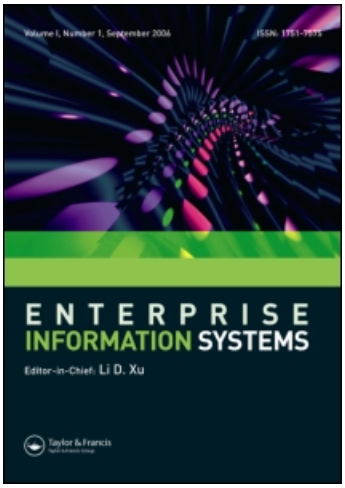

Enterprise Information Systems
Publication details, including instructions for authors and subscription information:
http://www.informaworld.com/smpp/title~content=t748254467

Modelling dependable collaborative time-constrained business processes

Alfredo Capozucca [a];Nicolas Guelfi [a]
[a] Laboratory for Advanced Software Systems, University of Luxembourg, Luxembourg

Online publication date: 21 April 2010

To cite this Article Capozucca, Alfredo andGuelfi, Nicolas(2010) 'Modelling dependable collaborative time-constrained business processes', Enterprise Information Systems, 4: 2, 153 — 214
To link to this Article: DOI: 10.1080/17517571003753266
URL: http://dx.doi.org/10.1080/17517571003753266

Enterprise Information Systems
Vol. 4, No. 2, May 2010, 153–214

Taylor & Francis
Taylor & Francis Group

Modelling dependable collaborative time-constrained business processes

Alfredo Capozucca* and Nicolas Guelfi

Laboratory for Advanced Software Systems, University of Luxembourg, 6, rue Richard Coudenhove-Kalergi, L-1359, Luxembourg

(*Received 31 December 2009; final version received 5 March 2010*)

The effectiveness of the information system that a particular organisation uses for running its business depends largely on the success in modelling such business. This is due to the fact that the business model defines the requirements of the information system that will support the running of the business. Nowadays, there exist many business process development methods that are supported by modelling notations and tools. Unfortunately, these methods are not capable of modelling jointly complex collaborations, time constraints and to offer means to support resilient business process engineering. This article presents a business process language called DT4BP, which has been designed to drive the modelling of dependable, collaborative and time-constrained business processes. The presentation of DT4BP is made to target end users, which are business processes modellers.

Keywords: business process; business process modelling language; resilience; dependability; time; collaboration; fault tolerance; transaction processing; exception handling

1. Introduction

In today's world, every organisation[1] makes use of an information system to run its business. This means that the information system has to be aimed at helping the organisation to succeed in providing the service or product it offers. The effectiveness of the information system depends largely on the success in modelling the business the organisation runs. A business model is a simplified view (i.e. abstraction) of the business that has to be supported by the information system. The business model thus can be considered as the requirement document for such information system.

A central concept used for modelling business is the *business process*. A business process is defined as *a set of one or more linked procedures or activities which collectively realise a business objective or policy goal, normally within the context of an organisational structure defining functional roles and relationships* (Workflow Management Coalition 1999). A *process definition* is a description or representation of what a particular business process is intended to do. The process definition then is the model that captures those relevant aspects of the business that are required to be supported by the information system.

When using the notion of business process to model the business a particular organisation runs, information technologies that focus on process management are

*Corresponding author. Email: alfredo.capozucca@uni.lu

ISSN 1751-7575 print/ISSN 1751-7583 online

DOI: 10.1080/17517571003753266
http://www.informaworld.com

good candidates to achieve the information system that will provide the required support for running such business. Workflow management systems (WfMS) and Enterprise resource planning (ERP) systems are two distinct solutions that focus on business processes that have received special interest in the past two decades (Cardoso *et al*. 2004). A WfMS is a system that defines, creates and manages the execution of business processes through the use of software, running on one or more workflow engines, which is able to interpret the process definition, interact with business process participants and, where required, invoke the use of information technology (IT) tools and applications (Workflow Management Coalition 1999). On the other hand, an ERP is a generic off-the-shelf system that supports most of the key functions (e.g. logistics, sales and financial management) any organisation requires (Soffer *et al*. 2003). Since an ERP is a generic system, its implementation in a particular organisation involves a process of customisation in order to align it with the specific needs of the organisation. The ERP capabilities are configured according to the information provided by the process definition, since it is the model that describes the requirements of the organisation. Therefore, the process definition plays a key role in the achievement of the appropriate information system (whether it is a WfMS or an ERP) that supports the running of the organisation business.

The person in charge of providing the process definition is usually called *business analyst*. As its name states, such person should be business oriented since the modelling of processes requires understanding about organisational issues, policies and corporate directions. People with a background in software engineering may also participate in the production of such process definition since the business solution is required to be aligned with the information system that supports its execution. The fact that people with two different profiles (i.e. business and IT) need to cooperate and communicate in the production of business processes, the comprehensibility of the modelling notation (i.e. the notation should ease the writing of models capable of being understood by every stakeholder) is an important aspect to be considered. Suitability and expressiveness are also other important characteristics to consider when choosing the modelling language. The former determines the easiness with which a concept can be captured, whereas the latter focuses on the facilities to express all those different concepts that can be of interest for the kind of business process being addressed.

The kinds of business processes we are interested in developing are those that are resilient. We consider *resilience* as the capacity of a business process to recover and reinforce itself when facing changes (Guelfi *et al*. 2008) such that it improves its dependability. A dependable business process is one whose failures (i.e. the business process misses its goal) are not unacceptably frequent or severe (from some given viewpoint). Business process reinforcement is achieved by following an iterative development process aiming at improving the dependability of the business process. Recovering capabilities are achieved by introducing recovery activities explicitly into the business process model. For this, we define a business process[2] as resilient when it has capabilities that allow its business objective to be reached either as initially promised or partially (but still good enough to satisfy the requester's expectactions) when facing situations that, without such capabilities, would lead the business process to fail (i.e. to miss its business objective or goal). Situations that may lead a business process to fail range from technological (e.g. Y2K problem) or financial circumstances (e.g. subprime mortgage crisis) to environmental ones (e.g. global greenhouse effect), but all of them have as common point the characteristic of

forcing *changes* in the way the business process is being performed. The ability of the business process, given by its capabilities to tolerate, recover and reinforce, at each iteration of the business process modelling, to these forced changes while providing its business objective, is what defines its resilience.

Once again, the capabilities that make a business process to be resilient should not only include features to allow it to avoid failing when facing such situations, but also to improve the way the business process responds to already-faced situations. This means that for a particular situation that led the business process to deliver a degraded business objective, the next time it faces with such situation a less degraded business objective would be provided. In this manner, it is expected that at certain point in time the business process has the ability to deal with the situation in a manner that its business objective is fully provided as promised originally. This points out specially that the resilience has to be evaluated over the life time of the business process as described by Hamel and Välikangas (2003): '*rather than go from success to success, most organisations go from success to failure and then, after a long, hard climb, back to success*', and not only with respect to a punctual situation (or change). Therefore, a business process is considered as resilient even when it does not succeed to cope with a particular change for first time, but it will do for the next time due to its capabilities for continuous reconstruction and improvement. In this article, we offer this capacity at development level but future work will focus on improving our modelling language with reflexive concepts thus reaching a reflexive modelling language for resilience.

Since our goal is to rely on resilience to improve the dependability of business processes, we will refer to the targeted business processes as *dependable* business processes.[3]

It is up to the stakeholders (i.e. owners of the business, business analysts and IT managers and developers) to define what are those expected changes the business process might face with, and above all, what are the required activities that will have to be performed in each case to allow such business process to not to fail. Note that it will depend on the severity of each expected change whether the business process reaches its goal totally or partially. Those changes that stakeholders did not either foresee or deny as possible to happen define the set of unexpected changes. When the circumstances force the business process to face with an unexpected change, the business process (usually) will fail. In this scenario, stakeholders have to make the choice to turn such unexpected change to an expected one, and then modify the business process to allow it to manage the situation properly next time it comes. If no action is taken (i.e. the business process will fail again when the situation comes), it is said that the business process has not improved its resilience. It is worth explaining that not always a business process that faces with an unexpected change will unavoidably fail. It may be the case the business process has the ability to cope with an unexpected change if it appears that such change is implicitly included in the total set of expected changes.

Once stakeholders have made their choices regarding those changes for which the business process will have the ability to accommodate while providing its business objective either as promised originally (in the best case) or at least partially, the modelling of resilient business processes is reduced to describe every expected change, along with the activities that will allow the business process to accommodate to such change when it takes place (aka resilience policy or handler).

Other characteristics that narrow the kind of business process being targeted are the notion of *collaboration* and *time*. This means that we are not only interested in

dependable business processes but also in those that are *collaborative*: multiple participants take part along the process to reach its goal), and *time constrained*: at least one of the elements that constitutes the process is time-dependent.

A business process, as stated in its definition, takes place within the context of an organisational structure. It is such structure that determines how the work carried out by such organisation is divided amongst its staff. In today's world, companies require several people with different skills to succeed in providing the services they offer. This requirement leads business processes to be composed of multiple participants that collaborate along the process to succeed in providing the offered service or product (i.e. the goal of the process). Each participant is assumed to have its own processing capacity for performing activities. Thus, while a participant denotes a business entity capable of executing its activities in parallel with some other part, a collaborative business process denotes a set of participants that interact among them. Two participants interact if they communicate by exchanging messages. The fact that they interact is what defines them to be collaborative with respect to the reaching of the process's goal.

Time constraints are different rules that determine how a particular business process behaves from the time point of view (Marjanovic 2000). Such kind of constraints can be used to determine the maximum allowed duration of the overall business process, or just one of its activities. In this manner, a time-constrained business process is one that includes at least a time constraint in its definition. Indeed, it is very frequent to find business processes that are, in one way or another, time dependent. These time-related dependencies can be found in organisational rules, laws, commitments, policies, standards, or just in end-user's requirements.

Business analysts in charge of modelling this particular kind of business processes (we name them *Dependable Collaborative Time-Constrained* business processes (DCTC-BPs)) will look for a notation that embodies collaboration, dependability and time. Having a notation with these characteristics, business analysts expect the modelling of DCTC-BPs to turn easier. This work describes *DT4BP – Dependability and Time for Business Processes (DT4BP)*, a business process modelling language meant for fulfilling such requirement.

This article is organised as follows. Section 2 introduces the basic definitions and concepts that underlie the principles on which the language is founded. Section 3 presents the existing business process modelling languages and their support regarding the viewpoints of interest (i.e. dependability, collaboration and time). Section 4, which is the core of the article, describes the DT4BP business process modelling language. Since the intended audience is users of the language (i.e. business analysts and/or software engineers), and following the recommendations of Harel and Rumpe (2004), the language definition is given in natural language with many examples such that both its notation and semantics are carefully described. Finally, Section 5 presents our conclusions and future work.

2. Background on dependability

The aim of this section is to give a more detailed presentation of the background coming from the dependability computing area (IFIP WG 10.4 on Dependable Computing and Fault Tolerance 1960), with special emphasis on those concepts that underlie the notion of dependable collaborative time-constrained business processes as considered in this work. The section starts giving the definition of dependable

business process, and then introduces the concepts and principles coming from the dependable computing area on which such definition relies on.

2.1. Dependable business process

A business process definition is expected to capture all the possible paths that make its goal reached. A process instance that follows one of these paths is termed *well-behaved or normal*. It might be the case that a particular process instance follows a path which has not been considered in the definition. As it is assumed that all possible paths that make the business process to reach its goal are specified in the process definition, it can be concluded that instances following a non-specified path fail in reaching the business process' goal. It is assumed that once the instance has deviated from the process definition it will eventually fail, if no corrective activities are performed. A process instance missing the goal is termed *business process failure*. A *dependable business process* is one whose failures (i.e. process instances missing the goal) are not unacceptably frequent or severe (from some given viewpoint). Therefore, a business process *increases* its dependability when for those events that make it fail more often than expected, corrective activities are explicitly included in its definition such that instances do not miss the goal in spite of facing such undesirable events. In this manner, let BP_{def_1} be the original business process definition, and let BP_{def_2} be the new one obtained by including the corrective activities to deal with the undesirable event; it is said that BP_{def_2} is *more* dependable than BP_{def_2}. New definitions of the process are provided (i.e. $BP_{def_3}, \ldots, BP_{def_{(n-1)}}$) until a definition BP_{def_n} is reached such that it includes corrective activities for every undesirable event that make it fail more often than wished.

The corrective activities to be included in a process definition depend on the particular kind of undesirable event to deal with. Depending on the severity that such kind of event produces when it occurs, it might be possible to recover (1) completely, (2) partially or (3) not possible to recover at all. In any case, the required activities to deal with problems have to be included explicitly in the business process definition. The initial business process goal should always be considered while redefining the dependable business process. This is called *explicit dependability*.

2.2. Dependable computing

In the software engineering literature, the term *dependability* is defined as *the ability to deliver service that can justifiably be trusted* (Avizienis *et al*. 2004). Such service is delivered by a system: i.e. an entity made of hardware, software and humans. In the business domain, the service is the business process, whereas the system is the business organisation that owns the process.

As it is assumed that systems are not perfect, they are expected to fail from time to time. A system fails (aka system failure, or simply *failure*) when there is a service failure: i.e. the delivered service is judged (by a particular judgemental entity (Randell and Koutny 2007)) as different from what it is intended to do. As system failures are unavoidable, the challenge is to reduce their frequency and severity. A dependable system thus is one that has the ability to avoid service failures that are more frequent and more severe than is acceptable from the judgemental system's point of view. The term 'judgemental system' covers from failure detectors

implemented in hardware or software till courts of justice. Furthermore, since it is a system, it might also fail (as judged by another higher judgemental system). In this reported work, as it will be shown later, the role of judgemental system is played by the stakeholders that request the service to be provided. An *error* is the part of system state that might lead to a failure. The hypothesised cause of the error is a *fault*. An error does not necessarily lead to a failure as it may be avoided by chance or design, or simply because it does not constitute a fault for the enclosing system.

There exist four general means to achieve dependability (Avizienis *et al.* 2004): fault prevention, fault tolerance, fault removal and fault forecasting. Fault prevention deals with the objective of avoiding to introduce faults during the software development process. As such objective is part of the general aim of every software development methodology, fault prevention can be considered as an inherent part of it. Thus, good practices in software development (e.g. modularity with low coupling and high cohesion, information hiding, use of strongly typed languages for the specification, design and implementation) help reducing the number of faults when developing a system. Fault tolerance is aimed at allowing the system to provide the service in spite of the presence of faults. The basic activities required to achieve fault tolerance are error detection (i.e. to identify the presence of an error) and system recovery (i.e. to lead the system to a well-defined state – state without detected errors from where the system can continue its normal execution). Fault removal deals with uncovering faults that have happened at any phase of the development process. Activities covered by this means range from checking the specification of the system for uncovering specification faults till exercising the system (i.e. testing) to find development faults. Fault forecasting is aimed at evaluating the behaviour of the system under the occurrence of faults such that it can be concluded which ones would lead to system failure.

The effectiveness of each means depends on the context and nature of the fault. Thus, the dependability of a system can be increased by a combined used of these means. It is worth mentioning that a total dependable system (i.e. a perfect system) is something impossible to be reached, but it should be the objective that any development process must try to approximate. This work takes a view centred around fault tolerance as means to achieve dependability, which is complemented with the other existing ones. Based on the assumption that faults cannot be fully avoided or removed, the choice is to enrich the system with means to detect erroneous system states and then to perform the necessary recovery steps that lead the system to a well-defined state (fault tolerance view). Both the error detection means and the recovery steps are made part of the model that describes the system. This model is produced during the analysis phase of the development methodology that is being followed (fault prevention view). The model does not only describe the functional aspects of the system but also the fault tolerant ones. Such model can be used to perform an early evaluation of the system behaviour with respect to its adherence to the expected functional properties as well as the occurrence of faults (fault forecasting view).[4] If such early evaluation reports that the system model either does not adhere to certain functional property or does not behave as expected when facing a particular fault, then the model has to be corrected because it is faulty (fault removal view). This early evaluation is carried out till the model fulfils both the functional and fault-tolerant aspects. Once such point is reached, the next phase of the development process (i.e. design) is started.

2.3. Fault tolerance

Fault tolerance is achieved by error detection and recovery. There are two kinds of error detection techniques: concurrent and preemptive (Avizienis *et al.* 2004). Systems that allow error to be detected during the delivery of the normal service are said to support *concurrent error detection*, whereas those that can detect errors only while running on specific modes or at particular period of time (e.g. audit and start up, respectively) are said to support *preemptive error detection*.

The recovery phase aims at leading the system back to certain state such that it can continue executing. This can be done by either removing the fault from the system to avoid being activated again (aka *fault handling*), or modifying the system state such that it does not contain errors (aka *error handling*). Fault handling implies to identify the fault (i.e. diagnosis), isolate the faulty element (e.g. component, module, class) reassign the tasks performed by the faulty element among non-faulty elements (i.e. system re-configuration) and restart the system.

Error handling can be implemented by backward error recovery (sometimes called rollback), forward error recovery (sometimes called rollforward), compensation or any combination of them. Backward error recovery is aimed at returning the system into a saved state that existed before the error occurrence. Such state is assumed to be correct since in the past it allowed the system to be fully operative. Implementations of this technique are checkpoints (Elnozahy *et al.* 2002), conversations (Randell 1975) and transactions (Gray and Reuter 1992).

Forward error recovery is aimed at leading the system towards a new (i.e. not reached recently) correct state. Reaching such new state is only possible when there exists precise knowledge about the kind of error that has corrupted the system state. In that case, specific activities meant to deal with such particular kind of error are performed. By executing these activities, the correct new state should be reached allowing the system to resume its operation either as before the error detection (best case scenario) or in a mode where not all its services are available (aka degraded mode). Forward error recovery is usually achieved by using exception handling mechanisms (Cristian 1989, Buhr and Mok 2000) as they embody concepts (i.e. exception and handler) and capabilities (e.g. detection, control flow transfer, exception and handler categorisation) that make this type of recovery easy to implement.

Compensation is aimed at allowing the system execution to progress while failures occur. This technique is implemented under the assumption the system holds enough redundant information to mask the occurred failure. This means that in spite of the failure having occurred it is not externalised (i.e. such failure is imperceptible by system's users). Hardware redundancy includes supplementary (potentially similar) hardware to the system, whereas software redundancy includes additional components (i.e. programs, objects) or data. Software redundancy is complemented with software diversity to solve the problem of replicated design and implementation faults. N-version programming (Avizienis 1985) and recovery blocks (Horning *et al.* 1974, Randell 1975) are the original and basic techniques that implement software diversity.

2.4. Fault tolerance in distributed real-time systems

As already mentioned, the approach to achieve dependability in business processes is centred around fault tolerance. As such means will have to be used in business processes that are collaborative and hold timing constraints, it is logical to explore

how fault tolerace has been successfully applied in the computing area when engineering 'collaborative systems with timing constraints' (known as distributed real-time systems in the computing field).

A 'distributed system' (Lamport 1978, Burns and Wellings 2001) is defined as a system composed of multiple autonomous processing nodes cooperating in a common purpose or to achieve a common goal. These autonomous processing nodes communicate with one another by exchanging messages. It is a necessary condition for a distributed system to be considered as such, that the message transmission delay is not negligible compared with the time between events in a single processing node (multiprocessor computers are excluded as the message transmission delay is negligible). This definition is compatible with our way of considering collaboration in business processes.

A system is 'real-time' (Burns 1991) if at least one of the computational tasks that it executes is constrained somehow by time (compatible with our way of considering time constraints). Such definition points out that the correctness of the result provided for the task depends not only on its logical value, but also on the time at which it is provided. These timing constraints appear in the requirements specification in the form of deadlines. This definition is compatible with our way of considering timing constraints in business processes.

Distributed systems (being real-time or not) have the partial-failure property: the occurrence of a failure (it does not matter of which kind) usually affects only a part of the whole system (Tel 1994). Fault tolerance techniques exploit such property with the aim of coordinating the execution of the distributed system so that non-faulty processing nodes can take over the activities of those that are failing.

Fault-tolerant algorithms based on replication (i.e. systematic compensation for fault masking) are an option since (potentially) every processing node can be used for redundancy purposes. Every system component that needs to be replicated (nothing forbids to replicate the entire system) can be deployed on one of the processing nodes that compose the distributed system. There also exist fault-tolerant algorithms aimed at ensuring the correct behaviour of the system (while certain conditions hold) in spite of failure occurrences (Lamport *et al.* 1982), while others are meant for identifying the kind of occurred failure (even in the case of multiple occurrences) to perform the necessary recovery actions that will lead the system to either its normal behaviour (best scenario) or a graceful degradation, instead of reaching an overall malfunctioning (Campbell and Randell 1986). Such fault-tolerant algorithms have to coexist with scheduling algorithms that take care of the temporal behaviour of the distributed system, when timing constraints are required to be met. In this scenario, a trade-off between dependability and performance has to be made.

Fault tolerance techniques can be also applied within each processing node that is part of the real-time distributed system. The goal in doing so is to increase the dependability of the local computation carried out in the processing node. An approach is to combine exception handling principles with scheduling practices for providing fault tolerance by means of forward error recovery (Lima and Burns 2005). This approach categorises processes or activities as primary or alternative tasks: a primary task is one whose execution is required in error-free scenarios, whereas an alternative (i.e handler) is one that must be executed only when some error is detected (i.e. exception is raised). Such categorisation is aimed at helping in the scheduling of the tasks. Since an alternative task is expected to run less often than a primary one, different priorities can be assigned to them. It might be decided that,

for example, alternative tasks run with higher priorities as a way to increase the tolerance to detected errors.

Other alternative is to use the transaction processing paradigm for executing the timing constrained tasks within a processing node. Turning every task a transaction allows fault tolerance to be provided by means of backward error recovery (atomicity property – the A of the ACID[5] properties granted by the transaction processing paradigm). However, the transaction processing mechanism (whatever it is) embedded into the processing node has to include scheduling aspects so that the number of transactions missing their timing constraints are minimised (Abbott and Garcia-Molina 1992). Advanced transaction mechanisms (Romanovsky *et al*. 1999) can be used when some of the ACID properties need to be relaxed, while keeping the others. This is the case for long-running transactions (i.e. transactions intended to run over long periods of time) that keep consistency and durability (C and D), whereas relax atomicity and isolation (A and I) (Gray 1981).

3. Existing business process modelling languages

This section examines business process modelling languages that are currently being used to specify business processes. Among all the different existing languages oriented towards the modelling of business process, we considered only those that represent a standard and/or are widely used both in the industrial sector as in the academic field. Based on this selection criteria, the specific business process modelling languages to be examined are:

- Unified modelling language Activity Diagram (UML-AD) version 2.2 (Object Management Group (OMG) 2009)
- Business Process Modelling Notation (BPMN) version 1.2 (OMG 2009)
- Yet Another Workflow Language (YAWL) (Yet Another Workflow Language 2009)
- Event-driven Process Chain (EPC) as implemented by ARIS Express 1.0 (ARIS Community 2009)

It is worth explaining that standards like XPDL (Workflow Management Coalition 2008) or BPEL (OASIS 2007) are not considered in this examination because their aims make them to be out of the class of business process modelling languages as considered in this work: XPDL is aimed at facilitating the interoperability between workflow technologies, whereas BPEL targets the modelling of business process made of web services, with special emphasis on the description of the interactions between the web-services that compose the process.

These languages are analysed with respect to the kind of business processes this work targets (i.e. DCTC-BPs). Therefore, we put special emphasis on the features such modelling languages provide for describing participants and their collaboration, time constraints and dependabilities aspects.

3.1. UML-AD

3.1.1. Generalities

A UML-AD is one of the different kinds of diagrams the UML (OMG 2009) provides for describing behaviour. A UML-AD can be considered as a directed

graph, since it describes a set of nodes (aka activity nodes) being bound by directed connections (activity edges). Activity nodes connected by activity edges define the potential execution flows to be followed once the UML-AD where they are contained is initiated.

An activity node can be an *action*, or an *activity*, or a *flow-of-control construct*. An action represents a single step within an activity (i.e. it is not further decomposed within the activity diagram). An activity represents a step that is composed of individual elements that are actions. A flow-of-control construct is a node that allows coordinating the execution flow in a UML-AD (e.g. *DecisionNode, ForkNode, JoinNode, MergeNode*).

Incoming and outgoing activity edges are used in activity nodes not only to specify the control flow from and to other nodes, but also the data flow. In this manner, incoming edges can be used by an activity node to get its data inputs, whereas outgoing edges can be used to deliver data information to the enclosing context of such activity node.

3.1.2. Collaboration

Activity nodes can be grouped. An activity group is a construct that allows a set of activity nodes and edges to be joined in a same group. Nodes and edges can belong to more that one group. An activity partition is a kind of activity group for identifying actions that have some characteristics in common. A UML-AD then can be divided in partitions such that its contained nodes are organised in a way that the understanding of the UML-AD becomes easier. It is argued by OMG (2009) that a partition often corresponds to an organisational unit, when using UML-AD as means to model business processes.

3.1.3. Dependability

An activity node may have associated pre- and post-conditions. A pre-condition is a constraint that must be satisfied when the activity node execution is started, whereas a post-condition is a constraint that must be satisfied when its execution is completed.

An activity node is considered as *protected* when an *exception handler* is attached to such node. An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node. The handler body is an activity node, that does not have any explicit input or output edges, and may access its enclosing context as the protected node does. It is worth mentioning that an activity group can play the role of protected node. In this case, such activity group is named *interruptible activity region*.

A protected node might have more than one exception handler, and a handler might handle more than one exception. Once an exception is raised, a handler starts searching for handling such an exception. The selection policy is based on type matching: the type of the raised exception has to match with one of the types held by a handler. In case that more than one handler satisfies the selection policy, only one is selected in a non-deterministic manner.

If none of the handlers satisfies the selection policy, then the exception is propagated to the enclosing context that contains the protected node, and then the handling process starts again. This is repeated until either a handler is found or the

outmost level is reached and the exception is not caught (in whose case, nothing can be said about the UML-AD behaviour).

Once a handler body ends its execution, the control flow continues from the protected node in the same manner as the raised exception would have not occurred.

3.1.4. Time

Time information on UML-AD is described using the notion of *accept event action*. As every action, it describes a single step within an activity, but in this case, this kind of action happens only when its associated condition is satisfied. Thus, an Accept Event Action is used to capture the occurrence of a particular event that satisfies certain condition. When the event to be captured is a time event, the action produces as result (i.e. output parameter) the point in time at which the event occurred. In this case, the action is informally called a *wait time action*.

3.2. BPMN

3.2.1. Generalities

BPMN provides a Business Process Diagram (BPD) to allow business processes to be modelled. A BPD then is meant to contain one or more business processes, which are all described using a graphical notation. This graphical notation is a flowchart-like notation that has its elements divided in four categories:

(1) *Flow objects*: These are those elements used to define the behaviour of a business process. There are three flow objects: *events, activities* and *gateways*. Events are used to model the 'happening' of something during the course of a business process. Depending on when such an event happens, it is categorised as *start, intermediate* or *end*. An activity is used to describe a piece of work. When an activity is not divided in sub-activities it is called atomic, otherwise it is non-atomic. Gateways are used to control the flow of the business process by means of decision, forking, and merging points.
(2) *Connecting objects:* These are those elements used to connect flow objects. There are three kinds of connecting objects: sequence flow, message flow and association. A sequence flow is used to show the dependency order between flow objects. A message flow is used to show the flow of messages between two different participants. A participant represents a business entity or role that is modelled as a *Pool* in a BPD (see below).
(3) *Swimlanes:* they are elements used to group activities. There are two grouping elements: *pools* and *lanes*. A pool is used to group those activities that a same participant has to perform in a business process. A pool then allows a business process to be partitioned in such a way that the activities are divided according to the participant that performs them. A *lane* is a sub-partition within a pool, used to organise and categorise the activities a same participant performs.
(4) *Artifacts:* These are those elements used to provide additional information about the business process being modelled, without affecting the sequence flow or message flow of such process. The artifacts provided by the notation are: *data object*, *group* and *annotation*. A data object is used to model the

information an activity requires to be performed and/or what is the information produced once such an activity has been executed. Both group and annotation are artifacts meant for easing the documentation of the business process. While the former is used to define categories of activities, the latter allows the business analyst to provide additional information for the reader of the BPD.

These notation elements provide the necessary support to model concepts that are applicable to business process, only. This means that other kinds of modelling done by organisations like business rules, data models and organisational structures and resources are out of the scope of BPMN.

3.2.2. Collaboration

BPMN defines a collaboration as any BPD that contains two or more participants (shown by pools) and shows their interactions. These interactions are defined as the communication, in the form of message exchanges (shown by message flows) between two participants. Therefore, the pool and message flow element provided by the language are the necessary means to model the collaborative aspects of a particular business process.

It should not be surprising that BPMN considers collaboration as a first-class concern, because support for it was sought since its inception (White and Miers 2008).

3.2.3. Dependability

BPMN provides elements to support exception handling. An *intermediate event* attached to the boundary of an *activity* is used to model an *exception* that may occur during the execution of such activity. In case the exception is raised (i.e. the trigger the intermediate event holds is fired) the activity is interrupted. As the *normal flow* is the one followed by the process when the activity terminates its execution normally, the *exception flow* is the one to be followed when an intermediate event occurs. Intermediate events attached to an activity are used to denote the different exceptional flows such an activity can follow. It is worth noting that the flow from the intermediate events (i.e. exceptional flows) can go anywhere. It can go to a completely new path to perform some handling activities for the faced exception, or join the normal path as if the exception would have not occurred, or it can go back to attempt a new execution of the activity.

BPMN also provides support for the notion of *transaction*. In BPMN, a transaction is defined as *a formal business relationship and agreement between two or more participants* (White and Miers 2008). A transaction is considered as successfully executed when all the participants taking part have reached a common agreement point.

Since a transaction involves multiple participants, its definition is spread over the pools that describe each participant. Each participant (i.e. pool) includes a *transactional sub-process* to define those activities that form a part of the transaction. A double-lined boundary indicates that a sub-process is a transaction.

In case that any of the participants taking part in the transaction faces with a processing or technical error, then the transaction is interrupted. There are two

possibilities for interrupting a transaction: it can be terminated immediately or some compensation can be performed before terminating such as the transaction is considered as cancelled. Intermediate events attached to the transactional sub-process boundary are used to represent each kind of interruption. An *error intermediate event* is used to model the sudden interruption of the transaction, whereas a *cancel intermediate* represents the cancellation of the transaction.

As it was just mentioned, cancelling a transaction may require to perform some compensation. Compensation is the undoing of work that a particular activity has completed. Since compensation does not happen automatically, another activity is required to undo the work of the original activity. BPMN supports compensation by means of *compensation activity, compensation intermediate event* and *compensation association*. Every activity that requires compensation must have a compensation intermediate event attached on its boundary. A compensation association is used to bind the original activity which is the one that compensates its effects. The compensation association must start at the compensation intermediate event attached to the original activity and end into the compensation activity. It is worth noting that a compensation activity must not have any incoming or outgoing sequence flow, and that its use is not limited to activities that form a part of a *transactional sub-process*.

3.2.4. Time

Time is supported in BPMN by the notion of *timer start event* and *timer intermediate event*. A timer start event is used to start a business process, whereas a timer intermediate event is used to delay the starting of a particular activity or interrupt its execution. Both kind of timers hold a time condition that specifies when the event occurs (i.e. the event is triggered). When a timer is used to start a process (i.e. it is a timer start event) its time condition may be a specific data and time (e.g. 31 December 2009 at 8 am) or a recurring time (e.g. every Monday at 8 am). In any case, the condition is compared with a clock[6] that measures the passage of the physical time.

As said before, a timer intermediate event is used either to delay the execution of an activity or to interrupt its execution. A delay is modelled by inserting a timer intermeditate event between activities in a process. The time condition held by such timer describes the *delay* in starting the next activity of the process. This time condition specifies an absolute time (e.g. wait until 31 December 2009 at 8 am) or a relative time (e.g. wait 2 weeks) that might be repetitive (e.g. wait until next Monday at 8 am). Note that using absolute times for describing the condition of a timer should be avoided as much as possible as they inhibit the re-usability of the process (an absolute time condition will be valid only once). It is worth mentioning that among the flow objects that can precede a timer intermediate event are not only activities, but also gateways, or intermediate events.

A timer intermediate event attached to the boundary of an activity represents a deadline for the execution of such activity. This means that the activity has the time condition described by the timer as maximum allowed time for completing its execution. If the time condition becomes true before the activity is completed, then the activity is immediately interrupted. The time condition that a timer intermediate events used as *time-out* holds is always relative to the starting of the activity where the timer is attached to.

3.3. YAWL

3.3.1. Generalities

YAWL is a modelling language meant for supporting most workflow behaviours that can be commonly found in practice. Such common workflow behaviours are known as *workflow patterns* (Van Der Aalst *et al.* 2003), and they are categorised according to four different perspectives: control flow, data, resource, and exception handling. A business process specification in YAWL is determined by a set of one or more YAWL-nets, which define a hierarchical graph. Every YAWL-net describes part of the work the business process does[7] by means of tasks and conditions.

Tasks are either *composite* or *atomic*. A composite task in a YAWL-net is a reference to another YAWL-net at a lower level in the hierarchy of the graph, which describes the way in which such composite task is defined. An atomic task represents an unit of work that is not further subdivided into sub-tasks.

A condition represents a state of the business process, which can be used to make a choice in the flow of the process. A condition must be located in between tasks, except the mandatory conditions *input* and *output* that every YAWL-net must have to be considered valid. The input condition represents the starting point of the business process, whereas the output condition the end of such business process.

For each YAWL specification there exists one YAWL-net that does not have a *composite task* referring to it and forms the root of the graph. This YAWL-net is known as the *top level process* or *top level net*.

A task is connected to either a condition or another task by a *flow*, which is represented by an unidirectional arrow. A YAWL specification is considered as correct if every task is tied into a YAWL-net via flows that can be traced back to the YAWL-net's input condition, and which eventually lead to the YAWL-net's output condition (Bradford and Dumas 2007). The number of incoming/outgoing flows a particular task can have ranges from one to many. Special decorators have to be added over a task that has more than one incoming or outgoing flow. A *split* decorator is used to specify that the task that owns such decorator is followed for one or more tasks, whereas a *join* decorator specifies what are the required tasks to be completed before allowing the tasks to become available for execution. Both the split and join decorator have the *OR, AND* and *XOR* associated operators, which are used to determine their behaviour. Details about the behaviour of each of them can be found in Rusell's thesis (Russell 2007) on page 252.

3.3.2. Collaboration

Collaboration, as considered in the context of this work (i.e. message exchange between two different participants), is not supported in YAWL. It is worth explaining that in YAWL a participant refers to the actual resource that performs a particular task, whereas the notion of *role* is used to group different participants that share a same feature. Both concepts (i.e. participant and role) are the build blocks used to define the *organisational model* of the enterprise that owns and runs the business process being modelled. The organisational model along with additional constraints are the elements used to specify what are the user(s) that must perform certain task(s) (aka resource allocation policy).

3.3.3. Dependability

YAWL covers dependability by means of exception handling. The actions to be taken such that exceptions that may arise during the execution of a particular business process can be handled, are defined by a set of primitives incorporated into the modelling language. These primitives, which are chained in sequence, define the handling process (called *exlet*) for a particular exception. Such handling primitives are:

- *Suspend workitem*: a workitem (i.e. a process instance activity) is suspended until it is either continued, restarted, cancelled, failed or completed, or the entire process instance is cancelled or completed. It is worth noting that a process instance activity can be suspended only when it has a status of fired, enabled or executing).
- *Suspend case*: a case (i.e. a process instance) is suspended. Suspending a process instance means that every single activity owned by the suspended process instance is suspended.
- *Suspend all cases*: all process instances of a particular business process definition are suspended.
- *Continue workitem*: resumes the execution of a process instance activity, which was previously suspended.
- *Continue case*: resumes the execution of a process instance (i.e. every activity owned by such process instance resumes its execution).
- *Continue all cases*: resumes the execution of all process instances belonging to a same business process definition.
- *Remove workitem*: ends the execution of a process instance activity and marks its status as cancelled. Further activities that are in the same process path of the removed activity are not executed.
- *Remove case*: ends the execution of the process instance.
- *Remove all cases*: ends the execution of all process instance of a particular business process definition.
- *Restart workitem*: restarts the execution of a process instance activity under the same conditions as it was when executed for first time (i.e. the process instance activity is rolledback).
- *Force complete workitem*: completes a process instance activity that is currently under execution (i.e. its status is fired, enabled or executing). Such process instance activity is considered as successfully executed, and the execution proceeds to the next process instance activity.
- *Force fail workitem*: fails a process instance activity that is currently under execution (i.e. its status is fired, enabled or executing). Such process instance activity is considered as unsuccessfully executed (but not cancelled), and the execution proceeds to the next process instance activity.
- *Compensate*: executes a business process that, depending on the primitives used previously either runs in parallel along with its parent business process instance, or does it while its parent is suspended or removed.

A handling process that is defined using the above-listed primitives will take over the execution of the parent business process instance only when certain *exception* occurs. An exception is defined as the occurrence of a certain *kind of event* along with the existence of a *rule* that allows such event to be bound with a handling process. The kind of events for which a rule may exist are:

- **Pre and post case/WorkItem execution:** events that notify that a particular case (i.e. process instance) or workitem (i.e. process instance activity) is ready to be executed or it has just concluded its execution. Such events are meant specially to check four kind of rules:
  - **CasePreConstraint:** rules that are checked before the process instance begins its execution,
  - **ItemPreConstraint:** rules that are checked before the process instance activity begins its execution,
  - **CasePostConstraint:** rules that are checked just after the process instance concluded its execution,
  - **ItemPostConstraint:** rules that are checked just after the process instance activity concluded its execution.
- **External trigger:** events that notify the occurrence of something outside of the process instance execution that affects its continuation. Such events are categorised depending on whether they affect the entire process instance (aka *CaseExternalTrigger*) or just one of its activities (ItemExternalTrigger). Based on this categorisation, the event can be handled at the level of either the process instance or one of its activities.
- **Timeout:** event that notifies that a particular process instance activity owning an associated timer has reached its deadline.
- **Resource unavailability:** event that notifies that the selected resource to be allocated for carrying out certain process instance activity is unavailable to accept the allocation.
- **Item abort:** event that notifies that the execution of a particular process instance *automated activity* (i.e. the activity is automatically executed by an external application) has aborted before being completed.
- **Constraint violation:** event that notifies that certain data constraint associated to the process instance activity being executed has been violated.

A rule[8] is a predicate, which is used to determine which handling process, if any, to invoke. In case that for certain kind of event a rule exits and evaluates to true, the handling process owned by such rule is invoked. If there are no rules defined for a certain kind of event for a business process definition, then such kind of event is simply ignored when the process instances are executed.

3.3.4. Time

YAWL has been recently extended with features to define the time behaviour of a particular atomic task (Yet Another Workflow Language 2009 Foundation 2009). These features include the possibility to define a *timer* which is used to constrain the behaviour of an atomic task. The semantics of a timer depends on whether the atomic task is *manual* or *automated*. For a manual atomic task, a timer is used to define a time frame on which the life cycle of such manual atomic task is allowed to take place. For this kind of atomic task, the timer is initiated either when the task is enabled or started. Activation on enablement means that the timer begins as soon as the manual task is enabled, whereas activation on starting it starts only when the task has started (i.e. the task will be first offered, then allocated, and once it is started the timer is initiated).

A timer has an *expiry value* associated, which is used to specify the duration of the time frame such timer defines. Once such expiry value is reached by the timer, the task instance will complete whatever its current status is (i.e. offered, allocated, started). This expiry value of the timer represents either an absolute time (i.e. a specific date and time has to be provided) or a relative time (i.e. the timer will expire once such value has passed since the task was either enabled or started).

For an automated atomic task, a timer behaves as a delay: the task instance created to execute an automated task delays its execution until the expiry value owned by the timer is reached. Once the timer expires, the task is immediately executed and completed.

The other time-related feature provided by YAWL to constrain the time behaviour of a task is the data type *YTimerType*. This data type is used to define parameters which allow the timer settings to be assigned at runtime. The values a parameter of type *YTimerType* must provide are:

- *Trigger*: the starting policy of the timer (the only two valid values are *OnEnabled* and *OnExecuting*;
- *Expiry*: the duration of the timer.

3.4. EPC

3.4.1. Generalities

EPC is graphical language meant for describing business processes in a manner business people can easily understand and use. EPC has become a widespread modelling language because it is the notation used by some of the leading tools in the field of business process management such as SAP R/3 and ARIS (van der Aalst 1999).

A business process is described in EPC as set of *events* and *activities* which chained is sequence define the control flow structure of such business process. An event is used to describe both a condition that leads to an activity to be started and a result of having executed one. An activity describes a piece of work which needs to be done in order to get the entire business process successfully completed. An activity may require some information (i.e. *input parameter*) to be executed or produce some one (i.e. *output parameter*) due to its execution. The *document* element is the primitive provided by EPC that allows information to be modelled in business process definition.

A *connector* is the element used to link the different elements that compose a business process definition. While an activity can be connected to another one by using a connector, events are allowed to be connected to activities.

Rules are special kind of connectors that are used to bind both events and activities. There are three kinds of rules:

- AND rule: the processing steps that follows the rule are performed once all incoming processing steps were completed.
- OR rule: the processing steps that follows the rule are performed once at least one incoming processing step was completed.
- XOR rule: the processing steps that follow the rule are performed once one (and only one) incoming processing step was completed.

Other elements provided by the language are oriented to the specification of the resources in charge of carrying out the business process activities. These elements are *organisational unit*, *role* and *person*. While the element *organisational unit* is used to defined the organisational structure of the enterprise that runs the business process(s) being modelled, the *role* element is used as means to specify the profile, skills or characteristics of the people belonging to a particular organisational unit. The element *person* is used to define an actual person, who usually is bound to at least one role. Such binding means that the person has the abilities to play such role. Thus, it can be specified at modelling time whether an activity is performed always by the same person, or it is deferred until runtime. The former is made by linking the activity to such particular person, whereas the latter is done by linking a role or organisational unit to the activity.

3.4.2. Collaboration

Collaboration, as considered in the context of this work (i.e. message exchange between two different participants) is not supported in EPC.

3.4.3. Dependability

The language (at the writing time) does not provide any element for addressing dependability as considered within this work.

3.4.4. Time

The language (at the writing time) does not provide any element that allows time-related aspects of a business process to be modelled (e.g. activity duration, delay between activities, etc.).

4. The DT4BP business process modelling language

This section describes DT4BP, a novel business process modelling language meant for specifying DCTC business processes. The description of the language is given in two parts. The first part (Section 4.1) is aimed at giving a quick introduction to the essential elements of the language, but without getting bogged down in the details. The purpose is to allow the reader to have an overview of the language and its main constituting parts on which to hang the more detailed descriptions given in the second part.

The second part is covered from Sections 4.2 to 4.5. Through these sections the key concepts[9] that determine the relevant process-related information that business analysts commonly need to include in the modelling of business are detailed along with the primitives[10] that allow such concepts to be represented in a straightforward manner. It is worth emphasising here that these primitives have been chosen in a way that the resulting model not only remains clear and understable, but also small in terms of space to facilitate its reading.

This section concludes with a comparison between the DT4BP and those business process modelling languages that were introduced in Section 3 such that the reader can clearly see where this novel language exceeds the current existing notations.

4.1. A tutorial introduction

As mentioned above, this section gives a quick introduction to DT4BP. At this point, the aim is not to being complete or even precise, so that important features of the language are intentionally left out. The aim is to get the reader as quick as possible to the point where he/she can understand how business processes are modelled in DT4BP. The best way to do this (and probably the only one) is by modelling business processes in it. A fictitious process for diagnosing a patient, adapted from the one described by Mans *et al*. (2009), is used as running example to describe a DCTC business process. Such process is defined by the following activities:

(1) *registration:* a secretary checks the admission of the patient to the diagnosis unit,
(2.1) *examination:* an assistant and a nurse do the first examination of the patient (e.g. checking his temperature and blood pressure) to determine his overall health status, and in parallel,
(2.2) *make document:* a nurse prepares the patient sheet. A patient sheet is a document that includes not only personal information about the patient such as his name, address, phone and age, but also relevant medical information gathered from his local record stored in the hospital's database. The patient sheet is completed along with the diagnosis process. Once such process ends, this document is given to the patient as it describes the treatment to be followed, and the medicine to take, if required,
(3) *consultation:* a doctor evaluates the results of the examination, gives a diagnostic and sets a treatment to be followed by the patient,
(4) *give information:* a nurse gives more precise information to the patient about the prescribed treatment and medicine.

The collaborative aspect of the process is determined by the different kinds of people required to perform it: a secretary, a nurse, an assistant, and a doctor. Since the patient is required to perform some activities during the process, he is also considered a participant.[11] The interaction between participants takes place during the execution of each activity. In the ''registration'', for example, the secretary interacts with the person (i.e. potential patient) who wants to be diagnosed in order to certify that such person has medical insurance. In the ''examination'' activity, the interaction is between the patient, a nurse and an assistant. The patient also interacts with a doctor and a nurse in the ''consultation'' and ''give information'' activities, respectively. The keyword *within* is used to constrain the time execution of an activity. In this case, time constraints have been set over the activities as specified earlier such that the whole process takes less than two hours.

The time-related aspect of the process is determined by the need of ensuring that the patient gets his/her diagnosis in less than two hours. This requirement is achieved by setting time constraints over the process' activities in the following manner: ''registration'' must take less than 15 min, ''examination'' and ''make document'' less than 30 min, ''consultation' less than 1 h (but more than 15 min, to ensure some quality in the check), and ''give information'' less than 15 min.

The diagnosis process definition includes activities to deal with certain undesirable events such that it becomes more dependable. Examples of the considered events range from foreigners trying to get diagnosed, to a fire taking place at the hospital. The activities included in the diagnosis process to deal with

such events are meant either to allow the process' goal (i.e. the patient is diagnosed) to be reached totally or partially; or mitigate the negative effects of such event over the process. In this manner, while some alternative activities are added to allow a foreigner to be diagnosed, others are added to make less harmful the consequences of a fire at the hospital, even knowing that the process will fail.

In DT4BP, a business process is modelled by means of four different models. These models are the *process model*, the *data model*, the *resource model* and the *dependability model*. While the *process* and *resource* models are mandatory, the existence of the *data* and *dependability* models depends on whether data elements are required and undesirable events are being handled, respectively. The *process model* describes the control flow of the business process, whereas the *resource model* the actual business entities owned by the organisation such that the process can be enacted.

Figure 1 shows the *process model* of the diagnosis business process.[12] Such model contains information about the kind of participant in charge of performing the activities (i.e. *DiagnosisUnit* on line 7). This kind of participant has to be defined in the *resource model* (see Figure 2) along with the actual business entities that are available for playing the activities once a process instance is created. Each of the activities to be executed by the participant (i.e. "registration" on line 11,


```
1  ;;*************************************************************
2  PROCESS MODEL
3  ;;*************************************************************
4  business process diagnosis() {
5
6
7    participant DiagnosisUnit {
8
9      do{
10       Person prs;
11       registration(out prs) within[_,15min.],
12
13       Temperature t expire(1hs.);
14       BloodPressure bp expire(1hs.);
15       PatientSheet ps;
16       do{split examination(out t,bp)
17              makeDocument(in prs; out ps)
18          }within[_,30min.];
19
20       consultation(in ps,t,bp; out ps) within[15min.,1hs.];
21
22       giveInformation(in ps) within[_,15min.]
23
24     }deviation[EX_Fire|(fa|fa.oclIsNew() and fa = #ON)]
25   }
26
27 } post[not ps.d.oclIsUndefined and not ps.p.oclIsUndefined]
```

Figure 1. *Process model* of the diagnosis business process.

```
;;*************************************************************
RESOURCE MODEL
;;*************************************************************
resources{
        DiagnosisUnit = DU1;
}
```

Figure 2. *Resource model* of the diagnosis business process.

"examination" on line 16, "makeDocument" on line 17, "consultation" on line 20, and "giveInformation" on line 22) either requires some information to be executed, or produces some information once it has executed. Such information defines the data elements that take place within the business process (i.e. *prs, t, bp*, and *ps*, on lines 10, 13, 14 and 15, respectively). The *type* of each data element is required to be defined in the *data model* (see Figure 3). Since in the *process model* the undesirable event fire is being considered (named as *EX__Fire* on line 24), the *dependability model* contains the activities to be performed when such event occurs (see Figure 4). This model specifies that when the *EX__Fire* event takes place the process performs the *evacuation* recovery, which executes the *evacuateDiagnosisUnit* activity. Note that this activity does not help to reach the process' goal, but it does help to mitigate the effects of the fire.

Now that the reader has an overall view of the different models to be provided when describing a business process in DT4BP, a more detailed description of the language can be started.


```
;;*************************************************************
DATA MODEL
;;*************************************************************
type  Person{
        String name, surname, address, address, city, country;
        Calendar birthDate;
        Integer ssn;
}

type Calendar;
type Temperature { Integer t where (30 <= t) and (50 >= t)}
type BloodPressure { Integer bp where (50 <= bp) and (250 >= bp)}
type Treatment,Medicine,Diagnosis,MedicalHistory;

type PatientSheet{
        Integer ssn;
        String name, surname, address, city, country;
        MedicalHistory mh;
        Diagnosis d;
        Prescription p;
}

type Prescription{
        Treatment t, Medicine m;
}

type FireAlarm = enum{ON,OFF}
```

Figure 3. *Data model* of the diagnosis business process.

```
;;*************************************************************
DEPENDABILITY MODEL
;;*************************************************************
resolution{
        EX_Fire -> Evacuation;
}

recovery{
        Evacuation(){
                evacuateDiagnosisUnit();
        }Failure
}
```

Figure 4. *Dependability model* of the diagnosis business process.

4.2. Basic elements

The aim of this section is to introduce the basic features that any business process modelling language should provide to allow business analysts to describe comprehensible models.

A general business process is considered as a set of *activities* which are executed in certain order by a *participant*. The definition of a business process thus starts by describing the participant's activities as well as the execution dependencies between such activities. In DT4BP, this information is included in the ***process model***.

4.2.1. Modularity

Since an activity represents a piece of work, its description will depend on the complexity of such work. The divide and conquer strategy is often applied when a task is very complex. What is required then is a decomposition model that allows an activity to be divided in sub-activities such that their completion represent the completion of the enclosing activity. In other words, a business process should be able to embed another business process. This is achieved by introducing the notion of *composite* activity. A composite activity is an activity within a business process that refers to another business process. This allows a business process to be structured hierarchically using a top-down approach: complex activities are refined in sub-business processes until, at the lowest level business processes are only made of *atomic* activities (i.e. an activity that is not further subdivided into sub-activities).

4.2.2. Refinement

A composite activity thus defines two abstraction levels: the higher level hides the details about the refereed business process, whereas the lower shows its definition. Every time an activity is refined in a composite activity during the business process definition both abstraction levels are introduced. Thus, the *composite* construct (i.e. the primitive that indicates that the activity is actually another business process) can be considered as a refinement operator for such modular decomposition model. The use of such decomposition model not only allows to reduce the complexity of modelling non-simple activities but also reuse already defined activities. Furthermore, the different abstraction levels make the reading and understanding of the business process definition easier as the reader controls the details to be observed when going through the model.

4.2.3. Control flow operators

To specify the execution dependencies between activities control flow operators are required. The required basic operators are those that allow a modeller to represent a sequence of activities to be executed one after other (i.e. act_1; act_2), an alternative between two activities based on certain condition (i.e. **if** *cond* **then** act_1 **else** act_2), the execution of one or more activities while certain condition holds (i.e. **while** *cond* **do** $act_1, \ldots, act_n$ and **repeat** $act_1, \ldots, act_n$ **until** *cond*), and the parallel execution of some activities which either need to be joined synchronously later (i.e. **split** $(act_1, \ldots, act_n), \ldots, (act_1, \ldots, act_m)$;) or last until the end of the process (i.e. **spawn** $(act_1, \ldots, act_n), \ldots, (act_1, \ldots, act_m)$;).

In the running example introduced in Section 4.1 some of these control flow operators are used (see Figure 5, lines 1–7): the first activity to be performed is ''registration''. This is a *composite* activity; therefore, a business process definition for 'registration'' has to be also provided as part of the overall business process definition[13] (line 9 in Figure 5). The *split* operator is used to start ''examination'' and ''make documents'' composite activities in parallel. ''Consultation'' will wait for the completion of these parallel activities since they were started using a *split* primitive.

4.2.4. Participants and resources

The notion of *participant* is used as structuring mechanism to encapsulate a set of activities and their dependencies under the context of a particular kind of business entity. A business entity capable of doing work is known as *resource*. A resource can be either human (e.g. a worker) or non-human (e.g. plant or equipment) and be owned either by the organisation leading the business process or by a third-party entity on which such organisation relies on for doing so. A participant then prescribes the *class of resource* required to perform the activities it encapsulates, rather than the actual resource that performs such activities at run-time. In the running example, a participant of class *DiagnosisUnit* encapsulates the activities related to the diagnosis process (see Figure 6, lines 3–9), while a participant of class *Secretary* and another of class *Patient* are required by the registration business process to perform its activities (lines 13–14).

Since a participant does not specify which is the actual resource that performs the activities, some extra information that allows the binding participant–resource to be set has to be also included in the business process definition. Such information could

```
1  business process diagnosis(...){
2       ...
3       composite registration(...);
4       split composite examination(...),composite makeDocument(...);
5       composite consultation(...);
6       composite giveInformation(...)
7  }
8
9  business process registration(...){...}
```

Figure 5. *Process model* of the diagnosis business process.

```
1  business process diagnosis() {
2
3    participant DiagnosisUnit{
4       ...
5       composite registration(...);
6       split composite examination(...),composite makeDocument(...);
7       composite consultation(...);
8       composite giveInformation(...)
9    }
10 }
11
12 business process registration(...){
13      participant Secretary{...}
14      participant Patient{...}
15 }
```

Figure 6. Introducing a participant in the diagnosis business process.

be given at modelling time or be deferred until the participant instance needs to be executed at run-time. Specifying the identity of the resource to which instances of the participant will be assigned at run-time prevents the problem of unexpected or non-suitable resource allocation arising at run-time. The resources to be used by the participants are specified in the call of the business process, and not in its definition (i.e. $name_{BP}$ [**alloc**(res_1), ... , **alloc**(res_n)](...);). This allows to implement different resource allocations for the same business process definition. The main drawback of this approach is that if the specified resource is not available, then the process instance might suffer from non-acceptable delays. The deferred allocation approach overcomes with this problem, since the decision of which actual resource has to perform the participant's activities is postponed until the moment the instance becomes runnable. It is at that point that one resource belonging to the class prescribed by the participant in the process definition is chosen to execute the activities. Therefore, the deferred allocation approach based on class of resources is considered the by-default strategy to allocate resources on participants. However, any static binding between a particular resource and a participant overrides the default allocation strategy.

It might be required to constrain the resource population used to select the resource that will execute the participant's activities. Predicates written in first-order logic are used to refine the resources to be taken by the allocation strategy to select one (i.e. *nameBusinessProcess*[***alloc***(r | pred(r))]()). Furthermore, to allow cross reference between resource allocations of different business processes the notion of *resource variable* is introduced. A resource variable is used to store the resource that has been selected by the allocation approach (whatever it is) for performing the participant's activities (i.e. *nameBusinessProcess*[p = ***alloc***()]()). Resources variables are defined in the resource allocation area (i.e. the scope defined by the [ ]) and can be used in any further resource allocation area enclosed by the same participant.

The different classes of resources along with the actual resources that belong to each class[14] are defined in the *resource model*. A business process definition has an unique associated resource model, which is used for the process instances to find the needed resources. The capabilities or attributes (e.g. previous experience, skills), as well as standard information (e.g. costs, availabilities) related to each class of resource are also specified in the resource model. Such characteristics can be used to select at run-time the "best suitable" (according to some policy) resource element for the activities to be performed. Usually, the resources on which a business process can rely on to execute its instances are known in advance, so that they are defined at modelling time in the resource model. Figure 7 shows the resource model related to the diagnosis business process.

Resources can be also created upon request. This means that a *new* resource is created on demand to allow a particular process instance to be executed. A resource

```
1  resources{
2          DiagnosisUnit = DU1;
3          Secretary = Ann;
4          Nurse = Sue, Rose;
5          Patient;
6          Assistant = Jane;
7          Doctor{Integer cost} = Marc(50), Nick(80);
8  }
```

Figure 7. Resource model of the diagnosis business process.

variable can be used to keep a reference of this new resource in case that it wants to be used to execute another business process by the same participant instance. Note that the scope of a new resource is the process instance where it is created. Thus, once the process instance ends its execution, the resource is removed. Figure 8 shows how a new resource of class *Patient* will be created at runtime to undertake the execution of the registration business process. The same *Patient* resource that undertakes ''registration'' must be used to perform ''examination'', ''consultation'', and ''giveInformation'' business processes. Thus, once the *Patient* resource is created, it is stored in the resource variable *p*. Such variable is used upon call of the other business processes to specify the *Patient* resource to be used by them. An underscore symbol (i.e. '_') is used in the allocation area to denote that the default allocation approach has total freedom to select the resource that will undertake the business process.[15]

4.2.5. *Data*

An activity might require some data to be executed. It is also possible that such data are generated by an earlier performed activity. For example, in the running example, the doctor requires to know the temperature and blood pressure to be able to perform the ''consultation'' activity. This information is the outcome that the ''examination'' activity has to provide. This means that the data an activity requires or produces are also part of the business process, and therefore included in its definition. Thus, concepts that allow the representation and utilisation of data within a business process to be modelled are required.

As the data required by a business process depend strongly on the domain that is being targeted, the characteristics of such data may be very simple, or quite complex. A modelling language then should include a means to allow domain-specific data entities to be defined. The *abstract data type* principle is the chosen means to satisfy such requirement. The primitive **type** is used to introduce a new data type (i.e. **type** *typeName*). Basic types as *string*, *float* or *integer* are assumed to exist by default. A new data type that is defined by combining multiple elements of other types is referred to *structured data type* (i.e. **type** *typeName*{$typeName_1$ $attr_1$, ... , $typeName_n$ $attr_n$}. It might be required to impose conditions (aka invariants) for a particular data type to constraint the possible values that it can represent. These conditions are described using first-order logic (OCL (Warmer and Kleppe 2003) is the notation used to describe such conditions). The collection of all data types a particular business process makes use in its definition determines the *data model* associated to such business process.

```
1  business process diagnosis() {
2
3    participant DiagnosisUnit{
4       ...
5       composite registration[_,p = alloc(new Patient)](...);
6       split composite examination[p,_,_](), composite makeDocument(...);
7       composite consultation[p,_](...);
8       composite giveInformation[p,_](...)
9    }
10 }
```

Figure 8. Resource allocation in the diagnosis business process.

Figure 9 shows part of the data model associated to the "diagnosis" business process of the running example. This figure shows the definition of the *temperature* data type: such data type holds an integer attribute *t*. The predicate shown in line 2 defines the allowed values the attribute *t* may have.

Since a *type* presents a collection of data items (aka instances or objects), a way to denote such data items must be provided. A *data variable* (called from now on variable for simplicity) is used within a business process to denote an instance of a particular data type. The type of the variable is either one of the basic types or one of the types defined in the data model associated to the business process. The variable name is used to identify the value it contains. The scope of a variable is determined by the place where it is defined. A variable can be defined either in the context of a participant, or as parameter of a business process. A bussines process parameter is accessible for every enclosed participant in the business process, whereas a participant variable is only accessible for the activities such participant performs. Activities may carry *arguments* (i.e. $act(\textbf{\textit{in}}\ \arg_{in_1}, \ldots, \arg_{in_n}; \textbf{\textit{out}}\ \arg_{out_1}, \ldots, \arg_{out_m})$). An argument is used either to pass information to the activity (listed in the ***in*** part) or to get side effects of the activity execution (listed in the ***out*** part). Arguments in composite and nested activities are passed by value. It is possible to determine statically whether the activity arguments are well-typed with respect to the parameters expected by the inner business process such activity refers to. Figure 10 shows some of the variables defined in the *diagnosis* process (lines 5 and 8–10) that are used as arguments in "registration", "examination" and "makeDocument" activities (lines 6, 11, and 12, respectively)

Business process parameters are divided into *input* and *output*. Input parameters are used to receive information from the enclosing context and output parameters to return to the enclosing context any useful side effect produced by the execution of the process. Input parameters in the outmost business process are used to capture the information belonging to the environment where the process operates and is required for its execution.


```
1  type Temperature { Integer t where
2          (30 <= t) and (50 >= t)}
```

Figure 9. Part of the diagnosis business process data model.

```
1  business process diagnosis(){
2
3    participant DiagnosisUnit{
4
5       Person prs;
6       composite registration[_,p = alloc(new Patient)](out prs)
7
8       Temperature t;
9       BloodPressure bp;
10      PatientSheet ps;
11      split composite examination[p,_,_](out t,bp),
12            composite makeDocument(in prs; out ps);
13      ...
14   }
15 }
```

Figure 10. Use of arguments to pass/get information to/from inner business processes.

At run-time, for every variable a new instance will be created for every instance of the business process that is initiated. The definition of "makeDocument" business process, shown in Figure 11, specifies that it has an input parameter of type *Person* called *prs* (line 1) and an output parameter of type *PatientSheet* called *ps* (line 2).

4.2.6. Event-trigger business processes

In general, a *process instance* is created by an *event* coming from the environment (e.g. a patient arrives at the hospital) where it operates. Such information then has to be captured explicitly in the business process definition to allow other people involve in the management of such business process to know *when* a process instance is created.

Figure 12 shows how the event *patientArrives* is used to specify that the *diagnosis* business process is triggered when such event occurs. Instances of a business process that do not include event-related information are created immediately upon request (which is the case mainly for business processes called by *composite* or *nested* activities).

It is worth noting that the kind of business process being addressed in this report is usually enclosed in a highly non-deterministic environment which tends to evolve over the time, so that no information about event arrival patterns is known a priori.[16]

4.3. Collaboration between processes

As stated earlier, a collaborative business process is defined as the one that requires the interaction of multiple participants to reach the process' goal. Therefore, it is expected that definitions of the targeted business processes contain more than one participant. In the running example, every business process involved in its definition has more than one participant, except "makeDocument" and "diagnosis" which are the outmost business processes that enclose all the others.[17] In the targeted business processes, the interaction among participants is expected to be high in the sense that their progress depends on each other's progress. Therefore, operational dependencies between collaborative participants are expected to exist.

The interaction between participants is achieved by sending and receiving messages. Both sending and receiving a message are considered atomic activities. The reasons why a participant sends a message are: (1) to forward a message to another participant, (2) to synchronise with another participant or (3) to request an action from another participant. The first case requires sending a message without the need to wait for the receiver to get the message (i.e send-no-wait).[18] The second case requires sending a message and then blocking until the receiver gets the message

```
1  business process makeDocument(in Person prs;
2                                out PatientSheet ps){...}
```

Figure 11. Use of parameters to receive/return information from/to the enclosing context.

```
1  business process diagnosis() when(patientArrives) {...}
```

Figure 12. Event-trigger business process.

(i.e send-wait). The third case requires sending a message and then get blocked only when a reply is needed to continue its execution rather than immediately after sending the message (i.e. send-no-wait-receive). What is required then to cover these cases are communication primitives to allow sending and receiving messages along with some optional blocking mechanism that can be attached to them. A possible syntax for the sending primitive would be "***send*** *msg to* ***participant*** [***block***]", meaning that the message *msg*[19] is sent to participant, and if the optional clause ***block*** is used, then the sender participant gets blocked until the receiver participant receives the message. Regarding the receiving primitive, "***receive*** *msg* ***from*** *participant*" means that the message *msg* will be retrieved from *participant* and the receiver is blocked until the message arrives. Therefore, the "send-no-wait" is achieved by using the *send* primitive without the optional clause *block*,[20] the "send-wait" by using the *send* primitive with the optional clause *blocks*, and the third by using the *send* primitive without the optional clause *block* followed by a *receive* primitive.

Figure 13 shows how the *send* and *receive* clauses are used to model the interaction between the secretary and the patient during the "registration" process. This process starts by the secretary requesting the patient to provide his social security card. In order to be sure that the patient gets the request, it is done in a synchronous manner. That is the reason why the *send* has a *block* clause at the end (line 3). The patient, once has got such request (line 9), provides his social security card to the secretary (line 12), who is waiting for such information (line 5).

It might be the case that a subset of the participants involved in a collaborative business process need to perform some sub-collaborative work which must be seen as an "atomic action" (i.e. single logical unit of work) by the other participants of the same process. This means that those required activities to be performed by every participant engaged in the sub-collaboration have to be scoped and hidden such that participants not engaged in such sub-collaboration see them as a single activity that provides a consistent outcome. The solution is then to move those activities that define the sub-collaboration to another collaborative business process. Since this new collaborative business process encloses those activities that determine the sub-collaboration, the kind of participants that perform such activities must be also included in its definition. The notion of **nested** activity is introduced to allow a participant within a collaborative business process to "call" another participant of the same type, but defined within another collaborative business process.

The called participant will start performing its activities once every other participant within the same collaborative business process has also been called. The

```
1  business process registration(out Person p)
2    participant Secretary{
3      send reqSSCard to Patient block;
4      SocialSecurityCard ssCard;
5      receive reqSSCard(ssCard) from Patient;
6      ...
7    }
8    participant Patient{
9      receive reqSSCard from Secretary;
10     SocialSecurityCard ssCard;
11     searchSScard(out ssCard);
12     send reqSSCard(ssCard) to Secretary;
13     ...
14   }
15 }
```

Figure 13. Message exchange in the registration process.

caller participant will continue performing its activities once the called collaborative business process has completed its execution (i.e. every participant within the called collaborative business process has performed all its activities). Therefore, those participants that are called indirectly by using the concept of nested activity (i.e. **nested** *act*) synchronise their execution upon entry and exit to the inner collaborative business process. It is also important to emphasise that a *nested* business process must guarantee the achievement of a consistent outcome even when facing with situations that lead its goal not to be reached. This is required since the overall behaviour of the enclosing business process is dependent on such outcome.

Thus, *nested* business processes are required to provide the isolation and consistency transactional properties. Such requirement can be fulfilled by making such kind of business process behave as transactions. As a *nested* business process is a just particular kind of collaborative business process, the same principle (i.e. behave as transactions) can be applied to any collaborative business process. It is worth noting that while the consistency property can be provided in the same manner as it is done in ACID-transactions, the isolation property can be achieved only partially due to the long-lived nature of business processes. The isolation property can be applied only to those local elements that are not visible from outside of the business process. A deeper analysis of considering business processes as transactions is deferred until Section 4.5 when dependability-related concepts are introduced, since transaction processing is a well-known approach used to develop fault-tolerant software (Gray and Reuter 1992).

4.4. Time-related aspects

Timing constraints are expected to be included explicitly in the process definition. Examples of such time constraints are: ''the diagnosis of a patient should take less than 3 hours'', ''every Monday, nurses receive training from 13:00 to 14:00'', and ''consultations are given by doctors during week days from 9:00 to 21:00''. Such constraints are used to restrict the time behaviour of the process. The information held by a time constraint can be *relative* or *absolute*. Absolute time refers to the time as seen in the physical world, whereas relative time refers to the passage of the time with respect to a certain event.[21] It is important to underline that time constraints placed on models should be relative, otherwise the model turns valid only at one point in time in its entire life (which is not the idea, of course). However, at run-time, absolute deadlines are computed for any relative time constraint. This means that absolute time is the time frame to be considered during the execution of a business process. Therefore, at run-time, it is assumed the existence of a device capable of measuring the passage of the *physical time*. Such device is called **clock**.

In a business process where the participation of multiple participants is perfectly possible that each participant holds its own clock. Since we want participants to have a common view of the actual time, the *global time based on physical time* is assumed as time model. This model is defined as follows: let *bp* be a business process made of p_i ($i = 1, \ldots, n$) participants and $Clock_{\text{reference}}$ be a clock that measures the passage of the *physical time* accurately,[22] then it is assumed that for every p_i $\exists! Clock_{pi}$ such that

$$\left|Clock_{pi} - Clock_{\text{reference}}\right| < D,$$

where D is the maximum difference between any two clocks, provided by a clock coordination algorithm. Examples of such algorithms can be found in the literature (Network Time Protocol 1980, Gusella and Zatti 1989, Cristian and Fetzer 1994). Under this time model, participants' clocks are strongly synchronised to each other, allowing them to have the same view of the actual time.

Considering then the global time model previously defined, it is required now to find out all possible time constraints that can be specified when modelling a business process. The strategy to find possible time constraints consists of taking every concept (or modelling primitive) introduced so far and analysing (for those that make sense) what kind of time-related information can be associated to it. Knowing all possible time constraints that can be defined in a business process allows realising what are the required modelling primitives that make them easy to be modelled.

(1) *Business process*: a business process is started by a request. Such request is either an external event coming from the environment where it operates[23] or a simple call (like the one made by *composite* or *nested* activities).

Thus, taking as frame of reference the point in time in which such request is performed, time information can be used to constrain the starting of a business process in the following way:

- the process starts t time units after the request *req* has been performed (aka *delay*),
- the process starts at least t time units later than the request *req* has been performed (aka *minimum delay*),
- the process starts no later than t time units after the request *req* has been performed (aka *maximal delay*).

As already stated, in general there is no information about how often requests that produce the starting of a business process arrive (i.e. request patterns are not known in advance). An exception to this rule is *periodic* business processes. Such kind of business process is performed according to a recurring pattern, which specifies how often the business process executes and for how long the recurring patterns has to be applied (if any) once the business process is started. Therefore, for a periodic business process the time information to be provided is its periodicity along with the point in time where the periodic sequence stops (i.e. until when the recurring pattern has to be applied).

The actual point in time in which the business process starts its execution defines another time frame of reference. Such time frame can be used to constrain the duration of the business process in the following way:

- the process lasts exactly t time units (aka *elapse time*)
- the process lasts at least t time units (aka *minimum deadline*)
- the process lasts no more than t time units (aka *maximum deadline*)

(2) *Participant*: as a participant is used to specify the *kind of resource* required at run-time for performing those activities it encloses, a time constraint can be used to specify the time the resource to be assigned at runtime should work (as minimum and/or maximum) for completing the activities. The time frame of reference for this time constraint is determined by the point in time at which the resource allocation for the business process has been completed.

(3) *Activity*: an activity can have its starting and duration constrained by time. Regarding the starting of the activity, the alternatives are: (1) to start at time t

(aka delay), where t is a time relative to the completion of the previous activity, or, in its defect, to the start of the business process; (2) to wait for at least t time units before starting its execution (aka minimum delay); or (3) the activity execution starts no later than t time units (aka maximum delay).

It is assumed that the execution of certain activity takes time. The time it takes to be executed is called the *activity duration*. Such duration can be explicitly constrained by means of deadlines: an *earliest deadline* is used to specify the minimun amount of time the activity will be under execution, whereas a *latest deadline* defines the maximum allowed time to be spent in performing such activity.

It is worth noting that those time constraints set over composite or nested activities must be compatible with those time constraints enclosed in the definition of the business processes to be called by such kind of activities.

(4) *Resource:* time constraints over resources are used to specify their availability. Absolute time constraints would be the rule rather than the exception when specifying the availability pattern for certain resource.
(5) *Data:* the value contained in a variable might be valid only for certain time frame from its last update (e.g. the variable that holds the patient's temperature). Time-related information associated to variable is aimed at specifying a property p that determines for how long the variable value will be valid. If the property p must hold for every variable of the same type rather than for a particular one, then such property should be given in the definition of the data type. In any case, this time information defines a deadline for the value stored in a particular variable, which is relative to the last time the variable was updated. It is worth noting that the required activities that allow the variable value to remain valid are neither part of the data type definition nor variable declaration; such activities must be included in a particular business process meant for that. Furthermore, note that the value of a variable remains accessible even if it is out of date. It is up to the activity that accesses the variable to check whether its value is still valid.
(6) *Message exchange between participants:* messages are exchanged between participants by using the *send* and *receive* primitives. Possible time constraints over these primitives are: (1) how long a (blocking) sending participant is willing to wait for the receiver to get the message, (2) how long a receiving participant is willing to wait for a message to arrive, (3) how long a receiving participant is going to execute to produce a reply after a message has been received, and (4) how soon a reply should arrive to a sending participant after a message has been sent.

The remaining concepts to be analysed are: *control flow operators, participant* and *event*. As it is assumed that a *control flow operator* does not take time to be performed, it does not make sense to set time constraints over them. An *event* is used to specify the circumstances that make a business process to start its execution. Thus, a time constraint set over an event can be seen as a constraint over the start of the business process that such event triggers. Therefore, rather than attaching time constraints to an event, the approach is to use such event as time-reference for the time constraints that can be set over the start of a business process: exact, minimum and maximum starting delay.

Knowing what are all the possible time constraints that could be set over a business process, primitives that allow such constraints to be specified easily are proposed.

4.4.1. Timing constraints over a business process

The primitive **start** is used to specify constraints related to starting the business process. Such primitive is followed by either a single time value t_{Delay} or a pair $[t_{\text{Delay}_{\min}}, t_{\text{Delay}_{\max}}]$. *Start* followed by a single time value (i.e. ***start*** t_{Delay}) is used to specify the delay in starting the business process, whereas followed by a pair (i.e. ***start*** $[t_{\text{Delay}_{\min}}, t_{\text{Delay}_{\max}}]$) is used to specify the minimum delay ($[t_{\text{Delay}_{\min}}, \_]$), the maximum delay ($[\_, t_{\text{Delay}_{\min}}]$), or both ($[t_{\text{Delay}_{\min}}, t_{\text{Delay}_{\max}}]$).

Regarding periodic business process, the primitives **every** and **until** are used to specify information related to the period of the process and the end of the periodic sequence, respectively. Thus, while ***every*** $t_{\text{Period}_{\text{every}}}$ is used to specify how often the business process executes, ***until*** $t_{\text{Periodic}_{\text{until}}}$ specifies the end of the periodic sequence. It is worth noting the **start** primitive remains being the one to be used for specifying the beginning of the periodic business process.

The duration of a business process is specified using the primitive **last**. This primitive followed by a single time value (t_{Duration}) is used to determine the exact duration of the process, whereas followed by a pair ($[t_{\text{Deadline}_{\min}}, t_{\text{Deadline}_{\max}}]$) is used to determine the minimum deadline (i.e. $[t_{\text{Deadline}_{\min}}, \_]$), maximum deadline ($[\_, t_{\text{Deadline}_{\max}}]$), or both for such process. Figure 14 shows how this primitive is used to model in the "diagnosis" and process the requirement "the patient should get its diagnostic in less than two hours".

4.4.2. Timing constraint over a participant

The kind of time constraint a participant may hold is specified using the primitive **workFor**. This primitive is followed by a pair of values ($[t_{\text{WorkingTime}_{\min}}, t_{\text{workingTime}_{\max}}]$) which determine the minimun and/or maximum working time to be spent by the resource that will takeover the activities specified in the participant at runtime. Figure 15 shows how this primitive is used to specify the requirement that a doctor should work in the examination of the patient for at least 15 min in order to ensure certain quality of the checking service. Note that the time information held by this primitive is concerned with the actual time the resource will hold for completing the activities enclosed by the participant, whereas the primitive *last* is with the duration of "consultation" business process.


```
1  business process diagnosis() when(patientArrives) last[_,2hs.] {...}
```

Figure 14. Constraining the duration of the diagnosis process.

```
1  business process consultation(
2              in PatientSheet ps, Temperature t, BloodPreasure bp;
3              out PatientSheet ps) last[15min.,1hs.]{
4
5    participant Patient{
6          ...
7    }
8    participant Doctor {
9          ...
10   }workFor[15min.,_]
11 }
```

Figure 15. Constraining the working time of doctor participant.

4.4.3. Timing constraints over an activity

Timing constraints over activities are specified using the primitives *in* and *within*. The primitive *in* is used to specify time information related to the start of the activity, whereas *within* to the duration of it.

Timing information related to the start of an activity comprises: (1) exact delay, (2) minimum delay or (3) maximum delay. The *in* primitive followed by a time value t_{Delay} is used to specify (1), whereas followed by a pair specify (2) (i.e. [$t_{\text{Delay}_{\text{min}}}$, _]), (3) (i.e. [_, $t_{\text{Delay}_{\text{max}}}$]) or both (i.e. [$t_{\text{Delay}_{\text{min}}}$, $t_{\text{Delay}_{\text{max}}}$]).

The primitive *within* is used to specify the time information related to the duration of the activity: i.e. its *earliest* and *latest* deadlines. Following the same strategy when minimums and maximums need to be specified, a pair of time values following the primitive *within* is used to specify the duration of the activity. Thus, the pair [$t_{\text{Deadline}_{\text{earliest}}}$, $t_{\text{Deadline}_{\text{latest}}}$] means that the activity executes for at least $t_{\text{Deadline}_{\text{earliest}}}$ time units and it does not take more than $t_{\text{Deadline}_{\text{latest}}}$ time units. As used so far, the '_' symbol means that no constraint is set over the time value where such symbol is used. Figure 16 shows how this primitive is used to model the duration of the activity "consultation" according to the given requirement: "consultation' has to take less than 1 h, but more than 15 min.

The primitive *within* also allows specifying the time constraints related to message exchange between participants. More precisely, a participant p_{sender} holding the statement ***send*** *msg* ***to*** p_{receiver} ***block within***[_, *t*] describes the fact that p_{sender} is willing to wait for p_{receiver} to get the message *msg* for *t* time units, maximum. On the other hand, the participant p_{receiver} holding the statement ***receive*** *msg* ***from*** p_{sender} ***within***[_, *t*] describes the fact that p_{receiver} is willing to wait for message *msg* to arrive for *t* time units, maximum.

4.4.4. Timing constraints over a set of activities

It is not possible to model both (1) the maximum allowed time a participant should execute to produce a reply after a message has been received, and (2) the maximum allowed time for a sending participant to get the reply for a message it has sent, with the primitives introduced so far. What is required then is a new primitive that allows joining a set of activities in a common block such that information can be linked to a set of activities $act_1, \ldots, act_n$, rather than only one. The primitive **do{...}** is used to define a block of activities. In this manner, assuming that $a_1, \ldots, a_n$ are all the activities that a participant must execute to produce the expected reply, the statement ***do***{ $a_1, \ldots, a_n$ }***within***[_, *t*][24] allows (1) to be modelled. In the same way, assuming that $a_1, \ldots, a_n$[25] now represents the activities that follow a (no blocking) *send*, the constraint (2) can be modelled.

```
1  business process diagnosis() when(patientArrives) last[_,2hs.] {
2
3
4    participant DiagnosisUnit {
5       ...
6       composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
7       ...
8       }
9  }
```

Figure 16. Constraining the duration of the activity *consultation*.

Figure 17 shows how the primitives **do** and **within** are used to model the constraint 'the parallel activities "examination" and "makeDocument" should execute in less than 30 min'.

4.4.5. *Timing constraints over data*

Timing information related to data, if given, is meant for defining a time frame which determines the validity of such data. Since a data time frame is relative to the moment in which the data were last updated, its definition requires to specify a period of duration, only. The primitive **expire** is used to specify the duration of the value in a variable (i.e. *typeName var* ***expire(timeExpr)***). As explained earlier, if the duration of a value must be respected for every variable of the same type, such information can be given in the definition of the data type (i.e. ***type*** typeName ***expire(timeExpr)*****).**

In the running example, the temperature and blood pressure measured on a patient are considered as valid values until one hour after they have been taken. The modelling of this requirement using the primitive **expire** is shown in Figure 18.

In case the time frame associated to a particular variable may change through the business process, the means to check the age of the data held by a variable should be provided. The primitive **timestamp(*v*)** is used to get the time in which the variable *v* was last updated, whereas the primitive **clock()** is used get the current absolute time[26]. Using these two primitives is possible to know how long time ago a variable was last updated, and make decisions on such information (e.g. update the variable or use its value as it is).

4.4.6. *Timing constraints over resources*

The last kind of time constraints that could be required to be modelled are those used to specify the availability of a resource. Such time constraints determine the

```
1  business process diagnosis() when(patientArrives) last[_,2hs.] {
2
3
4    participant DiagnosisUnit {
5       ...
6       do{split composite examination[p,_,_](out t,bp),
7          composite makeDocument(in prs; out ps)
8       }within[_,30min.];
9       ...
10      }
11 }
```

Figure 17. Constraining a block of activities.

```
1  business process diagnosis() when(patientArrives) last[_,2hs.] {
2
3
4    participant DiagnosisUnit {
5       ...
6       Temperature t expire(1hs.);
7       BloodPressure bp expire(1hs.);
8       ...
9       }
10 }
```

Figure 18. Modelling time constraints over data elements.

availability of a resource by means of temporal patterns (e.g. Mondays and Fridays, or week days from 8:00 to 16:00, etc.) combined with valid periods (i.e. each pattern is valid only for a certain period of time (e.g. from 1 January 2009 to 31 December 2011). Since these patterns may be very different, the time constraints used to model them will turn complex, being not easy to read and understand. Furthermore, many different primitives should be given in order to allow such patterns to be modelled. The way to cope with this modelling issue is to rely on the notion of **calendar**[27] as means to describe the availability of a resource. In this manner, instead of using patterns to model the time frames for which a particular resource is available, the information can be now explicitly included[28] in a calendar. Therefore, the availability of a particular resource can be known by means of its associated calendar. It is assumed that every resource holds a calendar.

4.5. Dependability

A dependable business process has been defined as the one that either (preferably) does not miss its goal unacceptably frequently or, if it does, the consequences are not unacceptably severe. This section introduces those concepts that allow business analysts to drive explicit dependability when modelling business processes.

4.5.1. Pre- and post-conditions

A business process that misses its goal is considered as one that has failed. Without having a precise definition of the business process's goal, to judge whether it has failed or not may lead to ambiguous answers. Since precision is needed, a business process goal has to be described using certain formal notation. First-order logic is the chosen means to meet this requirement. It is also important to state clearly under which conditions such goal is expected to be reached by the business process. These conditions are expected to be fulfilled by the "client" that requests the business process. The relationship between the business process and its clients can be considered as a *contract* in the same sense as Meyer does in the "Design by Contract" approach (Meyer 1997). The *pre-condition* of the contract defines what every client must satisfy to get in return the business process' goal. Such goal is the *post-condition* of the contract.

In the running example, the client is the patient that arrives to the diagnosis unit with the intention of being diagnosed and getting a treatment to deal with the reported disease. The "diagnosis" business process has been engineered to satisfy the client's demands, i.e. its goal (or post-condition) is to diagnose the patient and give him a treatment. Figure 19, on line 9, shows how this post-condition is modelled in

```
1  business process diagnosis() when(patientArrives) take[_,2hs.]
2    pre[true]{
3
4      participant DiagnosisUnit {
5        ...
6        PatientSheet ps;
7        ...
8      }
9  } post[not ps.d.oclIsUndefined and not ps.p.oclIsUndefined]
```

Figure 19. Pre- and post-conditions for the diagnosis business process.

OCL. The personnel of the diagnosis unit assume that a person would come to the diagnosis unit because he believes to be carrying a particular disease. Thus, there is not any special pre-condition to be fulfilled by such person to be diagnosed at the unit, so that it is true[29] (line 2 in Figure 19).

It has been said that if the client requests the business process instance ensuring the pre-condition,[30] its post-condition will be met on the condition that every activity was performed properly. Once again, a judgement problem arises: how to know if a particular activity was properly done? To be consistent, the same principle used so far should be adopted: i.e. pre- and post-conditions. Unless the pre-condition evaluates to true, the activity with which it is associated cannot start its execution. An activity that ends its execution without holding its associated post-condition is considered as the one that has not been performed properly (i.e. it has failed). A failing activity produces a deviation in the process instance execution that leads it to miss its goal if corrective actions are not taken. Therefore, a deviation in the process instance execution can be detected when the post-condition associated to one of its prescribed activities evaluates to false. Associating a pre-condition and post-condition to an activity eases its judgement about correctness, but in addition the post-condition gives a means to detect a deviation in the process instance execution.

Figure 20 shows the pre-condition and post-condition associated to the activity ''consultation''. The pre-condition (lines 11–13) says that this activity can be executed only when a *patient sheet* for the patient to be diagnosed exists and his *temperature* and *blood pressure* are known. The activity is considered as properly executed when a diagnostic and treatment have been defined in the *patient sheet* (post-condition modelled in lines 14 and 15).

Since *composite* activities refer to business processes that hold their own pre- and post-conditions, further information regarding the execution assessment of such kind of activities has to be given. A *composite* activity starts its execution when both its associated pre-condition and the pre-condition held by the business process it refers to evaluate to be true. A *composite* activity is considered to be executed properly when both its associated post-condition and the one held by the referred business process are met. Therefore, to succeed in executing a *composite* activity may happen that $pre_{cmpAct} \Vdash pre_{\text{businessProcess}}$ and $post_{\text{businessProcess}} \Vdash post_{\text{cmpAct}}$, which means that the pre-condition of the *composite* activity has to be stronger than the

```
1  business process diagnosis() when(patientArrives) take[_,2hs.]
2    pre[true]{
3
4      participant DiagnosisUnit {
5          ...
6          Temperature t expire(1hs.);
7          BloodPressure bp expire(1hs.);
8          PatientSheet ps;
9          ...
10         composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
11                 pre[not ps.oclIsUndefined() and
12                     not t.oclIsUndefined()  and
13                     not bp.oclIsUndefined()]
14                 post[not ps.d.oclIsUndefined() and
15                      not ps.p.oclIsUndefined()]
16         ...
17     }
18 } post[not ps.d.oclIsUndefined and not ps.p.oclIsUndefined]
```

Figure 20. Pre- and post-conditions over activities.

one held by the referred business process, whereas the post-condition has to be weaker than the referred business process'. Pre- and post-conditions defined for the "consultation" business process are shown in Figure 21. Since no information is given about its pre-condition, it is assumed as ***true***. The pre-condition set over the *composite activity "consultation"* (Figure 20, lines 11–13) is not the trivial one. This implies that when the pre-condition set over the *composite activity* holds, the one set over the business process holds too. The post-condition set over the business process (Figure 21, line 17) not only says that the diagnostic and treatment have to exist, but also that such values must be equivalent to the ones defined by the doctor. Thus, the business process post-condition is stronger than the one set over the *composite activity* (Figure 20, lines 14–15), which only requires the existence of the diagnostic and treatment.

It is worth recalling that pre- and post-conditions are used as means to assess the execution of a particular process instance (or one of its activities) under the assumption that certain initial condition holds. This means that if such initial condition (aka *pre-condition*) holds, then the process instance is allowed to be executed. However, the actual execution of the process instance also depends on the existence and availability of the resources such instance requires to be performed. This underlines that, from the perspective of a state transition system, the state of the machine used to execute a particular process instance is not only defined by the information held in the pre- or post-conditions, but also for the status of the resources that allow it to be played.

Including pre- and post-conditions on activities and business processes allows bringing clarity about the circumstances on which a particular business process is expected to reach its goal and how it plans to do it. It can be concluded thus that their inclusion in a process definition is a way to increase its dependability since either a deviation or failure of the process can be assessed without ambiguities.

4.5.2. Transactional behaviour

A *business transaction* (Papazoglou and van den Heuvel 2007) is defined as a consistent change in the state of the business process that is driven by a well-defined

```
1  business process consultation(
2              in PatientSheet ps, Temperature t, BloodPreasure bp;
3              out PatientSheet ps) take[15min.,1hs.]{
4
5    participant Patient{
6          ...
7    }
8    participant Doctor{
9          ...
10     Diagnosis d;
11     diagnosePatient(d)
12         ...
13     Prescription p;
14     prescribeTreatment(out p)
15         ...
16   }
17 }post[ps.d = d and ps.p = p]
```

Figure 21. Pre- and post-conditions for the *consultation* business process.

activity. Business processes usually are composed of several business transactions. Business transactions exhibit the following characteristics:

(1) they provided a result that is critical to the overall success of the business process where they are embedded;
(2) they involve two or more participants, which interact in a coordinate manner to achieve a mutually desired outcome;
(3) they run over long periods of time, behaving as open nested transactions (Gray and Reuter 1992) (i.e. sub-business transactions can abort or commit independently of what the status of the final outcome of the parent business transaction is);
(4) they provide consistency and durability as ACID-transactions, but relax isolation (limited to those part of the business process' state that are local to the business transaction) and atomicity (limited to guarantee consistency during the business transaction progress).

Business processes belonging to our domain of interest (i.e. DCTC) share these properties: they include multiple collaborative participants; they are expected to last for hours, days or longer; and the special interest in making them dependable is evidence of the importance to succeed in providing the expected business result. Furthermore, as explained in Section 4.3 when the notion of *nested* business process was introduced, (reduced) isolation and consistency are transactional properties that any business process should hold.

It can be concluded that the assumption of considering a business process as a business transaction does not overconstrain the domain of interest. The motivation to consider business processes as business transactions (or transactions, for short) resides in exploiting the features provided by the transaction paradigm with the aim of increasing the dependability in business processes. In other words, we want to borrow from the transactions those nice features (i.e. ACID properties) that make them an approach to achieve fault-tolerant software, and adapt them to our context and needs. In the following, details about the tuned transactional properties that a DCTC business process provides are given.

4.5.2.1. Outcomes. There are four kinds of outcomes of a business process: ***normal***, ***degraded***, ***abort*** *and* ***failure***. A *normal* outcome is produced when the business process reaches its goal (i.e. its associated post-condition holds). When the business process' goal cannot be reached, a dependable business process should strive to provide a partial service that potentially satisfies every involved participant (Mustafiz *et al.* 2008). Such partial service represents a *degraded* outcome with respect to the initially promised business objective. A precise description (i.e. first-order predicate) of each degraded outcome has to be given in the same manner as it is done for the normal outcome. Such ''degraded post-condition'' allows the judgemental system to evaluate whether the degraded service was provided as expected.

In the running example, a degraded outcome arises when the doctor requests the patient to remain at the hospital for some days because his current health status does not allow him to determine a clear diagnostic. The doctor then prescribes a preliminary treatment to be followed by the patient during his stay in the hospital such that eventually he can be diagnosed. Figure 22,[31] on line 5, shows how such *degraded* outcome is modelled. Note that what makes it a degraded outcome with

```
1  business process diagnosis() when(patientArrives) take[_,2hs.] {
2       ...
3  } post[not ps.d.oclIsUndefined and not ps.p.oclIsUndefined]
4  ...
5  }degraded[ps.d.oclIsUndefined and not ps.p.t.oclIsUndefined]
```

Figure 22. Degraded outcome in the *diagnosis* business process.

respect to the expected *normal* outcome (Figure 22, line 3) – besides, the patient has to remain at the hospital longer than expected – is the fact that the business process ends without giving a diagnostic to the patient (i.e. *ps.d.oclIsUndefined*()), but at least he receives a preliminary treatment (i.e. *notps.p.t.oclIsUndefined*).

In case that is not possible to provide any partial service (i.e. *degraded* outcome), the business process should be aborted. Aborting a business process might imply to perform business-specific activities to clean-up the effects of having executed non-reversible activities. Thus, an *abort* outcome represents one whose effects of having executed the business process are considered as undone (i.e. the pre-condition of the business process holds), regardless of the side effects that remain after the end of the business process. A *failure* outcome is produced when the business process faced with problems that made it impossible to not only achieve its expected outcome or a degraded one, but also to abort.

While *normal* outcome represents the best result, a business process can produce and *failure* the worst one, *degraded* outcomes can vary from very close to the optimal expected result to very marginal ones. Despite an *abort* outcome can be considered as an extremely degraded one, the fact that it is the only outcome that guarantees to satisfy the business process pre-condition, makes it different.

4.5.2.2. Consistency. A business process guarantees to produce a consistent result: either it satisfies the business process post-condition (*normal* outcome); or one of the degraded post-conditions (*degraded* outcome), if any; or its associated pre-condition (*abort* outcome). The only situation where it does not offer any guarantees is when the business process fails (*failure* outcome). It is important to emphasise here that while a business process may or may not include *degraded* outcomes, *normal, abort* and *failure* are included by default. Therefore, in case that no *degraded* outcome is provided by the business process, the result of performing any instance of such business process is either *normal, abort* or *failure*. Including *degraded* outcomes in the business process definition, implies extending the guaranteed results to be provided.

4.5.2.3. Isolation. Owing to the long-lived nature of a business process, only local information (i.e. data variables) can be isolated from the outside. This means that any change made over a non-local data variable would be seen from the outside. It is important to emphasise here that changes on data variables are made by *atomic* activities only, since *composite* activities are just encapsulated business processes.

4.5.2.4. Locking. As different *atomic* activities might attempt to access the same data variable concurrently, a locking mechanism is required to avoid smuggling information. The locking mechanism proposed is one that works at the level of *atomic* activities by locking those data variables that are being passed as output arguments (as such variables are the ones being modified, only). Such data variables remain locked during the time the atomic activity performs, which is expected to be

much shorter than the time required by a composite activity (i.e. long-running business process). An *atomic* activity needs to gain access over every data variable being passed as output argument before starting its execution, otherwise it remains waiting (i.e. it is blocked) till such condition is met.

4.5.3. *Exception handling*

It is assumed that a process instance which has deviated from its prescribed paths (i.e. process definition) eventually misses its goal if no corrective activities are executed. Thus, to avoid a process instance to miss its goal (i.e. to fail), it is required first to *detect* that it has deviated from its definition, and second, to perform the necessary corrective activities that will *recover* the process instance execution to one of its prescribed paths.

Since detection and recovery are the bases of *fault tolerace* (see Section 2), techniques involved in fault tolerance should be considered as means to model dependable business process. One of the techniques that allows fault tolerance to be attained, and has strong and direct connection with the needed features[32] to avoid a business process to miss its goal, is exception handling. Therefore, in order to attain dependable business processes, the concepts of *exception* and *handler* embodied by such technique are borrowed and adapted in the following way: an *exception* denotes the notification of a deviation, whereas a *handler* the set of corrective activities to recover from such deviation.

4.5.3.1. Deviations and exceptions. It has been said that a deviation during the execution of a process instance can be detected when a post-condition associated to one of its prescribed activities evaluates to be false. Since an exception is used as means to notify the occurrence of a deviation, that is precisely what has to happen when a post-condition evaluates to false.

An activity that ends its execution without holding its associated post-condition represents one that has failed. While it is easy to know when an activity fails (i.e. the post-condition is false), it may not be obvious to realise *why* it has failed (i.e. what has made the post-condition to be false). Thus, precise information that allows the occurred deviation to be identified has to be somehow provided. The proposal to cope with this issue is that each activity provides information about every *expected* deviation that may occur during its execution. In this manner, while the post-condition of the activity describes the expected result to be achieved when it performs successfully, an expected **deviation** describes the condition (in first-order logic) that makes the activity fail. An expected deviation then is described by providing the condition that allows it to be detected along with the **exception** raised to notify the actual occurrence of the deviation (i.e. ***deviation***$[exName \mid deviationCondition]$).

Figure 23 shows the deviations that may arise during the examination of the patient. At this step of the process, a nurse and an assistant check the temperature (line 6) and blood pressure of the patient (line 14), respectively. With regard to the activity of checking the temperature, a value over 40 (line 8) or a problem in the thermometer (line 9) represent deviations with respect to the expected result of this activity (line 7). Regarding the assistant activity of checking the patient blood pressure, deviations with respect to the expected result (line 15) arise when the value of the blood pressure is over 200 (line 16) or the blood pressure monitor does not work properly (line 17).

```
1  business process examination(out Temperature temp, BloodPressure bp) take[_,30min.]{
2
3    participant Patient{...}
4    participant Nurse{
5          ...
6          checkTemp()
7             post[temp|temp.oclIsNew() and temp < 40];
8             deviation[EX_HighTemp|(temp|temp.oclIsNew() and temp > 40)]
9             deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]
10         ...
11   }
12   participant Assistant{
13         ...
14         checkBP()
15            post[bp|bp.oclIsNew() and bp < 200];
16            deviation[EX_HighBP|(bp|bp.oclIsNew() and bp > 200)]
17            deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]
18         ...
19   }
20 }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

Figure 23. Deviations that may arise in the *examination* business process.

It is worth noting that the conditions that determine a deviation have to be not only mutually exclusive between them, but also with respect to the post-condition of the activity. Therefore, once the activity is assumed to be finished, one (and only one) condition has to be true.

An activity may also fail because it does not meet one of the time constraints sets over it, if any. The time constraints an activity may own are *in* and *within*. The constraint *in* is used to constraint the starting of the activity, whereas *within* determines the duration of the activity (see Section 4.4.3 for more details about these time-relate primitives). Thus, the notion of *deviation* is also used to describe explicitly the condition that makes any of these time constraints to be missed. In this manner, a ***deviation***[exName | deviationCondition] following a time constraint *in* or *within* describes both the condition that determines its occurrence and the exception used to notify the actual occurrence of such deviation. It is worth noting that a time constraint that does not hold a deviation means that such time constraint is always expected to be met.

As the reason why a time constraint is missed is due to a **timeout**, special predicates are used to denote the time-related condition taking place: the special predicate ***timeout.min*** denotes the timeout of the lower time bound, whereas ***timeout.max*** the timeout of the higher time bound. The special predicate ***timeout*** alone denotes the timeout of any of the time bounds.

The same principle of using a *deviation* to denote the missing of a time constraint is extended to those time-related constraints that might be set over a participant (i.e. *workFor* – see Section 4.4.2 for more details about this time-related primitive) and/or business process (i.e. *start* and *last* – see Section 4.4.1 for more details about these time-related primitives).

4.5.3.2. Handling model. When an exception is raised, which means that a deviation has been detected, corrective activities have to be performed to avoid the overall business process to miss its goal. A set of corrective activities defines a **handler**. A handler is meant for dealing with one or more exceptions. The mechanism that determines how to find a handler for a particular exception is called *propagation mechanism*. The propagation mechanism adopted in this work adheres to the termination model (Buhr and Mok 2000): once the handler completes, the control flow is expected to continue as if the exception would have never occurred. However, it must be noted that how the control flow of the business process continues will

depend on the severity of the exception to be dealt with. When the severity of the exception makes impossible to continue with the execution of the business process as originally planned (best case), its handling has to be oriented to terminate the business process execution in a safe way (i.e. the negative effects of missing the goal should be mitigated as much as possible).

4.5.3.3. Participant handler. Since a deviation is part of the information to be held by an activity or a time-constraint, the scope of the exception to be raised when the deviation occurs is the participant that owns such activity. The propagation mechanism then searches for a handler at the level of the participant. The aim of a **participant handler** (*p-handler, for short*) is to deal with a deviation to allow the participant to continue its execution so that the business process can reach its goal. In this case both the exception and its handling are kept hidden to the other participants taking part in the same collaborative business process.

A *p-handler* is defined next to the exception it is meant to handle (i.e. ***deviation*** $[ex \mid condition]\{act_{hnd_1}, \ldots, act_{hnd_n}\}$). Figure 24 shows the *p-handlers* defined within the *examination* business process (lines 10 and 20) to deal with the deviations ''malfunction of the thermometer'' (line 9) and ''malfunction of the blood pressure monitor'' (line 19), respectively.

A *p-handler* executes within the context of the activity or time constraint it is associated with. Being nested inside the context of the activity allows the *p-handler* to have access to the same input parameters the activity receives. It is worth noting that a same *p-handler* can be used to handle different exceptions by joining with the disjunction operator ***or*** either the different deviations (i.e. ***deviation*** $[ex_1 \mid condition_1]$ ***or...or deviation*** $[ex_i \mid condition_i]$ $\{act_{hnd_1}, ..., act_{hnd_n}\}$ or the different conditions that produce each deviation (i.e. ***deviation*** $[ex \mid condition_1$ ***or ... or*** $condition_i]$ $\{act_{hnd_1}, ..., act_{hnd_n}\}$. Once a p-handler has completed its execution, the control flow of the participant continues as the activity would not have faced with any deviation.

4.5.3.4. Business process handler. When the propagation mechanism does not find a handler for the exception raised due to the activity or time constraint deviation,

```
1  business process examination(out Temperature temp, BloodPressure bp) take[_,30min.]{
2
3    participant Patient{...}
4    participant Nurse{
5          ...
6          checkTemp()
7             post[temp|temp.oclIsNew() and temp < 40];
8             ...
9             deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]{
10                      findNewThermometer();
11            }
12         ...
13   }
14   participant Assistant{
15         ...
16         checkBP()
17            post[bp|bp.oclIsNew() and bp < 200];
18            ...
19            deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]{
20                      findNewBPMonitor();
21                }
22         ...
23   }
24 }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

Figure 24. Deviations being handled locally by *p-handlers* in the *examination* business process.

the exception is propagated at the level of the business process. Propagating the exception to the entire collaborative business process makes (1) every participant to stop the execution of its current activity(ies) and (2) the propagation mechanism to start searching for a handler at the level of the business process to deal with such exception. The aim of a **business process handler** (*bp-handler*, for short) is to handle an exception in a collaborative manner so that the business process goal is reached either fully (i.e. *normal* outcome) or partially (i.e. *degraded* outcome).

A *bp-handler* involves all the participants taking part in the collaborative business process. This kind of handler is expected to be more complex in terms of interactions between participants. To avoid overloading the *process model* with information relative to the collaborative handling of exceptions, a new model called **dependability model** is defined to include such information. Therefore, for each exception that is not handled locally at the level of the participant (i.e. its occurrence is propagated at the level of the entire business process) a *bp-handler* must be defined in the **recovery** section of the *dependability model*.

Figure 25 shows (part of) the *dependability model* related to the *examination* business process. The ***recovery*** section of this model includes the handlers *Undress* (lines 2–11) and *CardiacUnit* (lines 14–27). These handlers are meant for dealing with the exceptions *EX_HighTemp* and *EX_HighBP*, which are used to notify the occurrence of the deviations ''temperature over 40'' and ''blood pressure over 200'', respectively (see Figure 23, on lines 8 and 16). In the next section, more precise information is given about how the binding between exceptions and bp-handlers is achieved.

A *bp-handler* executes within the context of the business process it is attached to. Participants collaborating in a *bp-handler* have not only access to the business process input data, but also to the local data produced until the moment the exception was raised (in the case of the deviated participant) or received (for participants being notified about the occurrence of the deviation). This semantics

```
1  recovery{
2     Undress(out Temperature temp, BloodPressure bp) {
3          participant Patient{
4                  receive reqUndress from Nurse;
5                  Undress();
6          }
7          participant Nurse{
8                  send reqUndress to Patient block;
9          }
10         participant Assistant{}
11    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
12
13
14    CardiacUnit(out Temperature temp, BloodPressure bp) {
15         participant Patient{
16                 receive reqChangeRoom from Assistant;
17                 goToSpecialUnit();
18         }
19         participant Nurse{
20                 receive reqChangeRoom from Assistant;
21                 takePatientToSpecialUnit();
22         }
23         participant Assistant{
24                 send reqChangeRoom to Patient,Nurse;
25                 notifyNewRoomPatient();
26         }
27    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
28 }
```

Figure 25. *Business process handlers* in the *dependability model* of *examination* business process.

allows every participant to use in the handling of the exception the work done during its normal execution within the business process until either the occurrence or notification of the exception. Depending on the severity of the exception being handled, a *bp-handler* will either reach the expected result (i.e. the business process post-condition is met) and provide some partial service that is described as a *degraded* outcome or abort. It is worth noting that nothing forbids *p-handlers* to be used within the participant section that belongs to a *bp-handler*.

When the severity of the exception does not allow the business process goal to be reached neither fully nor partially, the business process, as last alternative, is expected to be aborted (i.e. *abort* outcome). A *bp-handler* to abort the business process is included by default. This ***bp-Abort-handler*** can be overridden, when aborting a business process requires including specific activities to deal with non-reversible effects. When the exception is too severe or something was very wrong during its handling, the process is considered as one that has failed (i.e. *failure* outcome).

A handler then is the feature that allows a business process to guarantee that a consistent outcome (i.e. *normal, degraded* or *abort*) is provided to its enclosing context, when a deviation has occurred. Therefore, a dependable business process *must* include a handler to deal with each deviation that has been explicitly defined, whether at the level of the participant by providing a *p-handler*, or at the level of the overall business process by a *bp-handler*.

4.5.3.5. Parallel exceptions. Multiple exceptions may arise at the same time. This is possible not only due to the fact that participants execute their activities in parallel, but also because a same participant may execute multiple activities in parallel: the *split* and *spawn* control flow operators allow the parallel execution of activities by a same participant. Therefore, a participant that performs in parallel at least two *composite* activities[33] may face with this situation. Other source of multiple parallel exceptions takes place when an activity has a time-related constraints defined by the primitive *within*: in this case, a the time-related constraint can be missed along with an activity's deviation. When multiple parallel exceptions take place, whatever the case might be, the *propagation mechanism* has to determine what is the exception to be handled before starting to search a handler for it.

The resolution mechanism proposed in this work to determine what is the chosen exception to be handled when multiple exceptions are raised in parallel is based on the approach described by Campbell and Randell (1986). The principle followed by the authors is that all multiple exceptions being raised simultaneously, in conjunction, constitute a new exception that is symptomatic of a different and more complicated deviation than those that reported by each exception individually. Therefore, when multiple exceptions are raised in parallel (i.e. $ex_1, \ldots, ex_n$), the *propagation mechanism* will look for a handler with the capabilities to deal with the exception that the conjunction of all raised exceptions (i.e. ex_1 ***and*** $\ldots ex_{n-1}$ ***and*** ex_n) defines.

In large and complex collaborative business process as those we are targeting in this work, the number of potential exceptions that may arise in parallel grows exponentially. In line with the adopted principle of explicit dependability, every potential combination of parallel exceptions that could be raised and want to be handled *must* be explicitly determined in the business process definition. Those parallel exceptions that are not explicitly included in the business process definition are assumed (by the stakeholders) as impossible to occur.

Parallel exceptions at the level of the participant are defined by joining with the conjunction operator **and** those deviations that define each of them (i.e. ***deviation*** $[ex_1 \mid condition_1]$ **and**, ... , **and** ***deviation*** $[ex_i \mid condition_i]$). For those parallel exceptions that want to be handled within the context of the participant, a *p-handler* has to be defined as already explained for each of them, otherwise they are propagated to the business process context, as usual.

Parallel exceptions at the level of the business process are defined in the **resolution** section of the *dependability model*. In this section, every set of parallel exceptions (left part) is mapped to the respective *bp-handler* that is in charge of its handling. An arrow (→) is used to define the binding between left and right parts. Note that a single raised exception can be considered as a particular case of parallel exceptions: in this case the set of parallel exceptions contains only one. Therefore, the *resolution* section is used to define statically the binding between an exception and its associated handler, regardless of whether such an exception is defined as the conjunction of parallel raised exceptions or it is just a simple one.

Figure 26 shows the ***resolution*** section within the *dependability model* of *examination* business process. A parallel exception that may arise during the examination of the patient is that both his temperature and blood pressure are over the expected values (i.e. over 40 and 200, respectively). A specific handler called *UndressANDCardiacUnit* is defined to deal with such parallel exception (lines 14 and 15). The binding between this parallel exception and its associated handler is shown on line 4. The bindings *EX_HighTemp-Undress* (line 2) and *EX_HighBP-CardiacUnit* (line 3), which are considered when each exception is raised alone, are also included in this section.

4.5.3.6. Business process and composite activity binding. Since a *composite* activity is an abstraction of a business process, there must exist a clear connection between the different results the former can produce and the guaranteed outcomes of the latest (see Figure 27). A *composite* activity, as every activity in general, is considered as having executed successfully when its associated post-condition holds. Such result is achieved when the business process such *composite* activity refers to produces a *normal* outcome.

A deviated *composite* activity is one whose execution is considered as non-properly done. This means that the business process encapsulated by the *composite* activity has produced an outcome different to *normal*. A *degraded* outcome produced

```
1  resolution{
2          EX_HighTemp -> Undress;
3          EX_HighBP -> CardiacUnit;
4          EX_HighTemp and EX_HighBP -> UndressANDCardiacUnit;
5  }
6
7  recovery{
8     Undress(out Temperature temp, BloodPressure bp) {...
9     }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
10
11    CardiacUnit(out Temperature temp, BloodPressure bp) {...
12    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
13
14    UndressANDCardiacUnit(out Temperature temp, BloodPressure bp) {...
15    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
16 }
```

Figure 26. Binding single and parallel exceptions with their respective *bp-handlers* in the *dependability model* of *examination* business process.

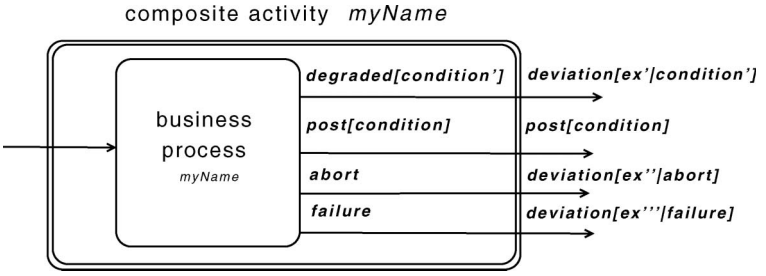

Figure 27. Relationships between the outcomes of a business process and its view as composite activity.

by a business process (i.e. ***degraded***[*condition*]), becomes a *deviation* at the level of the *composite* activity. This deviation carries the same condition defined by the degraded outcome along with the exception used to notify such deviation ***deviation***[*ex* | *condition*]). The *abort* and *failure* outcomes become special kind of deviations at the level of the *composite* activity. What makes them special is that their names are used to describe the deviation condition since their implicit semantics is enough to understand the kind of outcome produced by the encapsulated business process. Note that *abort* and *failure* deviations are part of any *composite* activity by default, as their respective related outcomes are always part of a business process.

4.6. *Comparison with existing business process modelling languages*

This section compares DT4BP with the existing business process modelling languages described in Section 3. The comparison criteria is based on those key concepts that govern the targeted business processes (i.e. collaboration, time and dependability) along with the relevant factors that should be considered when modelling a business enterprise (i.e. function, organisation, information and resource (Toh 1999; Liu *et al.* 2008)). This comparison allows the reader to identify in a compact manner not only what are the modelling concepts for which DT4BP offers support, but also to realise what is the current support of the selected existing business process languages with respect to these concepts.

The metric used to evaluate the support that a particular business process modelling language offers for certain concept is defined as follows: (+) indicates *direct support* for the concept, (+/−) indicates *partial support*, and (−) indicates that the concept is *not supported* at all. *Direct support* means that there exist constructs in the language that allow certain concept to be captured effectively in accordance with its semantics. *Partial support* means that a modelling language is able to capture a concept by combining or extending (if possible) some of its constructs, despite the fact that such constructs are aimed at capturing different concepts. It is important to underline that to consider that certain modelling language provides partial support for a concept that it originally did not target, the proposed workaround should correlate relatively closed with the concept being addressed. A modelling language that requires either complex workarounds to capture a particular concept or significant conceptual reorganisation of its constructs is considered as the one that does not provide support to capture such concept.

Table 1 lists the results of applying the assessment criteria both to the selected existing business process modelling languages as well as to DT4BP. This table shows also what are the aspects that each of key concepts being considered cover. The *modelling viewpoint* addresses the different types of perspectives oriented to the description of a business process. Russell (2007) identifies *control-flow*, *data*, *resource* and *exception handling* as the four dimensions that provide the basis for the modelling of business processes. DT4BP, by its *process*, *data*, *resource* and *dependability* models provides explicit support to allow the modeller to address each of these perspectives. The only comparable existing business process modelling language that also provides explicit support for these perspectives is YAWL.

The aspects related to the collaboration concept are those that determine not only the possibility of defining the different participants required to take part in the business process, but also the interactions between such participants (aka choreography) which determine their collaborative efforts for achieving the business process goal. DT4BP, by its built-in constructs *participant*, *send and receive* provides direct support for these aspects.

One of the characteristics that allows positioning DT4BP one step forward with respect to the existing business process modelling languages is its powerful set of time-related primitives for constraining the time behaviour of the business processes.

Table 1. Comparison between existing business process modelling languages and DT4BP.

Concept	**UML-AD**	**BPMN**	**YAWL**	**EPC**	**DT4BP**
Modelling viewpoint					
Control-flow	+	+	+	+	+
Data	−	−	+	−	+
Resource	−	−	+	−	+
Exception handling	−	−	+	−	+
Collaboration					
Multi-participants	+/−	+	−	−	+
Choreography	+/−	+	−	−	+
Time					
Business process min. delay	+	+	−	−	+
Business process max. delay	−	−	−	−	+
Business process min. duration	−	−	−	−	+
Business process max. duration	−	+	−	−	+
Periodic business process	−	+	−	−	+
Participant working time	−	−	−	−	+
Activity min. delay	+	+	+/−[a]	−	+
Activity max. delay	−	−	−	−	+
Activity min. duration	−	−	−	−	+
Activity max. duration	−	+	+/−[b]	−	+
Object duration	−	−	−	−	+
Dependability					
Pre-condition	−	−	+	−	+
Post-condition	−	−	+	−	+
Transaction	−	+	−	−	+
Deviation	+	+	+	−	+
Deviation handling	+	+	+	−	+
Multiple parallel deviations	−	−	−	−	+
Cooperative handling	+/−	+/−	−	−	+

[a]Only for atomic automatic activities.
[b]Only for atomic manual activities.

The fact that the primitives to constrain the business process are different from those used to constrain activities makes their semantics clearer, avoiding misconceptions about the time frame of reference associated to each primitive. Furthermore, time primitives for constraining the time a participant is able to work, as well as the duration of the value stored by an object are novel means that have never been considered by any other notation.

Another characteristic that makes DT4BP a superior language with respect to the existing BPMNs is the feature that allows to consider the occurrence of multiple parallel exceptions as a new exception along with its respective handling to avoid the failure of the business process. This feature combined with the support to consider business processes as long-lived transactions, and the explicit handling of deviations whether at the level of the participant or the business process represent a support for the dependability concept that goes beyond the one given by the existing languages.

Last, but not of least importance is the fact that DT4BP is a business process modelling language that provides support for every listed aspect in an integrated manner. The *Process Model* works as a pivot element where the data types defined in the *Data Model* are used to determine the data objects required to perform the different activities that compose the business process. The same principle applies between the resources defined in the *Resource Model* and the allocation policies defined in the *Process Model* used to select the actual resources in charge of performing the business process activities. On the other hand, the *Dependability Model* represents a complementary dimension for the *Process Model* that is used to drive the collaborative handling of simple and parallel deviations that take place during the enactment of the business process.

5. Conclusion and future work

We have presented in this article a business process modelling language called *DT4BP*. This language has been engineered to drive the modelling of dependable collaborative time-constrained business processes. Each of the dimensions that defines the targeted business process was deeply reviewed to extract the key concepts business analysts' and software engineers' need to manipulate during the business modelling and early phases of the software development process, respectively. In particular, reviewing the notion of dependability from the business process perspective has led us to provide a definition for *resilience* and *dependable business processes* and the position of the former with respect to the achievement of the latter. The *DT4BP* primitives used to represent these key concepts were carefully chosen such that the business process definition remains easy to read and understand for people involved not only in the modelling process, but also in its latter use. This is achieved by structuring a business process definition as a set of different models such that each model targets a specific concern.

Currently available business process modelling languages fall short when modelling dependability, collaboration and time aspects in an integrated way. While some of the existing languages directly lack in considering at least one of these dimensions, those that cover the three dimensions do it in a partial way. This article, for the first time, provides the definition of a business process modelling language that addresses explicit dependability in the description of collaborative business processes that in a way or the other must meet time constraints.

The feasibility of DT4BP for modelling the targeted business processes has been assessed by modelling a very simple (but complete) case study. Whether DT4BP is

desirable for modelling dependable collaborative time-constrained business processes can only be determined from future practical experience on modelling more complex case studies. This is a part of the future planned work along with the development of tools for easing the usability of the language (i.e. graphical editor) and validation of business process models. The validation of business process models will be performed by simulation. Both tools will be developed following a Model-Driven Engineering (Kent 2002, Schmidt 2006) approach, which means an intensive use of meta-modelling and model-to-model transformation techniques. A graphical concrete syntax is currently under development.

Notes

1. Here the term organisation is used as synonym of enterprise.
2. In this context, the notion of business process includes the organisation that owns it.
3. Having reflexive concepts for self-reinforcement of business processes will still be used to build dependable business processes. Means will nevertheless be more powerful since it will allow for dynamic self-adaptation of the business process to improve its resiliency.
4. Modelling and simulating error detection and recovery activities could be used as an effective method to estimate the consequences of a fault.
5. The acronym ACID means Atomicity, Consistency, Isolation and Durability.
6. This clock should be seen as a calendar that also provides information about the current physical time: i.e. YYYY-MM-DD:HH.MM.SS.
7. In case the specification is composed of only one YAWL-net, such net contains the entire definition of the business process.
8. A detailed explanation about rules can be found in YAWL Foundation (2009), in Section 7.5.
9. The set of concepts define the *abstract* syntax of the DT4BP language.
10. The word *primitive* is used as synonym of *construct*. The set of primitives define the *concrete* syntax of the DT4BP language.
11. A business entity that is required to perform at least one activity of the process is considered as a participant in such process.
12. This is a simplified version of the real *Process Model*. See the Appendix for the full description of the model.
13. Idem for ''examination'', ''make document'', ''consultation'' and ''give information''.
14. A resource might belong to more than one class.
15. This can be also denoted by using the **alloc**() primitive without argument.
16. An exception to this rule are business processes that are performed periodically. Such business processes are considered in Section 4.4, when time-related aspects in a business processes are introduced.
17. It is a good practice that the outmost business process includes only one participant, since it allows the reader to have a quick and simple overview about the business process being modelled.
18. It is assumed that for any two different participants P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent. It is also assumed that every sent message is eventually received.
19. A message can carry data. In that case the notation to be used is *msg(data)*.
20. When the *send* primitive is used to broadcast a message, more than one participant can be specified as receiver.
21. In theory, what we call *absolute time* is also relative, since it counts from a particular event. (e.g. the Christ's birth is the event on which the Gregorian calendar – unofficial global standard – relies on for numbering the years).
22. Atomic clocks are examples of such kind of clocks.
23. Time-related events are included in such consideration. Such events are launched automatically by the built-in clock that measures the passage of the time as seen in the physical world.
24. This statement means that t is the maximum allowed time for executing the activities $a_1, \ldots, a_n$.

25. The activity a_n must be the one in charge of receiving the message.
26. The primitive *clock* queries a clock device to get the current absolute time. The queried clock device is the one held by the participant where the primitive is included.
27. Dates and times are represented using the Gregorian calendar as it is the international standard.
28. The way in which information is set on a calendar is out the scope of this work.
29. A pre- or post-conditions that is not modelled is considered as true.
30. Otherwise the business process will not start executing.
31. The reader is encouraged to look at Section A.1 of the Appendix to see the complete example.
32. Detection of the deviation during the business process execution, and handling of the deviation invoked on demand after the deviation detection.
33. It is impossible to get raised multiple exceptions when a participant is prescribed to perform in parallel *atomic* activities only, since all these activities are to be performed by the same actual resource. In this scenario, the resource performs the parallel activities in a time-sharing fashion.

References

Abbott, R.K. and Garcia-Molina, H., 1992. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems*, 17 (3), 513–560.

ARIS Community, 2009. *ARIS Express 1.0* [online]. . Available from: http://www.ariscommunity.com/aris-express [Accessed 16 December 2009].

Avizienis, A., 1985. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11 (12), 1491–1501.

Avizienis, A., *et al*., 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1 (1), 11–33.

Bradford, L. and Dumas, M., 2007. *Getting started with YAWL* [online]. Available from: http://yawlfoundation.org/yawldocs/GettingStartedWithYAWL.pdf [Accessed 10 December 2009].

Buhr, P.A. and Mok, W.R., 2000. Advanced exception handling mechanisms. *IEEE Transactions on Software Engineering*, 26 (9), 820–836.

Burns, A., 1991. Scheduling hard real-time systems: a review. *Software Engineering Journal*, 6, 116–128.

Burns, A. and Wellings, A.J., 2001. *Real-time systems and programming languages: ADA 95, real-time Java, and real-time POSIX*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Campbell, R.H. and Randell, B., 1986. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12 (8), 811–826.

Cardoso, J., Bostrom, R.P., and Sheth, A., 2004. Workflow management systems and ERP systems: differences, commonalities, and applications. *Information Technology and Management*, 5 (3–4), 319–338.

Cristian, F., 1989. Exception handling. *In*: T. Anderson, ed. *Dependability of resilient computers*. Oxford: Blackwell Scientific Publications, 68–97.

Cristian, F. and Fetzer, C., 1994. Probabilistic internal clock synchronization. *In*: *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, October 1994, Dana Point, CA. Los Alamitos, CA: IEEE Computer Society, 22–31.

Elnozahy, E.N.M., *et al*., 2002. A survey of rollback-recovery protocols in message-passing systems. *ACM Computer Surveys*, 34 (3), 375–408.

Gray, J., 1981. The transaction concept: virtues and limitations (invited paper). *In*: *VLDB '1981: Proceedings of the Seventh International Conference on Very Large Data Bases*, Cannes, France VLDB Endowment, 144–154.

Gray, J. and Reuter, A., 1992. *Transaction processing: concepts and techniques*. San Francisco, CA: Morgan Kaufmann.

Guelfi, N., *et al*., 2008. *SERENE'08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, Newcastle upon Tyne. New York, NY, USA: ACM.

Gusella, R. and Zatti, S., 1989. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD. *IEEE Transactions on Software Engineering*, 15 (7), 847–853.

Hamel, G. and Välikangas, L., 2003. The quest for resilience. *Harvard Business Review*, 81 (9), 52–63.

Harel, D. and Rumpe, B., 2004. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37 (10), 64–72.

Horning, J.J., *et al*., 1974. A program structure for error detection and recovery. *In*: *Operating Systems, Proceedings of an International Symposium*. London, UK: Springer-Verlag, 171–187.

IFIP WG 10.4 on Dependable Computing and Fault Tolerance, 1960. *IFIP WG 10.4* [online]. Available from: http://www.dependability.org/wg10.4/ [Accessed 01 December 2009].

Kent, S., 2002. Model driven engineering. *In*: *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*. London, UK: Springer-Verlag, 286–298.

Lamport, L., 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21 (7), 558–565.

Lamport, L., Shostak, R., and Pease, M., 1982. The Byzantine generals problem. *ACM Transactions on Programming Language and Systems*, 4 (3), 382–401.

Lima, G.M. de A. and Burns, A., 2005. Scheduling fixed-priority hard real-time tasks in the presence of faults. *In*: *Proceedings of the second Latin-American symposium – LADC 2005*. Berlin/Heidelberg: Springer, 154–173.

Liu, X., *et al*., 2008. Manufacturing perspective of enterprise application integration: the state of the art review. *International Journal of Production Research*, 46 (16), 4567–4596.

Mans, R., *et al*., 2009. Schedule-aware workflow managment systems. *In*: *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE '09)*, Paris. Hamburg: University of Hamburg, Department of Informatics, 81–96.

Marjanovic, O., 2000. Dynamic verification of temporal constraints in production workflows. *In*: *ADC'00: Proceedings of the Australian Database Conference*. Washington, DC: IEEE Computer Society, 74. ISBN 0-7695-0528-7.

Meyer, B., 1997. *Object-oriented software construction*. 2nd ed. Upper Saddle River, NJ: Prentice-Hall.

Mustafiz, S., Kienzle, J., and Berlizev, A., 2008. Addressing degraded service outcomes and exceptional modes of operation in behavioural models. *In*: *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems*, Newcastle upon Tyne, United Kingdom. New York, NY, USA: ACM, 19–28.

Network Time Protocol, 1980. *NTP* [online]. Available from: http://www.ntp.org [Accessed 1 September 2009].

OASIS, 2007. *WS-BPEL 2.0* [online]. Available from: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf [Accessed 17 February 2010].

Object Management Group (OMG), 2009a. *Unified modeling language superstructure specification version 2.2*. Available from: http://www.omg.org/cgi-bin/doc?formal/09-02-02.pdf [Accessed 10 December 2009].

OMG, 2009b. Business process modeling notation (BPMN). Version 1.2. Available from: http://www.omg.org/spec/BPMN/1.2/PDF [Accessed 10 December 2009].

Papazoglou, M.P. and van den Heuvel, W.J., 2007. Business process development life cycle methodology. *Communications of the ACM*, 50 (10), 79–85.

Randell, B., 1975. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, (SE-1), 220–232.

Randell, B. and Koutny, M., 2007. Failures: their definition, modelling and analysis. *Theoretical Aspects of Computing ICTAC 2007*. Berlin, Heidelberg: Springer, 260–274.

Romanovsky, A., Xu, J., and Randell, B., 1999. Coordinated exception handling in real-time distributed object systems. *Computer Systems Science and Engineering*, 14 (4), 197–208.

Russell, N.C., 2007. *Foundations of process-aware information systems*. Thesis (PhD). Faculty of Information Technology, Queensland University of Technology.

Schmidt, D.C., 2006. Guest editor's introduction: model-driven engineering. *Computer*, 39, 25–31.

Soffer, P., Golany, B., and Dori, D., 2003. ERP modeling: a comprehensive approach. *Information Systems*, 28 (6), 673–690.

Tel, G., 1994. *Introduction to distributed algorithms*. New York, NY, USA: Cambridge University Press.
Toh, K.T.K., 1999. The realization of reference enterprise modelling architectures. *International Journal of Computer Integrated Manufacturing*, 12 (5), 403–417.
Van Der Aalst, W.M.P., *et al*., 2003. Workflow patterns. *Distributed and Parallel Databases*, 14 (1), 5–51.
van der Aalst, W.M.P., 1999. Formalization and verification of event-driven process chains. *Information & Software Technology*, 41 (10), 639–650.
Warmer, J. and Kleppe, A., 2003. *The object constraint language: getting your models ready for MDA*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.
White, S.A. and Miers, D., 2008. *BPMN modeling and reference guide: understanding and using BPMN*. Lighthouse Point, FL, USA: Future Strategies Inc.
Workflow Management Coalition, 1999. Terminology & Glossary. Document Number WFMC-TC-1011. Issue 3.0. Available from: http://www.wfmc.org/Download-document/WFMC-TC-1011-Ver-3-Terminology-and-Glossary-English.html [Accessed 17 February 2010].
Workflow Management Coalition, 2008. *XPDL 2.1* [online]. Available from: http://www.wfmc.org/xpdl.html [Accessed 17 February 2010].
YAWL Foundation, 2009. *YAWL User Manual 2.0* [online]. Available from: http://www.yawlfoundation.org/yawldocs/YAWLUserManual2.0.pdf [Accessed 10 December 2009].
Yet Another Workflow Language, 2008–2009. *YAWL* [online]. Available from: www.yawlsystem.com [Accessed 1 December 2009].

Appendix 1. The patient diagnosis running example

A.1.1. Diagnosis business process definition

A.1.1.1. Process model

```
;;*****************************************************************
business process diagnosis() when(patientArrives) last[_,2hs.] {
;;*****************************************************************
participant DiagnosisUnit {
  do{
    Person prs;
    composite registration[_,p = alloc(new Patient)](out prs) within[_,15min.],
      post[not prs.oclIsUndefined()];
    Temperature t expire(1hs.);
    BloodPressure bp expire(1hs.);
    PatientSheet ps;
    do{split composite examination[p,_,_](out t,bp)
        post[not t.oclIsUndefined() and
          not bp.oclIsUndefined()],
  composite makeDocument(in prs; out ps)
    post[not ps.oclIsUndefined()]
  }within[_,30min.];
    composite consultation[p,_](in ps,t,bp; out ps) within[15min.,1hs.];
        pre[not ps.oclIsUndefined() and
          not t.oclIsUndefined() and
          not bp.oclIsUndefined()]
        post[not ps.d.oclIsUndefined() and
          not ps.p.oclIsUndefined()]
        deviation[EX_Admission|ps.d.oclIsUndefined()]
    composite giveInformation[p,_](in ps) within[_,15min.]
        pre[not ps.p.oclIsUndefined]
        post[true]
      }deviation[EX_Fire|(fa|fa.oclIsNew() and fa = #ON)]
  }
} post[not ps.d.oclIsUndefined and not ps.p.oclIsUndefined]
```

A.1.1.2. Data model

```
type Person{
   String name;
   String surname;
   Calendar birthday;
   String address;
   String city;
   String country;
   Integer ssn;
}
type Calendar;
type Temperature { Integer t where
      (30 < = t) and (50 > = t)}
type BloodPressure { Integer bp where
      (50 < = bp) and (250 > = bp)}
type Treatment,Medicine,Diagnosis,MedicalHistory;
type PatientSheet{
   Integer ssn;
   String name;
   String surname;
   String address;
   String city;
   String country;
   MedicalHistory mh;
   Diagnosis d;
   Prescription p;
}
type Prescription{
   Treatment t;
   Medicine m;
}
type FireAlarm = enum{ON,OFF}
```

A.1.1.3. Resource model

```
resources{
   DiagnosisUnit = DU1;
   Secretary = Ann;
   Nurse = Sue, Rose;
   Patient;
   Assistant = Jane;
   Doctor{Integer cost} = Marc(50), Nick(80);
}
```

A.1.1.4. Dependability model

```
;;Binding EXEC-HANDLER
resolution{
   EX_Fire –> Evacuation;
   EX_Admission –> PatientAdmission;
}
```

```
;;HANDLER definition
recovery{
   Evacuation(){
      evacuateDiagnosisUnit();
   }Failure
   PatientAdmission(){
      addmit(in ps);
   }degraded[ps.d.oclIsUndefined and not ps.p.t.oclIsUndefined]
}
```

A.1.2. Registration business process definition

A.1.2.1. Process model

```
;;*************************************************************
business process registration(out Person p) last[_,15min.]{
;;*************************************************************
participant Secretary{
      send reqSSCard to Patient block;
   SocialSecurityCard ssCard;
      receive reqSSCard(ssCard) from Patient;
      Hospital h;
   checkSSCard(in ssCard,h)
         post[ssCard.country = h.country]
         deviation[EX_foreigner|ssCard.country <> h.country];
   String address,city;
   getPersonAddress(in ssCard.number; out address,city);
         post[address <>'' and city <>'']
   send reqAddress_and_City to Patient block;
   String p_address,p_city;
      receive dataRequested(p_address,p_city) from Patient;
   validatePerson(in p_address,p_city,address,city)
      post[p_address = address and
         p_city = city]
      deviation[Abort|p_address <>address or
         p_city <>city]
   registerPerson(in address,city,ssCard;out p)
      post[p|p.oclIsNew() and
         p.name = ssCard.name and
         p.surname = ssCard.surname and
         p.birthday = ssCard.birthday and
         p.address = address and
         p.city = city and
         p.country = ssCard.country and
         p.ssn = ssCard.number];
}
participant Patient{
      receive reqSSCard from Secretary;
      SocialSecurityCard ssCard;
      searchSScard(out ssCard) post[not ssCard.oclIsUndefined()];
            deviation[EX_notSSCard|ssCard.oclIsUndefined()]
      send reqSSCard(ssCard) to Secretary;
      receive reqAddress_and_City from Secretary;
      String p_address,p_city;
      send dataRequested(p_address,p_city) to Secretary;
   }
} post[not p.oclIsUndefined()]
```

A.1.2.2. Data model

```
type Person{
   String name;
   String surname;
   Calendar birthday;
   String address;
   String city;
   String country;
   Integer ssn;
}
type Calendar;
type SocialSecurityCard{
   Integer number;
   String name;
   String surname;
   Calendar birthday;
   Calendar expirationDate;
   Integer cardID;
   String country;
}
type Hospital{
   String name;
   String address;
   String city;
   String country;
}
type ForeignerForm{
   String address;
   String city where city <> Hospital.city;
   }
```

A.1.2.3. Resource model

```
resources{
   Secretary = Ann;
   Patient;
}
```

A.1.2.4. Dependability model

```
_____________________________________
;;Binding EXEC-HANDLER
_____________________________________
resolution{
   EX_notSSCard -> Abort;
   EX_foreigner -> FillSpecialForm;
}
_____________________________________
;;HANDLER definition
_____________________________________
recovery{
   Abort(out Person p) {
      participant Secretary{
         send goToAdminOffice to Patient block;
      }
```

```
    participant Patient{
      receive goToAdminOffice from Secretary;
    }
  }Abort
  FillSpecialForm(out Person p) {
    participant Secretary{
      ForeignerForm ff;
      createForeignerForm(out ff)post[ff|ff.oclIsNew()];
      send askFillForm(ff) to Patient block;
      receive filledForm(ff) from Patient;
      registerPerson(in ff,ssCard;out p)
        post[p|p.oclIsNew() and
          p.name = ssCard.name and
          p.surname = ssCard.surname and
          p.birthday = ssCard.birthday and
          p.address = ff.address and
          p.city = ff.city and
          p.country = ssCard.country and
          p.ssn = ssCard.number];];
    }
    participant Patient{
      ForeignerForm ff;
      receive askFillForm(ff) from Secretary;
      fillOutForm(in ff; out ff)
        post[ff.address <> '' and ff.city <> ''];
      send filledForm(ff) to Secretary block;
    }
  }post[not p.oclIsUndefined()]
}
```

A.1.3. Examination business process definition

A.1.3.1. Process model

```
;;*****************************************************************
business process examination(out Temperature temp,
        BloodPressure bp)
    last[_,30min.]{
;;*****************************************************************
participant Patient{
  receive reqProblemExplanation from Nurse;
  explainWhatIsWrong();
  spawn receive getTemp from Nurse,
    receive getBP from Assistant;
}
participant Nurse{
  send reqProblemExplanation to Patient;
  send getTemp to Patient block;
  repeat{
    checkTemp()
      post[temp|temp.oclIsNew() and temp <40];
      deviation[EX_HighTemp|(temp|temp.oclIsNew() and
        temp > 40)]
      deviation[EX_MalfunctionThermometer|temp.oclIsUndefined()]{
          findNewThermometer();
      }
  until(not temp.oclIsUndefined());
}
```

```
participant Assistant{
  send getBP to Patient block;
  repeat{
    checkBP()
      post[bp|bp.oclIsNew() and bp <200];
      deviation[EX_HighBP|(bp|bp.oclIsNew() and bp > 200)]
      deviation[EX_MalfunctionBPMonitor|bp.oclIsUndefined()]{
        findNewBPMonitor();
      }
until(not bp.oclIsUndefined());
  }
}post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
```

A.1.3.2. Data model

```
type Temperature { Integer t where
    (30 geq t) and (50 leq t)
}
type BloodPressure { Integer bp where
    (50 geq bp) and (250 leq bp)
}
```

A.1.3.3. Resource model

```
resources{
  Nurse = Sue, Rose;
  Patient;
  Assistant = Jane;
}
```

A.1.3.4. Dependability model

```
_____________________________________
;;Binding EXEC-HANDLER
_____________________________________
resolution{
  EX_HighTemp -> Undress;
  EX_HighBP -> CardiacUnit;
  EX_HighTemp and EX_HighBP -> UndressANDCardiacUnit;
}
_____________________________________
;;HANDLER definition
_____________________________________
recovery{
  Undress(out Temperature temp, BloodPressure bp) {
    participant Patient{
      receive reqUndress from Nurse;
      Undress();
    }
    participant Nurse{
      send reqUndress to Patient block;
    }
    participant Assistant{}
  }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
  CardiacUnit(out Temperature temp, BloodPressure bp) {
    participant Patient{
```

```
        receive reqChangeRoom from Assistant;
          goToSpecialUnit();
      }
      participant Nurse{
        receive reqChangeRoom from Assistant;
        takePatientToSpecialUnit();
      }
      participant Assistant{
        send reqChangeRoom to Patient,Nurse;
        notifyNewRoomPatient();
      }
    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
    UndressANDCardiacUnit(out Temperature temp, BloodPressure bp) {
      participant Patient{
        receive reqChangeRoom from Assistant;
        goToSpecialUnit();
        receive reqUndress from Nurse;
        Undress();
      }
      participant Nurse{
        receive reqChangeRoom from Assistant;
        takePatientToSpecialUnit();
        send reqUndress to Patient block;
      }
      participant Assistant{
        send reqChangeRoom to Patient,Nurse;
        notifyNewRoomPatient();
      }
    }post[not temp.oclIsUndefined() and not bp.oclIsUndefined()]
}
```

A.1.4. MakeDocument business process definition

A.1.4.1. Process model

```
;;*****************************************************************
business process makeDocument(in Person prs;
        out PatientSheet ps) last[_,30min.]{
;;*****************************************************************
participant Nurse{
  ;;Hx = abbr. Medical History
  MedicalHistory hx;
  getHxPatient(in prs.name,prs.surname,prs.birthday; out hx)
    post[not hx.oclIsUndefined()];
    deviation[EX_undefHx|hx.oclIsUndefined()]{
      createtHx(in prs)
        post[hx|hx.oclIsNew()];
    }
  makePatientSheet(in hx,prs; out ps);
    post[ps.ssn = prs.ssn and
      ps.name = prs.name and
      ps.surname = prs.surname and
      ps.addres = prs.address and
      ps.city = prs.city and
      ps.country = prs.country and
      ps.mh = hx]
  }
}
```

A.1.4.2. Data model

```
type Person{
   String name;
   String surname;
   Calendar birthday;
   String address;
   String city;
   String country;
   Integer ssn;
}
type Treatment,Medicine,Diagnosis,MedicalHistory,Calendar;
type PatientSheet{
   Integer ssn;
   String name;
   String surname;
   String address;
   String city;
   String country;
   MedicalHistory mh;
   Diagnosis d;
   Prescription p;
}
type Prescription{
   Treatment t;
   Medicine m;
}
```

A.1.4.3. Resource model

```
resources{
   Nurse = Sue, Rose;
}
```

A.1.5. Consultation business process definition

A.1.5.1. Process model

```
;;*************************************************************
business process consultation(
      in PatientSheet ps, Temperature t, BloodPreasure bp;
      out PatientSheet ps) last[15min.,1hs.]{
;;*************************************************************
   participant Patient{
      receive chek from Doctor;
      Diagnosis d;
      Prescription p;
      receive news(d,p) from Doctor;
   }
   participant Doctor{
      evaluateExaminationResults(in t,bp,ps);
      send check to Patient block;
      HealthStatus hs;
      checkPatient(out hs)
         post[hs|hs.oclIsNew() and hs = #GOOD]
         deviation[EX_Admission|(hs|hs.oclIsNew() and hs = #BAD)];
      Diagnosis d;
      diagnosePatient(d)
```

```
    post[d|d.oclIsNew() and not d.oclIsUndefined()];
  Prescription p;
  prescribeTreatment(out p)
      post[p|p.oclIsNew() and not p.oclIsUndefined()];
  send news(d,p) to Patient block;
  fillPatientSheet(in ps; out ps)
    post[not ps.d.oclIsUndefined() and
      not ps.p.oclIsUndefined()]
    }workFor[15min.,_]
  }post[ps.d = d and ps.p = p]
```

A.1.5.2. Data model

```
type Temperature { Integer t where
  (30 < = t) and (50 > = t)}
type BloodPressure { Integer bp where
  (50 < = bp) and (250 > = bp)}
type HealthStatus = enum{GOOD,BAD};
type Treatment,Medicine,Diagnosis;
type PatientSheet{
  Integer ssn;
  String name;
  String surname;
  String address;
  String city;
  String country;
  MedicalHistory mh;
  Diagnosis d;
  Prescription p;
}
type Prescription{
  Treatment t;
  Medicine m;
}
```

A.1.5.3. Resource model

```
resources{
  Patient;
  Doctor{Integer cost} = Marc(50), Nick(80);
}
```

A.1.5.4. Dependability model

```
_____________________________________
;;Binding EXEC-HANDLER
_____________________________________
resolution{
  EX_Admission –> HospitalAdmission;
}
```

```
;;HANDLER definition
recovery{
    HospitalAdmission(in PatientSheet ps,
            Temperature t,
            BloodPreasure bp;
        out PatientSheet ps) {
    participant Patient{
      receive reqAdmission from Doctor;
    }
    participant Doctor{
      send reqAdmission to Patient;
      notifyAdministration(in ps);
      Treatment t;
      prescribeAdmission(out t)
        post[t|t.oclIsNew() and
          not t.oclIsUndefined()];
        fillPatientSheet(in ps; out ps)
      post[not ps.p.t.oclIsUndefined()]
    }
  }degraded[ps|ps.d.oclIsUndefined() and ps.p.t = t]
}
```

A.1.6. GiveInformation business process definition

A.1.6.1. Process model

```
;;*****************************************************************
business process giveInformation(in PatientSheet ps)
  pre[not ps.p.oclIsUndefined()]
  last[_,15min.] {
;;*****************************************************************
participant Patient{
receive treatmentDetails from Nurse;
receive medecineDetails from Nurse;
}
participant Nurse{
checkTreatment(in ps.p.t);
send treatmentDetails to Patient block;
checkMedicine(ps.p.m);
send medecineDetails to Patient block;
  }
}
```

A.1.6.2. Data model

```
type PatientSheet{
  Integer ssn;
  String name;
  String surname;
  String address;
  String city;
  String country;
  MedicalHistory mh;
  Diagnosis d;
  Prescription p;
}
```

```
type Treatment,Medicine,Diagnosis,MedicalHistory;
type Prescription{
   Treatment t;
   Medicine m;
}
```

A.1.6.3. Resource model

```
resources{
   Nurse = Sue, Rose;
   Patient;
}
```