Service-Oriented Domain and Business Process Modelling

Gianna Reggio, Maurizio Leotta, Diego Clerissi, Filippo Ricca

Abstract:

We present Precise Service-Oriented Modelling (Precise SOM) – a novel lightweight method for integrated domain and business process modelling, which follows the service-oriented paradigm, and uses a UML profile as notation – and a detailed workflow to guide the production of the models. In our method the produced UML models are precisely defined by means of a metamodel, a set of constraints, and a limited set of UML constructs to help modellers to avoid common mistakes and to guarantee, by construction, a good quality. Precise SOM has been successfully used in an industry-academic project concerning the modelling of a big harbour.

Digital Object Identifier (DOI):

http://dx.doi.org/10.1145/3019612.3019621

Service-Oriented Domain and Business Process Modelling

Gianna Reggio, Maurizio Leotta, Diego Clerissi, Filippo Ricca
Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università di Genova, Italy
{gianna.reggio, maurizio.leotta, diego.clerissi, filippo.ricca}@unige.it

ABSTRACT

We present Precise Service-Oriented Modelling (Precise SOM) – a novel lightweight method for integrated domain and business process modelling, which follows the service-oriented paradigm, and uses a UML profile as notation – and a detailed workflow to guide the production of the models. In our method the produced UML models are precisely defined by means of a metamodel, a set of constraints, and a limited set of UML constructs to help modellers to avoid common mistakes and to guarantee, by construction, a good quality. Precise SOM has been successfully used in an industry-academic project concerning the modelling of a big harbour.

CCS Concepts: Applied computing $\rightarrow$ Business process modeling

Keywords: Service-Oriented, Precise Model, Method, UML, Domain

1. INTRODUCTION

Domain and *Business Process* models represent real-world concepts (entities involved in the business and business rules/processes) pertinent to the domain that needs to be modelled. They are often used in the initial phases of traditional software development as starting point for building software systems. Domain and business process modelling are nowadays well-established, and are also basic ingredients of *enterprise/business modelling* [9, 12]. Many different methods and notations have been proposed to support these activities [9], such as object-oriented approaches based on UML [22] class diagrams, and entity-relationships models for domain modelling; as well as, several proposals based on BPMN [23, 5] or UML activity diagrams [21, 15, 3, 19] for business process modelling.

Based on our experience in the context of domain and business process modelling, including also the participation to several joint industry-academic projects (e.g. a current one requiring to model a big harbour), we selected a set of *features* that we consider extremely relevant for a method having the goal of modelling both domains and business processes, and that can be successfully applied in industrial projects. Such a method should: (1) be integrated: both dynamic and static aspects must be modelled and cross-checked/synchronized; (2) lead to the creation of high-quality models by construction; (3) be fairly simple to apply; (4) provide a support for simulation and analysis; (5) avoid proprietary solutions; and (6) adopt a visual notation. To the best of our knowledge, no method in the literature seems to have all these features.

In our view, domains are composed of active and possibly autonomous dynamic entities interacting among them, and business processes are flows of events in which these dynamic entities take part. The need to model complex domains and processes led us to consider the *service paradigm* [7], since the concept of service offers a suitable way to modularize how the various entities composing a domain interact among them, and consequently to simplify business processes modelling [2]. We discarded the *object-oriented paradigm*, since it forces to express the interactions in terms of method calls, and as result the models contain classes with long lists of methods, that have to be structured using additional constructs, such as interfaces; furthermore, object-oriented classes do not offer a native way to model which methods can be called by their instances. Moreover, the fact that services may be provided or used allows to easily represent which are the possible interactions offered and which are those required. Services are also, in our opinion, better than *plain messages exchange*, again because they offer a natural way to structure complex interactions among dynamic entities.

In this paper, we present the result of our investigations and experiences, Precise SOM (Precise Service-Oriented Modelling), a method for the conceptual modelling of domains and related business processes using a specific UML profile. The *main novelty* of Precise SOM is that it has been conceived to support all the features mentioned before. Precise SOM is the result of previous works of the authors in the field of modelling using the UML [10, 4]. We developed and refined it working on a real industrial case study [1], where the domain is a harbour consisting of ships, people working on them, transporters of various kinds, officials, documents, and containers.

We have opted for the UML because it is widely known and used [20, 18], and provides many different diagrams suitable to cover all aspects of a domain (in contrast, e.g. to BPMN), including its business processes; furthermore, by means of the profile mechanism, UML provides a way to define its own variants.

For us, *precise* means that the form of the acceptable UML models is carefully defined by a metamodel, by a set of well-formedness constraints, and by restricting UML to only essential language constructs. We have found that precisely defining the models helps to avoid mistakes, and to improve their quality.

By following Precise SOM and the associated detailed workflow, the modeller can easily model all the aspects of a domain and obtain consistent and precise models of the associated business processes.

SAC 2017, April 03-07, 2017, Marrakech, Morocco
Copyright 2017 ACM 978-1-4503-4486-9/17/04…$15.00
http://dx.doi.org/10.1145/3019612.3019621

Precise SOM models are capable of representing both (1) the static/structural aspects of a domain as well as (2) the dynamic/behavioural ones in terms of services, and moreover they are also equipped with features to cover business process modelling (again in terms of services), having a good quality level by construction.

Precise SOM can be applied to any kind of domains and its models are particularly suited to be the starting point of the development of (software) systems following the service-oriented paradigm. Furthermore, the resulting models, being expressed with the UML, can be transformed into different notations by using model-driven tools, for example: executable code, inputs for simulation tools, and specification languages for formal analyses.

Precise SOM domain models can be employed in different activities including: (1) representing the know-how on a domain for getting a shared and mutually agreed starting point among the stakeholders before starting the development of some supporting HW/SW systems (also not service-oriented); (2) analysing and re-engineering the domain; (3) simulating/analysing the domain, either manually or by means of software tools/formal approaches, with the aim of getting a deeper understanding of the domain and improving it.

The paper is structured as follows: Sect. 2 introduces our service-oriented view of the domains, and Sect. 3 presents the form of the Precise SOM models and the workflow driving their productions. Then, Sect. 4 describes the related work, while conclusions and future work are drawn in Sect. 5.

Running Example: Dealer Network – Since the previously mentioned harbour case study is too complex to be presented in this paper, Precise SOM will be illustrated using a simple but meaningful case study, the Dealer Network, inspired by a similar example used in the SoaML Specification [14]. The Dealer Network is a business community including three primary parties: the dealers, the manufacturers, and the shippers. They are independent parties but they want to work together; manufacturers sell a product (e.g. gasoline) to the dealers, while the shippers deliver the product from the manufacturers to the dealers. The interactions among the parties of this business community are described as follows. A dealer wants to buy a quantity of product from a manufacturer. It may ask for the current price of the product before sending an order. When the manufacturer receives the order, it checks the product availability. If the product is available, the manufacturer sends a confirmation to the buyer, and asks a shipper to deliver the bought product to the dealer. When the shipper receives a request from a manufacturer, it informs the manufacturer of the date for picking up the shipment. Once the shipment is delivered to the dealer, the shipper sends a confirmation to the manufacturer. The dealers may check the status of the placed orders, and the manufacturers may check the status of the required shipments.

2. SERVICE-ORIENTED DOMAINS

A key ingredient of any modelling method is the definition of the collection of items that are going to be modelled [16]. The modelled items of Precise SOM are structured dynamic domains, composed by entities interacting following the service-oriented paradigm, that in the following of the paper we call *service-oriented domains*.

We follow the Oasis definition of *service* and *participant* [13], backed also by the OMG in the SoaML specification [14]: "*Service is defined as the delivery of value to another party, enabled by one or more capabilities. Here, the access to the service is provided using a prescribed contract and is exercised consistent with constraints and policies as specified by the service contract. A service is provided by a participant acting as the provider of the service for use by others. The eventual users of the service may not be known to the service provider and may demonstrate uses of the service beyond the scope originally conceived by the provider. As mentioned earlier, provider and user entities may be people, organizations, technology components or systems - we call these Participants.*"

Moreover, in our view, the *participants* may manipulate many different passive entities (that we name *objects*), either by means of internal actions (i.e. carried out directly by the participant) or through the services (i.e. carried out by other participants). We have preferred the term *objects* to data, since they are not always pure values but may have also a mutable state.

The value delivered by a service provided by a participant usually may involve entities different from the participant itself; for example, a participant providing many services for obtaining the public data about the students of a university will have to interact with the informative system of the university. We name the entities that a participant may observe and act on as *realms*, and say that it *reigns* on them. The realms are autonomous, and thus they may change their status not only as a consequence of a participant act. Note that the realms of a participant are not a subpart of the participant itself, and various participants may reign on the same realm; for example, a participant providing the services for enrolling to the courses and for recording the students marks will also reign on the university informative system.

The Dealer Network can be described as a service-oriented domain, whose participants are the dealers, the manufacturers, and the shippers. They will manipulate many different passive entities (i.e. objects), for example the orders, the quotes, and the shipping requests. The Dealer Network participants will interact using four services: ManageOrder, GetOrderStatus, RequestShipping and GetShipmentStatus. In detail, the dealers use the services provided by the manufacturers:

- ManageOrder to request a quotation and then place an order for what they want to buy from the manufacturers. In the former case, the service supplies the dealers with the quote of the product. In the latter case, the dealers will receive further information about the order, such as the order id and status.
- GetOrderStatus to get information about the status of the placed order (identified by the order id).

While the manufacturers use the services provided by shippers:

- RequestShipping to have the product relative to an order delivered to the dealer; the shipper informs the manufacturers of the picking and delivery dates. Then, the manufacturers will receive the delivery confirmations when the dealers receive the product.
- GetShipmentStatus to get information about the status of a shipment for the product ordered by a dealer (identified by the waybill number that they received together with the pickup info).

The realms of the manufacturers are their plants (where the industrial/manufacturing activities take place), whereas the realms of the shippers are their informative systems and their truck fleets. Notice that a participant is not obliged to reign on some realms, as in the case of the dealer, which also does not provide any service.

A participant P may be either *monolithic*, as in the case of a human being, or *structured*, whether it can be described as a set of (sub-)participants interacting among them by means of provided and used services. Thus, a *service architecture* presents how the sub-participants of P provide and use their services to allow P to provide its services, taking also advantage of the services used by P itself. A structured participant will reign on the realms of its

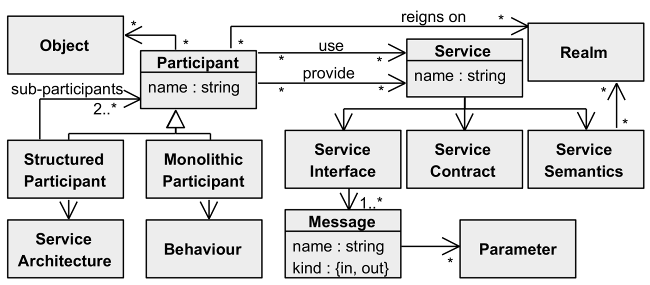


Figure 1: Service-Oriented Domains: Conceptual Model

sub-participants. A *domain* is a "closed" structured participant, i.e. it neither offers nor requires any service. Since the Dealer Network is structured and does not provide and use services, it can be seen as a domain.

As suggested by [16], the above informal introduction of the domains considered by Precise SOM is summarized and systematically presented by means of a conceptual model shown in Fig. 1 (for simplicity, we omit the association end multiplicities equal to 1, thus a service is linked exactly with one service contract and vice versa).

A *participant* is characterized by:

- a name;
- the objects that it (and its sub-participants, if any) may manipulate;
- the realms on which it reigns;
- the services that it *provides* (also none) or *uses* (also none);
- if monolithic, a *behaviour*, that will be based on sending and receiving the messages of the used and provided services, together with internal actions that may concern its realms or objects;
- if structured, a set of sub-participants, and a *service architecture*. Note that the behaviour of a structured participant is completely determined by its sub-participants and by its service architecture.

A *service* is characterized by an interface, contract, and a semantics.

The *service interface* contains the static information needed to access the service. A service interface is conceptually seen as a set of "in" and "out" messages, where each message is characterized by a name and a list of typed parameters; the "in-messages" are used to require the service capabilities to a service provider, and the "out-messages", if any, are used to answer such requests.

The *service contract* focuses on the protocol between the provider and the user of the service. The service contract specifies which are the allowed sequences of interactions between who uses the service and who provides it.

The *service semantics* describes the service essence. We assume that a service is able to act on some realms (also none) i.e. it may modify such realms as the result to having received an in-message from a service user, and its answers who uses the service (out-messages) may depend on the current status of such realms. Thus, the semantics of a service will be described by stating how the reactions to the in-messages depend on the previous received messages and on the realms statuses, and such reactions consist in acting on the realms and in sending out-messages.

3. Precise SOM MODELS

Precise SOM uses a UML profile as modelling notation. Thus, its models are UML models having the specific form presented in Fig. 2 by means of a metamodel, and a set of well-formedness constraints (for lack of room only a few are reported in Fig. 5, the others can be found in the technical report available in [1]).

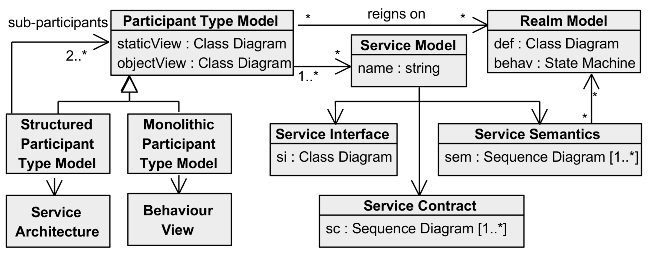


Figure 2: Precise SOM Metamodel

The structure of the Precise SOM models strictly mimics the structure of the domains presented in the previous section and summarized in Fig. 1. Indeed, Precise SOM provides two different kinds of models for structured and monolithic participants; both offer the possibility to model the services used and provided by a participant; moreover, the models of the structured participants include those of the sub-participants, and how they are organized into a service architecture, whereas the models of the monolithic participants contain their behaviours.

Notice that here we are actually speaking of participant *types* instead of specific participants, since very frequently many participants have a common structure and behaviour (i.e. they have the same type). So, when we say "participant P", we actually mean its type.

A Precise SOM model of a participant P is decomposed in several views listed below:

- *object view*: the definitions of the objects used by P;
- *realm view*: the collection of the models of the realms (Realm Model in Fig. 2) on which P reigns;
- *service view*: the collection of the models of the services (Service Model in Fig. 2) provided and used by P;
- *static view*: the class defining the P type and any class and datatype needed to define it, to show which services are provided and used by P, on which realms P reigns, and, in case P is monolithic, its internal actions and attributes.

If P is structured, a *service architecture* showing the sub-participants of P and their respective connections will be depicted together with the static view. If P is monolithic, then the model includes the *behaviour view* defining the behaviour of the P instances by means of activity or state machine diagrams.

3.1 Object View

The object view describes the types of the objects manipulated by a participant. Technically, the *object view* is a UML class diagram containing classes stereotyped by ≪object≫, plus any other class and datatype needed to define them. The operations of such classes/ datatypes may be defined either by methods or pre-post conditions.

In Dealer Network there are seven types of objects, modelled by the corresponding classes stereotyped by ≪object≫, as shown in

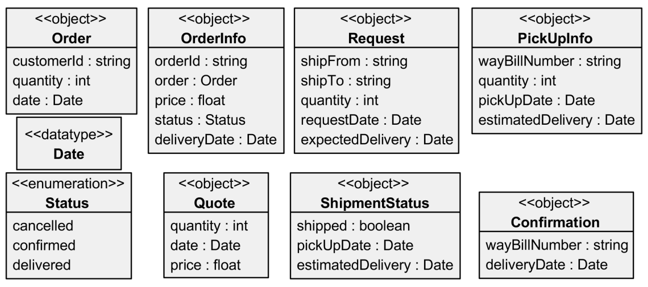
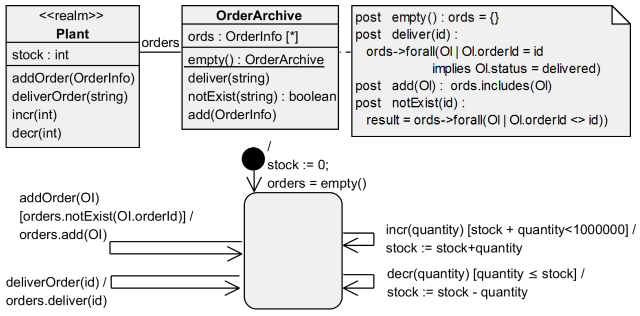


Figure 3: Dealer Network Model: Object View



<<realm>>
Plant
stock : int
addOrder(OrderInfo)
deliverOrder(string)
incr(int)
decr(int)
orders
OrderArchive
ords : OrderInfo [*]
empty() : OrderArchive
deliver(string)
notExist(string) : boolean
add(OrderInfo)
post empty() : ords = {}
post deliver(id) :
ords->forall(OI | OI.orderId = id implies OI.status = delivered)
post add(OI) : ords.includes(OI)
post notExist(id) :
result = ords->forall(OI | OI.orderId <> id))
/ stock := 0; orders = empty()
addOrder(OI) [orders.notExist(OI.orderId)] / orders.add(OI)
deliverOrder(id) / orders.deliver(id)
incr(quantity) [stock + quantity<1000000] / stock := stock+quantity
decr(quantity) [quantity ≤ stock] / stock := stock - quantity

Figure 4: Plant: Realm Model

Fig. 3. Date and Status are datatypes used to define some of the object classes.

3.2 Realm View

The *realm view* is the collection of the models of the realms under the control of a participant. Each realm is modelled by a class diagram including a class named as the realm itself and stereotyped by ≪realm≫, and all other classes and datatypes needed to define it. The realm is usually a dynamic entity exhibiting an autonomous behaviour, that will be modelled by means of a state machine having as context the realm class. A realm model is an abstract view of that entity taking into account only the features related with the participants acting on it. The visible operations of a realm class represent the possible activities of the realm itself, and will be used to define the events of the associated state machine.

The Dealer Network reigns on three (types of) realms: the manufacturers' plants, the shippers' information systems and truck fleets. Fig. 4 presents the model of the Plant realm, which is characterized by the quantity of product in stock (modelled by the attribute stock), and by the order archive (modelled by the association orders with the class orderArchive). The operations of orderArchive are defined by pre-post conditions (see the note). The state machine models the autonomous behaviour of the plant: the stock may be freely increased/decreased (it will be never negative nor bigger than 1000000), and the orders may be added and later delivered.

3.3 Service View

The *service view* contains the models of all the services provided and used by a participant. A *service model*, see Fig. 2, consists of a name, an interface, a contract, and a semantics. We use the service ManageOrder provided by the manufacturers as an example.

A *service interface* is defined by a UML class stereotyped ≪service≫ and named as the service itself. It should realize and use two UML interfaces, defining the in/out-messages by means of operations, named respectively S_IN and S_OUT, if the service is named S. *Realization* is represented by a dashed arrow with closed head stereotyped by ≪in≫, whereas *usage* is represented by a dashed arrow with open head stereotyped by ≪out≫. The operations of the service interfaces are used to model the in/out-messages. Messages may have parameters typed either by primitive types or by "objects" (introduced in the Object View). For instance, the message requestQuote contains an object of type Order. Thus, a service interface consists of a class diagram containing a class stereotyped by ≪service≫ and the two interfaces connected to it by the proper UML relationships (realization and use). The S_OUT may have no operation whenever the service provider never communicates with the service user; in this case it may be omitted.

Object View
- each classifier in the object view is either a class or a datatype

Realm View
- the realm view of a structured participant is the collection of the realm views of its sub-participants
- each realm model must contain a class stereotyped by ≪realm≫

Service View let SM be the model of a service S consisting of a service interface SI, a service contract SC and a service semantics SS:
- SM belongs to the service view of P iff it or its conjugate types a port of P
- the realms appearing in SS are included in those of P providing S
- the sequence diagrams in SS are corresponding bijectively with those of SC, and the corresponding sequence diagrams must have exactly the same messages in the same order, and all their guards are a conjunction of those of SC with possible further formulas based on the involved realms statuses

Let SD be a sequence diagram in SC/SS:
- SD must have exactly two lifelines typed by S_IN and S_OUT respectively
- all SD messages are calls of the operations of either S_IN or S_OUT
- the first message of SD is a call of an operation from S_IN (i.e. an in message)

Static View let SV be the static view of a participant P:
- SV includes exactly one class named as P, and stereotyped by ≪participant≫, whereas all other elements are either non stereotyped classes and datatypes used to define it or classes stereotyped by ≪realm≫
- each realm in SV must appear in the realm view of P, and must be connected by exactly one association with P, and the corresponding association end must be named
- any port of P must be either stereotyped by ≪service≫ and typed by a service interface, or by ≪use≫ and typed by the conjugate of a service interface

Behaviour View let BV be a behaviour view of a participant P (state machine):
- the events in BV correspond to in-messages (out-messages) of the interfaces of the services provided (used) by P
- the effects in BV are built out of internal actions of P, operation calls on the realms associated with P, and sending of messages

Service Architecture let SA be the service architecture of a structured participant SP:
- each part of SA must be typed by a sub-participant type of SP
- each port of SP stereotyped by ≪service≫ (≪use≫) must be connected with a port of a part of SA typed by the same service interface and stereotyped by ≪service≫ (≪use≫)
- each port stereotyped by ≪use≫ of a part of SA must be connected at least with either one port of another part typed by the conjugate of its service interface and stereotyped by ≪service≫ or with a port of SP stereotyped by ≪use≫ and typed by the same service interface
- each port stereotyped by ≪service≫ of a part of SP can be connected only with some port of another part typed by the conjugate of its service interface and stereotyped by ≪use≫ or with a port of SP stereotyped by ≪service≫ and typed by the same service interface (notice that thus it may also be unconnected, since not all the services offered by sub-participants must be used)
- each sub-participant class must type at least one part of SA

Figure 5: Precise SOM Model: Well-Formedness Constraints

Fig. 6 presents the interface of the service ManageOrder. The two operations of ManageOrder_IN represent the in-messages for requesting a quotation and for placing an order; whereas the two operations of the interface ManageOrder_OUT correspond to the out-messages and return a quotation and an order info, respectively.

A *service contract* consists of a set of UML sequence diagrams defining all the possible ways of using the service, showing which messages the provider and the user exchange and in which order. Those sequence diagrams have exactly two lifelines (service provider role and service user role) typed by the provided and used interfaces respectively (S_IN and S_OUT, if the service is named S); moreover, all their messages are calls of those interfaces operations, and the first message must be an in-message, i.e. a call of an operation of the provided interface.

Fig. 7 presents the contract of the service ManageOrder. We can see that the service may accept a request for a quotation for an order O, to which it answers providing a quotation containing, among other info, the proposed price (Q.price). Moreover, it may accept an order O, to which it answers with an order info OI, containing the order status (confirmed or cancelled). The guards are used to explicit the constraints of the contract between input and output; for instance,

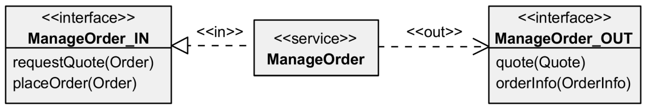


Figure 6: ManageOrder: Service Interface

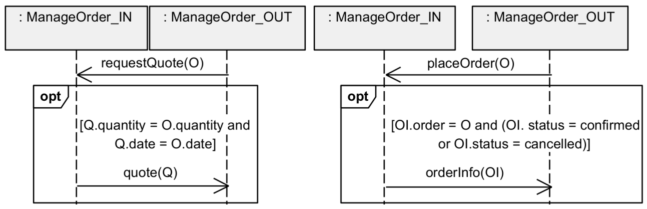
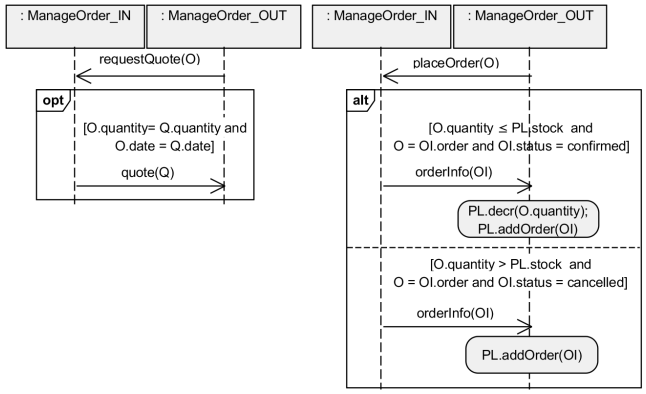


Figure 7: ManageOrder: Service Contract



: ManageOrder_IN
: ManageOrder_OUT
requestQuote(O)
opt
[O.quantity= Q.quantity and O.date = Q.date]
quote(Q)
: ManageOrder_IN
: ManageOrder_OUT
placeOrder(O)
alt
[O.quantity ≤ PL.stock and O = OI.order and OI.status = confirmed]
orderInfo(OI)
PL.decr(O.quantity); PL.addOrder(OI)
[O.quantity > PL.stock and O = OI.order and OI.status = cancelled]
orderInfo(OI)
PL.addOrder(OI)

Figure 8: ManageOrder: Service Semantics

the quote Q must be computed according to the same quantity and date of the order O for which a quotation was requested.

The value provided by a service may concern some realms, thus the *service semantics* should be defined by modelling how the in-messages will result in modifications of that realm's status, and how that realm's status will influence the out-messages. Then, the sequence diagrams of the contract should be refined by adding execution specifications to represent the modifications of the realms statuses, and further guards on the statuses of the realms to influence the decision of which messages to send out and which parameters they are carrying.

The realm of the service ManageOrder is denoted by PL:Plant (see Fig. 4), and its semantics, reported in Fig. 8, states that an order is accepted only in the case the required quantity of product is less or equal to that in stock. An execution specification shows how the stock and the order archive are modified whenever an order is received (e.g. if it is accepted, then the ordered quantity of product is eliminated from the stock).

3.4 Static View

The static view of a participant type P is used to show which services are provided and used by P, and on which realms P reigns. It is a class diagram containing only one class stereotyped by ≪participant≫, named as the type itself, and any number of classes stereotyped by ≪realm≫, linked by associations to the former.

A class stereotyped ≪participant≫ introduces a *participant type*, that will be used to type the specific participants (instances) of that kind, and that we name *participant class*.

The UML mechanism of the ports is used to indicate the points of interaction through which participants interact with each others to enact services, and the needed ports are added to the participant class. There are two kinds of port that a participant class may have, one is stereotyped by ≪service≫ where a service is provided, and the other by ≪use≫ where a participant uses a service provided by another participant. A port is then typed by a service interface. A service port has the type of the provided service interface, and a use

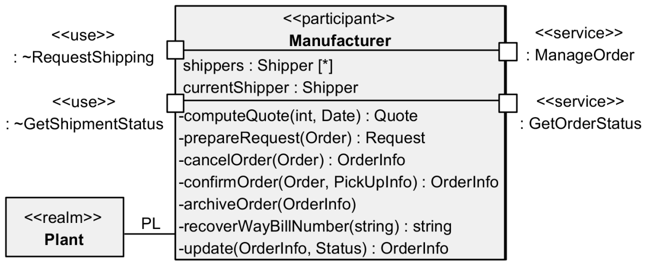


Figure 9: Manufacturer: Static View

port has the type of the conjugate of the interface of the required service. A *conjugate* service interface is suggested as a mechanism to connect the using participant and the providing participant. Each service interface has one conjugate service interface that is named by the name of the corresponding service interface prefixed with "∼", and it is defined transforming the in-messages into out-messages, and similarly the out-messages into in-messages, i.e. the realized interface becomes the used one and vice versa. The static view of a monolithic participant, the Manufacturer, can be seen in Fig. 9.

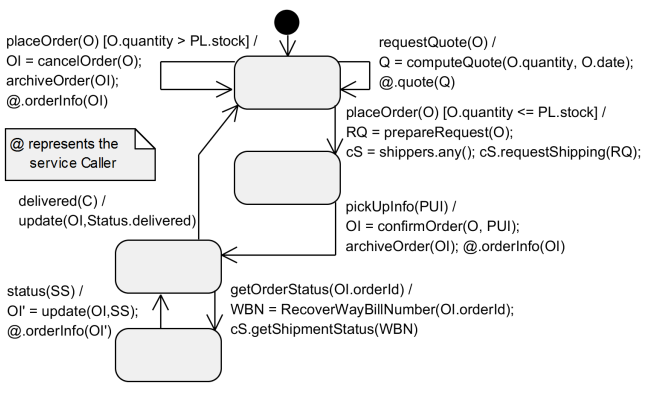
3.5 Behaviour View

The behaviour of the monolithic participants can be modelled using an activity diagram (e.g. in the case of an orchestrator of services) or a state machine (e.g. for a human being).

Fig. 10 presents a simplified, for space reason, behaviour of the manufacturers (a monolithic sub-participant of Dealer Network) by means of a state machine describing the handling of only one order per time. Such machine has 4 states and its events are determined by (a) the S_IN messages of the provided services (see, e.g. Fig. 6 for what concerns ManageOrder service), (b) and the S_OUT messages of the used services. The guards may involve the associated realms (in fact, PL is the association connecting the Manufacturer to the Plant) and the attributes depicted in its Static View. About the effects, they are built combining: (1) calls of private operations of the Manufacturer class (see the static view in Fig. 9), (2) sending of out messages of provided services to answer a request from a service user (for example, @.quote(Q) is the message containing a quote Q sent to the service caller @, e.g. a dealer, in compliance with ManageOrder contract), (3) sending of in messages of the required services (e.g. cS.requestShipping(RQ) is the message containing the shipping request RQ sent to the currently selected Shipper cS).

3.6 Service Architecture

The service architecture of a structured participant is modelled by a composite structure diagram containing the participant class con-



placeOrder(O) [O.quantity > PL.stock] / OI = cancelOrder(O); archiveOrder(OI); @.orderInfo(OI)
requestQuote(O) / Q = computeQuote(O.quantity, O.date); @.quote(Q)
placeOrder(O) [O.quantity <= PL.stock] / RQ = prepareRequest(O); cS = shippers.any(); cS.requestShipping(RQ);
@ represents the service Caller
pickUpInfo(PUI) / OI = confirmOrder(O, PUI); archiveOrder(OI); @.orderInfo(OI)
delivered(C) / update(OI,Status.delivered)
status(SS) / OI' = update(OI,SS); @.orderInfo(OI')
getOrderStatus(OI.orderId) / WBN = RecoverWayBillNumber(OI.orderId); cS.getShipmentStatus(WBN)

Figure 10: Manufacturer: Behaviour View

sidered as a structured class, having a part for each sub-participant role (typed obviously by its class). A multiplicity is associated with a part with the obvious meaning of constraining the number of sub-participants simultaneously playing that role.

Fig. 11 presents the service architecture of the Dealer Network (together with its static view); a participant role is displayed as a solid rectangle (the icon for the UML parts) that contains the optional role name, the mandatory participant class typing the role, and the multiplicity (omitted if equal to 1), e.g. : Dealer[*].

The fact that a sub-participant (role) uses a service provided by another sub-participant (role) is modelled by means of a connector linking the port where the service is provided (stereotyped by ≪service≫ and typed by the service interface) with the port where the service is used (stereotyped by ≪use≫ and typed by the conjugate of the service interface). Any port stereotyped by ≪use≫ must be connected either with a matching port of another sub-participant or with a port of the structured participant, whereas a port of a participant stereotyped by ≪service≫ may be connected with any number of matching port of sub-participants or with a port of the structured participant. If a structured participant provides a service, the port where it is provided should be connected with a port typed and stereotyped in the same way of a sub-participant (the one which will be able to provide such service); differently, if it uses a service, the port where it is used may be connected with any number (also none) of ports typed and stereotyped in the same way of some sub-participant (the one which will be able to use such service).

In Fig. 11, we can see how any service provided by a sub-participant of Dealer Network is used by another one (e.g. ManageOrder provided by the manufacturers and used by the dealers).

3.7 Business Processes in a (Service-Oriented) Domain

In the context of the service-oriented domains, the entities taking part in a business process are a subset of the participants of the service-oriented domain. To model a business process we consider *roles* for the process participants, not specific instances, thus we speak of *process roles*, that will be typed by participant classes. A *business process* in a service-oriented domain is essentially a workflow of service message calls (between pairs of process roles), and of internal actions of the participant roles (named *tasks*) aimed to obtain the process goal; moreover, since the participants manipulate objects and data, the model includes also roles typed by object classes. The workflow is modelled by an activity diagram, whereas the tasks are modelled by means of UML actions.

UML activity diagrams offer a large number of constructs (49 in the last version [22]). However many of them are relative to very specific cases or may be derived by other ones and moreover, a large number of them are scarcely known and used [20] and many of them have an unclear static and dynamic semantics. For this reason, to avoid that the modeller wastes time in studying and deciding which ones to use, to prevent common mistakes, and to improve the readability of the produced diagrams, Precise SOM strongly suggests to use only the following subset of the 49 constructs, that in our experience are sufficient to model the business processes: action nodes, control nodes (i.e. initial, decision/merge, fork/join, flow and activity final), time events, control flows, the rake construct (that allows to reuse an activity defined elsewhere by means of another activity diagram), and swimlanes.

The activity diagram modelling the business process should include a vertical swimlane to represent each participant process role (more detailed processes may use swimlanes to represent also the involved realms). The label of such lane must have the form RName: ParticipantClass. The actions will be placed in the swimlane of the participant performing them. The sending of a service message represented by the operation op by a process role R1 to a process role R2 is represented by a signal sending action having form R2.op(...) that must be placed in the swimlane of the sending role R1.

Furthermore, business processes require often non-deterministic choices among several alternatives and, since the UML activity diagram does not offer a specific construct to represent it, but it may be obtained by a choice where all the guards of the alternatives are true, we introduced the black diamond to represent such construct.

See Fig. 13 to view a partial list of well-formedness constraints regarding business processes in Precise SOM.

Fig. 12 shows the Buying business process of Dealer Network. The layout depicted in the figure is not the optimal one we suggest [17]; however, it has been adopted, due to space limits, to include all the relevant information.

It may be surprising, but the list of the suggested activity diagram constructs does not include the object nodes; they are not needed since we explicitly represent the things used in the processes by means of objects/data roles (see, e.g. the flowing of O in the top-left part of the diagram in Fig. 12). Moreover, because our activity diagrams want to precisely model the flowing of the activities in a business process, if we decide to use the object nodes to model the passing of something between two activities, they should be used for any possible passage, hence making the activity diagrams immediately unreadable

3.8 Workflow of Precise SOM

Now we introduce the workflow that a modeller should follow in order to properly apply our method. We identified two main approaches for building a model in the context of service-oriented domains. The *business first* approach is characterized by a top down process: starting from interviews with stakeholders, informal messages exchanged between domain participants are identified and some business processes are drafted. More details about those partic-

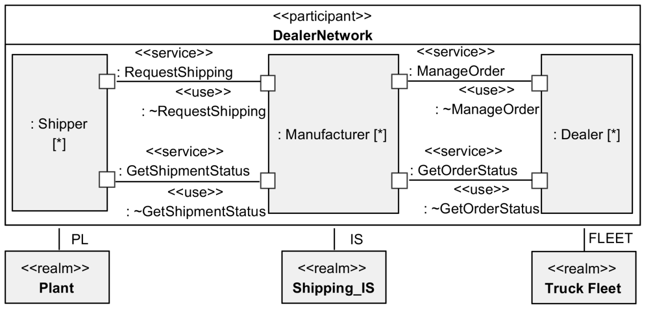


Figure 11: Dealer Network: Service Architecture & Static View

Let a business process model consisting of the roles R_1:PC_1, ..., R_m:PC_m, and of the activity diagram AD:

- $m \geq 1$
- for $i = 1, \ldots, m$ PC_i is a participant class
- AD is built using only action nodes, control nodes (i.e. initial, decision/merge, fork/join, flow and activity final), control flows, time events, the rake construct, and swimlanes. Moreover:
- a guard on a flow leaving a decision node is labelled by [else] or a well-formed OCL expression of type bool without side effects, where only the process roles appear freely
- any activity recalled by means of the rake construct must be defined in another diagram part of the business model

Figure 13: Business Process: Well-Formedness Constraints

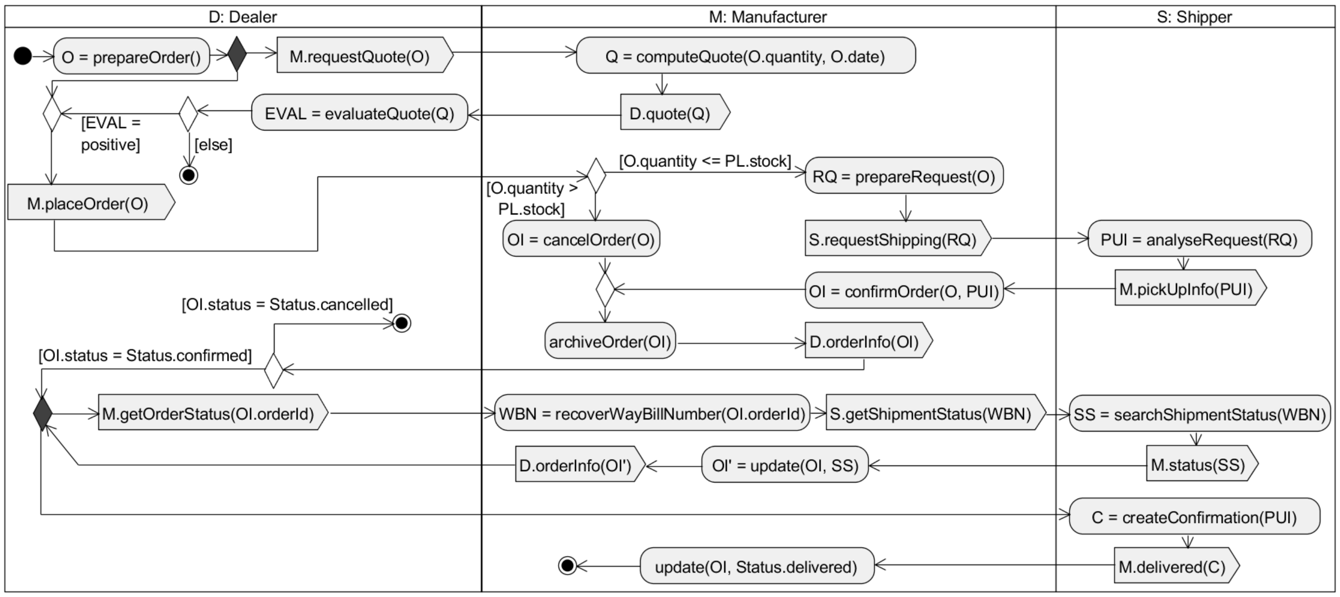


Figure 12: Dealer Network: Business Process Buying

ipants, services, realms and objects are later gathered and introduced to support the previously modelled elements, eventually refining the business processes themselves. The *service first* approach is a bottom up process, where the domain modelling is the first step. Here, services are identified at the beginning, aiming at having more detailed information while modelling the business processes at the end, which makes them consistent and precise.

In Fig. 14 the *service first* approach is detailed. We adopted it for modelling the running example, since initially more know-how about participants and services was available than that about the process. As shown, each constitutive element of Precise SOM meta-model (see Fig. 2) is modelled, having a draft list of the services of the domain as input. Notice that, whenever a structured participant is found, each sub-participant will be modelled by recursively applying the Model Participant Type activity. At the end of the activity, if any, business processes are easily obtained. Even though we chose not to show it in Fig. 14 for simplicity, it is important to consider that the modeller may move back to a step of the process in case a refinement is needed (e.g. a new realm is identified while working on the service architecture).

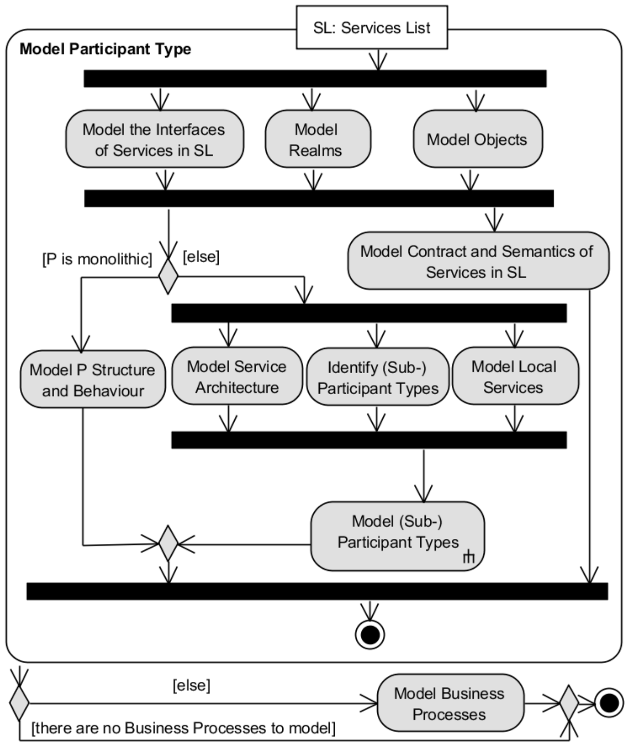

SL: Services List
Model Participant Type
Model the Interfaces of Services in SL
Model Realms
Model Objects
Model Contract and Semantics of Services in SL
[P is monolithic]
[else]
Model P Structure and Behaviour
Model Service Architecture
Identify (Sub-) Participant Types
Model Local Services
Model (Sub-) Participant Types
[else]
Model Business Processes
[there are no Business Processes to model]

Figure 14: The Service First Approach

3.9 The Harbour Project

The method has been prompted and evaluated in a joint industry-academic projects where the domain to model was a harbour consisting of ships, people working on them, carriers, officials, documents, containers and so on. The processes were modelled around the interactions among those participants. We applied the “service first” approach to obtain precise business processes. The complete model of the harbour (see [1]) included more than 10 participant classes and more than 20 object classes, 3 realms and half a hundred of messages exchanged from about 20 services. The main business process included more than 40 nodes, considering both internal actions and messages. Feedback from professionals was positive and promising. The involved stakeholders were satisfied with the quality of the obtained artefacts and, more generally, with the usefulness of our method.

4. RELATED WORK

In the context of the SENSORIA project[1], a number of model-driven approaches have been proposed for developing service-oriented systems. For example, MDD4SOA (Model-Driven Development for Service-Oriented Architectures) [11] adopts UML4SOA profile[2], an extension of UML which overcomes UML limitations in modelling service-specific elements and services orchestration. As last step, the produced UML4SOA artefacts are transformed into code. The

[1]http://www.sensoria-ist.eu

[2]http://www.pst.ifi.lmu.de/projekte/Sensoria/del_54/D1.4.b.pdf

main difference with our proposal is that MDD4SOA proposes an approach for modelling and implementing executable service-oriented systems, while we provide a precise method for modelling domains and business processes, including both (software) systems and humans.

Service-Oriented Architecture Framework (SOAF) [8] has been proposed to improve the quality and effectiveness of the produced service-oriented systems by means of a quite structured process, consisting of input data (e.g. business use cases), activities to perform (e.g. business process modelling) and produced artefacts (e.g. business architecture). Even though, similarly to Precise SOM, SOAF is structured and precise, it differs on the following points: no well-formedness constraints are provided, UML is not adopted, and the goal of the framework is not on domain modelling.

Delgado *et al.* propose a conceptual methodology to derive software services from BPMN business processes [6]. It is composed of two main steps: 1) services identification and, 2) orchestration/choreography modelling starting from business processes. In another work, the same authors present the MINERVA framework [5], which adopts the previous methodology and combines Model Driven Development with Service-Oriented Computing paradigms to derive executable services from business processes. These works adopts the SoaML profile, but neither the methodology nor the framework introduce ways to represent the behaviour of a human/software participant in the process as Precise SOM does.

IBM Service-oriented modelling and architecture (SOMA) [2] is largely used in the industry, in particular for developing end-to-end SOA solutions and for modelling software systems based on services. SOMA is a flexible software engineering method composed of several iterative phases, each one following a structured flow of tasks. Flexibility is provided by choosing solutions templates, i.e. customized decisions to specific problems that depend on the client's needs while building a SOA solution. Similarly to UML4SOA and SOAF, SOMA is not oriented to conceptual modelling and its phases are not UML-based. The absence of well-formedness constraints is another difference with Precise SOM.

5. CONCLUSION AND FUTURE WORK

In this work we proposed a novel method for the conceptual modelling of domains integrated with business processes, using a specific UML profile and adopting the service paradigm. The novelty is given by the characteristics of our method that, to the best of our knowledge, differentiate Precise SOM from any other service-oriented domain modelling method found in the literature. First, Precise SOM integrates both dynamic and static aspects of domain entities, making the modelling of the behaviours of (human) participants flexible. Second, Precise SOM models are precisely defined by means of a metamodel, a set of constraints and restricting UML to only essential language constructs, with the final goal of helping the modellers to avoid common mistakes, and to reduce the time needed to build a model. The workflow we provided is simple and may help the modeller to apply the method with little effort. Since we chose the well-known UML visual notation, the resulting artefacts may be also used for future activities, such as code transformations, simulations and formal analyses. Finally, the method avoids proprietary solution, thus it can be freely and completely applied.

As future work, we intend to: 1) experiment the method with further industrial case studies, 2) improve/refine our workflow, 3) empirically compare Precise SOM with other modelling methods, and finally 4) develop a set of tools using open source Model-Driven Engineering technologies for automatically verifying the constraints on the various parts of the models and supporting the automatic generation of source code for the software system participants.

6. REFERENCES

[1] *Precise SOM Website*. http://sepl.dibris.unige.it/PreciseSOM.php.

[2] A. Arsanjani, S. Ghosh, A. Allam, T. Abdollah, S. Gariapathy, and K. Holley. SOMA: A method for developing service-oriented solutions. *IBM Syst. J.*, 47(3):377–396, 2008.

[3] E. Astesiano, G. Reggio, and F. Ricca. Modeling business within a UML-based rigorous software development approach. In *Concurrency, Graphs and Models*, volume 5065 of *LNCS*, pages 261–277. Springer, 2008.

[4] C. Choppy, G. Reggio, and K. Tran. Formal or not, but precise modelling of services with CASL4SOA and SoaML. In *Proc. of KSE 2012*, pages 187–194. IEEE, 2012.

[5] A. Delgado, F. Ruiz, I. de Guzmán, and M. Piattini. MINERVA: model driven and service oriented framework for the continuous business process improvement and related tools. In *Service-Oriented Computing*, volume 6275 of *LNCS*, pages 456–466, 2009.

[6] A. Delgado, F. Ruiz, I. de Guzmán, and M. Piattini. Towards a service oriented and model-driven framework with business processes as first-class citizens. In *Business Process, Services Computing and Intelligent Service Management*, volume 147 of *LNI*, pages 19–31. GI, 2009.

[7] T. Erl. *Service-oriented architecture: concepts, technology, and design*. Prentice Hall, 2005.

[8] A. Erradi, S. Anand, and N. Kulkarni. SOAF: An architectural framework for service definition and realization. In *Proc. of SCC 2006*, pages 151–158. IEEE, 2006.

[9] G. M. Giaglis. A taxonomy of business process modeling and information systems modeling techniques. *Int. J. of Flexible Manufacturing Systems*, 13(2):209–228, 2001.

[10] M. Leotta, G. Reggio, F. Ricca, and E. Astesiano. Towards a lightweight model driven method for developing SOA systems using existing assets. In *Proc. of WSE 2012*, pages 51–60. IEEE, 2012.

[11] P. Mayer, A. Schroeder, and N. Koch. MDD4SOA: Model-driven service orchestration. In *Proc. of EDOC 2008*, pages 203–212. IEEE, 2008.

[12] J. Meekel, T. B. Horton, R. B. France, C. Mellone, and S. Dalvi. From domain models to architecture frameworks. In *Proc. of SSR 1997*, pages 75–80. ACM, 1997.

[13] Oasis. *Reference Model for Service Oriented Architecture 1.0*, 2006.

[14] OMG. *SoaML Specification, Version 1.0.1*, 2012.

[15] M. Razavian and R. Khosravi. Modeling variability in business process models using UML. In *Proc. of ITNG 2008*, pages 82–87. IEEE, 2008.

[16] G. Reggio, E. Astesiano, and C. Choppy. A framework for defining and comparing modelling methods. In *Software, Services, and Systems*, volume 8950 of *LNCS*, pages 377–408. Springer, 2015.

[17] G. Reggio, M. Leotta, and F. Ricca. "Precise is better than light" a document analysis study about quality of business process models. In *Proc. of EmpiRE 2011*, pages 61–68. IEEE, 2011.

[18] G. Reggio, M. Leotta, and F. Ricca. Who knows/uses what of the UML: A personal opinion survey. In *Proc. of MODELS 2014*, volume 8767 of *LNCS*, pages 149–165. Springer, 2014.

[19] G. Reggio, M. Leotta, F. Ricca, and E. Astesiano. Business Process Modelling: Five Styles and a Method to Choose the Most Suitable One. In *Proc. of EESSMod 2012*, pages 8:1–8:6. ACM, 2012.

[20] G. Reggio, M. Leotta, F. Ricca, and D. Clerissi. What are the used UML diagram constructs? A document and tool analysis study covering Activity and Use Case diagrams. In *Model-Driven Engineering and Software Development*, volume 506 of *CCIS*, pages 66–83. Springer, 2015.

[21] N. Russell, W. M. P. van der Aalst, A. H. M. ter Hofstede, and P. Wohed. On the suitability of UML 2.0 activity diagrams for business process modelling. In *Proc. of APCCM 2006*, pages 95–104. ACS, 2006.

[22] UML Revision Task Force. *OMG UML, V2.5*, 2015.

[23] P. Wohed, W. M. van der Aalst, M. Dumas, A. H. ter Hofstede, and N. Russell. On the suitability of BPMN for business process modelling. In *Business Process Management*, volume 4102 of *LNCS*, pages 161–176. Springer, 2006.