Business Processes Definition Metamodel Concepts and Overview

Joachim H. Frank
Tracy A. Gardner
Simon K. Johnston
Stephen A. White
Sridhar Iyengar

all of IBM Software Group

version 2004-05-03 (Draft)

Abstract

This paper defines the fundamental concepts, key elements, and semantics of a metamodel that is being proposed as a response to the OMG's RFP for a Business Process Definition Metamodel [BPDM]. It also introduces notations that allow to view a process model at varying levels of detail, while maintaining consistency with the underlying model and semantics. A formal definition of the metamodel and its relationship with UML 2.0 [UML2] will be given in a forthcoming paper.

1 Introduction

Business processes are "behavioral building blocks" of a business: They describe the activities a business performs in pursuing its objectives. They can be broken down into subordinate processes and eventually tasks that are considered atomic. Business processes do not exist in isolation. In implementing a value chain business processes interact with partners, and these interactions may cross organizational boundaries.

We therefore model two aspects of a business process: its internal behavior, using a flow model, and its external interactions, using collaborations and associated protocol specifications. The first view is that of the process execution environment, the second view is required to connect processes with other components in a service-oriented architecture.

In introducing the modelling concepts, we will use the following example:

> A producer of electronics assemblies implements a "Just-in-Time Manufacturing" process: Incoming customer orders are evaluated based on the available manufacturing capacity and the parts inventory: If sufficient quantities for all parts are on stock and assembly line time slots are available, an order confirmation is returned to the customer, the assemblies are produced, and finally delivered together with an invoice. If assembly line capacity is not available, the order is declined. If manufacturing capacity is available but some parts are not, replenishment orders are first issued to preferred suppliers, including a deadline that leaves enough time for production. For parts that the preferred supplier cannot deliver within the allotted time frame, an auction is launched accepting bids from suppliers in the open market. If all parts can be procured before the production start deadline, the order is confirmed and the assemblies are produced, delivered, and invoiced. Otherwise, the order is rejected.

The *internal view* (or behavioral view) of a process shows "how it is performed", at the level of detail that the modeler knows or chooses to reveal: it is modeled as a partially ordered set of tasks. Figure 1 shows the internal view of the *Just-in-Time Manufacturing* process.[1] The meaning of symbols used in the diagram, and their implied semantics,[2] will be defined in chapter 3 "Operational Processes (Modeling Process Behavior)".

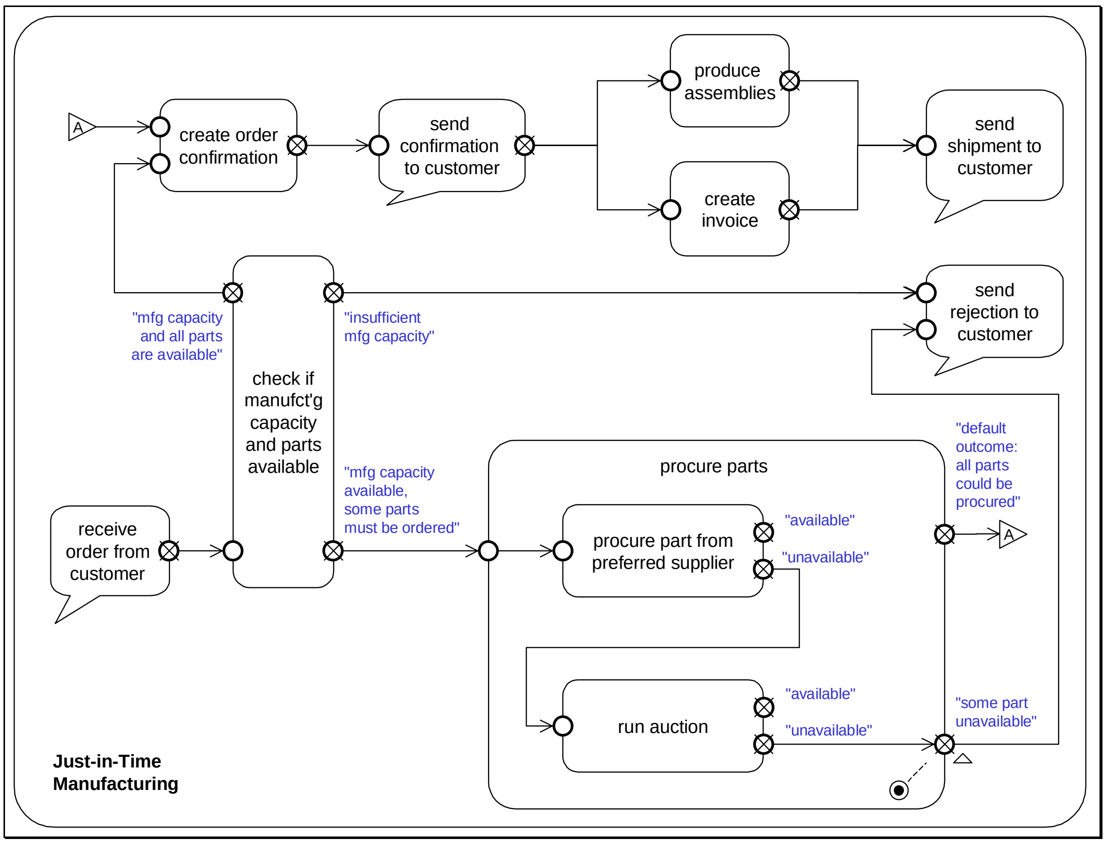
A
create order confirmation
send confirmation to customer
produce assemblies
create invoice
send shipment to customer
send rejection to customer
"mfg capacity and all parts are available"
"insufficient mfg capacity"
check if manufct'g capacity and parts available
"mfg capacity available, some parts must be ordered"
receive order from customer
procure parts
procure part from preferred supplier
"available"
"unavailable"
run auction
"available"
"unavailable"
"default outcome: all parts could be procured"
A
"some part unavailable"
Just-in-Time Manufacturing

Figure 1: Just-in-Time Manufacturing Process (internal / behavioral view)

The *external* view (or collaborative view) of a process shows "how it interacts". It specifies the partner roles that a process expects to collaborate with, and their interaction protocol. A high-level collaborative view of the *Just-in-Time Manufacturing* process is shown in Figure 2. The detailed interaction sequences associated with the collaborations (Order Fulfillment, Parts Provisioning and Open Auction) are not shown in this overview diagram, but will be introduced in chapter 2 "Process Components".

[1] The notation employed in this paper is not normative, but was chosen to provide a clear illustration of the modelling concepts we would like to support. In particular, it is not the standard notation of UML 2.0.

[2] Please note that while our process definitions have well-defined semantics, and hence are executable in principle, this does not imply that they are automated. On the other hand, they may be refined to the level of detail required by an execution engine, for complete or partial automation.

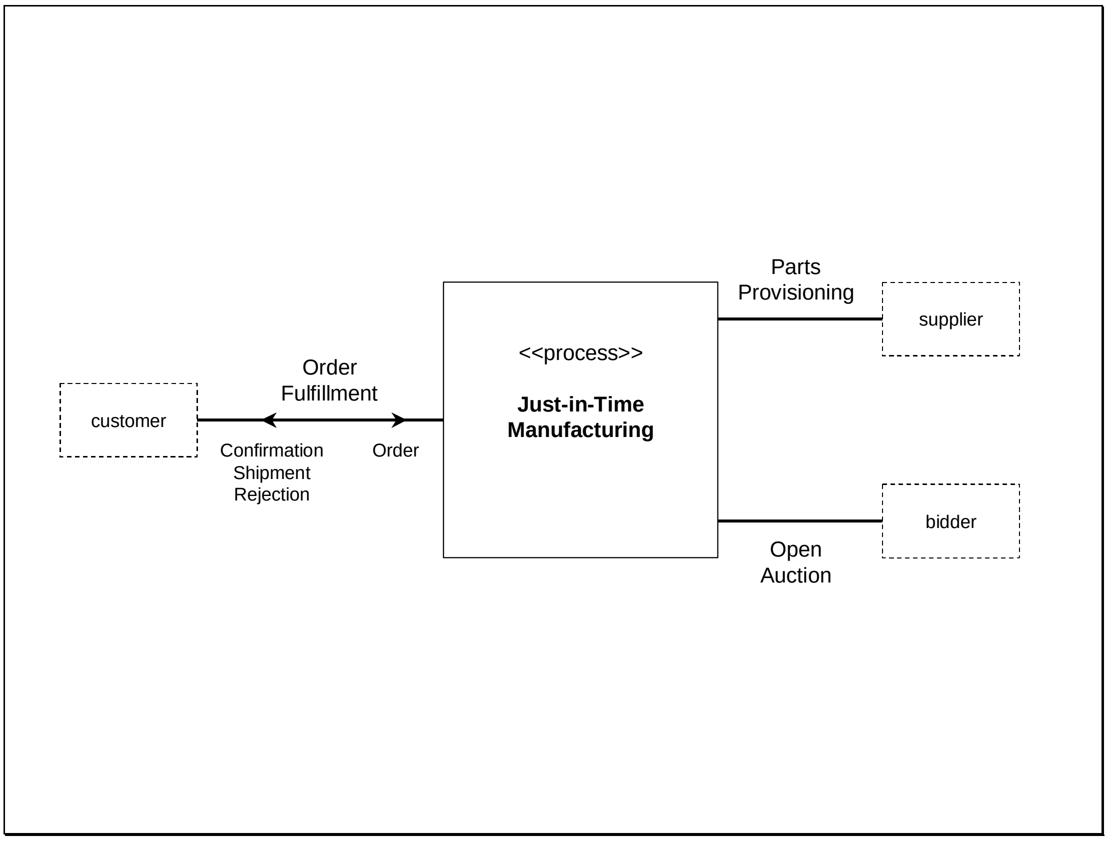

Figure 2: Just-in-Time Manufacturing process (external / collaborative view)

2 Process Components

To enable business processes to collaborate with partners it is important for them to be defined in a composable fashion. To this end, we apply the paradigm of service-oriented architecture to business process modeling: We treat a business process as a definition of a behavioral component[3] that connects to and interacts with other components (its partners). The internal behavior of a process then includes tasks that conduct the corresponding partner communications.

A complete solution is modeled as a network of interconnected components interacting via long-running conversations.[4] The same process can appear in multiple solutions and can be connected to different partners in each case, as long as the "partner roles" they provide match those required by the process. The partner interaction protocol is described using the concept of "collaborations". Partner roles and collaborations are introduced in the following sections.

The *Just-in-Time Manufacturing* process engages in three collaborations with partners: order fulfillment (with customers), parts provisioning (with suppliers), and open auction (with bidders). This is shown in Figure 2. In the following sections we

[3] The term component is used in this paper to denote the view of a process as a encapsulated behavioral element that can be connected with other such elements. Connections represent communication channels amongst these. Please note that this is not exactly the "Component" class of UML 2.0.

[4] Strictly speaking, instantiations of these components interact via long-running conversations; see section 3.4 "Process Life-Cycle and Inter-Process Communication" for a more in-depth description.

demonstrate how to model the external view of the process, including its partner dependencies and collaboration protocols.

2.1 Partner Roles

The component view of a process states its dependencies on its partners through "partner roles", which define roles to be fulfilled by components external to the process. They may be realized inside or outside thesame organization. In our *Just-in-Time Manufacturing* example, the manufacturing process collaborates with three partner roles: the customer who sends an order and must receive a response, a set of preferred suppliers who may provide parts, and potentially bidders participating in an open market auction.[5]

A graphical view of this is shown in Figure 2, where partner roles are denoted by dashed rectangles. Note that partner roles are not components, but must be substituted by components playing these roles in order to obtain a complete solution. The connection between a partner role and the process is labeled with the name of a collaboration defining the details of the underlying communication protocol. Modeling the protocol is described in the following sections.

2.2 Collaborations

A collaboration describes an interaction pattern between partner roles. It is defined from the viewpoint of an external observer of a group of interacting partners. Collaborations between roles are specified independently of any implementation of these roles. The same collaboration can be used in many scenarios that require the same interaction pattern, and can be implemented by different components each time.

A process component can play multiple partner roles, participating in different collaboraitons at the same time. The *Just-in-Time Manufacturing* process for example plays the manufacturer role in the Order Fulfillment collaboration, the customer role in the Parts Provisioning collaboration, and the auctioneer role in the Open Auction collaboration (This is shown in Figure 3).

A collaboration diagram shows communicating roles together with an overview of the items they exchange. Figure 3 shows that as part of the Order Fulfillment collaboraiton a customer sends orders to the manufacturer and the manufacturer returns order confirmations, shipments, or rejections back to the customer. In this overview, however, we cannot see the time ordering of the items exchanged (see section 2.3.2 "Collaborative Processes" for how this is specified).

In the Parts Provisioning collaboration, the *Just-in-Time Manufacturing* process plays the customer role (note that it is different from the customer role it requires as part of the Order Fulfillment collaboration), and in the Open Auction collaboration it plays the role of the auctioneer.

[5] Note that in Figure 1 process steps communicating with the customer are shown, while communication with other partners—which would become apparent as one refines the *procure part from preferred supplier* and *run auction* tasks—has not been modelled explicitly.

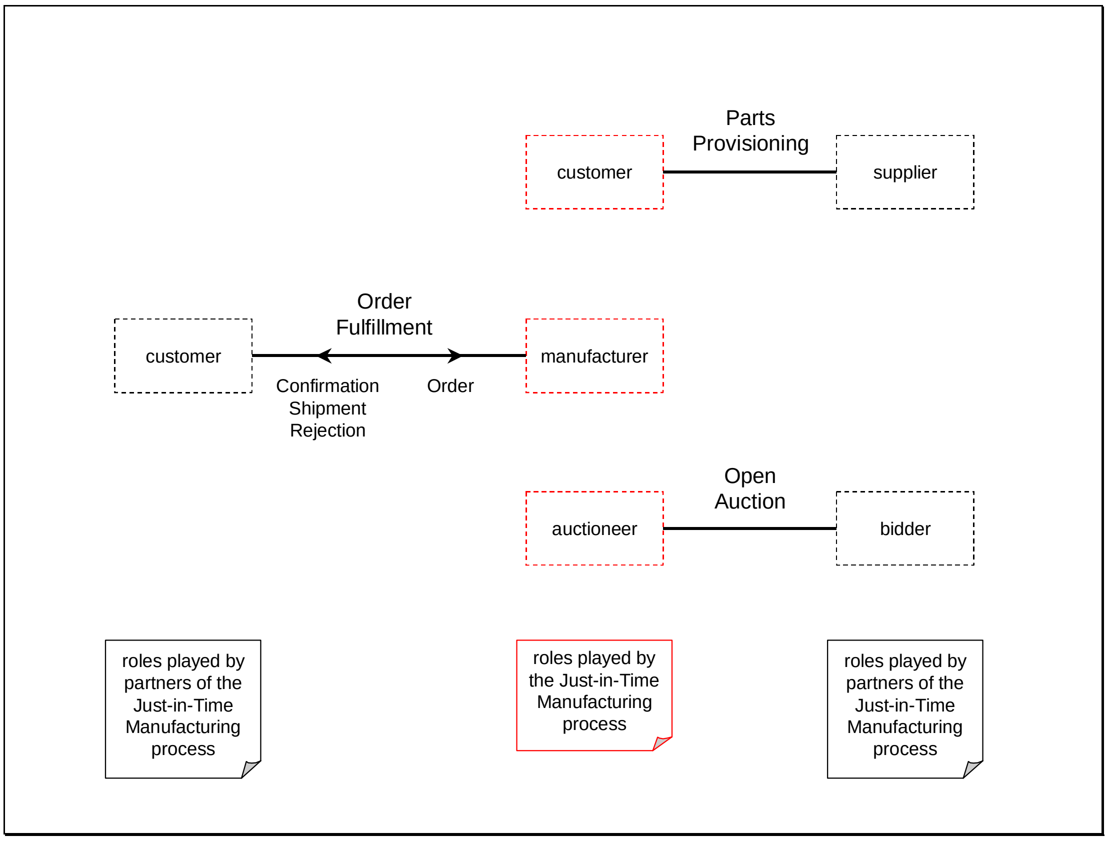
Parts Provisioning
customer
supplier
Order Fulfillment
customer
manufacturer
Confirmation
Shipment
Rejection
Order
Open Auction
auctioneer
bidder
roles played by partners of the Just-in-Time Manufacturing process
roles played by the Just-in-Time Manufacturing process
roles played by partners of the Just-in-Time Manufacturing process

Figure 3: Collaborations of the Just-in-Time Manufacturing process

2.3 Protocol Specification

A protocol between interacting partner roles can be defined either by specifying the externally visible behavior of each participant, or from the central viewpoint of an independent observer. In the first case, we define abstract processes, in the second case a collaborative process. Both cases are explained below using the *Just-in-Time Manufacturing* example.

2.3.1 Abstract Processes

An abstract process defines *public* behavior of a process. It describes what a process is expected to do in order to participate in a collaboration, but does not define how this is accomplished. An abstract process must be realized by an operational process, and *constrains its behavior*. Note that an abstract process can be realized by multiple operational processes, and in many different ways.

We can associate an abstract process with each role in a collaboration (in Figure 4 its name is placed after the name of the role, separated by a colon). It describes the behavior that must be exhibited by any process playing that role. In Figure 4 we see that in order to play the manufacturer role in the Order Fulfillment collaboration, the *Just-in-Time Manufacturing* process must implement the Fulfillment abstract process, while any process playing the customer role must implement the Ordering abstract process (whose behavior is not shown).

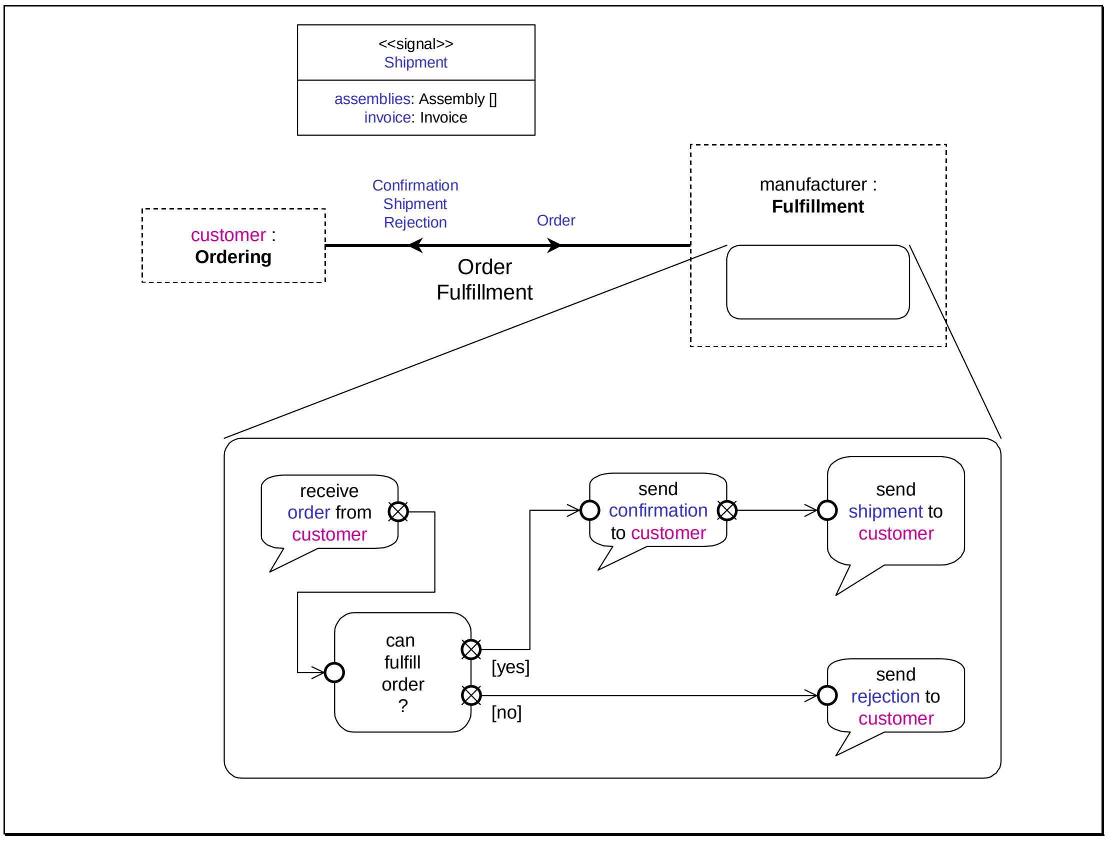

Figure 4: Abstract process for the "manufacturer" role of the Order Fulfillment collaboration

Also shown in Figure 4 are the items that the customer and the manufacturer may exchange in this collaboration: order, confirmation, shipment, and rejection. Note that the term "item" is used in a very general sense here, and may represent messages, documents, or business artifacts of any kind. Their type is defined using Classes, as exemplified for Shipment in Figure 4. We see that a Shipment consists of two parts: a set of assemblies and an invoice.

The flow of the Fulfillment abstract process is shown in Figure 4. The tasks receiving and sending items from / to the customer (denoted as callouts in this simplified view) refer to the corresponding partner role, which is highlighted in purple. They must be matched with actual send and receive tasks in an operational process realizing this abstract process. Note that the sequencing of communicationtasks in an abstract process only represents an ordering constraint: an operational process realizing the abstract process in Figure 4, for example, will likely introduce additional tasks between *send confirmation to customer* and *send shipment to customer*.

Also note that the decision whether or not an order can be fulfilled may be quite complex: it may involve an assessment of available manufacturing capacity, sending replenishment orders to preferred suppliers, and running open auctions. This level of detail, however, while part of the operational[6] process definition, is irrelevant for its

[6] Operational processes are described in chapter 3 "Operational Processes (Modeling Process Behavior)".

communication protocol with customers and therefore omitted in the abstract process.[7]

2.3.2 Collaborative Processes

Collaborative processes are used to detail the interaction protocol of a collaboration from the global viewpoint of an external observer. They specify the sequence of items exchanged by associating a process flow with the collaboration itself whose individual steps represent sections or "phases" of the interaction protocol.

Like operational and abstract processes, collaborative processes are described using activity graphs.[8] However, where abstract and operational processes use one-sided constructs like sending a message or receiving a message, a collaborative process uses symmetrical concepts, or joint actions, that show a message sent by one role and received by another. A collaborative process for the Order Fulfillment collaboration is shown in Figure 5.Note that it involves a decision *can fulfill order?* which it declares to be made by the manufacturer at some point in the protocol. It is the same decision that is shown in the abstract process in Figure 4.

[7] The formal definition of what it means for an operational process to satisfy one or more abstract processes and the creation of algorithms to verify this automatically are topics that still require some further investigation.

[8] There are restrictions on what is permitted within a collaborative process, for example, they cannot define global state that is visible by all roles and hence must not contain variables.

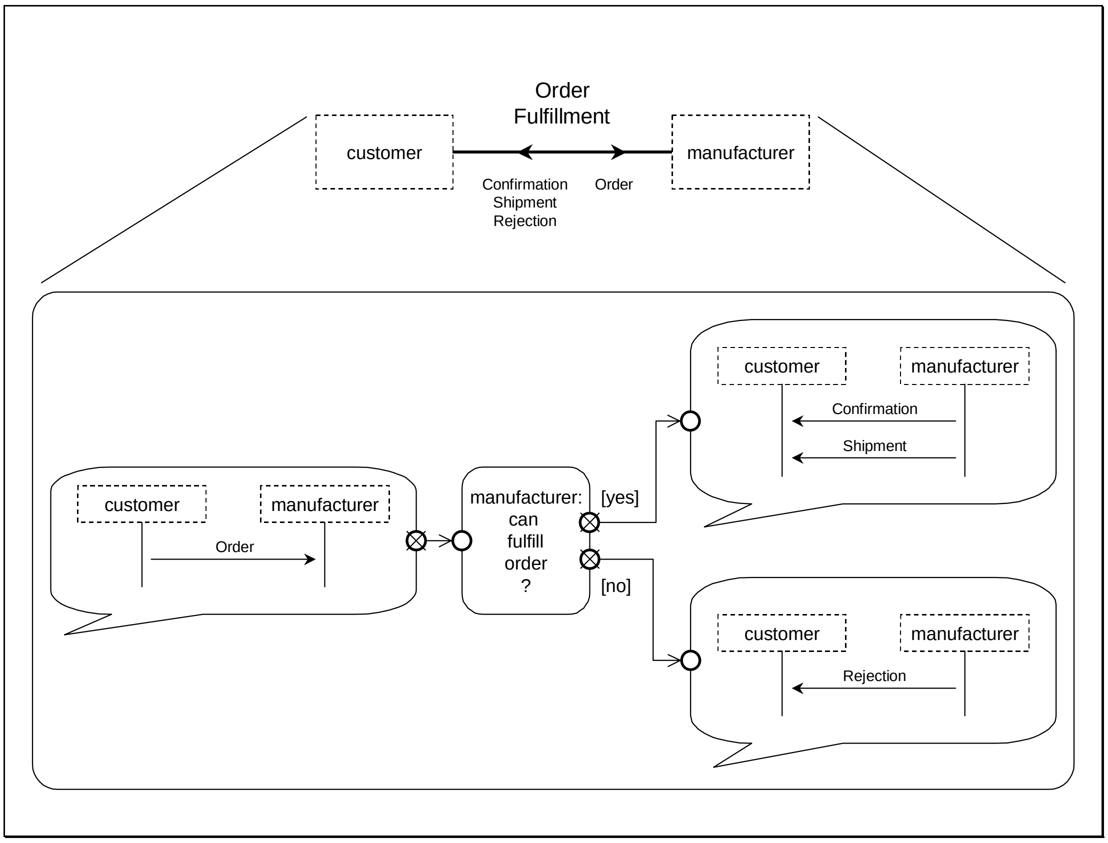

Figure 5: Collaborative process of the Order Fulfillment collaboration

The collaborative process describing the overall behavior of a set of interacting roles must be consistent with any abstract processes defined for those roles. A collaborative process for a set of interacting partner roles can be derived from the abstract processes defining their communication behavior, and vice versa.[9]

3 Operational Processes (Modeling Process Behavior)

In this chapter, we describe the fundamentals of modeling process behavior. Just as the preceding chapter on the component view of processes, this makes intensive use of concepts found in UML 2.0

3.1 Tasks

The behavioral view of a business process describes the sequencing, input, output, and resource requirements of business tasks.

Tasks can be structured (revealing subordinate steps to some level of detail) or unstructured (opaque). A structured task is sometimes referred to as an "embedded sub-process"; an example is the *procure parts* task in Figure 1. An unstructured task is

[9] We will need to formally specify this isomorphism (bi-directional mapping) between a collaborative process and a set of abstract processes defined for its partner roles ...
.

atomic from the modeler's point of view, who at some point decides to not further refine its description.[10] Tasks may be executed by humans or automated, and some may involve partner communication.

3.1.1 Task Entry Points and Exit Points

Business tasks may have several *entry points* (representing alternative ways to provide input and potentially spawn a new execution) and several *exit points* (representing alternative outcomes). In Figure 1, entry points are represented as open circles and exit points as circles with a cross.

For example, the *check if manufacturing capacity and parts available* task, which represents a three-way decision, has three possible outcomes whose conditions are indicated in parentheses. The *send rejection to customer* task has two entry points, one of which is triggered when there is insufficient manufacturing capacity, and one when some parts cannot be procured.

A task can be triggered via any one of its entry points, and upon termination provides output through any one of ts exit points.

3.1.2 Task Input and Output

We are going to refine the *Just-in-Time Manufacturing* process shown in Figure 1 by zooming into each task's entry points and exit points and specifying the types and quantities of input and output items. The resulting picture is shown in Figure 6.

The circles denoting entry points and exit points in Figure 1 have become dashed rectangles, which group smaller rectangles representing the task's actual inputs and outputs. The dashed rectangles are in one-one correspondence with the task's entry points and exit points, and are referred to as input sets and output sets.

Small rectangles drawn with thick lines represent object input or output, and the type of objects they receive or emit is denoted in blue. Small rectangles drawn with thin lines represent control input or output. Control output originating from an output set is used to signal the task outcome that this output set represents (denoted in quotes).[11]

Following [UML2], we collectively refer to entities exchanged amongst tasks as "tokens". Control inputs and outputs receive and emit control tokens, object inputs and outputs receive and emit object tokens. Control tokens are not typed and indistinguishable, object tokens are typed, carry content, and are distinguishable.

Adding token flow detail to the *Just-in-Time Manufacturing* process will result in the picture presented in Figure 6. Some of the features shown have not been discussed so far:

[10] Few tasks, if any, are really atomic. For automated tasks, machine instructions could be considered the atomic level, but even those may be realized as "micro programs". For human tasks, one could consider an action like "sign contract" atomic, but upon closer inspection this involves finding a pen, opening it, pointing it to the paper, moving it, ... Atomicity of a task always depends on the modeller's view point

[11] We will need a direction indicator (arrow ...) differentiating task inputs from outputs.

- The triangles labeled "A" are continuation symbols, providing a convenient way to show where flow continues and avoid diagram clutter. They are a notational convenience and don't introduce any additional semantics.

- The cylinders represent repositories for tasks to deposit input / output items. Please refer to section 3.3 "Repositories" for additional detail.

- Outcomes marked by small, upright triangles (for example, the "some part unavailable" output set of *procure parts*) denote exceptions or faults. When one of these is reached, compensation may be triggered for the task that ended abnormally. See section 3.5 "Exeptions and Compensation" for additional detail.

- The termination symbol (bull's eye) associated with the control output in the "some part unavailable" output set of the *procure parts* task indicates that when this output receives a token, that outcome has been reached. The output set then fires (releases any tokens), and the structured task and any subordinate task in progress are terminated. This is treated in more detail in section 3.1.4 "Structured Task Termination: Solicited and Default".

- Task inputs and outputs may have multiplicity ranges shown next to their object types (the "manufacturing capacity available, some parts must be ordered" output of the initial checking task, for example, will issue one or more part requisitions). They denote the multiplicity range of object tokens that a single execution of the task may consume or produce. The default is 1, which is also the token multiplicity for control inputs and outputs. Note that the minimum quantity may be zero, making the input or output optional.

- Individual inputs or outputs may be part of more than one input set or output set: The *create order confirmation* task, for example, has two entry points, one is triggered when manufacturing capacity and all parts are available, the other one after all missing parts have been obtained in the market. In both cases, the *create order confirmation* task will need to pull the customer order from the repository and hence that input is shared by the two input sets. Outputs that are shared amongst output sets will deliver their tokens when either outcome was reached.

- The "available" outcomes of the *procure part from preferred supplier* and *run auction* tasks are not wired to any follow-on tasks, because there is no immediate follow-on activity after a part was successfully procured (see section 3.1.4 "Structured Task Termination: Solicited and Default" for further discussion).

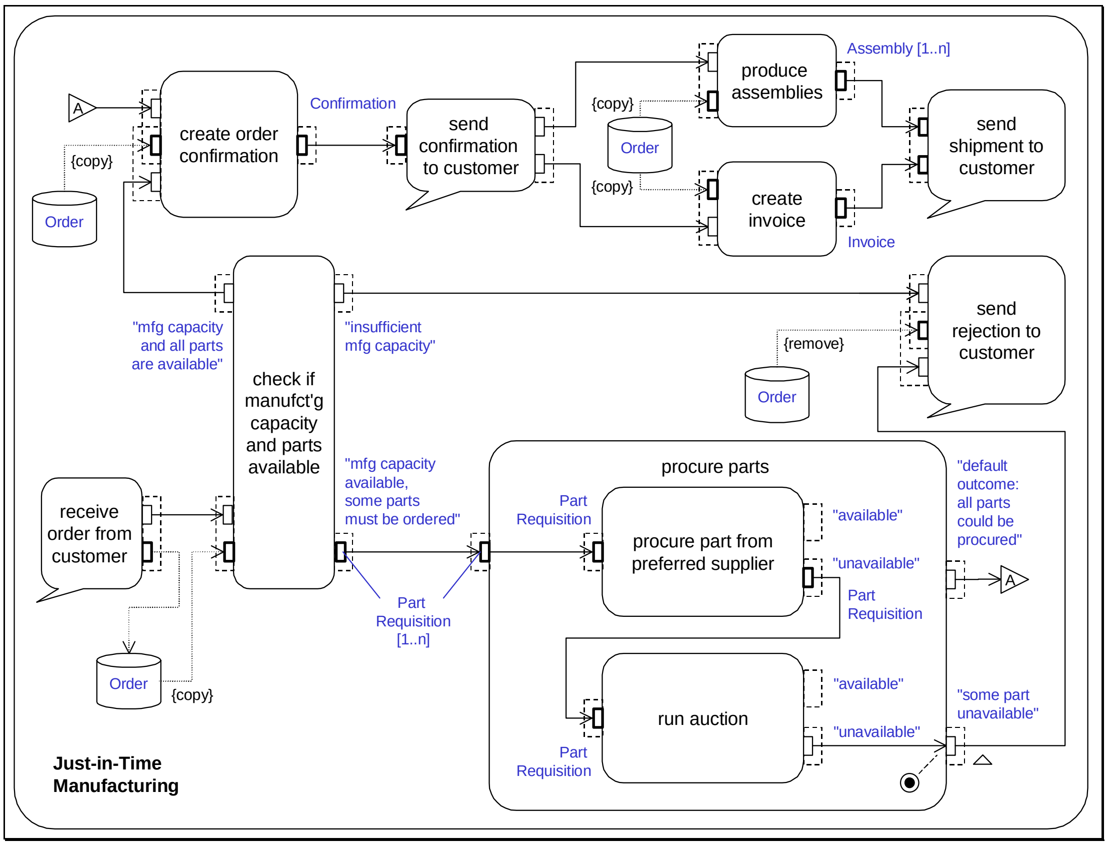
A
create order confirmation
Confirmation
send confirmation to customer
{copy}
Order
{copy}
Order
{copy}
produce assemblies
Assembly [1..n]
create invoice
Invoice
send shipment to customer
check if manufct'g capacity and parts available
"mfg capacity and all parts are available"
"insufficient mfg capacity"
send rejection to customer
{remove}
Order
"mfg capacity available, some parts must be ordered"
receive order from customer
Order
{copy}
Part Requisition [1..n]
procure parts
Part Requisition
procure part from preferred supplier
"available"
"unavailable"
Part Requisition
run auction
"available"
"unavailable"
Part Requisition
"default outcome: all parts could be procured"
A
"some part unavailable"
Just-in-Time Manufacturing

Figure 6: Just-in-Time Manufacturing process, after adding task input / output detail

3.1.3 Task Lifecycle

An execution of a task is started when one of its input sets (entry points) receives the input (tokens) it needs.

When execution ends, a task provides output at one of its output sets, which represent the task's alternative exit points or outcomes. Unstructured tasks end "whenever they are done" and the events or time by which this happens are not modeled. Modeling the termination of a structured task is described in section 3.1.4 "Structured Task Termination: Solicited and Default".

The *procure part from preferred supplier* task in Figure 6 has no multiplicity specified for its input and thus will accept part requisitions only one at a time. If more than one part is out of stock, multiple concurrent executions of this task will be started. (The *procure parts* task represents all procurement activity on behalf of a manufacturing process; it can take any number of part requisitions and is instantiated only once.)

Note that thereis a hierarchy of task executions: Several procurement tasks from preferred suppliers and several auctions may be in progress within a single *procure parts* structured task. Whenthe latter terminates, all nested procurement tasks and auctions still in progress will be terminated as well.

3.1.4 Structured Task Termination: Solicited and Default

When modeling the behavior of a structured task with several outcomes, there must be a way of indicating when a particular outcome was reached. To this end, we allow to mark a task output with a termination symbol (bull's eye). When this output is reached by a token, task execution ends and the (single) output set containing it fires: it releases a control token at each control output and any object tokens that have arrived at object outputs within this set.[12] We call this a solicited termination.

In general, each output set of a structured task must have at least one terminating output, so that one can "trigger" it via object or control flow. However, a single default output set may be defined for a structured task, which is characterized by not containing any terminating output. It represents the task's implicit or default outcome, which is considered to be reached when all token flow within the structured task and any subordinate task executions have ended.

In Figure 6 the default outcome of the *procure parts* structured task ("all parts available") thus occurs when all executions of the nested procurement tasks have ended and no token has reached the output marked "some part unavailable". If the latter happens—indicating that an auction has failed to obtain a part that was unavailable from the preferred supplier—the *procure parts* structured task and all subordinate procurement activities will be terminated. This outcome is also marked as a fault, which causes it to trigger compensation for nested procurement tasks that have successfully completed. (See chapter 3.5 "Exeptions and Compensation".)

Note that the output from the "available" outcomes of *procure part from preferred supplier* and *run auction* is not required. The default outcome of *procure parts* will fire and signal that all parts could be procured, when all procurement activity inside it has ended; wthere is no need to "collect and count" tokens reporting individual purchases.

3.2 Flow Control

The semantics of input sets and output sets can be used to model AND and OR conditions for inbound and outbound flow. Consequently, explicit flow control constructs, such as decision, merge, fork, join, can usually be avoided. In Figure 1 and Figure 6, for example, the *check if manufacturing capacity and parts available* task represents a decision, characterized by its three alternative outcomes; the *create order confirmation* and *send rejection to customer* tasks merge their inbound control flows (two entry points and two corresponding input sets are modeled), while the *send shipment to customer* task joins its two inbound flows (only one entry point / input set has been specified); note that the matching fork is provided by the output set of *send confirmation to customer*.

If needed, however, the "classic" decision, merge, fork, and join elements can be modeled explicitly using special-purpose tasks.[13] A summary of patterns for modeling flow control is presented in Figure 7. The first row shows the standard control

[12] Note that it is the task implementation's responsibility to ensure that the multiplicities of object tokens emitted fall with the "promised" multiplicity ranges. A violation represents a runtime error (the process's behavior does not meet the specification) which is handled in an implementation-dependent fashion.

[13] This approach is also supported by the observation that decisions in real-world processes require time and effort; the same is true for "forks" (for example, someone makes copies and distributes them); similarly, a join may be realized by a person tracking the completion of several concurrent tasks and triggering subsequent steps only when all are done [...] In all cases, these "control steps" take time and resources—and thus can be considered tasks.

elements using the notation of [UML2]. The second row has them represented as tasks with entry points and exit points, and the third row after their expansion into input sets and output sets.[14]

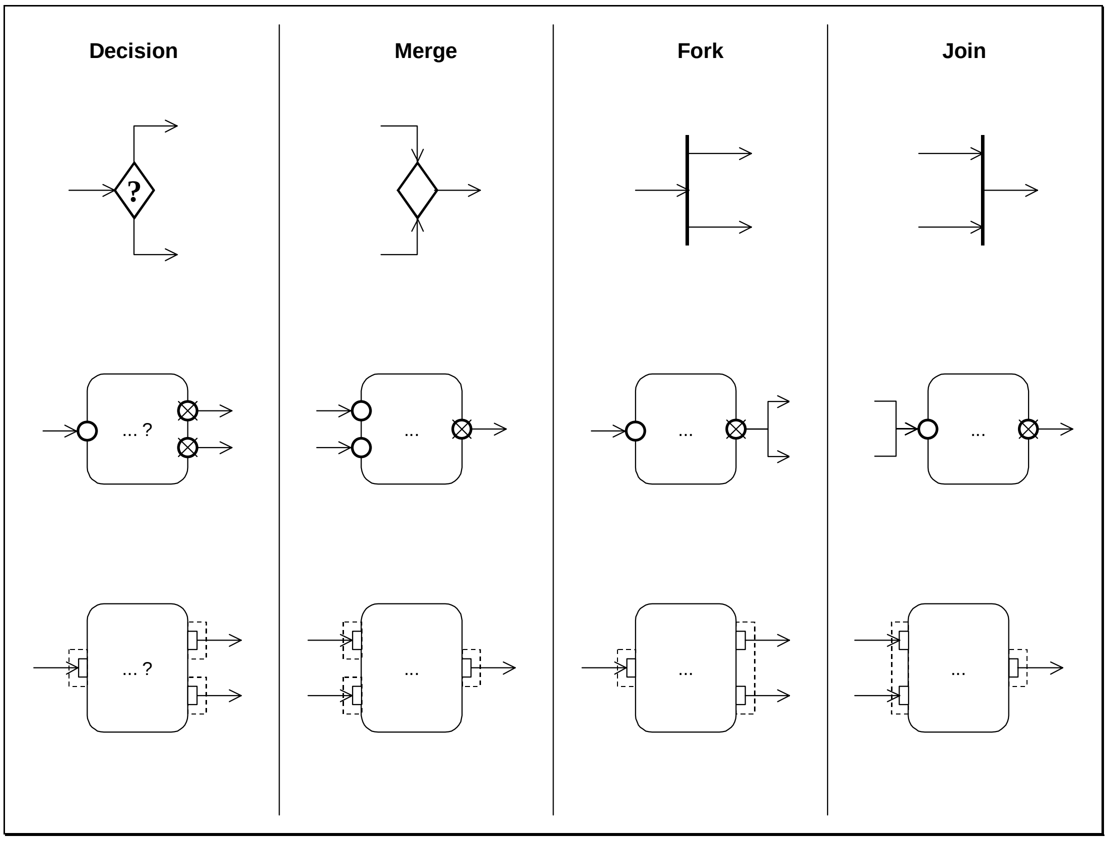

Figure 7: Summary of flow control constructs: flow chart notation, simplified task notation with entry points and exit points, and expanded task notation showing input sets and output sets

3.2.1 Loops

Loops in business processes are usually required to iterate over a set of objects. This is usually modeled by sending individual tokens representing these objects to the task processing them. (As explained in section 3.1.3 "Task Lifecycle", multiple executions of a task may run concurrently, unless they compete for a shared resource that enforces serialization.) Loops can also be constructed explicitly by "routing back" control or object flow after a decision.[15] Note that it needs to be *merged* into the flow where it rejoins (e.g., enter a task that is being re-executed via a different entry point) in order to avoid a dead-lock situation.

[14] The control elements are only shown for control flow in Figure 7. Branching, merging, forking, and joining object flow implies some constraints on the types and multiplicities of the inbound and outbound flows, which we skip in this overview paper. An example of an input set joining two object flows is shown for the *send assemblies and invoice to customer* task in Figure 6.

[15] Note that we do not restrict task interconnection in a process in any way, permitting arbitrary flow graphs. Tokens will move along the object and control flows that the model defines, tasks will be instantiated whenever one of their input sets has received its required set of tokens. It is the modeller's responsibility to ensure that this results in the intended behavior.

3.3 Repositories

In Figure 6, a repository for orders is shown at different places in the diagram (note that the cylinder symbols represent "proxies" or "access points" to a single repository). Orders are stored by the *receive order from customer* task, and retrieved at later stages in the flow. The {copy} annotation indicates that the order's content is read, but it remains in the repository for access by further tasks (a copy is made and passed out). The {remove} annotation indicates the order's removal from the repository. Task inputs connected to a repository access it ("pull tokens") when all other inputs in the same input set have received the tokens they require. The *create order confirmation* task in Figure 6, for example, will read the order content from the repository when either of its two input sets receives a control token (the object input is shared).[16]

Repositories are defined for objects of a given type and have a given capacity, which may be unlimited (the default capacity is 1). Tasks deposit tokens by routing them from an output to the repository, and retrieve them by connecting the repository to one of their inputs.

Deposit operations can either replace the previous content of the repository or add to it. Retrieve operations can either copy or remove the repository's content. The type of task inputs or outputs connected to a repository must match the type of objects stored in it, and their multiplicity range determines how many tokens are stored or retrieved at a time (task inputs retrieve at least their minimum multiplicity, and then take as many tokens as available, up to their maximum multiplicity).

Repositories whose capacity is larger than 1 always keep their tokens in an ordered collection; tokens can be added and removed at its beginning or its end, which allows to model stack or queue behavior. More selective retrieval algorithms (queries) can be modeled by adding a boolean constraint to the retrieving input set; it may involve the content of other task input that is present when token retrieval is triggered.

Repositories shown within a process (like the one for orders in Figure 6) represent temporary storage whose life time is a single process execution. They are also called process variables. Repositories can also be defined outside of any process context and then represent permanent datastores, which different process executions can use to share and asynchronously exchange items.

Clearly, if several concurrent tasks try to remove tokens from a repository they may enter in competition and race conditions may result.

3.4 Process Life-Cycle and Inter-Process Communication

An execution of a process is started when a "receive" task that has no incoming flow is targeted by inbound communication, and the incoming request is not correlated with an existing process execution. Correlation is discussed in section 3.4.2 "Addressing Process Executions (Correlation)". Note that such "initial" receive tasks are not permitted within structured tasks, which must be initiated via token flow (spawning a sub-flow within a process that has not even started does not make sense).

[16] Replicating the object input in two (disjoint) input sets instead of sharing it would have had the same effect, but would have resulted in additional diagram clutter as well as the need to merge the object flows emerging from these two inputs if one ever wanted to model the internal behavior of the *create order confirmation* task.

A process execution is terminated either explicitly when a task is reached that is marked as "terminating" (typically, this task is a final send returning a result), or implicitly after all token flow and executions of nested tasks have ended. Note that this is analogous to the termination of a structured task, explained in section 3.1.4 "Structured Task Termination: Solicited and Default" above. Just like initial receive tasks, terminating tasks must not be nested within structured tasks.

The *receive order from customer* task in Figure 1 and Figure 6 is an example of an initial receive task. The *Just-in-Time Manufacturing* process does not contain a terminating task, but ends implicitly after execution of all subordinate tasks has finished.

The next section describes how requests are routed to a process component, which acts like a "factory" and request broker for its executions.

3.4.1 Partner Selection (Routing)

Addressing a particular process execution conceptually comprises two steps:

First, a modeled process can be implemented by different partners (or in different locations): the *Just-in-Time Manufacturing* process, for example, could be implemented by two companies, cheapAssemblies.com and quickAssemblies.com. Addressing partners or locations is accomplished via request routing.

Then, at any given time, cheapAsemblies.com and quickAssemblies.com may have several executions of the process running, executing different customer orders. Addressing these is accomplished via correlation.

Addressing a process location is discussed in this section; correlation is explained in section 3.4.2 "Addressing Process Executions (Correlation)" below.

Consider Figure 8. It shows the *Just-in-Time Manufacturing* process component with a static attribute companyLocator. The component is subclassed by components representing the installed processes at cheapAssemblies.com and quickAssemblies.com, which give specific values for their company locators. Note that these process components' behaviors may vary as well, but must conform to the behavior of the parent component. [17] In particular, they must implement the abstract processes of all collaboration roles which the parent component declares to play.

[17] A precise definition of what it means for a process to "conform" to another process is needed. While this is lacking, "conforms to" can be interpreted as "is identical" for process flows.

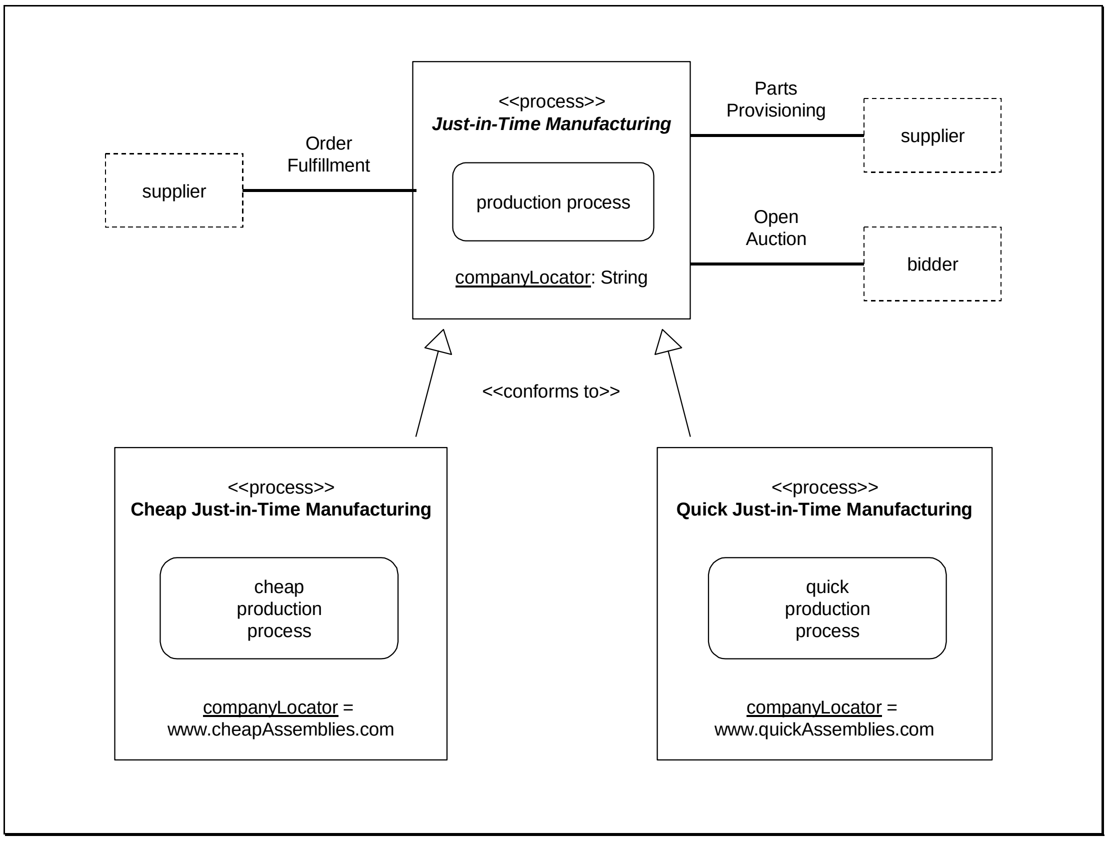

Figure 8: Process components conforming to a common teamplate (parent component)

The static attribute values can be used to identify (or locate) a process component representing a specific partner. In general, we support four possible options to model request routing or partner selection:

- **ignore at modeling time** — the runtime environment will "somehow know", there is only one possible target, or the specific partners involved are irrelevant for the modelling task being carried out

- **hard-wire in the model** — by wiring components whose identity (locator, location, ...) is known at modeling time

- **define selection criteria in the model** — by specifying a "selector algorithm" which, based on information known to the requester or implicit in the environment, calculates a set of (static) attribute values identifying a specific partner. Using an explicit component reference that was provided as input (for example, to route an asynchronous response) is a special case of this.

- **publish-subscribe** — by wiring a "distributor" partner role of the publishing and subscribing processes to a topic component acting in this role

An illustration of the publish-subscribe scenario is shown in Figure 9. The two collaborations News Publishing and News Subscription define the interaction between the publisher and distributor, and distributor and subscriber, respectively. The multiplicities indicated at the connector ends indicate that more than one publisher or subscriber can be connected to the same distributor. Process components implementing the publisher and subscriber roles, and a topic component playing the distributor role are shown in the right half of the figure. Note that the instantiations of the topic component represent "publications".

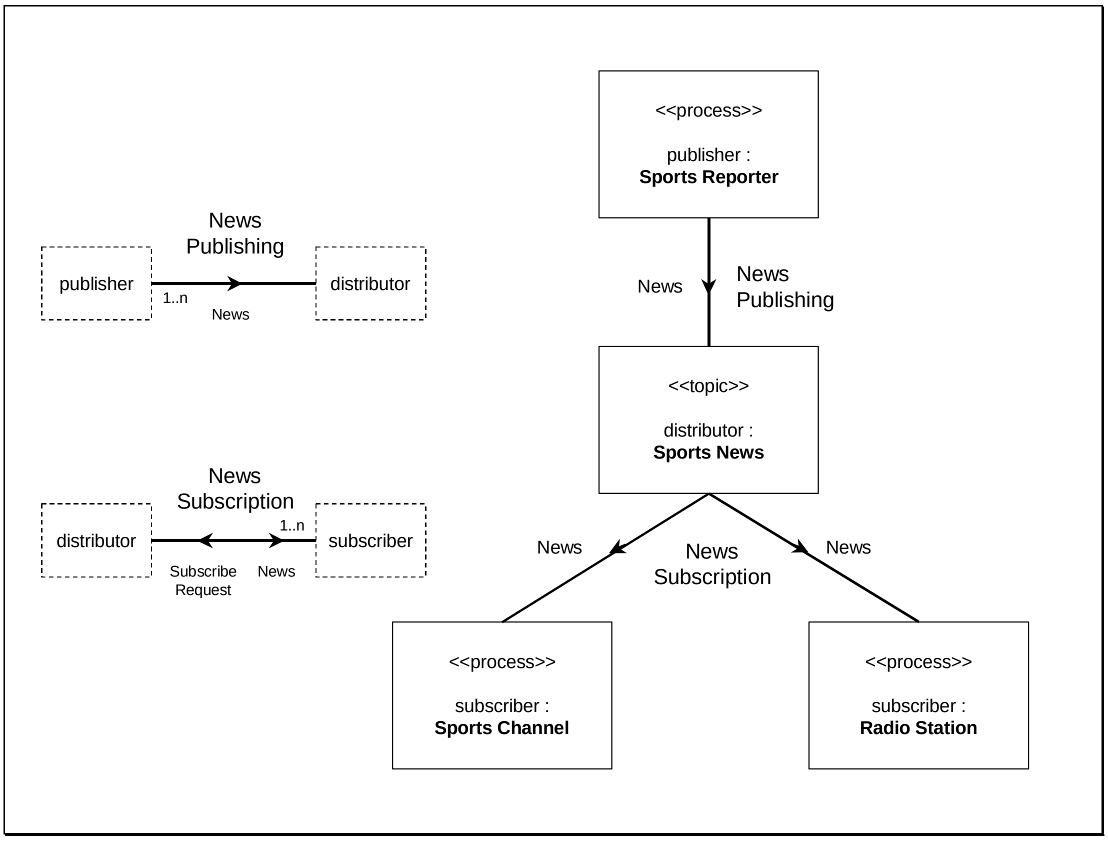

Figure 9: Puslish - subscribe routing

3.4.2 Addressing Process Executions (Correlation)

Implicit correlation is the direction of a "response" to a receive task within the process execution that sent the original request. Explicit correlation is the routing of an unsolicited request to one or more ongoing executions of a process.

We discuss implicit correlation first. With the component interaction model introduced in chapter 2 "Process Components", where processes exchange unidirectional signals whose sequencing is goverened by a collaborative process (protocol), request-response correlation must be generalized: The first exchange between two process executions that establishes a collaboration (it must be an initial request according to the collaboration's protocol) implicitly "binds" two executions threads to the collaboration instance. They may be represented by executions of the process proper or nested sub-processes. Further execution threads, of potentially other partners, may join as the collaboration proceeds. All messages issued by these executions in accordance with the collaboration protocol will be routed to the "partner" executions bound in this way, until the collaborative process has ended.[18] [19] Clearly, an instance of an operational process may participate in several collaborations at the same time.[20]

To explain explicit correlation, we revisit the *Just-in-Time Manufacturing* process. Note that multiple open auctions may be running concurrently on behalf of a single manufacturing process. They represent ongoing negotiations for different parts, which bidders must be able to address individually.

Therefore, the actual auction should be modeled as an independent process that is *controlled* by the manufacturing process on whose behalf it was launched. This is shown in Figure 10, where we assume that the manufacturing process plays the requester role (a refinement of *run auction* would show the pertinent communication tasks).

In the Procurement Auction operational process, in the lower half of Figure 10, only the communication tasks are shown. The three receiver tasks have the following correlation predicates:

- The *receive part requisition from requester* task has no correlation predicate and is the receiver of the initial request in the Auction Control collaboration protocol. Input received here will spawn a new Pocurement Auction execution.

- The *receive bid from bidder* task has a correlation predicate that matches the part number of an incoming bid with the one of the requisition that initially spawned the auction. Incoming bids will thus be routed to the auction for that part.

- The *receive termination signal from requester* task matches an order number supplied as part of the termination signal with the order number of the requisition. The termination signal thus targets all auctions that are in progress procuring parts for a particular order.

[18] Design issue: In order to enable protocol tracking, a send task needs to specify more information than the partner and type of signal exchanged ("send X to Y") because the same signal may be permitted in different branches of a collaborative process, and without additional information it will be impossible to tell which branch was taken—which undermines protocol tracking.

[19] We may be able to avoid "reply tokens" by agreeing that all communication task instances that are spawned by these execution threads and map to an abstract process of an ongoing collaboration (that is, all potential past and future senders / receivers participating in the conversation) remain bound to the message exchange prescribed the collaboration protocol, and blocked for any other communication, until the collaboration instance is finished.

[20] Work is needed to precisely define the mapping of "execution threads" in interacting operational processes to a collaboration instance; ultimately, we must ensure that the question "which *instance* of a receive task will get a particular *instance* of an inbound signal" can always be answered. One can easily conceive process models where this is all but clear (e.g. by spawning several receive tasks after sending).

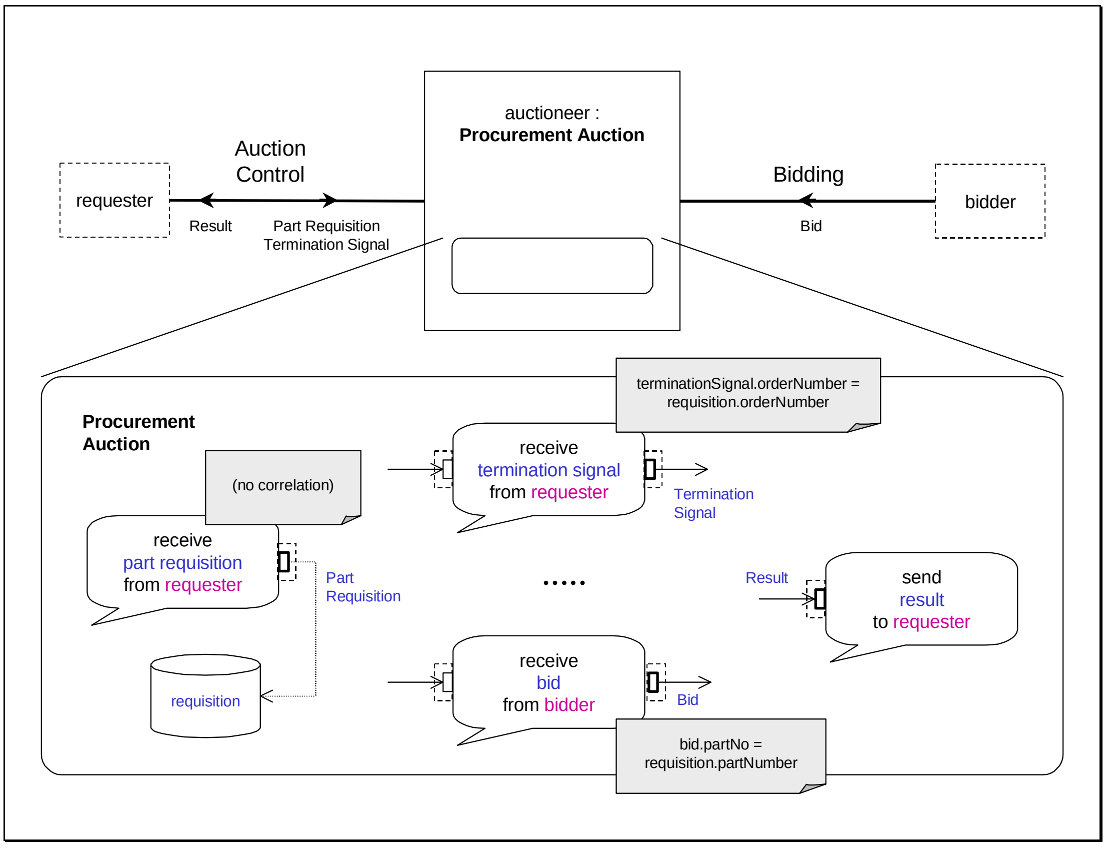

Figure 10: Correlation example

In general, a correlation predicate may be associated with individual receive tasks (compare Figure 10). Whenever such a task becomes the target of an inbound request, the predicate is evaluated for all of its existing instantiations. If it evaluates to true for exactly one of them, then this task instance will get the inbound request. Otherwise, the user-defined settings for "no correlation matches" and "multiple correlation matches" determine what happens, as specified in the following table:

Attribute	Setting	Effect
no correlation matches (applies if there is no execution for which the correlation predicate evaluates to true)	create new instance	a new process execution is started and receives the request (not for receive tasks defined within a nested sub-process)
	raise exception	a "no correlation matches" exception is raised
	ignore	the request is ignored
multiple correlation matches (applies if there is more than one execution for which the correlation predicate evaluates to true)	deliver to all	the request is delivered to all process executions for which the correlation predicate is true
	deliver to any	the request is delivered to one (arbitrarily chosen) process execution for which the correlation predicate is true
	raise exception	a "multiple correlation matches" exception is raised
	ignore	the request is ignored

In the example in Figure 10, the no correlation matches setting is "ignore" for the task receiving the termination signals, and "raise exception" for the task receiving the bids. The reason is that termination signals can safely be ignored if no auction is found to be terminated, while invalid bids should cause an exception notification back to the bidder. The multiple correlation matches settings for the two tasks are "deliver to all" and "raise exception", respectively.

3.5 Exeptions and Compensation

3.5.1 Exceptions

A task can have one or more exception output sets. Completion via an exception output set indicates an undesired or "fault" outcome: the task did not complete normally and has not successfully performed its work. The "some part unavailable" outcome of the procure parts task in Figure 6 is an example.

Declaring an output set an exception can be purely informational. When used in conjuncation with compensation, however, additional semantics apply, as discussed in the following section.

3.5.2 Compensation

It sometimes becomes necessary to compensate work that has already been performed. This may be due to a failure of a subsequent piece of work, or because the objective to which this work contributed is now kown to be not achievable. In some cases this compensation will be an undo, in other cases it will be a repair task (for example, a letter cannot be unsent so a correction letter must be sent).

We refer to a task that can be compensated as a primary task. A primary task can have a compensation task that is installed on successful completion of the primary task (a primary task is permitted to have multiple compensation tasks, one per non-exception output set, that is, one for each of its non-exception outcomes).

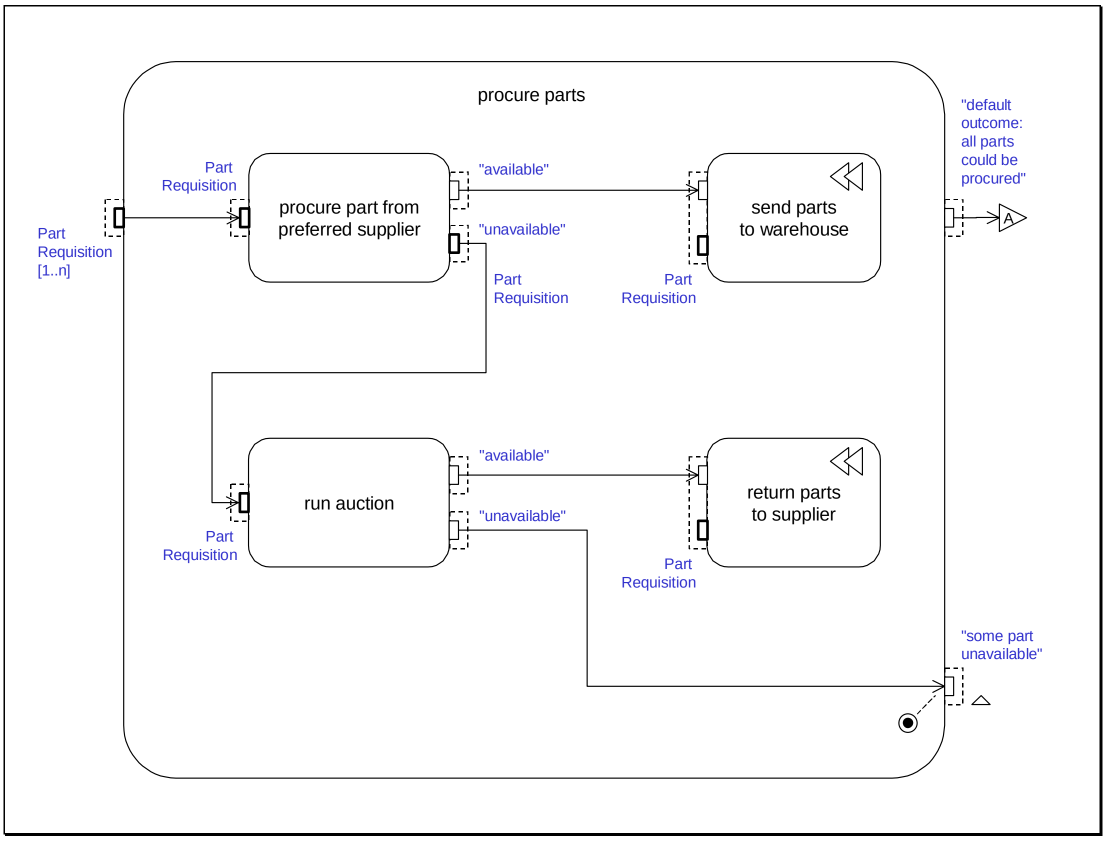

Figure 11: Compensation example

Figure 11 shows an extended view of the *procure parts* task of the *Just-in-Time Manufacturing* example (it is at the same refinement level and could be considered an extension of Figure 6). When it fails with the "some part unavailable" outcome, the procurement sub-tasks that have completed successfully should be compensated as follows: parts bought from preferred suppliers should be sent to a warehouse; parts bought in the open market should be returned to their suppliers.

Two compensation tasks (marked by "rewind" symbols) are added to achieve this. They receive control flow when the tasks they compensate have completed successfully. However, this flow does not trigger their execution but only "install" them: thus, there will be a *send parts to warehouse* task installed for each successful execution of *procure part from preferred supplier*, and a *return parts to supplier* task for each successful execution of *run auction*.

When an execution of *procure parts* task ends abnormally, all compensation tasks installed in it will be run, in reverse order of their installation.

Compensation tasks have access to "snapshot" copies of the (input) object tokens that triggered the task they compensate, as well as any (output) object tokens from the output set that delivered their install token. They also have access to variables within their context, whose values are "frozen" at the time the primary task completes. Snapshots of the part requisitions that triggered the individual procurement tasks are thus provided to the compensation tasks in Figure 11. Note that multiple executions of a primary task lead to multiple installations of the compensation task, each with its own associated "snapshot" data.

Once compensation has been completed, flow will continue from the exception output that triggered it (the "some part unavailable" output set in the example).

Compensation is propagated to nested scopes: If a structured task is to be compensated, then any nested tasks that have compensation tasks installed are compensated first before the task's own compensation is run.

4 Modelling Resources

4.1 Resources

The most straightforward description of a business task is a sentence: Jill approves the order. John delivers the package. etc. These sentences have subjects (Jill, John), verbs (approves, delivers), and objects (the order, the package). In business process definitions, we use tasks to describe the "verbs", items to describe the "objects", and resources to describe the "subjects" or actors. In addition, actors sometimes need collaborators or tools to accomplish a task, which are also modelled as resources.

We differentiate between individual resources, which are discernible and have an identity, and bulk resources, which are provided and used in some quantity. A human worker or employee is a special kind of individual resource, other kinds would be a truck, a sledgehammer, a slide projector, a conference room, or a computer system. Examples for bulk resources would be fuel, floor space, budget, cement, or memory.

Resources, both individual and bulk, are typed (have user-definable attributes) but have three fundamental properties in common:

- qualifications—the set of roles they can fulfill
- availability—the set of time slots during which they can be deployed
- cost—the cost of using them, which may depend on usage time and quantity (for bulk resources)

In business process models we allow modelling resource types (e.g., Employee) as well as instances (e.g., John Doe).

4.2 Roles

A role describes a functional capability, which tasks require and resources provide. A rescue task, for example, may require someone in the role of a pilot, something in the role of an "aircraft", and someone in the role of a physician; different people or items may be qualified to fulfill these roles.

Oftentimes roles are further qualified by properties of the task at hand: a pilot may have to be licensed for a specific type of airplane, an expense approver role be further qualified by the amount to be approved and the type of expense request. We use the notion of "role scope" to describe such instance-dependent restrictions or "down-scoping" of a fundamental qualification which a role represents. Tasks requiring a role may define a required scope (pilot to fly a B-59 airplane; approver for a $21,000 group travel request, etc.), and resources declare the scope for roles they can play (airplanes they are trained to fly; expense requests they are authorized to approve, etc.).

Roles are different from resource types, although the two are often used interchangeably in everyday language: a "team leader" role may be fulfilled by a corporate employee, a contractor, or a summer student. Each of these resources may be qualified for the role, but they have different resource types (some have employee serial numbers, other don't ...)

4.3 Resource Requirements

Tasks have three ways to specify their resource requirements:

- require a specific resource
  examples: Dr. William H Smith; the manager of the request submitter

- require a resource of a specific type, potentially subject to constraints
  examples: an employee of dept D27X; a conference room that can seat 20

- require a role to be fulfilled, potentially specify scope values
  examples: a pilot for a B-59 airplane; an approver for a $21k travel expense

Furthermore, resource requirement specifications sometimes need to refer to resources used in a preceding task ("... the same user that performed this preceding step"). This is accomplished by modelling special-purpose task outputs, which emit object tokens representing the resources employed after task execution has finished. Subsequent tasks may use the information carried by these tokens to formulate their resource requirements in terms of the resources used by their predecessors.

5 References

[BPDM] Business Process Definition Metamodel - Request for Proposal; OMG Document bei/2003-01-03; http://www.omg.org/cgi-bin/doc?bei/2003-01-03

[UML2] Unified Modelling Language: Superstructure, version 2.0, Final Adopted Specification, c/03-08-02

[BPEL] Business Process Execution Language for Web Services, Version 1.1, 5 May 2003, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/