied to the process model to yield the following $ACP_\varepsilon^\mathcal{C}$ equations:

$$\begin{array}{lcllcl} E_{p_0} & = & \partial_H(M) & \sigma_{p_1,s} \mid \sigma_{p_2,s} & = & E_y \\ M & = & E_{p_1} \parallel E_{p_2} & E_y & = & E_{p_3} \\ E_{p_1} & = & a \cdot \sigma_{p_1,y} & E_{p_3} & = & c \\ E_{p_2} & = & b \cdot \sigma_{p_2,y} & & & \end{array}$$

where $H = \{\sigma_{p_1,y}, \sigma_{p_2,y}\}$.

The reduction of the system is:

$$\begin{array}{rcl} E_{p_0} & = & \partial_H(a \cdot \sigma_{p_1,y} \parallel b \cdot \sigma_{p_2,y}) \\ & = & \partial_H(a(\sigma_{p_1,y} \parallel b \cdot \sigma_{p_2,y})) + \partial_H(b(\sigma_{p_2,y} \parallel a \cdot \sigma_{p_1,y})) \\ & = & a \cdot \partial_H(b(\sigma_{p_1,y} \parallel \sigma_{p_2,y})) + b \cdot \partial_H(a(\sigma_{p_2,y} \parallel \sigma_{p_1,y})) \\ & = & ab \cdot \partial_H(\sigma_{p_1,y} \mid \sigma_{p_2,y}) + ba \cdot \partial_H(\sigma_{p_2,y} \mid \sigma_{p_1,y}) \\ & = & abc + bac \end{array}$$

4.2.7 Translation of decisions

To allow moments of choice, simple decisions have a set of outcomes which are essentially triggers associated with rules. The execution of an outcome depends on the truth satisfaction of the rule, recalling that non-deterministic execution applies when more than one rule is satisfied. The resulting execution can involve either the triggering of other process elements, generating an abort or the termination of execution without further triggers.

The translation of a simple decision which is not part of a complex decision $d \in \mathcal{D}$ and $\mathsf{ComDec}(d)\uparrow$ is:

$$E_d = \sum_{\mathsf{Trig}(d,x)\downarrow} \lessdot \mathsf{Choice}(d,x) \gtrdot \cdot \zeta_x + \sum_{\mathsf{Abt}(d,p)\downarrow} \langle \lessdot p \gtrdot \rangle \leadsto \langle E_{\mathsf{Abt}(d,p)} \rangle + \lessdot \mathsf{Choice}(d, \multimap) \gtrdot$$

Each possible outcome for a decision is preceded by a conditional process corresponding to the rule associated with the outcome. To discriminate (as we have done previously) between the different translations resulting from triggers associated with normal outcomes - i.e. resulting

in the execution of synchronisers, elements which subsequently abort and normal elements - we use ζ_x:

$$\zeta_x = \begin{cases} \sigma_{d,x} & \text{if } x \in \mathcal{Y} \\ T_x & \text{if } x \in \mathcal{Z} \setminus \mathcal{Y} \\ E_x & \text{otherwise} \end{cases}$$

The translation of aborting decisions involves an abort expression (defined in section 4.2.2) which results in the current decomposition level being terminated and an abort handler being executed.

$\nleftarrow \mathsf{Choice}(d, \multimap) \nrightarrow$ relates to decision termination. Recall, we have forced $\nleftarrow \mathsf{Choice}(d, \multimap) \nrightarrow$ to fail truth satisfaction for non-terminating decisions thereby causing this expression to result in deadlock.

Complex decisions were introduced to abstract decision processing such that their outcomes are determined from within their decomposition. As we have seen, a normal process model structure can exist inside complex decisions with exception that the final triggers belong to simple decision outcomes only. These outcomes either abort the whole decision, finish without aborting or terminate (that execution path). With exception to termination, the internal outcomes are correlated with the external outcomes.

Aborting outcomes and the *last* finishing outcome trigger (communicate with, as we will shortly see) their correlated, external outcomes. For this we use the `exitdec` action. Outcomes which finish but are not final do not trigger anything. To incorporate this type of outcome partiality, we use a transient action `finish`, which as axiomatised in Table 9, raises an `exitdec` if no other execution is left. Since the external actions correlated with finishing outcomes can be any type of processing element, the parameter of a `finish` action has $x \in \mathcal{X}$.

$$\ll \mathtt{finish}(x) \gg = \mathtt{exitdec}(x) \quad \text{(F1)}$$

$$\ll \mathtt{finish}(x) \cdot aX \gg = \ll aX \gg \quad \text{(F2)}$$

$$\ll \mathtt{finish}(x) \cdot (X + Y) \gg = \ll \mathtt{finish}(x) \cdot X \gg + \ll \mathtt{finish}(x) \cdot Y \gg \quad \text{(F3)}$$

Table 9: Axioms for complex decisions

The translation of a simple decision inside a complex decision $d \in \mathcal{D}$ and $\mathsf{ComDec}(d)\downarrow$ is:

$$E_d = \sum_{\mathsf{Trig}(d,x)\downarrow} \nleftarrow \mathsf{Choice}(d, x) \nrightarrow \cdot \zeta_x + \nleftarrow \mathsf{Choice}(d, \multimap) \nrightarrow + \sum_{\mathsf{Abt}(d,p)\downarrow} \langle \nleftarrow p \nrightarrow \rangle \looparrowright \langle \mathtt{exitdec}(\mathsf{Abt}(d,p)) \rangle + \sum_{\mathsf{Fin}(d,p)\downarrow} \nleftarrow p \nrightarrow \mathtt{finish}(\mathsf{Fin}(d,p))$$

For outcomes which do not conclude an execution path inside complex decisions, the conditional triggering of other process elements and terminations apply (as usual). Following our informal description, the translation of aborting outcomes requires `exitdec` in the aborting handling part of an abort expression. It is targeted at an external process element determined through $\mathsf{Abt}(d, p)$. The translation of finishing outcomes requires a `finish` to cater for transient signalling. $\mathsf{Fin}(d, p)$ yields its target.

As in the translation of a superprocess, the translation of a complex decision $d \in \mathcal{D} \wedge \mathsf{Name}(d) \in \mathtt{V_d}$ involves a create preemption. The external actions activated to await signals from within the complex decision through `waitdec`:

$$E_d = \ll \mathtt{cr}(\mathsf{Name}(d)) \gg \cdot \left(\sum_{d\ \mathsf{Trig}\ x} \mathtt{waitdec}(x) \cdot \zeta_x \right)$$

ζ_x is used to discriminate between the different translations of aborting and finishing outcome actions:

$$\zeta_x = \begin{cases} T_x & \text{if } x \in \mathcal{Z} \setminus \mathcal{Y} \\ \sigma_{d,y} \text{if } y \in \mathcal{Y} & \\ E_x & \text{otherwise} \end{cases}$$

Given $\mathsf{Name}(d) = D$, instance α that the decision is created in and the decision's identifier β, the initialisation function serves to generate a ρ^α for individualisation within the decision, to initialise its environment and to trigger its initial elements:

$$\varphi(D, \beta, \alpha) = \rho^\alpha(\mathtt{initdec}(D, \beta) \left(\mathop{\Big\|}_{\substack{\mathsf{Init}(x) = D \\ x \notin \mathcal{Z}}} E_x \mathop{\Big\|}_{\substack{\mathsf{Init}(x) = D \\ x \in \mathcal{Z}}} T_x \right))$$

The state operator effects environment updates associated with an `initdec`, which like the initialisation of processes involves inheriting α's variables and performing local initialisations through a transaction. The details are similar to an `initproc` environment update described in section 4.2.1 and are not repeated here.

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{initdec}(D, \beta)@\alpha \cdot X) = \Delta^{e,c \oplus \langle \beta, \alpha \rangle}_{i,o,v',b,d}(X)$$

The complex decision environment c is also updated to associate decisions (instances) and their outcomes actions. This is necessary to avoid an outcome inside a decision resulting in an outcome of another decision (same type but different instance) being run. In this regard, an unrestricted set of *effective* outcomes for complex decisions are generated when decision exits communicate with (pending) external outcomes:

$$\mathtt{exitdec}(p)@\alpha \mid \mathtt{waitdec}(p)@\beta = \mathtt{outcome}(\beta, \alpha)$$

The state operator, of course, provides the restriction that effective outcomes reflect compatible decision exit and external outcome communication:

$$\Delta^{e,c}_{i,o,v,b,d}(\mathtt{outcome}(\beta, \alpha) \cdot X) = \begin{cases} \Delta^{e,c \ominus \langle \beta, \alpha \rangle}_{i,o,v,b,d}(X) & \text{if } \langle \beta, \alpha \rangle \in c \\ \delta & \text{otherwise} \end{cases}$$

Example 4.7

The process model of Figure 18 depicts a a complex decision with two external outcomes, namely processes p_1 and p_2 being triggered. Inside the decision, exiting outcomes come from decisions d_1 and d_2, where d_1 is run in parallel with d_2's triggering process, p_3. Both

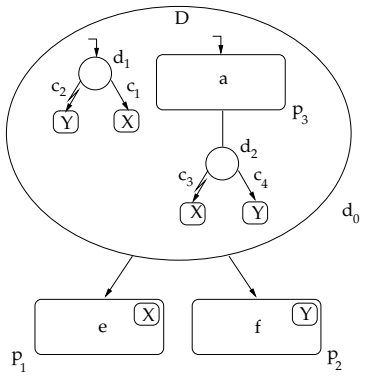

Figure 18: Complex decision example

decisions having aborting and finishing outcomes correlated (X,Y) with the external outcomes. Aborting outcomes are associated with conditions c_2 of d_1 and c_3 of d_2. Finishing outcomes are c_1 of d_1 and c_4 of d_2.

Buffer interaction and process pre- and postconditions are irrelevant and are ignored in the translation. Although the state operator plays a part in validating communication between decision exits and external outcomes, for simplicity we ignore it, and instead point out where communication occurs in the reduction. Also process creation does not play a vital part (no dynamic duplication occurs for the complex decision) and is therefore ignored. At the outset, its effect in passing over creation preemption of the complex decision would be to run the external outcomes in parallel. Accordingly, we denote this in the main equation. It follows, then, that no action individualisation will take place in the reduction.

Using the previous translations for the process model yields the following set of ACP_ε^c equations:

$$\begin{aligned}
E_{d_0} &= \partial_H(\ll D \gg \|(\mathtt{waitdec}(p_1)\cdot E_{p_1} + \mathtt{waitdec}(p_2)\cdot E_{p_2})) \\
D &= E_{d_1} \| E_{p_3} \\
E_{p_3} &= a \cdot E_{d_2} \\
E_{d_1} &= \langle \nmid c_2 \nmid \rangle \leadsto \langle \mathtt{exitdec}(p_2)\rangle + \nmid c_1 \nmid \mathtt{finish}(p_1) \\
E_{d_2} &= \langle \nmid c_3 \nmid \rangle \leadsto \langle \mathtt{exitdec}(p_1)\rangle + \nmid c_4 \nmid \mathtt{finish}(p_2) \\
E_{p_1} &= e \\
E_{p_2} &= f
\end{aligned}$$

The reduction of the complex decision is:

$$\begin{aligned}
E_{d_0} = \partial_H(\ll(\langle \nmid c_2 \nmid \rangle \leadsto \langle \mathtt{exitdec}(p_2)\rangle + \nmid c_1 \nmid \mathtt{finish}(p_1)) \\
\| a(\langle \nmid c_3 \nmid \rangle \leadsto \langle \mathtt{exitdec}(p_1)\rangle + \nmid c_4 \nmid \mathtt{finish}(p_2)) \gg \\
\|(\mathtt{waitdec}(p_1)\cdot e + \mathtt{waitdec}(p_2)\cdot f))
\end{aligned}$$

$$
\begin{aligned}
= \quad & \partial_H(\ll \langle \lessdot c_2 \gtrdot \rangle \looparrowright \langle \texttt{exitdec}(p_2) \rangle \cdot a(\langle \lessdot c_3 \gtrdot \rangle \looparrowright \langle \texttt{exitdec}(p_1) \rangle + \lessdot c_4 \gtrdot \texttt{finish}(p_2)) + \\
& \quad \lessdot c_1 \gtrdot \texttt{finish}(p_1) \cdot a(\langle \lessdot c_3 \gtrdot \rangle \looparrowright \langle \texttt{exitdec}(p_1) \rangle + \lessdot c_4 \gtrdot \texttt{finish}(p_2)) + \\
& \quad a(\langle \lessdot c_2 \gtrdot \rangle \looparrowright \langle \texttt{exitdec}(p_2) \rangle + \lessdot c_1 \gtrdot \texttt{finish}(p_1) \\
& \quad \| \langle \lessdot c_3 \gtrdot \rangle \looparrowright \langle \texttt{exitdec}(p_1) \rangle + \lessdot c_4 \gtrdot \texttt{finish}(p_2)) \gg \\
& \| (\texttt{waitdec}(p_1) \cdot e + \texttt{waitdec}(p_2) \cdot f)) \\
= \quad & \partial_H(\lessdot c_2 \gtrdot \texttt{exitdec}(p_2) + \lessdot c_1 \gtrdot a(\lessdot c_3 \gtrdot \texttt{exitdec}(p_1) + \lessdot c_4 \gtrdot \texttt{exitdec}(p_2)) + \\
& \quad a(\lessdot c_2 \gtrdot \texttt{exitdec}(p_2) + \lessdot c_1 \gtrdot (\lessdot c_3 \gtrdot \texttt{exitdec}(p_1) + \lessdot c_4 \gtrdot \texttt{exitdec}(p_2)) + \\
& \quad \lessdot c_3 \gtrdot \texttt{exitdec}(p_1) + \lessdot c_4 \gtrdot (\lessdot c_2 \gtrdot \texttt{exitdec}(p_2) + \lessdot c_1 \gtrdot \texttt{exitdec}(p_1))) \\
& \quad \| (\texttt{waitdec}(p_1) \cdot e + \texttt{waitdec}(p_2) \cdot f)) \\
= \quad & \lessdot c_2 \gtrdot \cdot f + \lessdot c_1 \gtrdot a(\lessdot c_3 \gtrdot \cdot e + \lessdot c_4 \gtrdot \cdot f) + \\
& a(\lessdot c_2 \gtrdot \cdot f + \lessdot c_1 \gtrdot (\lessdot c_3 \gtrdot \cdot e + \lessdot c_4 \gtrdot \cdot f) + \\
& \quad \lessdot c_3 \gtrdot \cdot e + \lessdot c_4 \gtrdot (\lessdot c_2 \gtrdot \cdot f + \lessdot c_1 \gtrdot \cdot e))
\end{aligned}
$$

5 Conclusion

At the outset, the paper sought to explore the meaning of process-centric workflow specifications extended with information passing through buffering and messaging, abort handling and decision decompositions. Previous research confirmed that such extensions enhance considerably the business processing cognition imparted in workflow specifications, specifically those of coordinative workflow domains. Our inspiration for conceptualisation, even down to detailed levels of process specifications, guided the choice of (Hydra) Task Structures. Included in its expressive artillery is an ORM-based data modelling technique and a conceptual specification language, augmenting the specification of intricate, if deceptively simple, processing constructs, e.g. synchronisation, choice, iteration, (recursive) decomposition, parallelisation and processing exclusivity. Consequently, all these aspects were inherited into our goal of assigning a formal semantics for - Aquino process modelling.

Clearly, the major hurdle was dynamic duplication. This is possible through the repeated triggering of process elements of the same type. For this, we applied Process Algebra's process creation to generate decomposition-grained identifiers which served in the individualisation of all created actions. In contrast, other solutions are much more complicated and are difficult to exemplify. For example, the previous formalisation of Task Structures uses three state operators and infinite summations over all possible environments as the right environment cannot be determined "locally" but only at the global level. And the formalisation of Jacobson's Objectory involves a precreation of an infinite set of uniquely identified objects, allocated when actual creation takes place.

A further problem is the necessary parallelisation of process element creation within a decomposition. We saw that without an intervention like protected sequential composition, process elements required to be triggered after the processes being created, will also run in parallel. And further problematic is unicity of identification for expressions which follow those of abort contexts; and which do not require protected sequencing. Since process creation had to be distribute over these expressions, we were faced with the difficulty of preallocating their identifiers. The difficulty arose because the number of creations in abort contexts and monitors

cannot be determined statically. Gödel numbering provided unique identifier spaces, and so it was adapted for general identification.

Beyond creation, the translation of other constructs was a little more straightforward. For aborts and exclusive processes, adaptions of Process Algebra's mode transfer, led to aborting expressions and monitors respectively. Messaging and complex decisions were involved applications of communication. The state operator provided an instance's data space preservation and the dynamic tracking of instanced interactions.

In general, Process Algebra proved to be a flexible enough framework to specify what is unquestionably a complex formalisation. It is with little hesitation that we recommend its use, particularly in the generally informal discipline of information systems.

As we alluded, business processing is complex and any journey which embarks on its conceptualisation cannot claim to be final. Ours is no exception. Through our suitability synthesis, time, service encapsulation, and with it, the service coordination of "islands" of workflows, and process recovery are further areas which are inherently grounded in business requirements. Without early conceptualisation, these too can slip into unvalidated programming decisions. Also our integration of data and control flow is very much "first cut". Further varieties of buffering and messaging, notably with temporal constraints need to be canvassed. Finally, an actor resource model which appears in workflow products like InConcert and enterprise modelling techniques, e.g. [Ram94], needs to be factored into the formal semantics.

A Mathematical notation

In this appendix the mathematical notation used in this paper, as far as it is non-standard, is explained briefly.

A.1 Functions

If f is a partial function from A to B, i.e. $f\colon A \rightarrowtail B$, then:

$$\begin{aligned} f(a)\downarrow &= \exists_{b\in B}\,[f(a) = b] \\ f(a)\uparrow &= \neg\, f(a)\downarrow \\ \mathsf{dom}(f) &= \{a \in A \mid f(a)\downarrow\} \\ \mathsf{ran}(f) &= \{b \in B \mid \exists_{a\in A}\,[f(a) = b]\} \end{aligned}$$

If f and g are partial functions, then $f \oplus g$ is also a partial function defined by:

$$f \oplus g = g \cup \{\langle x, y\rangle \in f \mid g(x)\downarrow\}$$

The function $f \oplus \{a\colon b\}$ therefore behaves the same as function f except that its value in a is b. This can be generalised to $f \oplus g$. In some cases, we will overload this notation and write $S \oplus s$, where S is a set and s an element, as an abbreviation for $S \cup \{s\}$. The notation $S \ominus s$ is an abbreviation for $S \setminus \{s\}$.

To avoid having to use many parenthesis as a result of repetitive function applications, the function composition notation may be applied. The function $f \circ g$ is defined by:

$$f \circ g(x) = f(g(x))$$

Naturally, it is required that $\mathsf{ran}(g) \subseteq \mathsf{dom}(f)$.

The function $f[A']$ is the function f restricted to a subdomain $A' \subseteq \mathsf{dom}(f)$. This function is defined by:

$$f[A'] = \left\{ \langle a, b \rangle \in f \mid a \in A' \right\}$$

To avoid the frequent use of tuple brackets ($\langle\rangle$), a shorthand for denoting functions is used. For example, the function f, defined by $f = \{(p, a), (q, b)\}$, is denoted as $\{p{:}a, q{:}b\}$.

If f is a function from $A \times B$ to C, the curry of f for a specific element $a \in A$, is defined by $\overline{f}(a) = \{(b, c) \in B \times C \mid f(a, b) = c\}$, hence the signature of $\overline{f}$ is $A \to (B \to C)$ and for any $a \in A, b \in B$ we have $\overline{f}(a)(b) = f(a, b)$ (see e.g. [Mey90]).

The power set of a set A, i.e. the set of all subsets of A, is denoted as $\wp(A)$. The i-th element of a tuple $t = \langle a_1, \ldots, a_i, \ldots, a_n \rangle$, i.e. a_i, can be found by the projection $t_{\langle i \rangle}$: The length of a tuple, i.e. its number of elements, can be found through $|t| = n$.

B Graphical notation

In this appendix the graphical notation used for process modelling in this paper, is summarised. Figure 19 contains the elementary symbols.

Figure 20 elaborates on some of the common relationships between process modelling constructs which are graphically depicted.

References

[Bae90] J.C.M. Baeten, editor. *Applications of Process Algebra.* Cambridge University Press, Cambridge, United Kingdom, 1990.

[BCN92] C. Batini, S. Ceri, and S.B. Navathe. *Conceptual Database Design - An Entity-Relationship Approach.* Benjamin Cummings, Redwood City, California, 1992.

[Ber89] J.A. Bergstra. A Mode Transfer Operator in Process Algebra. Report P8808b, University of Amsterdam, The Netherlands, 1989.

[Ber90] J.A. Bergstra. A process creation mechanism in process algebra. In J.C.M. Baeten, editor, *Applications of Process Algebra*, pages 81–88. Cambridge University Press, Cambridge, United Kingdom, 1990.

[BH96] A.P. Barros and A.H.M. ter Hofstede. A Business Transaction Workflow Case Study: Road Closures in Queensland. Technical Report 393, Department of Computer Science, The University of Queensland, Brisbane, Australia, December 1996.

[BHP97a] A.P. Barros, A.H.M. ter Hofstede, and H.A. Proper. Essential Principles for Workflow Modelling Effectiveness. In G.G. Gable and R.A.G. Webber, editors, *Proceedings of the Third Pacific Asia Conference on Information Systems (PACIS'97)*, pages 137–147, Brisbane, Australia, April 1997.

[BHP97b] A.P. Barros, A.H.M. ter Hofstede, and H.A. Proper. Towards Real-Scale Business Transaction Workflow Modelling. In A. Olivé and J.A. Pastor, 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, June 1997. Springer Verlag.

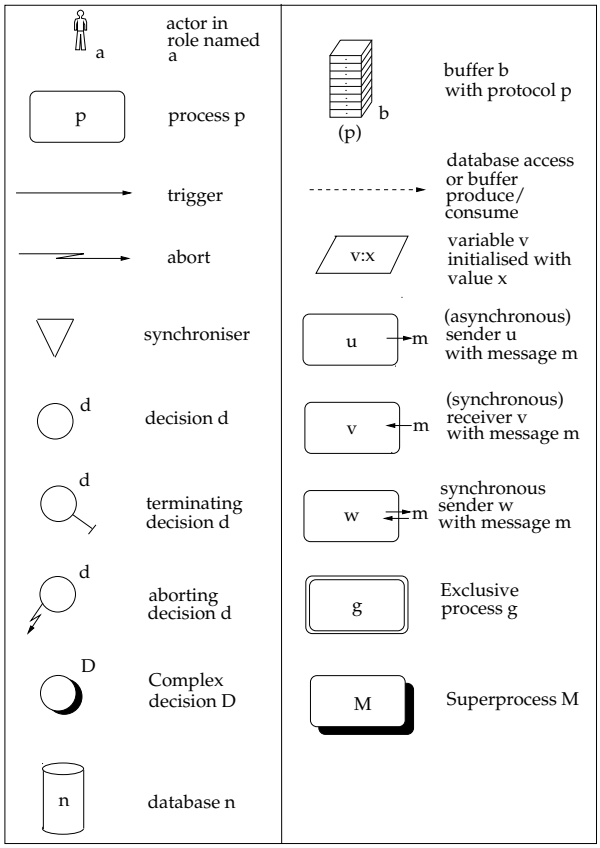

Figure 19: Summary of graphical notation for process modelling constructs

[Bro87] F.P. Brooks Jr. No silver bullet: essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[BS87] D. Benyon and S. Skidmore. Towards a Tool Kit for the Systems Analyst. *The Computer Journal*, 30(1):2–7, 1987.

[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, Cambridge, United Kingdom, 1990.

[CCPP95] F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modeling of Workflows. In M.P. Papazoglou, editor, *Proceedings of the OOER'95, 14th International Object-Oriented and Entity-Relationship Modelling Conference*, volume 1021 of *Lecture Notes in Computer Science*, pages 341–354. Springer-Verlag, December 1995.

[CP96] P.N. Creasy and H.A. Proper. A Generic Model for 3-Dimensional Conceptual Modelling. *Data & Knowledge Engineering*, 20(2):119–162, 1996.

[Dav90] A.M. Davis. *Software Requirements: Analysis & Specification*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[FL80] F. Flores and J.J. Ludlow. Doing and Speaking in the Office. In *Decision Support Systems: Issues and Challenges*. Pergamon, 1980.

[Gen87] H. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig, and G. Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, volume 254 of *Lecture Notes in Computer Science*, pages 207–247. Springer-Verlag, Berlin, Germany, 1987.

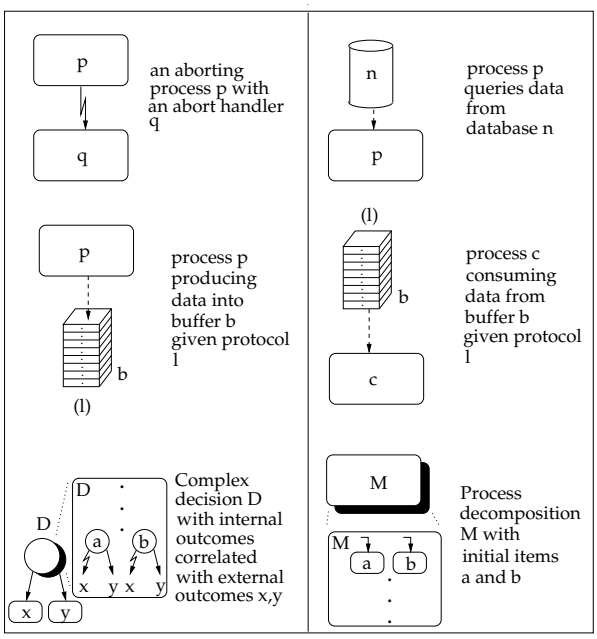

Figure 20: Common relationships between process modelling constructs

[GHS95] D. Georgakopoulos, M. Hornick, and A. Sheth. An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[Gri82] J.J. van Griethuysen, editor. *Concepts and Terminology for the Conceptual Schema and the Information Base*. Publ. nr. ISO/TC97/SC5-N695, 1982.

[GW96] R.J. van Glabbeek and W.P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, May 1996.

[HH97] J.W.G.M. Hubbers and A.H.M. ter Hofstede. Formalization of Communication and Behaviour in Object-Oriented Analysis. *Data & Knowledge Engineering*, 23(2):147–184, August 1997.

[HN93] A.H.M. ter Hofstede and E.R. Nieuwland. Task structure semantics through process algebra. *Software Engineering Journal*, 8(1):14–20, January 1993.

[Hoa78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[Hof93] A.H.M. ter Hofstede. *Information Modelling in Data Intensive Domains*. PhD thesis, University of Nijmegen, Nijmegen, The Netherlands, 1993.

[HPW93] A.H.M. ter Hofstede, H.A. Proper, and Th.P. van der Weide. Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7):489–523, October 1993.

[InC97] XSoft InConcert. *InConcert Technical Product Overview*. XSoft Inc, 1997.

[JCJO92] I. Jacobson, M. Christerson, M. Jonsson, and P. van Overgaard. *OO Software Engineering, A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusets, 1992.

[Jen91] K. Jensen. Coloured Petri Nets: A High Level Language for System Design and Analysis. In G. Rozenberg, editor, *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 342–416, Berlin, Germany, 1991. Springer-Verlag.

[KG94] S. King and R. Galliers. Modelling the CASE Process: empirical issues and future direction. *Information and Software Technology*, 36(10):587–596, 1994.

[Kob97] James G. Kobielus. *Workflow Strategies.* IDG Books Worldwide, Foster City, California, 1997.

[Lin95] DEC LinkWorks. *LinkWorks User Guide Version 3.0.* Digital Equipment Corporation, 1995.

[LR94] F. Leymann and D. Roller. Business Process Management with FlowMark. In *Proceedings of IEEE Compcon*, pages 1–21, March 1994.

[Mak96] P. Makey, editor. *Workflow: Integrating the Enterprise.* Report of the Butler Group, June 1996.

[McC92] S. McCready. There is more than one kind of work-flow software. *Computerworld*, November 1992.

[Mey90] B. Meyer. *Introduction to the Theory of Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[ML83] J.L. Malouin and M. Laundry. The Mirage of Universal Methods in Systems Design. *Journal of Applied Systems Analysis*, 10:47–62, 1983.

[Moh96] C. Mohan. State of the Art in Workflow Management System Research and Products. In *Proceedings of ACM SIGMOD/PODS 96 Joint Conference*, New York, New York, June 1996. ACM. Tutorial presentation.

[MWFF92] R. Medina-Mora, T. Winograd, R. Flores, and F. Flores. The ActionWorkflow Approach to Workflow Management Technology. In J. Turner and R. Kraut, editors, *Proceedings of the ACM 1992 Conference on Computer Supported Cooperative Work (CSCW)*, pages 281–288, Toronto, Canada, November 1992. ACM Press.

[Pet81] J.L. Peterson. *Petri Net Theory and the Modelling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Ram94] G.J. Ramackers. *Integrated Object Modelling, an Executable Specification Framework for Business Analysis and Information System Design.* PhD thesis, University of Leiden, Leiden, The Netherlands, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[Rei85] W. Reisig. *Petri Nets: An Introduction.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany, 1985.

[Rod91] T. Rodden. A survey of CSCW systems. *Interacting with Computers*, 3(3):319–353, 1991.

[RS94] M. Rusinkiewicz and A. Sheth. Specification and execution of transactional workflows. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press, Cambridge, Massachusetts, 1994.

[Vaa90] F.W. Vaandrager. Process Algebra Semantics of POOL. In J.C.M. Baeten, editor, *Applications of Process Algebra*, volume 17 of *Cambridge Tracts in Theoretical Computer Science*, pages 173–236. Cambridge University Press, Cambridge, United Kingdom, 1990.

[WMP+76] A. van Wijngaarden, B.J. Mailloux, J.E.L. Peck, C.H.A. Koster, M. Sintzoff, C.H. Lindsey, L.T. Meertens, and R.G. Fisker. *Revised Report on the Algorithmic Language ALGOL 68.* Springer-Verlag, Berlin, Germany, 1976.

[WW93] D.G. Wastell and P. White. Using process technology to support cooperative work: Prospects and design issues. In D. Daiper and C. Sanger, editors, *CSCW in Practice: An Introduction and Case Studies*, pages 105–126. Springer-Verlag, London, United Kingdom, 1993.

Contents