A Case Study of Workflow Reconfiguration: Design, Modelling, Analysis and Implementation

Faisal Abouzaid, Anirban Bhattacharyya, Nicola Dragoni, John S. Fitzgerald, Manuel Mazzara and Mu Zhou

TECHNICAL REPORT SERIES

No. CS-TR-1265 July 2011
A Case Study of Workflow Reconfiguration: Design, Modelling, Analysis and Implementation

F. Abouzaid, A. Bhattacharyya, N. Dragoni, J.S. Fitzgerald, M. Mazzara and M. Zhou

Abstract

This paper describes a case study involving interference between application activities and reconfiguration activities in an office workflow. We state the requirements on a system implementing the workflow and its reconfiguration, and describe the system's design in BPMN. We then use a number of computational formalisms of different kinds, including VDM and an asynchronous pi-calculus, to model the design and verify whether or not it will meet the requirements. In the process, we evaluate the formalisms for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. We include in the evaluation, two process algebras we have been developing, namely, Webpi and CCSdp. Finally, we give an implementation of the system in BPEL.
Abstract

This paper describes a case study involving interference between application activities and reconfiguration activities in an office workflow. We state the requirements on a system implementing the workflow and its reconfiguration, and describe the system's design in BPMN. We then use a number of computational formalisms of different kinds, including VDM and an asynchronous pi-calculus, to model the design and verify whether or not it will meet the requirements. In the process, we evaluate the formalisms for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. We include in the evaluation, two process algebras we have been developing, namely, Webpi and CCSdp. Finally, we give an implementation of the system in BPEL.

About the authors

Faisal Abouzaid obtained his 3rd cycle Doctorate from University of Bordeaux (France) and his PhD in 2010 at the Polytechnic School of Montreal (Canada). His thesis was based on Formal Verification of Web Services Orchestations. From 1990 to 2002 he was Professor of Computing Science and Director of the Computer Science Department at University of Casablanca (Morocco). Since 2004 he is a Research Associate at the CRAC laboratory of the Polytechnic School of Montreal.

Ani Bhattacharyya received a B.Sc. (Hons) in Mathematics from the University of London King's College in 1982, and an M.Sc. in Information Systems Engineering from South Bank Polytechnic in 1984. Ani subsequently worked on a variety of R & D projects: at MARI Advanced Microelectronics Ltd (1985 - 1989), he researched into the instrumentation of an object-oriented distributed system, specification of a distributed meta IPSE using Z, enterprise modelling, and a documentation system for IBCN software. At York University (1989 - 1992), he worked on a modelling framework for hard real-time systems. Ani subsequently developed a variety of database applications for accounting, HR and payroll. Ani is currently a PhD student in the School of Computing Science, Newcastle University under the supervision of Dr. John Fitzgerald. His research interests include modelling and verification of dynamically reconfigurable dependable real-time systems using pi-calculi and model checking.

Nicola Dragoni obtained a M.Sc. Degree and a Ph.D. in computer science, respectively in 2002 and 2006, both at University of Bologna, Italy. He visited the Knowledge Media Institute at the Open University (UK) in 2004 and the MIT Center for Collective Intelligence (USA) in 2006. In 2007 and 2008 he was post-doctoral research fellow at University of Trento, working on the S3MS project. Between 2005 and 2008 he also worked as freelance IT consultant. In 2009 he joined Technical University of Denmark (DTU) as assistant professor in security and distributed systems. Since May 2011 he has been associate professor at the same university.

John Fitzgerald is Reader in Computing Science and Deputy Director of the Centre for Software Reliability at Newcastle University. He is the Newcastle Principal Investigator in the DESTECS project, in which he leads work on methods for exploring design spaces for dependable embedded systems in terms of co-simulation. He is Chairman of Formal Methods Europe.

Manuel Mazzara achieved his Masters in 2002 and his PhD in 2006 at the University of Bologna. His thesis was based on Formal Methods for Web Services Composition. During 2000 he was a Technical Assistant at Computer Science Laboratories (Bologna, Italy). In 2003 he worked as Software Engineer at Microsoft (Redmond, USA). In 2004 and 2005 he worked as a freelance consultant and teacher in Italy. In 2006 he was an assistant professor at the University of Bolzano (Italy) and in 2007 a researcher and project manager at the Technical University of Vienna (Austria). Currently he is a Research Associate at the Newcastle University (UK) working on the DEPLOY project.

Mu Zhou did her Masters at Technical University of Denmark.
A Case Study of Workflow Reconfiguration: Design, Modelling, Analysis and Implementation

F. Abouzaid\textsuperscript{1}, A. Bhattacharyya\textsuperscript{2}, N. Dragoni\textsuperscript{3},
J. S. Fitzgerald\textsuperscript{2}, M. Mazzara\textsuperscript{2} and M. Zhou\textsuperscript{3}

\textsuperscript{1} CRAC Laboratory
Computer and Software Engineering Department
École Polytechnique de Montréal,
Montreal, Canada
m.abouzaid@polymtl.ca

\textsuperscript{2} Reconfiguration Interest Group
School of Computing Science
Newcastle University
Newcastle upon Tyne, UK
\{Anirban.Bhattacharyya, John.Fitzgerald, Manuel.Mazzara\}@ncl.ac.uk

\textsuperscript{3} Embedded Systems Engineering Section
DTU Informatics
Technical University of Denmark (DTU)
Copenhagen, Denmark
ndra@imm.dtu.dk, mu.zhou31@gmail.com

Abstract. This paper describes a case study involving interference between application activities and reconfiguration activities in an office workflow. We state the requirements on a system implementing the workflow and its reconfiguration, and describe the system’s design in BPMN. We then use a number of computational formalisms of different kinds, including VDM and an asynchronous $\pi$-calculus, to model the design and verify whether or not it will meet the requirements. In the process, we evaluate the formalisms for their suitability for the modelling and analysis of dynamic reconfiguration of dependable systems. We include in the evaluation, two process algebras we have been developing, namely, $\text{Web}\pi$ and $\text{CCS}^{dp}$. Finally, we give an implementation of the system in BPEL.

1 Introduction

Competition drives technological development, and the development of dependable systems is no exception. Thus, modern dependable systems are required to be more flexible, available and dependable than their predecessors, and dynamic reconfiguration is one way of achieving these requirements.

A significant amount of research has been performed on hardware reconfiguration (see [11] and [21]), but little has been done for reconfiguration of services, especially regarding computational models, formalisms and methods appropriate to the service domain. Furthermore, much of the current research assumes
that reconfiguration can be instantaneous, or that the environment can wait during reconfiguration for a service to become available. These assumptions are unrealistic in the service domain. For example, instantaneous mode change in a distributed system is generally not possible, because the system usually has no well-defined global state at a specific instant (due to communication delays). Also, waiting for the reconfiguration to complete is not acceptable if (as a result) the environment becomes dangerously unstable or the service provider loses revenue by the environment aborting the service request.

These observations lead to the conclusion that further research is required on dynamic reconfiguration of dependable services, and especially on its formal foundations, modelling and verification. In particular, the problem of interference between old configuration activities, new configuration activities and reconfiguration activities that occurs due to overlapping modes needs to be addressed. In a preliminary paper [33], we examined a number of well-known formalisms for their suitability for reconfigurable dependable systems. We observed that some reconfigurability features of \( \pi \)-calculi are appropriate for this purpose. However, although link passing models link reconfiguration well, it is clumsy in modelling process reconfiguration. Therefore, existing \( \pi \)-calculi are not entirely satisfactory for modelling system reconfiguration. In this paper, we take a deeper look at existing formalisms than was possible in [33]. We use a more complex case study involving the reconfiguration of an office workflow for order processing, define the requirements on a system implementing the workflow and its reconfiguration, and describe the design of a system in BPMN (see section 2). We then use formalisms selected from different categories to model the design and to verify whether or not the design will meet the reconfiguration requirements. We use VDM (a model-based formalism) in section 3; an asynchronous \( \pi \)-calculus, Web\( \pi \infty \) and CCS\( dp \) (process algebras) in section 4. We outline an implementation of the design in BPEL in section 5. Thus, the contribution of this paper is to identify strengths and weaknesses of the formalisms and their categories in the modelling and verification of reconfiguration requirements, discussed in section 6. This evaluation may be useful to system designers intending to use formalisms to design dynamically reconfigurable systems, and also to researchers intending to design better formalisms for the design of dynamically reconfigurable systems.

2 Office Workflow: Requirements and Design

This case study describes dynamic reconfiguration of an office workflow for order processing that is commonly found in large and medium-sized organizations [14]. These workflows typically handle large numbers of orders. Furthermore, the organizational environment of a workflow can change in structure, procedures, policies and legal obligations in a manner unforeseen by the original designers of the workflow. Therefore, it is necessary to support the unplanned change of these workflows. Furthermore, the state of an order in the old configuration may not correspond to any state of the order in the new configuration (as we shall see). These factors, taken in combination, imply that instantaneous reconfigu-
ration of a workflow is not always possible; neither is it practical to delay or abort large numbers of orders because the workflow is being reconfigured. The only other possibility is to allow overlapping modes for the workflow during its reconfiguration.

2.1 Requirements

A given organisation handles its orders from existing customers using a number of activities arranged according to the following procedure:

1. **Order Receipt**: an order for a product is received from a customer. The order includes customer identity and product identity information.
2. **Evaluation**: the product identity is used to perform an inventory check on the availability of the product. The customer identity is used to perform a credit check on the customer using an external service. If both the checks are positive, the order is accepted for processing; otherwise the order is rejected.
3. **Rejection**: if the order is rejected, a notification of rejection is sent to the customer and the workflow terminates.
4. If the order is to be processed, the following two activities are performed concurrently:
   (a) **Billing**: the customer is billed for the total cost of the goods ordered plus shipping costs.
   (b) **Shipping**: the goods are shipped to the customer.
5. **Archiving**: the order is archived for future reference.
6. **Confirmation**: a notification of successful completion of the order is sent to the customer.

In addition, for any given order, **Order Receipt** must precede **Evaluation**, which must precede **Rejection** or **Billing** and **Shipping**.

After some time, managers notice that lack of synchronisation between the Billing and Shipping activities is causing delays between the receipt of bills and the receipt of goods that are unacceptable to customers. Therefore, the managers decide to change the order processing procedure, so that **Billing** is performed before **Shipping** (instead of performing the two activities concurrently). During the transition interval from one procedure to the other, the following requirements must be met:

1. The result of the **Evaluation** activity for any given order should not be affected by the change in procedure.
2. All accepted orders must be billed and shipped exactly once, then archived, then confirmed.
3. All orders accepted after the change in procedure must be processed according to the new procedure.
2.2 Design

In this section we present a design of the office workflow case study by means of the Business Process Modeling Notation [8]. The choice of using BPMN as a design tool is motivated by its wide adoption as graphical representation for specifying business processes. Indeed, BPMN is currently maintained by the Object Management Group (OMG) [41], representing a standard for business process modeling.

![BPMN Diagram](image)

**Fig. 1.** Office workflow - BPMN diagram of the original configuration

The BPMN diagram representing the original configuration of the office workflow is shown in Figure 1. The diagram is based on six pools representing different functional entities. It is worth noting that this is not strictly imposed by our requirements. Indeed, only the credit check service is supposed to be external (Section 2.1). Thus, each other service might be included in a lane of a
single pool, representing in this way a specific activity within the organization. However, we decided to adopt a pool for each service to design a more generic situation where the different services are offered by external parties.

The coordinating entity is represented by the pool Office Workflow. When a customer’s request is received, an order is created by calling the Order Generator entity. This order is then sent to both the credit check handler (Credit Check pool) and the inventory check handler (Inventory Check pool). The former is used to check the customer credit’s availability, the latter to verify the availability of the product. Note that the inventory check is performed only in case the credit check is successful. This has been expressed by means of an Exclusive Data-Based Gateway. In case of a negative reply from Credit Check, a notification is sent to the customer, the order is rejected and the overall workflow terminates. We do the same with the results from Inventory Check: by means of an Exclusive Data-Based Gateway we proceed only in case of a positive reply. We notify the user and reject the order in case of a negative reply. Thus, according to the requirements, if both the checks are positive the order is processed; otherwise the order is rejected and the customer notified. The Bill&Ship pool represents the entity responsible for both the billing and shipping activities, which are represented as two different lanes (Bill and Ship, respectively) of this pool. Note that when the order is received by Bill&Ship, the two activities are called concurrently by means of a Parallel gateway. The same gateway is used to merge the results from Bill and Ship. The bill and ship details are then sent to the caller (Office Workflow) which, according to the requirements, calls the Archive service for storing the order. Finally, a successful notification is sent to the customer and the workflow terminates.

It is worth noting that, for the sake of simplicity and readability of the overall workflow, we assume that neither the billing activity nor the shipping activity provides a negative result. This explains why we do not check the results of such activities, for instance notifying the user in case of a negative result. Adding these further checks would be straightforward, since it could be done by means of two Exclusive Data-Based Gateways (as in the Office Workflow pool).

Let us now focus on the reconfiguration problem. The key change concerns the order of billing and shipping activities: instead of calling the two activities concurrently, the organization now requires that billing is performed before shipping. Looking at the design we have presented so far, it should be not so difficult to realize this reconfiguration requires a change in the main lane of Bill&Ship only (that is, where the billing and shipping activities are called, and therefore their invocations ordered), while the rest of the workflow remains unaltered. The resulting BPMN diagram is shown in Figure 2. Technically speaking, the Parallel gateways have been removed and the two activities are now called synchronously.

The key issue which remains unanswered is how the transition from the original configuration (Figure 1) to the new one (Figure 2) can be done. In other words, how the reconfiguration that we have applied can be designed. Figure 3 shows exactly this, i.e., the overall workflow during its reconfiguration. The basic idea is that we have a default flow that is exactly the same of the one of the
original configuration. This default flow can be altered through an interrupting message event contained in a "Determine configuration" activity, an activity that determines which configurations should be used when Bill&Ship is called. This activity has been included in a separate pool (Reconfig. region) to highlight where the flow can take one the two different directions. Moreover, in this way we represent a possible authority in charge of deciding the reconfiguration. Thus, if the interrupting event in the "Determine configuration" activity happens, this will affect the flow activating the new configuration instead of the original one.

In section 6.1 we will discuss how the modeling phase influences the design and we will explain the reasons behind the choice of this specific design. We will also show an alternative design and we will motivate why we found the alternative less suitable for this project.
3 Model-based Formalisms

Model-based formalisms are oriented towards the abstract description of data and functionality. The long tradition of methods in this category includes VDM [6],
Z [5], the B-method [3], and, lately, Event-B [4]. Models in these formalisms typi-
cally describe structured data in terms of abstract and unconstrained types
such as real numbers, arbitrarily large sets, sequences and mappings. From
these types, application-specific data types are defined, constrained by invari-
ants stated as logical predicates. Persistent data are described as state variables,
again restricted by invariants. Functionality is primarily defined in terms of op-
erations ("events" in Event-B) defined as relations over states. Abstraction in
operation definition is provided by loose specification, for example in terms of
postconditions. Restrictions on the use of operations are defined as guards or pre-
conditions. In Z, the emphasis is on formal specification, whilst the B-method
emphasizes the "method" itself. Both B and Event-B focus on the application
of stepwise refinement ("reification" in VDM). That is, the verifiable transfor-
mation of a high level formal specification into an executable program. In this
paper VDM will be used as a representative of this category.

3.1 VDM

The Vienna Development Method (VDM) [25, 16] is a collection of formal tech-
niques for the model-based specification and analysis of general computing sys-
tems. At its heart is an ISO Standardised Base Language for which denotational
semantics [24] and a proof theory [19] have been developed. The language has
been extended to support object-orientation [17] and aspects of real-time [45].

The VDM modelling language provides abstract base data types and collec-
tions, including finite but arbitrarily large sets, mappings and sequences. Type
membership may be restricted by arbitrarily complex invariant predicates. Per-
sistent state is modelled by state variables. Functionality is described primarily
by operations that may change the state. Operations may be specified implicitly
by predicates (postconditions) defining relations between "before" and "after"
states, or by means of explicit algorithms. Assumptions about the states in which
operations and functions may be applied are defined in preconditions over the
state variables and input parameters.

Each VDM model gives rise to proof obligations that may be discharged
in the proof theory. Aside from obligations ensuring internal consistency, such
as ensuring preconditions of partial operations are respected, include invariant
preservation, data and operation refinement.

VDM is supported by both commercial and open source tools (VDMTools [18]
and Overture [27] respectively). Both tools, in response to current industry needs,
provide strong support for static analysis and dynamic testing. Direct proof sup-
port has not yet been integrated, but is the subject of current study [46]. Industri-
al applications of VDM include applications as diverse as financial systems
and embedded processor firmware design [20]. Few, if any, applications have fo-
cussed on reconfiguration. However, the support of reconfiguration of threads
allocated to abstract CPUs is a topic of current work [40].

In spite of the "Method" in its name, VDM does not have a strong associ-
ated methodology. Indeed, the leitmotif of the formalism over the last decade
has been the provision of modelling methods and tools that are accessible to
practitioners and are well supported by robust tools. The industry applications have concentrated on the improvement of requirements and abstract designs by modelling, rather than on formal verification of implementations or refinement steps. We illustrate this kind of application in the remainder of this section.

3.2 Basic VDM Model of Order Processing Workflow

A classical VDM approach to the workflow discussed in our case study starts with the textual requirements stated in Section 2.1 and begins by identifying the purpose for which the model is being constructed. The explicit statement of a guiding purpose is essential, as it determines the model’s competence, and hence governs the abstraction decisions that may need to be made. We consider the purpose of the VDM model to be to identify and specify the services required in order to implement the normal and reconfigured workflows. The construction of such a first formal model tends to reveal areas of incompleteness in the informal requirements description.

In order to maintain an appropriate level of abstraction, we try not to prejudge the workflow design (shown in Section 2.2), but focus exclusively on the informal requirements description in Section 2.1. In approaching this set of requirements, we begin by identifying a system boundary. We consider that the workflow covers the functionality of a task that receives orders from outside the system, and choreographs the order processing. There are several forms of potential external interaction: inventory checking, stock checking, communicating outcomes to the customer, and archiving. We assume that inventory and credit checking functions are outside the system boundary, while the archiving function remains within it. Thus we do not model the functionality of stock and credit checking, treating these as underspecified external operations, but we do model the content of the archive. Communications with the customer are assumed to take place over an external medium over which we do not have control. We therefore record the sending of messages, but not their receipt. In practice, the assumptions made above would be made in collaboration with a real client.

A review of the requirements suggests that the model will need to describe orders and customers, maintaining data about each of these. We therefore begin by defining abstract data types to represent these identifiers:

\[
\text{CustId} = \text{token};
\]
\[
\text{OrderId} = \text{token};
\]

Requirement 1 indicates that orders come from customers. We introduce a composite data type to represent orders:

\[
\text{Order} :: \text{custid} : \text{CustId}
\]
\[
\text{inventoryOK} : [\text{bool}]
\]
\[
\text{creditOK} : [\text{bool}]
\]
\[
\text{accept} : [\text{bool}];
\]
For the moment, we just consider the custid (customer identifier) field. The remaining fields are used to record the order’s acceptance status (considered below). Each order contains the record of the sending customer. In order to relate order identifiers to the information contained on each order, we introduce a persistent state variable that maintains the mapping between them:

```plaintext
state Office of
orders : map OrderId to Order
...
end
```

Requirement 1 identifies the functionality needed to create an order within the bounds of the system. This is described by a simple operation specification:

```plaintext
ReceiveOrder(c:CustId)o:OrderId
post o not in set dom orders~ and
     orders = orders~ union {o |-> mk_Order(c,...)};
```

The operation, given a customer identifier, returns a new order identifier and adds a corresponding new order to the state. Note that, at this level of abstraction, the value of the order is not strictly required in the model, nor are the details of the product ordered.

Requirement 2 demands the performance of two external checks: one on the availability of stock and one on the availability of customer credit. It is not clear from the requirements whether these are performed by systems that are outside the boundary of the system under development, or not. If they are outside, they are modelled as nondeterministic operations. This is how we interpret the current requirements. We note that the BPMN design does record certain assumptions about the operation of these two checking tasks by modelling them as concurrent workflow streams, but this is minimal.

The inventoryOK and creditOK fields indicate the inventory and credit check outcomes respectively. They are set (arbitrarily, as far as the office workflow system is concerned), by the external EvaluateOrder operation.

```plaintext
EvaluateOrder(oid:OrderId)
ext wr orders
pre oid in set dom orders and
   orders(oid).inventoryOK = nil and
   orders(oid).creditOK = nil and
   orders(oid).accept = nil
post exists iOK, cOK : bool & orders = orders~ ++
   {oid |-> mu(orders(oid),
       inventoryOK |-> iOK, creditOK |-> cOK)};
```

Note this operation specification assumes that the evaluation process terminates in that the external checks both return results. We do not take account of the possibility that one or neither of the inventory or credit checks fails to return a conclusive value. This is likely to be an oversimplification, for example if the
item or customer identifiers are unknown to the external checks. The existential quantification in the postcondition introduces looseness into the specification, which is treated as nondeterminism. Further, the operation definition does not specify an order in which the evaluation tasks are to be executed.

We interpret the requirements as suggesting that acceptance or rejection may be handled as a separate operation from evaluation. We defined two separate accept and reject operations, allowing these to be performed as and when required, rather than defining a single operation that performs both together for a given order. We further extend the Order type with a field to record the accepted status:

```
Order :: custid : CustId
     inventoryOK : [bool]
     creditOK : [bool]
     accept : [bool];
```

The operations to manage acceptance and rejection are as follows:

AcceptOrder(oid:OrderId)

```text
ext wr orders
pre oid in set dom orders and
    orders(oid).accept = nil and
    orders(oid).inventoryOK <> nil and
    orders(oid).creditOK <> nil and
    orders(oid).inventoryOK and orders(oid).creditOK
post orders = orders~ ++
    {oid |-> mu(orders(oid), accept |-> true)};
```

RejectOrder(oid:OrderId)

```text
ext wr orders
pre oid in set dom orders and
    orders(oid).accept = nil and
    orders(oid).inventoryOK <> nil and
    orders(oid).creditOK <> nil and
    not orders(oid).inventoryOK or not orders(oid).creditOK
post orders = orders~ ++
    {oid |-> mu(orders(oid), accept |-> false)};
```

Both operations have preconditions requiring that the order is defined, and that the order has not yet been accepted/rejected. We therefore do not specify behaviour if the order is unknown or if the status of the order is to be changed. The requirements are silent on this point. There may be an argument for seeing the change of order status as a different operation anyway.

Requirement 3 indicates that, in the case of rejection, we need to inform the customer. It is not clear whether communications with the customer are or are not part of the system being modelled. We assume that this is the case. We introduce a record of customer communications by extending the state as follows:
The type `Msg` represents messages that can be sent to the customer:

```plaintext
Msg ::= oid : OrderId
     content : <REJECT> | <BILL> | <SHIPMENT> | <COMPLETION>;
```

All messages contain a field with the identifier of the order to which the message relates. The content of the message is modelled as an enumerated type. At present, we are only concerned with a rejection message, but we also include several other message types. We model the shipment of goods as a message. Note that the use of the set type in the `orders` state component indicates that we do not record whether more than one message of each type has been sent per order. This appears to be consistent with the requirements, in that the status of an order changes once instance of each message type has been sent, but again this would require clarification in a realistic development process. The operation to notify a client of rejection is defined thus:

```plaintext
NotifyRejectOrder(oid:OrderId)
```

```plaintext
pre oid in set dom orders and
not orders(oid).accept
post custcomms = custcomms~ munion
{orders(oid).custid ->
custcomms(orders(oid).custid) union
{mk_Msg(oid,<REJECT>)});
```

Requirement 4 indicates that the customer is to be billed for the order costs “plus shipping costs”. Up to this point, there has been no mention of the actual costs associated with an order, or of how shipping costs are to be determined. Since the declared purpose of our model is to analyse workflow rather than calculation, we deliberately omit from the model the process of cost calculation. We add a message type corresponding to a bill. The billing and shipping operations are as follows:

```plaintext
Bill(oid:OrderId)
```

```plaintext
pre oid in set dom orders and
orders(oid).accept <> nil and
orders(oid).accept and
mk_Msg(oid,<BILL>) not in set custcomms(orders(oid).custid)
post custcomms = custcomms~ munion
```
Ship(oid:OrderId)
  ext rd orders
  wr custcomms
  pre oid in set dom orders and
  orders(oid).accept <> nil and
  orders(oid).accept and
  mk_Msg(oid,<SHIPMENT>) not in set custcomms(orders(oid).custid)
  post custcomms = custcomms \union
  {orders(oid).custid |->
    custcomms(orders(oid).custid) union
    {mk_Msg(oid,<BILL>)}};

We have assumed that repeat billing is undesirable and so have added a precondi-
tion to the Bill operation. Similarly for shipping, we model the despatch of a
shipment as a single customer communication. For shipping, we believe that, re-
alistically, the issuing of a shipment will only take place after some sub-protocol
has been completed with the stores and shipper. However, we abstract from that
process.

Requirement 5 does not indicate what the conditions are on archiving. We
extend the state with archiving information. Orders are moved from the general
orders mapping to the archive once they are closed out.

state Office of
orders : map OrderId to Order
custcomms : map CustId to set of Msg
archive : map OrderId to Order
inv mk_Office(orders,-,archive) ==
  dom orders inter dom archive = {}
end

The state invariant records the constraint that an order can not be in both the
active orders set and the archive simultaneously. The operation to move an
order to the archive is as follows:

Archive(oid:OrderId)
  ext wr orders
  wr archive
  pre oid in set dom orders and
  (mk_Msg(oid,<BILL>)) in set custcomms(orders(oid).custid) and
  mk_Msg(oid,<SHIPMENT>) in set custcomms(orders(oid).custid))
or
  mk_Msg(oid,<REJECT>) in set custcomms(orders(oid).custid)
  post orders = {oid} <-: orders and
  archive = archive \union {oid |-> orders~(oid)};
Note the extensive precondition here, which records the constraint that, to be archived, an order must have been either billed and shipped, or else rejected. Again, these conditions need to be precisely clarified with a client.

Requirement 6 does not indicate what the criteria are for “successful completion” of an order. We assume that a completion message may be issued iff it has been billed and shipped and no previous completion message has been sent.

\[
\text{NotifySuccess}(oid:\text{OrderId})
\]
\[
\text{pre oid in set dom archive and}
\]
\[
\text{mk}_{\text{Msg}}(oid,\text{<BILL>}) \text{ in set custcomms(orders(oid).custid) and}
\]
\[
\text{mk}_{\text{Msg}}(oid,\text{<SHIPMENT>}) \text{ in set custcomms(orders(oid).custid) and}
\]
\[
\text{mk}_{\text{Msg}}(oid,\text{<COMPLETION>}) \text{ not in set custcomms(archive(oid).custid)}
\]
\[
\text{post custcomms = custcomms} \cup \{\text{orders(oid).custid} \rightarrow \text{custcomms(orders(oid).custid)} \cup \{\text{mk}_{\text{Msg}}(oid,\text{<COMPLETION>})\};
\]

There is an ordering constraint on the operations: order receipt before evaluation before rejection or billing and shipping. We argue that the ordering is respected by the preconditions of the operations. Evaluation is defined only if the order exists and the \text{inventoryOK}, \text{creditOK} and \text{accept} fields are all \text{nil}. The only way that nils are assigned to all these fields is through \text{ReceiveOrder}. So order receipt must occur before evaluation. Evaluation must occur before rejection because only evaluation is capable of setting the \text{inventoryOK} and \text{creditOK} fields to non-nil values. The precondition of \text{RejectOrder} requires them both to be set to Boolean values and so \text{RejectOrder} must occur after \text{AcceptOrder}. A similar argument applies to \text{AcceptOrder}. Only these operations can set the \text{accept} field of an order. A similar form of argument applies to the billing and shipping operations.

In the present model, there is no constraint on the ordering of Billing and Shipping. If we wish to incorporate an ordering constraint, as required for the reconfigured workflow, this is easily done by modifying \text{pre-Ship} to ensure that Billing must be performed first:

\[
\text{ConstrainedShip}(oid:\text{OrderId})
\]
\[
\text{ext rd orders}
\]
\[
\text{wr custcomms}
\]
\[
\text{pre oid in set dom orders and}
\]
\[
\text{orders(oid).accept} \neq \text{nil and}
\]
\[
\text{orders(oid).accept and}
\]
\[
\text{mk}_{\text{Msg}}(oid,\text{<BILL>}) \text{ in set custcomms(orders(oid).custid) and}
\]
\[
\text{mk}_{\text{Msg}}(oid,\text{<SHIPMENT>}) \text{ not in set custcomms(orders(oid).custid)}
\]
\[
\text{post custcomms = custcomms} \cup \{\text{orders(oid).custid} \rightarrow \text{custcomms(orders(oid).custid)} \cup
\]
\[
\{\text{mk}_{\text{Msg}}(oid,\text{<COMPLETION>})\};
\]
3.3 VDM: Workflow and Reconfiguration

The VDM-SL model presented so far does not have an explicit description of the workflow. Instead, it defines sufficient services to allow any workflow that respects the constraints outlined in the requirements. A specific workflow may be specified by means of an explicit operation specification that calls the operations currently specified implicitly. This can readily be specified in VDM-SL, and executed if the current implicit operation definitions are replaced by explicit ones.

The VDM model is essentially an abstract description of a sequential program to perform the workflow. For example, we might describe a sequential workflow by an operation with the general following form:

```
OrderWorkflow:() ==> ()
OrderWorkflow() == (
  dcl oid: OrderId;
  dcl c: CustId;
  oid := ReceiveOrder(c);
  EvaluateOrder(oid);
  if orders(oid).inventoryOK and orders(oid).creditOK
    then (Bill(oid); Ship(oid); Archive(oid); NotifySuccess(oid))
    else (RejectOrder(oid); NotifyRejectOrder(oid); Archive(oid))
```

The reconfiguration, if described in a model at this level, must be hard-coded as an alternative flow, as in the BPMN design. If we wish to build in a capacity for arbitrary dynamic reconfiguration, the model needs to step up a level to give a description of a run-time system for executing workflows in general.

4 Process Algebras

Model-based formalisms are mainly concerned with functional properties and sequential behavior. In contrast, process algebras are concerned with interaction between concurrent processes. Among the original methods in this field, we can mention CSP [22] and CCS [36]. Mobile process algebras (e.g. Milner’s π-calculus [38]) represent a further development by addressing mobility. In this paper we will use the π-calculus as a representative of this category. Furthermore we will analyze Webπ∞ and CCSdp, two novel process algebras that have been used to specifically address dynamic reconfiguration (among other things).

4.1 Asynchronous π-Calculus

The asynchronous π-calculus ([23], [7]) is a subset of Milner’s π-calculus [38], and it is known to be more suitable for distributed implementation. It is considered a rich paradigm for asynchronous communication, although it is not as expressive as Milner’s π-calculus in modelling mixed-choice constructs, such as $\pi.P + b.P'$ (see [43]).
Syntax and Semantics  We recall the (monadic) asynchronous π-calculus.

Let \( \mathcal{N} \) (ranged over by \( a, b, \ldots \)) be a set of names and \( \mathcal{V} \) a set of variables (ranged over by \( x, y, z, \ldots \)). The set of the asynchronous π-calculus processes (ranged over by \( P, Q, R, \ldots \)) is generated by the following grammar:

\[
P, Q, \ldots ::= \bar{x}z \mid G \mid P | Q \mid [a = b]P \mid (\nu x)P \mid A(x_1, \ldots, x_n)
\]

where guards \( G \), ranged over by \( G, H, \ldots \), are defined as follows:

\[
G, H, \ldots ::= 0 \mid x(y).P \mid \tau.P \mid G + G
\]

Intuitively, an output \( \bar{x}z \) represents a message \( z \) tagged with a name \( x \) indicating that it can be received (or consumed) by an input process \( x(y).P \) which behaves, upon receiving \( z \), as \( Pz/y \). Furthermore, \( x(y).P \) binds the names \( y \) in \( P \) and the restriction \( (\nu x)P \) declares a name \( x \) private to \( P \) and thus binds \( x \). Outputs are non-blocking.

The parallel composition \( P | Q \) means \( P \) and \( Q \) running in parallel. \( G + G \) is the non-deterministic choice that is restricted to \( \tau \) and input prefixes.

\([a = b]P \) behaves like \( P \) if \( a \) and \( b \) are identical.

\( A(y_1, \ldots, y_n) \) is an identifier (also call, or invocation) of arity \( n \). We assume that every such an identifier has a unique, possibly recursive, definition \( A(x_1, \ldots, x_n) \overset{\text{def}}{=} P \) where the \( x_i \)'s are pairwise distinct, and the intuition is that \( A(y_1, \ldots, y_n) \) behaves as its \( P \) with each \( y_i \) replacing \( x_i \). We shall presuppose finitely many such definitions.

Furthermore, for each \( A(x_1, \ldots, x_n) \overset{\text{def}}{=} P \) we require

\[
fn(P) \subseteq \{x_1, \ldots, x_n\}.
\]

We use the standard notations \( bn(P) \) for the set of bound names in \( P \), and \( fn(P) \) for the set of free names in \( P \). The set of names in \( P \) is defined as \( n(P) = fn(P) \cup bn(P) \).

We let \( \sigma, \upsilon, \ldots \) range over (non-capturing) substitutions of names on processes.

The structural congruence \( \equiv \) is the least equivalence relation on processes that is a congruence and satisfying the rules in Table 1.

A labeled transition system (see Table 2) is used to give an operational semantics for the π-calculus. The transition system is defined modulo α-equivalence on processes. Rules (Sum, Par, Com and Rec) have symmetric versions that are omitted.

Asynchronous π-Calculus The model in asynchronous π-calculus needs to keep the synchronization between actions in sequence coherent with the workflow definition. So sequence is implemented by using parallel composition with prefix and postfix on the same channel. Channel names are not restricted since the full system is not described here and has to be put in parallel with the detailed implementation of the environment process described (that will be here omitted).
\[ P[0] \equiv P \quad P[Q] \equiv Q[P] \quad P[(Q|R)] \equiv (P|Q)|R \]

\[ G + 0 \equiv G \quad G_1 + G_2 \equiv G_2 + G_1 \quad G_1 + (G_2 + G_3) \equiv (G_1 + G_2) + G_3 \]

\[ (\nu x)0 \equiv 0 \quad (\nu x)(\nu y)P \equiv (\nu y)(\nu x)P \quad (\nu x)(P|Q) \equiv (\nu x)|P \quad \text{if} \ x \notin fn(Q) \]

\[ A(y_1, ..., y_n) \equiv P[y_1, ..., y_n/x_1, ..., x_n] \quad \text{if} \ A(x_1, ..., x_n) \overset{\text{def}}{=} P \]

**Table 1.** \( \pi \)-calculus Structural Congruence.

<table>
<thead>
<tr>
<th>Out ( \pi = \pi_0 )</th>
<th>Tau ( \tau.P \overset{\text{out}}{\rightarrow} P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi = a(x), P \overset{a(u)}{\rightarrow} P'[u/x])</td>
<td>Res ( P \overset{\mu \in P'}{\rightarrow} P' [\mu] )</td>
</tr>
<tr>
<td>( P \overset{a^n}{\rightarrow} P' )</td>
<td>( a^u )</td>
</tr>
<tr>
<td>( (\nu a)P \overset{\pi}{\rightarrow} P' )</td>
<td>( (\nu a)P \overset{\pi}{\rightarrow} P' )</td>
</tr>
<tr>
<td>( G_1 + G_2 \overset{\alpha}{\rightarrow} G_1' )</td>
<td>Close ( P</td>
</tr>
<tr>
<td>( G_1 + G_2 \overset{\alpha}{\rightarrow} G_1' )</td>
<td>( G_1 + G_2 \overset{\alpha}{\rightarrow} G_1' )</td>
</tr>
<tr>
<td>( P_1 \overset{\alpha}{\rightarrow} P_1' )</td>
<td>( P_1 \overset{\alpha}{\rightarrow} P_1' )</td>
</tr>
<tr>
<td>( P_1 \overset{\alpha}{\rightarrow} P_1' )</td>
<td>( P_2 \overset{\alpha}{\rightarrow} P_2' )</td>
</tr>
<tr>
<td>( P_1 \overset{\alpha}{\rightarrow} P_1' )</td>
<td>( P_2 \overset{\alpha}{\rightarrow} P_2' )</td>
</tr>
<tr>
<td>Match ( \alpha = \alpha )</td>
<td>( \alpha = \alpha )</td>
</tr>
</tbody>
</table>

**Table 2.** Early Asynchronous Operational Semantics.

The entire model is expressed in asynchronous \( \pi \)-calculus as follows:

**Entire Model**

Let \( \text{params} = \{ \text{customer, item, Archive, ArchiveReply, Bill, BillReply, BillShip, Confirm, CreditCheck, CreditOk, CreditReject, InventoryCheck, InventoryOk, InventoryReject, OrderGenerator, OrderGeneratorReply, OrderReceipt, Reject, Ship, ShipReply, reco, recn} \} \)

We can define the \textit{Workflow} process as follows:

\[ \text{Workflow}(\text{params}) \overset{\Delta}{=} (\nu \text{order}) \text{OrderReceipt(customer, item)}. \text{OrderGenerator(customer, item)} | \text{OrderGeneratorReply(order)}. \text{CreditCheck(customer)} | (\text{creditOk()}, \text{InventoryCheck}(\text{item}) + \text{CreditReject}(), \text{Reject(order)}) \]

We can define the \textit{Workflow} process as follows:

\[ \text{Workflow}(\text{params}) \overset{\Delta}{=} (\nu \text{order}) \text{OrderReceipt(customer, item)}. \text{OrderGenerator(customer, item)} | \text{OrderGeneratorReply(order)}. \text{CreditCheck(customer)} | (\text{creditOk()}, \text{InventoryCheck}(\text{item}) + \text{CreditReject}(), \text{Reject(order)}) \]
In the model, the old region is identified as follows:

\[
\text{record}.\text{BillShip}().(\text{Billcust}\text{omer, item, order} | \text{Shipcustomer, item, order}) \\
| \text{BillReply(order)}.\text{ShipReply(order)}.\text{Archive order}
\]

And the new region is:

\[
\text{record}.\text{BillShip}().(\text{Billcustomer, item, order} | \text{Shipcustomer, item, order}) \\
| \text{BillReply(order)}.\text{Shipcustomer, item, order}) | \text{ShipReply(order)}.\text{Archive order}
\]

In the asynchronous $\pi$-calculus, two outputs cannot be in sequence. In order to impose ordering between $\text{Bill}$ and $\text{Ship}$, in the new region, it is necessary to put a guard on $\text{Ship}$, which requires enlarging the boundary of the old region to include the processes in the environment of the workflow that synchronize with $\text{Bill}$ and $\text{Ship}$. We did not model these processes because they are outside the system being designed, but the limitations of the asynchronous $\pi$-calculus imply that we must be able to access the logic of external services for which we know only the interfaces. For a description of this problem, please see [29].

The entire model represents a specific instance of the workflow that spawn concurrently another instance with fresh customer and item which here are assumed to be fresh names but in reality will be user entered (but it is not relevant to our purposes). We have to assume the existence of an “higher level” process (at the level of the BPEL engine) that activate the entire workflow and bounds the names that are free in the above $\pi$-calculus process. In this model channels $\text{creditOK, creditReject, InventoryOK}$ and $\text{InventoryReject}$ are used to receive the result of the credit check and inventory check, respectively. The old/new region is externally triggered using specific channels $\text{rec}_n$ and $\text{rec}_c$ chosen according to the value $x$ received on channel $\text{region}$:

\[
(\nu x)(\nu \text{region})\text{Workflow}(\text{params}) \mid \text{region}(x).([x = \text{new}]\text{rec}_n \mid [x = \text{old}]\text{rec}_c)
\]

In the following (see Section 4.2) we will show a more efficient solutions using a formalism tailored for reconfiguration.

**Verification** In $\pi$-calculi, the notion of bisimulation and congruence enables equational reasoning to be used. Thus, one may use some kind of equational theory based on functions, as in the $\pi$-applied calculus [1]. So, we may use the ProVerif tool (for example). One may also use the $\pi$-logic to specify desired properties and then verify them with the HAL Toolkit model-checker [15].
Syntax of the π-logic The π-logic has been introduced to specify the behavior of systems written in π-calculus in a formal and unambiguous manner. This logic has been introduced in [15] to express temporal properties of π-processes. The logic integrates modalities defined by Milner ([39]) with EFφ and EF{χ}φ modalities on possible future. The π-logic syntax is:

\[ \phi ::= \text{true} \mid \sim \phi \mid \phi \land \phi' \mid EX{\mu}\phi \mid EF\phi \mid EF\{\chi\}\phi \]

where \( \mu \) is a π-calculus action and \( \chi \) could be \( \mu \), \( \sim \mu \), or \( \bigvee_{i \in I} \mu_i \) and where \( I \) is a finite set.

Semantics of π-formulae is given below:

- \( P \models \text{true} \) for any process \( P \);
- \( P \models \sim \phi \) iff \( P \not\models \phi \);
- \( P \models \phi \land \phi' \) iff \( P \models \phi \) and \( P \models \phi' \);
- \( P \models EX{\mu}\phi \) iff there exists \( P' \) such as \( P \xrightarrow{\mu} P' \) and \( P' \models \phi \) (strong next);
- \( P \models EF\phi \) iff there exists \( P_0, \ldots, P_n \) and \( \mu_1, \ldots, \mu_n \), with \( n \geq 0 \), such as \( P = P_0 \xrightarrow{\mu_1} P_1 \cdots \xrightarrow{\mu_n} P_n \) and \( P_n \models \phi \). The meaning of \( EF\phi \) is that \( \phi \) must be true sometimes in a possible future.
- \( P \models EF\{\chi\}\phi \) if and only if there exists \( P_0, \ldots, P_n \) and \( \nu_1, \ldots, \nu_n \), with \( n \geq 0 \), such that \( P = P_0 \xrightarrow{\nu_1} P_1 \cdots \xrightarrow{\nu_n} P_n \) and \( P_n \models \phi \) with:
  - \( \chi = \mu \) for all \( 1 \leq j \leq n \), \( \nu_j = \mu \) or \( \nu_j = \tau \);
  - \( \chi = \sim \mu \) for all \( 1 \leq j \leq n \), \( \nu_j \neq \mu \) or \( \nu_j = \tau \);
  - \( \chi = \bigvee_{i \in I} \mu_i \): for all \( 1 \leq j \leq n \), \( \nu_j = \mu_i \) for some \( i \in I \) or \( \nu_j = \tau \).

The meaning of \( EF\{\chi\}\phi \) is that the truth of \( \phi \) must be preceded by the occurrence of a sequence of actions \( \chi \).

Some useful dual operators are defined as usual: \( \phi \lor \phi \), \( AX{\mu}\phi \), \( < \mu > \phi \) (weak next), \([\mu]\phi \) (Dual of weak next), \( AG\phi \) (\( AG\{\chi\} \)) (always).

Properties of the dynamic reconfiguration model

In the context of the dynamic reconfiguration, we need to verify that during the transition interval things continue to unfold normally, as stated in section 2. For this purpose we need to formally express, if possible, the desirable properties using the π-logic. Examples of desirable properties that may illustrate the dynamic reconfiguration capabilities of the π-calculus could be the requirements presented in section 2.1:

The result of the Evaluation activity for any given order should not be affected by the change in procedure. The following formula means whatever the chosen path (old or new region), an order will be billed, shipped and archived or refused:

\[
AG\{EF\{OrderReceipt()\}\text{true}\}
AG\{EF\{Bill\ customer,\ item,\ order\}\text{true} \land EF\{Ship\ customer,\ item,\ order\}\text{true} \land EF\{Archive\ order\}\text{true}\} \lor EF\{Reject\}\text{true}\]
All accepted orders must be billed and shipped exactly once, then archived, then confirmed. The following formula means that after an order is billed and shipped, it is archived and confirmed, and cannot be billed nor shipped again:

\[ AG\{EF\{BillShip()\}\text{true}\} \]
\[ AG\{EF\{Bill\ customer,\ item,\ order\}\text{true} \land EF\{\overline{Ship\ customer,\ item,\ order}\}\text{true} \land EF\{Archive\ order\}\text{true}\} \land EF\{Confirm\ order\}\text{true}\} \]
\[ AG\{\{Bill\ customer,\ item,\ order\}\text{false} \land \{Ship\ customer,\ item,\ order\}\text{false}\} \]

All orders accepted after the change in procedure must be processed according to the new procedure We can express in the \( \pi \)-logic the following requirement: “after a reception on the channel \( rec_n \), no other reception on channel \( rec_0 \) will be accepted”. This meets the desired requirement since it is obvious from the model that, if a signal is received on channel \( rec_n \), the order will be processed according to the new procedure.

\[ AG\{\{rec_n()\}\text{true} AG\{rec_0()\}\text{false}\} \]

However, since the choice between the old procedure and the new one is nondeterministic, this formula will not be true, although it is an essential requirement for the model. This result illustrates the difficulty of the asynchronous \( \pi \)-calculus to model the dynamic reconfiguration properly. A first attempt to answer this problem is presented in the next section.

Discussion of the model in asynchronous \( \pi \)-calculus Since the reconfiguration has to be triggered from outside the process, the asynchronous \( \pi \)-calculus, but more generally the \( \pi \)-calculus, shows its weakness in term of reconfiguring processes dynamically. Moreover, some relevant properties could not be expressed until we include some timed extensions.

The HAL-Toolkit

The verification environment, HD Automata Laboratory (HAL) \(^4\) [15], allows for the finite state verification of systems specified in the \( \pi \)-calculus. It is based on the theory of History Dependent automata (HD-automata) which peculiarity resides in the fact that states and transitions are equipped with names allowing thus one to model explicitly name creation/deallocation, and name extrusion. In the HAL-Toolkit, HD-automata are translated into ordinary automata that are handled by the JACK verification environment. JACK permits to calculate behavioural equivalences, and supports verification by model checking of properties expressed as formulae written in the \( \pi \)-logic.

The HAL Toolkit can be used as a Web application\(^5\) for simple verification or as a standalone application.

\(^4\) http://matrix.iei.pi.cnr.it:80810/hal/
\(^5\) http://matrix.iei.pi.cnr.it:8080/hal/bin/HALOnLine/
4.2 Web$\pi_\infty$

Web$\pi_\infty$ is a conservative extension of the $\pi$-calculus developed for modelling and analysis purposes in the context of Web Services and Service Oriented Architectures. The basic theory has been mainly developed in [35] and [32] while its applicability has been shown in other works: [29] gives a BPEL semantics in term of Web$\pi_\infty$, [13] clarifies some aspects of the Recovery Framework of BPEL and [34] exploits a web transaction case study.

In this paper we will use this formalism to model the workflow case study. Verification issues will stem form this formalization and we will discuss what is doable with Web$\pi_\infty$ and what is not. In the conclusion section the effectiveness of this formalism will be compared with the other ones presented in this work.

The work previously done on Web$\pi_\infty$ is vast, as mentioned above where some of the papers have been referenced. For this reason, it would take too much space to recall entirely the language, its theory and its applications. Due to the nature of this work, which includes also some of the aspects of a survey, we strongly encourage the reader to look at least at [35]. However, for a quicker reference, we report here syntax, structural congruence and reduction semantics of Web$\pi_\infty$ to simplify the understanding of the model shown below.

**Syntax and Semantics** The syntax of web$\pi_\infty$ processes relies on a countable set of names, ranged over by $x, y, z, u, \cdots$. Tuples of names are written $\tilde{u}$. We intend $i \in I$ with $I$ a finite non-empty set of indexes.

$$P ::= 0 \quad (\text{nil})$$
$$| \pi \tilde{u} \quad (\text{output})$$
$$| \sum_{i \in I} x_i(\tilde{u}).P_i \quad (\text{alternative composition})$$
$$| (x)P \quad (\text{restriction})$$
$$| P | P \quad (\text{parallel composition})$$
$$| !x(\tilde{u}).P \quad (\text{guarded replication})$$
$$| \{P : P\}_x \quad (\text{workunit})$$

A process can be the inert process $0$, an output $\pi \tilde{u}$ sent on a name $x$ that carries a tuple of names $\tilde{u}$, an alternative composition consisting of input guarded processes that consumes a message $\pi\tilde{w}$ and behaves like $P_i\{\tilde{w}/\tilde{u}\}$, a restriction $(x)P$ that behaves as $P$ except that inputs and messages on $x$ are prohibited, a parallel composition of processes, a replicated input $!x(\tilde{u}).P$ that consumes a message $\pi\tilde{w}$ and behaves like $P\{\tilde{w}/\tilde{u}\} |x(\tilde{u}).P$, or a workunit $\{P : Q\}_x$ that behaves as the body $P$ until an abort $\pi$ is received and then behaves as the event handler $Q$. Names $x$ in outputs, inputs, and replicated inputs are called subjects.
of outputs, inputs, and replicated inputs, respectively. It is worth to notice that the syntax of \texttt{webπ}_∞ processes simply augments the asynchronous \( \pi \)-calculus with workunit process. The input \( x(u).P \), restriction \( (x)P \) and replicated input \( !x(u).P \) are binders of names \( u, x \) and \( u \) respectively. The scope of these binders is the process \( P \). We use the standard notions of \( \alpha \)-equivalence, \textit{free} and \textit{bound names} of processes, noted \( \text{fn}(P) \), \( \text{bn}(P) \) respectively.

We give the semantics for the language in two steps, following the approach of Milner [37], separating the laws that govern the static relations between processes from the laws that rule their interactions. The first step is defining a static structural congruence relation over syntactic processes. A structural congruence relation for processes equates all agents we do not want to distinguish. It is introduced as a small collection of axioms that allow minor manipulation on the processes’ structure. This relation is intended to express some intrinsic meanings of the operators, for example the fact that parallel is commutative. The second step is defining the way in which processes evolve dynamically by means of an operational semantics. This way we simplify the statement of the semantics just closing with respect to \( \equiv \), i.e. closing under process order manipulation induced by structural congruence.

\textbf{Definition 1.} \textit{The structural congruence} \( \equiv \) \textit{is the least congruence satisfying the Abelian Monoid laws for parallel and summation (associativity, commutativity and 0 as identity) closed with respect to \( \alpha \)-renaming and the following axioms:}

\begin{enumerate}
\item \textit{Scope laws:}
\begin{equation*}
(u)0 \equiv 0, \quad (u)(v)P \equiv (v)(u)P, \\
\langle (u)Q \rangle_x \equiv (u)\langle (P | Q) \rangle_x, \quad \text{if } u \not\in \text{fn}(P)
\end{equation*}
\item \textit{Workunit laws:}
\begin{equation*}
\langle 0 ; Q \rangle_x \equiv 0 \\
\langle P ; Q \rangle_x | R \equiv R | \langle P ; Q \rangle_x
\end{equation*}
\item \textit{Floating law:}
\begin{equation*}
\langle z \tilde{u} \rangle P \equiv z \tilde{u} \langle P \rangle_x
\end{equation*}
\end{enumerate}

The scope laws are standard while novelties regard workunit and floating laws. The law \( \langle 0 ; Q \rangle_x \equiv 0 \) defines committed workunit, namely workunit with 0 as body. These ones, being committed, are equivalent to 0 and, therefore, cannot fail anymore. The law \( \langle \langle P ; Q \rangle_y | R ; R' \rangle_x \equiv \langle P ; Q \rangle_y | \langle R ; R' \rangle_x \) moves workunit outside parents, thus flattening the nesting. Notwithstanding this flattening, parent workunits may still affect the children ones by means of names. The law \( \langle z \tilde{u} \rangle P ; Q \rangle_x \equiv z \tilde{u} \langle P ; Q \rangle_x \) floats messages outside workunit boundaries. By this law, messages are particles that independently move towards their inputs. The intended semantics is the following: if a process emits a message,
this message traverses the surrounding workunit boundaries until it reaches the
 corresponding input. In case an outer workunit fails, recoveries for this message
 may be detailed inside the handler processes.

The dynamic behavior of processes is defined by the reduction relation where
we use the shortcut:

\[ \langle P ; Q \rangle \overset{\text{def}}{=} (z) \langle P ; Q \rangle_z \text{ where } z \not\in \text{fn}(P) \cup \text{fn}(Q) \]

**Definition 2.** The reduction relation \( \rightarrow \) is the least relation satisfying the fol-
lowing axioms and rules, and closed with respect to \( \equiv \), \( (x)_x \), \( x \), \( \langle \cdot ; Q \rangle \), and \( \langle \cdot ; Q \rangle_z \):\

\[ \text{(com)} \]
\[ \pi \overline{v} | \sum_{i \in I} x_i(\overline{u}_i).P_i \rightarrow P_i^{(\overline{v}/\overline{u}_i)} \]

\[ \text{(rep)} \]
\[ \pi \overline{v} | !x(\overline{u}).P \rightarrow P^{(\overline{v}/\overline{u})} | !x(\overline{u}).P \]

\[ \text{(fail)} \]
\[ \pi | \prod_{i \in J} x_i(\overline{u}_i).P_i | \prod_{j \in J} !x_j(\overline{u}_j).P_j ; Q \rangle_z \rightarrow \langle Q ; \emptyset \rangle \]

where \( J \neq \emptyset \lor (I \neq \emptyset \land S \neq \emptyset) \)

Rules (com) and (rep) are standard in process calculi and models input-
output interaction and lazy replication. Rule (fail) models workunit failures:
when a unit abort (a message on a unit name) is emitted, the corresponding
body is terminated and the handler activated. On the contrary, aborts are not
possible if the transaction is already terminated (namely every thread in the
body has completed its own work), for this reason we close the workunit restricting
its name.

**The model in Webπ∞** Since Webπ∞ has been used to encode WS-BPEL (see
[29]) it is no surprise that its basic structure has been partly inspired by this com-
position language and that model and implementation can look somehow similar.
However, the fundamental basic idea behind Webπ∞ was more profound than
just mimicking BPEL itself. In [35] one of the authors of this paper exploited
Webπ∞ to precisely formalize a simplification of the BPEL Recovery Frame-
work unifying all the mechanisms (fault, compensation and event handling) into
a single one.

For the modeling purposes of this work, the ideas of workunit and event han-
der model turn out to be particularly useful and practical. Webπ∞ uses the mechanism of workunit to bound the identified (old and new) regions and the event raising is
exploited to operate the non immediate change (reconfiguration). The model
can be expressed as follows (as a shortcut we will use here process invocation):
Here is where Webπ∞ shows a subtle feature which turn out to be very important for modeling a reconfigurable system. Since the floating laws of structural congruence of Webπ∞ (definition 1) allow the asynchronous outputs in a workunit to freely escape the workunit itself, what happens is that, once the region to reconfigure has been already entered and the BillShip has been triggered, Bill customer, item, order and Ship customer, item, order will be not constrained inside the workunit and will not be killed by any incoming rec signal intended to change the old region with the new one. This means that, once the region has been entered by an order, that order will go through without being interrupted by reconfiguration events and the old order will be processed according to the old procedure, not the new one. Future orders will find instead only the new procedure Workflow_n waiting for orders, and the change is not immediate:

\[
\text{Workflow}_n(\text{customer}, \text{item}) \triangleq (\nu \text{order})(\nu \text{r}_c)(\nu \text{r}_i) \text{OrderReceipt}(\text{customer}, \text{item}) \text{OrderGenerator customer, item} \\
\text{| OrderGeneratorReply(order), CreditCheck customer} \\
\text{| (CreditCheckReply(order), InventoryCheck item}} \\
\text{+ CreditCheckReply_j(order), Reject order) \\
\text{| (InventoryCheckReply_l(order), BillShip) \\
\text{+ InventoryCheckReply_l(order), Reject order) \\
\text{| (BillShip(), Bill customer, item, order | Ship customer, item, order} \\
\text{| (\nu \text{customer})(\nu \text{item}) \text{Workflow}(\text{customer}, \text{item}))} \\
\text{; (\nu \text{customer})(\nu \text{item}) \text{Workflow}_n(\text{customer}, \text{item})}, \text{rec} \\
\text{| BillReply(order), ShipReply(order), Archive order} \\
\text{| ArchiveReply(order), Confirm order}
\]

Like for the π-calculus model we have to assume the existence of an “higher level” process (at the level of the BPEL engine) that activate the entire workflow and bounds the names that are free in the above π - calculus process:

\[
(\nu \text{customer})(\nu \text{item})(\nu t)(\nu f)(\nu \text{rec}) \text{Workflow}(\text{customer}, \text{item}) | \text{rec}
\]

This process is also responsible for triggering the reconfiguration.

**Verification in Webπ∞** Analysis in Webπ∞ is done in terms of equational reasoning, in [35] and [32] the full theory has been thoroughly developed. Inter-
ested readers may find all the definitions and proofs with an extensive explanation for the extensional semantics, the notions of barb, process contexts and barbed bisimulation in [32]. Definitions for Labelled Semantics, asynchronous bisimulation, labelled bisimilarity and the proof that it is a congruence are also present. Finally, results relating barbed bisimulation and asynchronous labeled bisimulation as well as many examples are discussed. A core BPEL is encoded in \texttt{webπ∞} and a few properties connected to this encoding are proved for it. At the moment, one severe weakness of Webπ∞ is the lack of tool support, i.e. automatic system verification. That means that any kind of verification is only possibly by hands, limiting its effectiveness and efficiency. However, it is clearly possible encoding Webπ∞ into the π-calculus being the only technical complication the encoding of the workunit and its asynchronous interrupt. The basic idea behind the encoding is the use of summation to unfold, at each step, all the possible traces of execution (normal or “abnormal” trace). It is not difficult to imagine how impractical would be to model such behavior in terms of basic asynchronous π-calculus. Once a compilation into the π-calculus is done, we can proceed in many ways, for example using HAL as detailed in this paper. The overall problem should be clear now: from one side Webπ∞ simplifies the modelling of dependable systems expressing with one additional constructs the recovery behavior, on the other makes the verification more difficult. Luckily, there is a tradeoff. If we chose Webπ∞ as the modelling language and the π-calculus as the intermediate language, a sort of verification bytecode, we can offer a practical modelling suite to the designer and still use the tool support for the π-calculus. Although this plan has been not implemented so far and it is not in the scope of this paper, it is clearly a desirable plan for future work.

In this section we will now discuss in detail all the three requirements which have to be met during and after the reconfiguration phase. We will discuss the requirements in terms of equational reasoning as explained in [35] and [32] for smaller examples. The case study of this paper is interesting at showing both the modelling power of Webπ∞ and, unfortunately, the weaknesses of its reasoning system. This issues will be discussed in more detail in Section 6.

The result of the Evaluation activity for any given order should not be affected by the change in procedure. The change in procedure in the Webπ∞ model consists in triggering the \texttt{rec} channel which, in turn, spawn the workunit handler. The handler generates new names and then activates a new instance of the workflow based on the new procedure scheme, here called \texttt{workflown}. As the reader can see following the model, the acceptability of an order (analyzed in the Evaluation activity) is not based on a computation performed in the model itself but it is delegated to external processes which interact with the main workflow through the channels representing the interface with the environment: \texttt{CreditCheck}, \texttt{CreditCheckReply}, \texttt{CreditCheckReplyf}, \texttt{InventoryCheck}, \texttt{InventoryCheckReply}, \texttt{InventoryCheckReplyf}.

Since the new procedure \texttt{workflown} does not affect those other processes there is no possibility for the change to alter the behavior of the inventory and
credit check. Furthermore, the region of the model where the interaction with these external processes is performed is outside the region to be reconfigured. That means that this part of the model in the old procedure workflow is exactly a verbatim copy of the same part in the new procedure workflow, i.e. the checks are performed in the same exact order. In conclusion, the change in procedure cannot alter in any way the acceptability of an order being this acceptability performed through unaltered and unmodelled external checks that are still called in the same exact order. More precisely, this requirement is not related to the model, i.e. it is about aspects of the workflow that have not been modeled. The reasons for this modelling approach are detailed in section 4.1.

We can formally express, in term of equational reasoning, that the checking part of the model in the old procedure workflow is exactly a verbatim copy of the same part in the new procedure workflow as follows:

\[(\nu \text{order})(\nu \text{rc})(\nu \text{ri}) \text{OrderReceipt}(\text{customer}, \text{item}) \mid \text{OrderGenerator} \text{customer}, \text{item} \mid \text{CreditCheck} \text{customer} \mid (\text{CreditCheckReply}(\text{order}), \text{InventoryCheck} \text{item} + \text{CreditCheckReply}(\text{order}), \text{Reject} \text{order}) \mid (\text{InventoryCheckReply}(\text{order}), \text{BillShip} + \text{InventoryCheckReply}(\text{order}), \text{Reject} \text{order}) \approx_n \]

where the symbol \(\approx_n\) has to be intended as a binary relation representing the fact that the process on its left and the one on its right exhibit the same behavior. Details of this definition are in [35] and [32]. This statement is trivially true exactly because the two processes are syntactically equivalent and, as a consequence, semantically equivalent.

All accepted orders must be billed and shipped, then archived. While the first requirement is about the Evaluation activity, the second is about the Billing, Shipping and Archiving activities. Therefore, the second part of the workflow can be independently analyzed provided an order has successfully passed the Evaluation activity. There are actually two possibilities for an order to be processed: either it falls in the old procedure or it falls in the one. If we isolate those parts of the modeling related to the two procedures we obtain:

**Old Configuration**

\[\langle \text{BillShip}(\nu \text{order}).(\text{Bill} \text{customer}, \text{item}, \text{order} | \text{Ship} \text{customer}, \text{item}, \text{order} \mid (\nu \text{order})(\nu \text{item}) \text{Workflow}(\text{customer}, \text{item})) ; (\nu \text{order})(\nu \text{item}) \text{Workflow}_n(\text{customer}, \text{item})) \rangle_{\text{rec}} \mid \text{BillReply}(\text{order}), \text{ShipReply}(\text{order}), \text{Archive} \text{order} \mid \text{ArchiveReply}(\text{order}), \text{Confirm} \text{order} \]
New Configuration

\[BillShip), (\overline{Bill} customer, item, order | BillReply(order) \overline{Ship} customer, item, order) \]
\[| ShipReply(order) Archive order | ArchiveReply(order) Confirm order \]
\[| (\nu \text{ customer}) (\nu \text{ item}) \text{Workflow (customer, item)} \]

As discussed when the model has been shown first, in the case of the old configuration (constrained inside the workunit) once the region has been entered by an order, that order will go through without being interrupted by reconfiguration events and the order will be processed according to the old procedure, not the new one. This means that the presence of a workunit does not affect how the order itself is processed. The workflow of actions described by the requirement can be formally expressed as follow:

\[(\nu x)(\nu y) \ (\overline{Bill} customer, item, order | \overline{Ship} customer, item, order \overline{BillReply} (order), \overline{ShipReply} (order), \overline{Archive} order | \overline{ArchiveReply} (order), \overline{Confirm} order) \]

In plain words this process describes billing and shipping happening in any order but both before the archiving. The channels \(x\) and \(y\) are there precisely to work as a joint for billing and shipping. If we want to express the requirements in term of equational reasoning, we might require that both the old and the new procedure are bisimilar with the above process in the terms explained in the discussion of the first requirement. However, this would be too strict since the above process allows a set of traces which is a superset of both the set of traces of the old configuration and the new one. So in this case we talk about one process being similar (but not bisimilar) to the other.

All orders accepted after the change in procedure must be processed according to the new procedure This point has been already partly discussed when the model has been previously presented. However, the emphasis there was on the subtlety of that Webπ∞ feature turning out to be important for modeling reconfigurable systems, i.e. the mechanism behind the workunit. We need to make the point clearer here. As already explained before, the change in procedure is here modeled by triggering the \(rec\) channel and spawning the workunit handler. The handler then activates a new instance of the workflow based on the new procedure scheme which has been called \(\text{workflow}_n\). The floating laws of structural congruence of Webπ∞ (definition 1) allow the asynchronous outputs in a workunit to freely escape the workunit itself. Thus, once the region to reconfigure has been already entered and the \(BillShip\) has been triggered, \(Bill customer, item, order\) and \(Ship customer, item, order\) will not be killed by any incoming \(rec\) signal. Shortly, once the region has been entered by an order, that order will be not interrupted by reconfiguration events so that old order will be processed according to the old procedure and not the new one. More precisely, here the requirement is to have all orders accepted after the change being processed according to the new procedure. However, this is just a consequence of the analysis we have just done since, once the invocation of \(\text{Workflow}\)
has been killed by the message rec, only invocations of Workflow\textsubscript{n} (the new procedure) will be possible. So future orders will be processed only according to new procedure.

4.3 CCS\textsubscript{dp}

The Calculus of Communicating Systems with dynamic process reconfiguration (CCS\textsubscript{dp}) is a process algebra based on CCS [36]. It is being developed for the purpose of modelling and analysis of dynamic reconfiguration of dependable systems, in which application and reconfiguration activities can interfere. CCS\textsubscript{dp} extends CCS with a single process construct (i.e. the fraction process \(\frac{P}{P'}\)), so that both planned and unplanned replacements of a process can be modelled simply and without the use of fictitious process behaviour. In this paper, for reasons of simplicity, we use the basic version of CCS\textsubscript{dp} in which process actions do not pass values.

**Syntax and Semantics** Let \(P\) be the set of processes in basic CCS\textsubscript{dp}. Let \(N\) be the countable set of input port/action names (e.g. \(a, b, c\)) of the processes in \(P\). Let \(\overline{N}\) be the countable set of complementary output port/action names (e.g. \(\overline{a}, \overline{b}, \overline{c}\)) of the processes in \(P\). Let \(PN\) be the countable set of names (e.g. \(A, B, C\)) of the processes in \(P\). The sets \(N, \overline{N}\) and \(PN\) are assumed to be pairwise disjoint.

The syntax of a process \(P\) in \(P\) is defined as follows:

\[
P ::= PN<\bar{\beta}> \mid M \mid P | P \mid (\nu\bar{\beta})P \mid \frac{P}{P'}
\]

\[
M ::= 0 \mid \alpha.P \mid M + M
\]

where \(PN \in PN\), \(\bar{\beta}\) is a tuple of \(\beta\)-values and \(\beta \in N \cup \overline{N}\), and \(\alpha \in N \cup \overline{N} \cup \{\tau\}\).

As in CCS, 0 is the NIL process, which has no behaviour.

\(\alpha.P\) models sequential action, with \(\alpha \in N\) representing the input action on the input port \(\alpha\) of a process, \(\overline{\alpha} \in \overline{N}\) representing the complementary output action on the output port \(\overline{\alpha}\) of a process, and \(\tau\) representing the internal action of a process.

Summation (e.g. \(M + M'\)) models non-deterministic choice of actions by a process.

\(A<\bar{\beta}>\) models the invocation of a constant process named \(A\), instantiated with a tuple of port/action names \(\bar{\beta}\). \(A(\bar{\beta})\) has a unique definition, which can be recursive.

Parallel composition \((P | P')\) models the execution of concurrent processes and their interaction. Interaction between processes is synchronous and point-to-point.

\((\nu\bar{\beta})P\) models restriction of the scope of a tuple of port/action names \(\bar{\beta}\) to a process \(P\).
fraction \( P' \) models process replacement and deletion. On creation, the fraction \( P' \) identifies any instance of a process matching its denominator process \( P \) with which it is composed in parallel, and replaces that process atomically with the numerator process \( P' \). If no such process instance exists, the fraction continues to exist until such a process is created (or the fraction is itself deleted or replaced). If there is more than one such process instance, a non-deterministic choice is made as to which process is replaced. Similarly, if more than one fraction can replace a process instance, a non-deterministic choice is made as to which fraction replaces the process. Deletion of a process \( P \) is achieved by parallel composition with \( 0 \). If \( P \) progresses to \( Q \), then \( P' \) will not replace \( Q \) by \( P' \) (unless \( Q \) matches \( P \)). Notice that a fraction process has no communication behaviour; its only behaviour is to replace a process with which it is composed in parallel that matches its denominator. The matching is done by behaviour, using the bisimulation \( \sim \) defined below, in order to increase the terseness of expressions modelling process reconfiguration.

We define the semantics of \( CCS^{dp} \) in two ways, using structural congruence (\( \equiv \)) and a labelled transition system (LTS). The structural congruence is based on the static structure of processes, and it is used for equational reasoning. The definitions equate structurally different process expressions that can be used interchangeably. The LTS is based on the behaviour of processes, and it has two uses: to define the bisimulation \( \sim_{cf} \) for matching processes for reconfiguration; and to define process transitions for model checking.

Structural congruence is the least process congruence over \( P \) that satisfies the rules given in Table 3.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )-conversion</td>
<td>( P[\bar{a}] \equiv P ) ( P[Q] \equiv Q[P] ) ( P[(Q</td>
<td>R)] \equiv (P</td>
</tr>
<tr>
<td>Addition</td>
<td>( M + 0 \equiv M ) ( M_1 + M_2 \equiv M_2 + M_1 ) ( M_1 + (M_2 + M_3) \equiv (M_1 + M_2) + M_3 )</td>
<td>Binary addition of process expressions</td>
</tr>
<tr>
<td>Scope</td>
<td>( (\nu \bar{a})0 \equiv 0 ) ( (\nu \bar{a})(\nu \bar{b})P \equiv (\nu \bar{b})(\nu \bar{a})P ) ( (\nu \bar{a})P' \equiv (\nu \bar{a})P' )</td>
<td>Restriction on process variables in process expressions</td>
</tr>
<tr>
<td>Restriction</td>
<td>( (\nu \bar{a})(P</td>
<td>Q) \equiv (\nu \bar{a})P</td>
</tr>
<tr>
<td>Transition</td>
<td>( A &lt;\bar{b}&gt; \equiv P[\bar{b}/\bar{a}] ) if ( A(\bar{a}) \equiv P \land</td>
<td>\bar{b}</td>
</tr>
</tbody>
</table>

Table 3. \( CCS^{dp} \) Structural Congruence.

The structural congruence rules for \( CCS^{dp} \) are the rules of \( CCS \) plus the scope rule for fraction processes. This rule states that a restriction \( (\nu \bar{a}) \) on a fraction \( P' \) is congruent to the fraction of the restrictions \( (\nu \bar{a})P' \).
0 is the identity of the summation and parallel composition operators in the equivalences and congruences of CCS, and it is desirable to retain this property of 0 in CCS\textsuperscript{dp} because it helps to manipulate process expressions during reasoning. However, the identity property of 0 in combination with fraction processes with a 0-valued denominator is problematic: if a fraction with denominator 0 has reconfiguration behaviour then processes that should be behaviourally equivalent (e.g. $\frac{a}{0} \parallel a.0$ and $\frac{a}{0} \parallel \tau$) can behave differently (e.g. $\frac{a}{0} \parallel a.0 \stackrel{\tau}{\rightarrow} a$ and $\frac{a}{0} \parallel \tau \nrightarrow a$).

Therefore, we exclude 0 and processes behaviourally equivalent to 0 (such as $a.0$) from reconfiguration transitions defined by the LTS rules. Hence, we distinguish processes that can perform a reconfiguration transition (positive processes) from processes that cannot perform a reconfiguration transition (zero processes).

Let $\mathcal{L}$ be the countable set of names that represent both ports and actions of the processes in $\mathcal{P}$, where $\mathcal{L} \triangleq N \cup \mathcal{N}$.

Let $\mathcal{I}$ be the countable set of input and output actions of the processes in $\mathcal{P}$, and their internal action, where $\mathcal{I} \triangleq \mathcal{L} \cup \{\tau\}$.

Given $p \in \mathcal{P}$, let $\mathcal{I}_p$ be the set of initial actions in $\mathcal{I}$ that $p$ can perform.

$: \mathcal{I}_{a.0} = \{a\}, \mathcal{I}_0 = \emptyset$ and $\mathcal{I}_{\tau} = \emptyset$.

Thus, restricting $\mathcal{I}_p$ in a simple way (e.g. $\mathcal{I}_p \neq \emptyset$) is not sufficient to isolate 0 and processes with 0-like behaviour. Therefore, we define the set of reconfiguration actions.

Let $\mathcal{R}$ be the set of reconfiguration actions that create a process in $\mathcal{P}$; and let $\overline{\mathcal{R}}$ be the set of complementary reconfiguration actions that delete a process in $\mathcal{P}$ (see the Creat and Delet rules in Table 4).

Given $p \in \mathcal{P}$, let $\mathcal{R}_p$ be the set of initial reconfiguration actions in $\mathcal{R}$ that $p$ can perform that create a process in $\mathcal{P}$; and let $\overline{\mathcal{R}}_p$ be the set of complementary initial reconfiguration actions in $\overline{\mathcal{R}}$ that $p$ can perform that delete a process in $\mathcal{P}$.

The LTS is defined by the structural operational semantic (SOS) rules given in Table 4, for which we need the following definitions.

Let $\mathcal{C}$ be the set of reconfiguration actions of the processes in $\mathcal{P}$, where $\mathcal{C} \triangleq \mathcal{R} \cup \overline{\mathcal{R}}$.

And let $\mathcal{A}$ be the set of actions of the processes in $\mathcal{P}$, where $\mathcal{A} \triangleq \mathcal{I} \cup \mathcal{C}$.

The SOS rules for CCS\textsuperscript{dp} are a superset of the SOS rules for CCS, consisting of unchanged rules of CCS (i.e. Sum and Res) plus CCS rules applicable to reconfiguration transitions (i.e. React, L – Par, R – Par and Ident) plus additional rules to describe new reconfiguration behaviour (i.e. Creat, Delet, CompDelet, L – React, R – React, ResFract and ResRecon).

The React rule states that if two processes can perform complementary labelled or reconfiguration transitions, then their parallel composition can result in a $\tau$ transition in which both processes undergo their respective complementary transitions atomically.
\[
\sum_{k \in I} a_k \frac{p_k}{p_k^*}
\]

\[
\sum_{i \in I} a_i \cdot \frac{p_i}{p_i^{\mu}} = P^i
\]

\[
\lambda \in \text{L } \cup \text{C } \land P.Q' \rightarrow P'^Q
\]

\[
\alpha \in A \land P.\frac{\mu}{\mu'} = P'
\]

\[
\beta \in \text{Set}() \cup \text{Set}() = P
\]

\[
\rho \in \{\tau_r Q, \tau_r Q\} \land P.\frac{\mu}{\mu'} = P'
\]

<table>
<thead>
<tr>
<th>Sum</th>
<th>React</th>
<th>L-Par</th>
<th>R-Par</th>
<th>Ident</th>
<th>Creat</th>
<th>Delet</th>
<th>CompDelet</th>
<th>L-React</th>
<th>R-React</th>
<th>Res</th>
<th>ResFract</th>
<th>ResRecon</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k \in I$</td>
<td>$\sum_{i \in I} a_i \cdot P_i$</td>
<td>$\frac{\alpha}{\alpha^*} = P_k$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\lambda \in \text{C } \land P.\frac{\lambda}{\lambda'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.Q \rightarrow P'^Q$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu \in A \land P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P(Q \rightarrow P'^Q$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\frac{\alpha}{\alpha^*} = P_k$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$A(b) = P$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$P.\frac{\mu}{\mu'} = P'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4. CCSdp Structural Operational Semantics.

The $L - Par$ and $R - Par$ rules state that parallel composition preserves the labelled, internal and reconfiguration transitions of constituent processes.

The $Ident$ rule states that if a relabelled constant process can perform a given labelled, internal or reconfiguration transition, then the constant process instantiated with the new labelling can also perform the same transition.

The $Creat$ rule states that a fraction process $\frac{\alpha}{\alpha^*}$ can perform a reconfiguration transition $\tau_r Q$ that results in the creation of $P'$, where $Q$ is strongly of-bisimilar to $P$ (strong of-bisimilarity is defined below).

The $Delet$ rule is complementary to the $Creat$ rule. It states that a non-0 process $P$ can be deleted by performing a reconfiguration transition $\tau_r Q$ that is complementary to the reconfiguration transition $\tau_r Q$ by some fraction that creates a process, where $Q$ is strongly of-bisimilar to $P$ (see below). Thus, strongly
of-bisimilar non-0 processes can be deleted by the same reconfiguration transitions (∴ ∼ ∼ is an equivalence relation). Notice that reconfiguration transitions do not involve any communication. Therefore, the interaction between complementary reconfiguration transitions does not require a port or a communication channel.

The CompDelet rule states that consecutive delete transitions of a process can be composed into a single delete transition of the process. This rule is necessary in order to prove the associativity of the parallel composition operator with respect to strong of-bisimulation, which supports equational reasoning in CCS._dp_.

In a process expression, the denominator of a fraction process can match the parallel composition of processes located on both sides of the fraction. The L – React and R – React rules state that a reconfiguration reaction can occur in this case, with all the processes participating in the reaction undergoing their respective transitions atomically. These two rules are necessary in order to prove the commutativity of the parallel composition operator with respect to strong of-bisimulation, which supports equational reasoning in CCS._dp_.

The ResFract rule for restricted fraction processes corresponds to the Creat rule for fraction processes. It is also the LTS form of the structural congruence scope rule for fraction processes. It states that if a fraction process P′_ν_a is perform a reconfiguration transition τ_rQ to create P′, then the restricted fraction process (ν̃ a)_P can perform the reconfiguration transition τ_r(ν̃ a)_Q to create (ν̃ a)_P, provided (ν̃ a)_P is a positive process. Thus, the restricted fraction (ν̃ a)_P behaves like the fraction of the restrictions (ν̃ a)_P′ (ν̃ a)_P.

The ResRecon rule is the Res rule modified for reconfiguration transitions. It states that a restriction (ν̃ a)_P preserves the reconfiguration transitions τ_rQ, τ_rQ of the process P, provided the free names of Q are not restricted by (ν̃ a). This condition (on Q) is stronger than the corresponding condition (on α) in Res, because τ_rQ and τ_rQ depend on the behaviour of the entire process Q for matching.

A feature of CCS._dp_ is the use of behavioural matching to determine whether or not a reconfiguration transition can occur. Behavioural matching helps to increase the terseness of models. The matching is done using the strong of-bisimulation (∼ ∼), which is defined as follows.

We define S to be a strong of-simulation on P iff S ⊆ P × P and the following conditions hold ∀(p, q) ∈ S:

1. ∀α ∈ I_p(∀p′ ∈ P(p −→_α p′ ⇒ ∃q′ ∈ P(q −→_α q′ ∧ (p′, q′) ∈ S))

2. ∀τ_rX ∈ R_p(∀p″ ∈ P(p −→_τ_rX p″ ⇒ τ_rX ∈ R_q ∧ ∃q″ ∈ P(q −→_τ_rX q″ ∧ (p″, q″) ∈ S)))

We define S to be a strong of-simulation on P iff S ⊆ P × P and the following conditions hold ∀(p, q) ∈ S:

1. ∀α ∈ I_p(∀p′ ∈ P(p −→_α p′ ⇒ ∃q′ ∈ P(q −→_α q′ ∧ (p′, q′) ∈ S)))

2. ∀τ_rX ∈ R_p(∀p″ ∈ P(p −→_τ_rX p″ ⇒ τ_rX ∈ R_q ∧ ∃q″ ∈ P(q −→_τ_rX q″ ∧ (p″, q″) ∈ S)))
Condition 1 is intended for processes that can behave like processes in CCS, and it is the same as the condition for strong simulation in CCS. It states that in order for \( q \) to simulate \( p \), any input or output or \( \tau \) action that \( p \) can perform to become \( p' \), must be also performable by \( q \) to become \( q' \), and \( q' \) must simulate \( p' \).

Condition 2 is intended for processes that can behave like fraction processes. It states that in order for \( q \) to simulate \( p \), any reconfiguration action that \( p \) can perform to create \( p'' \), must be also performable by \( q \) to create \( q'' \), and \( q'' \) must simulate \( p'' \).

The two conditions of strong of-simulation are very similar, and (therefore) can be readily combined into a single condition. However, we prefer to keep them separate in order to show the difference between strong observation equivalence in CCS and strong of-equivalence in CCS\(^{dp} \).

We define \( S \) to be a strong of-bisimulation on \( P \) iff both \( S \) and \( S^{-1} \) are strong of-simulations on \( P \).

Following convention, we represent the largest strong of-bisimulation on \( P \) by \( \sim_{of} \), where \( \sim_{of} \triangleq \bigcup \{ S \mid S \) is a strong of-bisimulation on \( P \} \).

We define two processes \( p \) and \( q \) in \( P \) to be strongly observationally and fractionally equivalent (\( p \sim_{of} q \)) iff there exists a strong of-bisimulation \( S \) on \( P \) containing \((p, q)\).

The use of \( \sim_{of} \) as a hypothesis in the \textit{Create} transition rule and the use of this rule to define \( \sim_{of} \) raises the issue of circular (and therefore undecidable) transitions. We avoid this problem by making the depth of recursion of the denominator of fraction processes finite.

**Entire Model** In modelling the design of the reconfiguration given in Figure 3, we represent pools in terms of their activities (modelled as processes), because CCS\(^{dp} \) has no facility for composing processes into a higher-level construct. We subsume the \textit{Order Generator} pool into the \textit{ORDERS} process, because this simplifies the model and does not affect the reconfiguration. Each workflow handles a single order.

Let \( C \) be the set of possible customer identifiers,

let \( I \) be the set of possible item identifiers,

let \( O \) be the set of possible order identifiers.

Let \( \text{ORDERS} \triangleq \prod_{c \in C, i \in I, o \in O} \text{OrderReceipt}_{c,i,o}(\text{Confirm}_{c,i,o} + \text{Reject}_{c,i,o}) \)

Let \( \text{WF} \triangleq \text{OR} | \text{CC} | \text{CCH} | \text{IC} | \text{ICH} | \text{BILL} | \text{SHIP} | \text{BSH} | \text{AR} | \text{ARH} \)

Let \( \text{OR} \triangleq \sum_{c \in C, i \in I, o \in O} \text{OrderReceipt}_{c,i,o}(\text{WF} | \overline{\text{CreditCheck}_{c,i,o}}) \)

Let \( \text{CC} \triangleq \sum_{c \in C, i \in I, o \in O} \text{CreditCheck}_{c,i,o,\tau}(\text{CreditCheckNotOK}_{c,i,o} + \text{CreditCheckOK}_{c,i,o}) \)

Let \( \text{CCH} \triangleq \sum_{c \in C, i \in I, o \in O} \text{CreditCheckNotOK}_{c,i,o,\tau}(\text{Reject}_{c,i,o} + \text{CreditCheckOK}_{c,i,o,\tau}) \)

Let \( \text{InventoryCheck}_{c,i,o} \)
Let $IC \triangleq \sum_{c \in C, i \in I, o \in O} \text{InventoryCheck}_{c,i,o} \cdot \tau$. 
Let $ICH \triangleq \sum_{c \in C, i \in I, o \in O} \text{InventoryCheckNotOK}_{c,i,o} \cdot \text{Reject}_{c,i,o} + \text{InventoryCheckOK}_{c,i,o} \cdot (\text{Bill}_{c,i,o} \mid \text{Ship}_{c,i,o})$

Let $BILL \triangleq \sum_{c \in C, i \in I, o \in O} \text{Bill}_{c,i,o} \cdot \tau \cdot \text{BillOK}_{c,i,o}$

Let $SHIP \triangleq \sum_{c \in C, i \in I, o \in O} \text{Ship}_{c,i,o} \cdot \tau \cdot \text{ShipOK}_{c,i,o}$

Let $BSH \triangleq \sum_{c \in C, i \in I, o \in O} \text{BillOK}_{c,i,o} \cdot \text{ShipOK}_{c,i,o} \cdot \tau \cdot \text{Archive}_{c,i,o}$

Let $AR \triangleq \sum_{c \in C, i \in I, o \in O} \text{Archive}_{c,i,o} \cdot \tau \cdot \text{ArchiveOK}_{c,i,o}$

Let $ARH \triangleq \sum_{c \in C, i \in I, o \in O} \text{ArchiveOK}_{c,i,o} \cdot \text{Confirm}_{c,i,o}$

**Old Region**

$WF$, $OR$, $ICH$ and $BSH$

**New Region**

$WF'$, $OR'$, $ICH'$ and $BSH'$, where

$OR' \triangleq \sum_{c \in C, i \in I, o \in O} \text{OrderReceipt}_{c,i,o} \cdot (WF' \mid \text{CreditCheck}_{c,i,o})$

and $ICH' \triangleq \sum_{c \in C, i \in I, o \in O} \text{InventoryCheckNotOK}_{c,i,o} \cdot \text{Reject}_{c,i,o} + \text{InventoryCheckOK}_{c,i,o} \cdot \text{Bill}_{c,i,o}$

and $BSH' \triangleq \sum_{c \in C, i \in I, o \in O} \text{BillOK}_{c,i,o} \cdot \text{ShipOK}_{c,i,o} \cdot \text{ShipOK}_{c,i,o} \cdot \text{Archive}_{c,i,o}$

and $WF' \triangleq OR' \mid CC \mid CCH \mid IC \mid ICH' \mid BILL \mid SHIP \mid BSH' \mid AR \mid ARH$

And the reconfiguring process is $RM_0$, where $RM_0 \triangleq \text{Trigger}.(\frac{WF'}{WF} \mid RM_1)$ and $RM_1 \triangleq \text{Loop}.(\frac{ICH'}{ICH} \mid \frac{BSH'}{BSH} \mid RM_1) + \text{Stop}$

We assume there is a process in the environment of $WF$ that triggers $RM_0$ if $WF$ is to be reconfigured. The same process keeps track of the number of orders currently being processed, and causes $RM_1$ to loop and (finally) to stop.
5 WS-BPEL Implementation of the Office Workflow

In [29] the mapping from BPEL to $\pi$-calculus has been investigated. The idea was there to design the system at the BPEL level and then verifying it at the $\pi$-calculus level. In [2], the opposite direction has been instead explored. This more recent work supports the idea that building the $\pi$-calculus model, check it and only then map it into BPEL seems to be a more effective way to tackle the problem of verification for BPEL systems. In that work the asynchronous $\pi$-calculus has been used. This means that, for example, the asynchronous $\pi$-calculus model presented in this paper could be easily used to obtain a BPEL implementation, provided a few annotations are added to store information like partner link, port type, and actions.

However, here we have decided to follow a different approach based on the BPMN design because we think it is a powerful design tool which should not be ignored.

In this section we will present a BPMN derived BPEL implementation of the case study and the basic ideas behind it. The same ideas can be used (and actually are used) to model the case study in $Web\pi_\infty$. What the reader can visibly understand looking at the previous sections, is that the original $\pi$-calculus is less intuitive for modelling certain kind of systems where reconfiguration, or even just recovery, has to be specified. This is exactly the thesis we want to support in this paper. The problem of the $\pi$-calculus is its lack of structural information as we have explained in [33]. This does not mean that the $\pi$-calculus cannot be used to model recovery and reconfiguration, it certainly can being all these formalisms Turing-complete [9]. It is just a matter of practicality: with the $\pi$-calculus the model should be based on link passing and network reconfiguration, playing with link passing and name hiding. It would result in a clumsy model. Instead, with interrupt/compensation operators able to impose a structure over the processes, we obtained much more easy to read models, as visible in the previous sections of this paper. A formal investigation of the expressive power of interruption and compensation operators can be found in [10]. The decision on what formalism to use for modelling a case study has always to be based on a tradeoff between expressiveness and practicality. Otherwise, obviously, everything can always be modelled in term of Turing Machines or $\lambda$-calculus.

Our intuition was that, although BPEL itself has not been designed to cope with dynamic reconfiguration, it presents some features which can be used to this purpose. This idea has emerged because of similar considerations we have done about $Web\pi_\infty$. Since $Web\pi_\infty$ has been used to encode WS-BPEL, we have suspected that the basic mechanism of the BPEL recovery framework would work as the $Web\pi_\infty$ mechanism worked for this purposes. This was just an intuition but we worked to make it work and the results will be presented in this section.

The basics principles, derived from the $Web\pi_\infty$ experience, on which our implementation is constructed are:

- The regions to be reconfigured have to be represented by BPEL scopes
- Each BPEL scope (i.e. region) will be associated with termination and event handlers.
- Interference ("overlapping modes") will be implemented by the combined use of event and termination handlers.

For a better understanding of how event handlers work please have a look at [13]. However, that paper does not investigate termination handlers (please see [26] for more details on this). Event handlers run in parallel with the scope body and are available more than once to be called (one single call does not suspend further availability). Thus, the new region has to be triggered by the event handler while the old region will be then terminated by the termination handler. As said, the body scopes run separately from the event handler so the old region can be terminated separately while the event handler brings the new region into play. This has not to be immediate. We think we can implement this way the synthetic cut-over change as defined in terms of Petri nets in [14].

While so far we have just presented the general principles on which the implementation is based, readers who are familiar with BPEL and who are interested in the details can find them in the following of this section. Readers who are not interested in the details of the implementation can just skip this section without missing to grasp the general concept of our research.

5.1 Mapping BPMN Models to BPEL

The first problem we have encountered when mapping the BPMN design into a BPEL implementation comes from the evident observation that BPMN and BPEL are representative of two different classes of languages. BPMN is indeed graph oriented while BPEL is mainly block-structured [42], at least in its commonly used XLANG [44] derived subset (however, BPEL has been also influenced by the graph oriented WSFL [28]). A consequence of this divergence is that the mapping from BPMN to BPEL is hard and it has a number of limitations since BPMN is able to express process patterns which cannot be expressed in BPEL. As a general comment we could say that the block structured nature of a BPEL process is too limited for modeling purposes.

However, we believe that BPEL cannot be ignored when it comes to workflow modelling because, although the business analysts more easily work with BPMN as modeling language and use its graphical notation to describe a business process (Task, Activity, sequence flow, etc), the system developers manage better to work with an executable language like BPEL to define the composite structure of a business process. In BPEL such a structure is defined in terms of a flow of structured activities (Sequence, Parallel, etc) where each activity, in turn, can contain a nested list of other activities being those Web service invocations or other structured activities.

In this work the structure mismatch between BPMN and BPEL has been resolved following the approach presented in [42] consisting of a complete translation based on the identification of patterns of BPMN fragments which can...
be directly mapped onto BPEL code. The transformation approach will be described in the following of this section where we will give an overview of our mapping and how it is intended to work for the case study we are considering.

**Basic Activities Translation** BPMN basic activities (the ones based on messages, events and assignments) can be directly mapped to BPEL according to the following translation schema:

<table>
<thead>
<tr>
<th>BPMN</th>
<th>BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Task, Service Task, Message Event</td>
<td>Invoke</td>
</tr>
<tr>
<td>Receive Task, Message Event</td>
<td>Receive</td>
</tr>
<tr>
<td>Send Task, Message Event</td>
<td>Reply</td>
</tr>
<tr>
<td>Assignment</td>
<td>Assign</td>
</tr>
<tr>
<td>Termination end event</td>
<td>Exit</td>
</tr>
</tbody>
</table>

**Structured Activities Translation** We can classify different types of well-structured BPMN patterns resembling BPEL structured activities - sequence, flow, switch, pick and while - and translate them as follows:

<table>
<thead>
<tr>
<th>BPMN</th>
<th>BPEL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence Flow</td>
<td>Sequence</td>
</tr>
<tr>
<td>Parallel Fork-Join Gateway</td>
<td>Flow</td>
</tr>
<tr>
<td>Exclusive Data-based Gateway</td>
<td>Switch</td>
</tr>
<tr>
<td>Exclusive Event-based gateway, Message/Timer Event</td>
<td>Pick</td>
</tr>
<tr>
<td>Loops</td>
<td>While, RepeatUntil</td>
</tr>
</tbody>
</table>

### 5.2 WSDL Descriptions of the Involved Processes

As the reader can see in section 2.2, the BPMN design of the office workflow is made up of a set of independent components, which are shown as separate pools (using BPMN terminology) with separate sequence flows. It is interesting to note that this is not an abstract process since it includes the specific service calls involved, i.e. it includes the interactions points between the different participants. There are actually six participants involved (implemented as asynchronous Web services) as shown in the table below.
WSDL [12] requires the specification of operations and port types the service is offering, the accepted messages and their types. Consequently, a precise WSDL descriptions of the involved services can be derived from the table above. We now present in detail the WSDL description for each of the six processes.

- **Office Workflow**: *OrderReceiptPortType* allows the order message to be received by means of the *OrderReceipt* operation. To return the result, the Web service specifies a second port type: *NotifyPortType*. This port type specifies *Confirm* and *Reject* operations to return notification messages back to the customer.

```xml
<wsdl:portType name="OrderReceiptPortType">
<wsdl:operation name="OrderReceipt">
  <wsdl:input message="OrderRequest" name="Input"/>
</wsdl:operation>
</wsdl:portType>

<wsdl:portType name="NotifyPortType">
  <wsdl:operation name="Confirm">
    <wsdl:input message="ConfirmNotify" name="Input"/>
  </wsdl:operation>
  <wsdl:operation name="Reject">
    <wsdl:input message="RejectNotify" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```
– **Order Generator**: `OrderGeneratorPortType` allows generating the order code by means of the `OrderGenerator` operation. The result is returned through the `OrderGeneratorReply` operation specified by `OrderGeneratorReplyPortType`.

```xml
<wsdl:portType name="OrderGeneratorPortType">
  <wsdl:operation name="OrderGenerator">
    <wsdl:input message="OrderRequest" name="Input"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="OrderGeneratorReplyPortType">
  <wsdl:operation name="OrderGeneratorReply">
    <wsdl:input message="GeneratorReply" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```

– **Credit Check**: `CreditCheckPortType` allows checking the identity of the customer with the operation `CreditCheck`. `CreditCheckReplyPortType` is instead used to return the check result through the operation `CreditCheckReply`.

```xml
<wsdl:portType name="CreditCheckPortType">
  <wsdl:operation name="CreditCheck">
    <wsdl:input message="Customer" name="Input"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="CreditCheckReplyPortType">
  <wsdl:operation name="CreditCheckReply">
    <wsdl:input message="CheckResult" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```

– **Inventory Check**: Similarly to the Credit Check service `InventoryCheckPortType` is used to check the identity of the product by means of the operation `InventoryCheck`. `InventoryCheckReplyPortType` is instead used to return the result of the check through the operation `InventoryCheckReply`.

```xml
<wsdl:portType name="InventoryCheckPortType">
  <wsdl:operation name="InventoryCheck">
    <wsdl:input message="Item" name="Input"/>
  </wsdl:operation>
</wsdl:portType>

<wsdl:portType name="InventoryCheckReplyPortType">
  <wsdl:operation name="InventoryCheckReply">
    <wsdl:input message="CheckResult" name="Input"/>
  </wsdl:operation>
</wsdl:portType>
```
Bill and Ship: BillShipPortType is used to trigger the bill and ship activity through the operation BillShip. The bill activity is performed using the operation Bill and the ship activity is performed using the operation Ship. To return the result, the service specifies a second port type: BillShipReplyPortType. The bill details are returned through the operation BillReply while the ship details are returned through the operation ShipReply. The overall bill and ship details are returned through the operation BillShipReply.

Archive: ArchivePortType allows archiving the ordered product for further reference using the operation Archive. ArchiveReplyPortType is specified to return the result through the operation ArchiveReply.
To make all the services working together BPEL requires the definition of a partnerLink section as follows:

```xml
<partnerLinks>
  <partnerLink myRole="OrderReceiptServiceProvider"
    name="OfficeWorkflow"
    partnerLinkType="OfficeWorkflow:OfficeWorkflowPLT"
    partnerRole="NotifyServiceRequester"/>
  <partnerLink myRole="OrderGeneratorReplyServiceRequester"
    name="OrderGenerator"
    partnerLinkType="OrderGenerator:OrderGeneratorPLT"
    partnerRole="OrderGeneratorServiceProvider"/>
  <partnerLink myRole="CreditCheckReplyServiceRequester"
    name="CreditCheck"
    partnerLinkType="CreditCheck:CredtCheckPLT"
    partnerRole="CreditCheckServiceProvider"/>
  <partnerLink myRole="InventoryCheckReplyServiceRequester"
    name="InventoryCheck"
    partnerLinkType="InventoryCheck:InventoryCheckPLT"
    partnerRole="InventoryCheckServiceProvider"/>
  <partnerLink myRole="BillShipReplyServiceRequester"
    name="BillShip1"
    partnerLinkType="BillShip:BillShipPLT"
    partnerRole="BillShipServiceProvider"/>
  <partnerLink myRole="ArchiveReplyServiceRequester"
    name="Archive"
    partnerLinkType="Archive:ArchivePLT"
    partnerRole="ArchiveServiceProvider"/>
</partnerLinks>
```

In this way, the interfaces of the other services which interacts with Office Workflow are linked to the main BPEL process of the workflow itself.

### 5.3 Office Workflow BPEL Main Body

The main process describing the workflow starts with the reception of an order request coming from a customer, then it asynchronously invokes Order Generator and, after having received a reply from it, Credit Check is asynchronously invoked. It continues asynchronously invoking different services one by one, according to the specification. Two structure patterns can be identified: the sequence pattern involving the whole process and the If-else pattern for handling both the credit check reply and the inventory check reply. The BPMN Message Start Event initiates the process receiving a message. This is mapped into BPEL using a receive activity.
Next, we have to prepare the request message for the Order Generator service. We have to send a message consisting of customer and item parts built through the corresponding BPEL assignment activity.

Now, the Order Generator service will be invoked. Because it is an asynchronous service, the callback will be received using the BPEL receive activity. We have so to invoke the OrderGeneratorReply operation on the OrderGeneratorReplyPortType. The callback message contains a Order number (OrderID) which is used to initiate the correlation set.
After having received the response message from the Order Generator service, the process will invoke the Credit Check service. This involves checking customer identity. Mapping the call of the Credit Check service is similar to mapping the Order Generator service. Again, we start with the preparation of the input message for the Credit Check service and then we invoke the service itself.

```bpelexport
<assign name="Callcreditcheck">
  <copy>
    <from>$Order.part/CustomerName</from>
    <to>$Customer.part/CustomerName</to>
  </copy>
  <copy>
    <from>$Order.part/OrderID</from>
    <to>$Customer.part/CustomerID</to>
  </copy>
</assign>
<invoke partnerLink="CreditCheck" operation="CreditCheck" portType="CreditCheck:CreditCheckPortType" inputVariable="Customer">
  <correlations>
    <correlation set="OrderId" pattern="out"/>
  </correlations>
</invoke>
<receive name="Receivecreditcheckresult" createInstance="no" partnerLink="CreditCheck" operation="CreditCheckReply" portType="CreditCheck:CreditCheckReplyPortType" variable="CreditCheckResult">
  <correlations>
    <correlation set="OrderId" initiate="no"/>
  </correlations>
</receive>

The BPMN exclusive gateway following the "Receive credit check result" message event is mapped into a BPEL If-else structured activity:

```bpelexport
<if name="If1">
  <condition>$CreditCheckResult.Part</condition>
  <sequence name="Sequence1">
    ...
  </sequence>
</if>
<else>
  ...
</else>
```
The Inventory Check works exactly in the same way as the Credit Check. The BPMN process then moves to "Receive item check result", it goes through the "Yes" condition and the Bill&Ship operation is invoked on the BillShipPortType of the Bill And Ship service. At this point, two operations bill and ship are invoked in parallel. Both bill and ship return their details by means of the operation BillShipReply and then the Archive service is invoked. After having received the return message ACK from ArchiveReplyPortType an invoke activity on NotifyPortType is performed to send a confirmation message back to the customer.

```xml
<invoke name="Notifycustomerconfirm"
       partnerLink="OfficeWorkflow"
       operation="Confirm"
       portType="OfficeWorkflow:NotifyPortType"
       inputVariable="ConfirmNotify"/>
```

Change Configuration To implement in BPEL the BPMN model depicted in figure 2 we have just to replace Bill Ship. So we need to define a new interface for it and then map it onto a new partner link. We have to do the same as before and finally we get BillandShip2. This is also an asynchronous process, containing both the invocation of bill and ship operations and the invocation of a callback operation.

```xml
<partnerLink
    myRole="BillShipReplyServiceRequester"
    name="BillandShip2"
    partnerLinkType="BillShip2:BillShipPLT" partnerRole="BillShipServiceProvider"/>
```

The process is simpler than the former "Bill Ship", only one structure pattern is now involved: Sequence. After the BillShip operation is invoked on BillShipPortType of the Bill And Ship service, the Bill operation is invoked. Then, the return message from BillReplyPortType is received and the Ship operation is invoked. Ship details are returned by the operation ShipReply and the return message sent from BillShipReplyPortType is received. Finally, the Archive service is invoked.

```xml
<sequence name="Sequence3">
  <assign name="Callbill">
    <copy>
      ...
    </copy>
  </assign>
  <invoke inputVariable="BillShipOrder"
          operation="Bill"
```
Transition between Configurations  The most interesting part of the BPMN design is the one depicted in figure 3. To map this to BPEL we have to define a new partner link for a new participant Reconf.region which will be then used to invoke the new configuration. We define a partner link with the role provider to change configuration:

```
<partnerLink myRole="provider"
    name="Reconf.region"
    partnerLinkType="Reconf.region:Reconf.RegionPLT" />
```

Within Reconf.region there is a BPMN Activity "Determine configuration" with a Non-Interrupting Intermediate Message Event, which can be mapped to a BPEL scope with an event handler [8].
Let us describe here in details how this works. If the process receive the Rec change message once the BillAndShip1 scope has been entered, it will execute the new process defined within the scope BillAndShip2. This other process is exactly the new configuration. Otherwise, the order will be processed accordingly with the original procedure.

In order to distinguish between these two situations — receiving the event before billing and shipping activities have started or after — we use scopes to define different event handlers: Scope1 represents the procedure running before billing and shipping, BillShip1 represents the concurrent billing and shipping and BillShip2 represents the sequential billing and shipping. When a management decision is made, the event handler for Scope1 will be invoked and it will terminate Scope1, which contains the procedure for order receipt, order evaluation and BillShip1 activities. We use termination handler to replace Scope1 with a new scope representing the new procedure for order receipt, order evaluation and, this time, BillShip2 activities. In this way, after its termination, the process will restart calling the new procedure.

We declare individual variables for BillShip1 and BillShip2. These are the request messages used to invoke the billing and shipping services and they are only visible within their own scope. This means that, if the request message for billing and shipping has already been created, this activity can be invoked without any interrupt. Technically, the event handler is used to implement the management decision for change. When the event is received, BillShip2 will be enabled. However, if the event is received after Scope1 has been executed, BillShip2 will not be run because no request message has been initialized and Scope1 only callsBillShip1. If the event is received while Scope1 is running, Scope1 will be terminated and Scope2 will start redoing order receipt, order evaluation. After that, BillShip2 will start because the receiving event, and also the request message, have been initialized exactly for it.

In the real word, after the management decision is made to switch to BillShip2, BillShip1 would be not available anymore. It is like ending to offer the BillShip1 service. However, in BPEL, we cannot model exactly this situation. All the services remain available. If we want to ensure all the instances of the
workflow created after the change run BillShip2 instead of BillShip1, the process needs to continue receiving the “change reconfiguration” event.

5.4 Tool-based Mapping BPMN Models to BPEL

The BPMN to BPEL mapping we have presented so far has been obtained by following the approach given in [42]. This allowed us to have some flexibility but the process had to be entirely manually executed. Another option, although more restrictive, is to use some automatic tool for the translation. In this section we will indeed discuss this option using the Intalio BPMS Designer version 6.0.

Intalio BPMS Designer is a set of Eclipse plugins allowing process designers to model processes with BPMN and to use several graphical tools to manage the data. It includes most of the BPMN elements which are relevant to executable business process models. External activities and message flows are mapped into specific interface operations and message definitions using WSDL. The message structures are indicated by XML Schema elements. Service calls are modelled by introducing Pools containing the operations of the WSDL. The process interacts with this external participants through message flows. After the process has been modelled and concrete services, messages and data have been defined, Intalio Designer will automatically generate a BPEL description.

To model the office workflow with the Intalio Designer the first thing we have to do is creating a 'Business Process Project' containing Business Process diagrams, XML Schemas, WSDL files, etc. Once the project has been created, we can then create a BPMN diagram with the embedded BPMN modeler. After the BPMN modelling for the office workflow will be completed, we can start implementing the process Office Workflow by integrating all the operations from the existing Web services, creating the interface to define how it will be exposed to the external users and defining the graphical mappings to invoke the services.

Integrating web services The tool integrates a full WSDL visual browser which allows to edit and introspect WSDL documents. To implement the case study we have to create WSDL documents for Office Workflow Service, Order Generator Service, Credit Check Service, Inventory Check Service, Bill&Ship Service and Archive Service respectively. Then we have to create pools representing all the external Web services and set them to ’Non-executable’ as these pools represent the sequence of service operations that will be invoked from the main business process Office Workflow.

It is very important to make the distinction between operations invoked by a process and operations that will invoke a process. An operation in a non-executed pool, represented as a BPMN task, it either provides the operation or invokes the operation. The operations like OrderGenerator, CreditCheck, InventoryCheck, etc are operations that the Office Workflow process will invoke; whereas OrderGeneratorReply, CreditCheckReply, InventoryCheckReply, etc are operations that will invoke the process.

Finally, we have to connect the process tasks to the Web service operations. The order is defined by creating the links. For the operations of Order Generator
we want the message received from the customer to go from the executable task ("Invoke order generator") to the corresponding Order Generator operation and the response message to go from the Order Generator operation to the message intermediate event ("Receive order"). In the same way, we have to integrate Credit Check, Inventory Check, Bill&Ship, Archive. All the data involved in the Office Workflow process are created automatically when integrating the WSDL files.

Generate BPEL code Once the Office Workflow process is ready to be executed we can easily deploy it. There are several artifact being generated at this point: the BPEL code corresponding to the Office Workflow process, the WSDL files used by the process to represent its interactions with the other participants and the different WSDLs used to represent external services.

Change Configuration As before, we have to deal with the Office Workflow reconfiguration, i.e. the process will invoke Bill and Ship in sequence instead parallel. The remaining parts like partner links, external services, WSDLs are not altered by this but the BPEL is. We need indeed a new participant Reconf.region used to send a reconfiguration message and invoke the new procedure. We also have to create a WSDL for it.

We need two use sub-process to include the two configurations and to add an Exclusive Event-based Gateway to make the choice. If the process receives the change message then the configuration2 sub-process will execute (the new configuration) otherwise the process will automatically executes the old configuration sub-process configuration1. The generated BPEL code, partner links and, in general, all the material related to this project could not be reported in this document due to its size but it is available upon request, so the interested reader can contact the authors.

As we can see from the generated BPEL code, the interaction between the Reconf.region Web service and BPEL process is mapped into a pick activity.

```xml
<pick ...>
  <onMessage partnerLink="...">
    operation="Change"
    portType="ReconfigService:ReconfigService"
    variables="...">
      <sequence>
        <scope ...>
          <variables>
            </variables>
            <sequence>
              <assign>
               ...
              </assign>
              <invoke name="...">
                <receive name="...">
                  <invoke name="...">
                    <receive name="...">
```
Thus, if the process receives the change message before invoking the BillAndShip operation on BillAndShipPortType, the order will be processed according to the new procedure, otherwise it will be processed according to the old one.

6 Discussion

To our knowledge, there is no computational formalism that can model both dynamic reconfiguration of a system and its real-time features such as periods and deadlines. Moreover, such a formalism is required in order to perform safely dynamic reconfiguration of hard real-time systems, which is still an open issue. Therefore, for future work, we intend to investigate the issues involved in developing computational formalisms capable of modelling and analysing dynamic reconfiguration of dependable real-time systems. Future work will also need to focus on bigger size industrial case studies to collect further evidence about the paper statements. In this section we will discuss the lesson learnt from the modelling, analysis and implementation of the office workflow by means of different formalisms.

6.1 How Modelling Influences Design

Modelling is part of the design phase and as such it necessarily influences design decisions. It is certainly obvious and well accepted that design, like every other phase of the software process, is not completely independent but it is strongly interconnected to the other preceding and following phases as it is influenced by other external constraints and tools in use. Given that, it is no surprise that the design you see in section 2.2 has been influenced and, as a consequence, altered by and during the modelling.

During this work we had to go through a number of design decisions that would be too long to detail here and it would be certainly out of the scope of this work. However, in this paper, we want to illustrate the different perspectives
that different formalisms have on a design, so there is a point we would like to
make about design decisions and this will also connect with some future work we
intend to do. It is not difficult to imagine that different formalisms have different
biases to design precisely because of their different perspectives. For example,
look at picture 4 where a different design is proposed for the transition. We
only show here, for brevity, the alternative transition diagram but the reader
can easily figure out that coherent changes need to be done also for the other
two diagrams. Basically what has been changed are the boundaries of the region
to be reconfigured. In this alternative design the Bill and Ship pools are now
separate from the region of reconfiguration.

We went through this (and others) design option during this work but we
have finally decided for the option presented in section 2.2 and there is a specific
technical reason for which we chose this particular design. We want to explain our
choice here to define better the role of modelling in design and what different
formalisms with different perspectives bring to the final decision, as well as
formalisms strengths and weaknesses with respect to a specific design.

The formalists job is to model what the system designers produce and tell
them to change their design if it cannot be modelled or is unverifiable. Our ex-
perience with asynchronous $\pi$-calculus first and Web$\pi_\infty$ then suggested us that
extending the boundaries of the region to reconfigure to include billing and ship-
ning was a practical choice. This is because, in the asynchronous $\pi$-calculus (and
in Web$\pi_\infty$ as a consequence), two outputs cannot be in sequence so, in order
to impose ordering between Bill and Ship we had to enlarge the boundaries
of the region to include the processes in the environment of the workflow that
synchronize with them. This very technical aspect is detailed in section 4.1. So
it is easy to see the negative side of this technical solution: we have been forced
to include in the region to be reconfigured parts of the system that were not
intended to be changed. Here the asynchronous $\pi$-calculus shows its weakness in
term of reconfiguring processes dynamically. A $CCS^{dp}$ driven design might have
been different since $CCS^{dp}$ has a synchronous nature. However, we have decided
to proceed with the design that worked better for the majority of formalisms
in this paper. We intend, as a future work, to proceed with a deeper analysis
of alternative design choices for this case study analyzing different design op-
tions according to multiple design dimensions, some lower level like single/multi
threaded workflows, while other higher level like reconfigurations from/to sepa-
rate/disjoint/overlapping workflows.

Since the design has been here modeller-driven the choice presented in this
section has been finally discarded. However, there is a strong possibility that a
(BPEL) developer driven choice might have been very different. We suspect that,
although the chosen version is easier to understand, a BPEL designer would have
had a go for the other one. This because keeping the configuration and the service
modules separate is quite a natural choice in Service Oriented Architectures. Also
people designing reconfigurable systems tend to do the same. For example, the
reader might have a look at the approach followed by the researchers working
on CONIC [31] and Darwin [30] at Imperial College.
6.2 Hierarchy of Correctness Criteria

The entire software cycle, including design and implementation, can benefit from the use of formal methods and in this paper we aimed at showing how this is achievable focusing on a specific case study and using general purposes formalisms as well as tailored ones. Modelling itself is interesting and useful, especially for specification and design purposes, but even more interesting and worth to be investigated is how a specific formalism allows (possibly automatic) verification of properties. In the past sections of this paper models of the case
study have been presented using different categories of formalisms and, for each of those, the requirements presented in section 2.1 have been discussed.

When it comes to evaluate correctness criteria, we can organize them in three different categories: formalism-dependent, formalism-independent and domain-specific. Some correctness criteria certainly depend on the class of formalisms which has been used for the modelling: language inclusion is appropriate for automata or Petri Nets but it would not work for process Algebra for example. On the other side, syntactic inclusion and equivalence works better for processes like in \( \pi \)-calculus and the same for behavioral inclusion and bisimulation. Despite the strong emphasis on bisimulation in some of our previous works, we agree on the fact that bisimulation is not “the” correctness criterion for dynamic change, in the sense it might be one but probably in the case of evolution is not always applicable.

Others correctness criteria should instead not be depending on the specific formalism, for example when we want to detect deadlock or termination (when possible). It is certainly possible to think about a hierarchy of correctness criteria where bisimulation place itself quite high in the hierarchy. Although a complete categorization of correctness criteria is far beyond the scope of this paper we still think it is worth saying something on the topic of bisimulation before proceeding with modelling.

**Bisimulation for Evolution and Change** Bisimulation is a possible definition for program equivalence, i.e. it is fine for proving correctness of refinement or of service/component substitutions, where you want to show that the behavior is preserved. This is not necessarily the case in evolution: you want the program to do new/different things. In some cases bisimulation can be good anyway: if you want to improve only non functional properties, e.g. response time, then you can use bisimulation to show that your new implementation is functionally equivalent, abstracting w.r.t. response time (probably using a weak bisimulation). If instead you want functional improvement, it is clear that you want a different, non bisimilar, behavior, thus you don’t want bisimilarity as correctness criterion.

We investigated bisimulation as a correctness criterion for Process Algebra involved in dynamic change. One of the author exploited it as a criterion for Web services “swap”, or dynamic replacement ([35], [32]). That is because, without altering the BPEL description, one (or more) of the services in an orchestration might have been dynamically replaced, provided that the behavioral interface is not altered. And this behavioral interface turned out to be bisimulation. In this paper, instead, we did not constraint ourselves with replacing ”processes” (in \( \pi \)-calculus terms) that are in some strict relationship each other. In [14], workflow replaces regions of the Petri net that, expressed in \( \pi \)-calculus, would not necessarily be bisimilar, although this concept is not explicitly addressed in their work.
Acknowledgments

This work is partly funded by the EPSRC under the terms of a graduate studentship. The paper has been improved by very useful conversations with Cliff Jones, Alexander Romanovsky, Jeremy Bryans, Gudmund Grov, Mario Bravetti, Massimo Strano, Michele Mazzucco, Anjan Pakhira, Peter Andras and Paolo Missier. We also want to thank members of the Reconfiguration Interest Group (in particular, Kamarul Abdul Basit, Carl Gamble and Richard Payne), the Dependability Group (at Newcastle University) and the EU FP7 DEPLOY Project (Industrial deployment of system engineering methods providing high dependability and productivity).

References


