Distribution Concerns in Service-Oriented Modelling

N.Aoumeur, J.L.Fiadeiro, C.Oliveira
Department of Computer Science
University of Leicester
Leicester LE1 7RH, UK
{na80,jwf4,co49}@leicester.ac.uk

ABSTRACT

Service-oriented development offers a novel architectural approach that addresses crucial characteristics of modern business process development such as dynamic evolution, intra- and inter-enterprise cooperation, and distribution/mobility. In previous papers, we have shown how the mechanisms that regulate the relationships, functioning and cooperation of business activities in such architectural models can be externalised from business rules in terms of connectors that can be superposed dynamically on stable core business entities. That is to say, we focused on what, in the literature, has been called the "service composition layer" of service-oriented architectures or, for short, their "composition logic". Our emphasis in this paper is on the distribution aspects: we show how a corresponding "distribution logic" can be defined in terms of another set of architectural primitives that address the way business rules depend on "locations". These primitives address what are sometimes called "business channels" (ATMs, PDAs, Pay-TV, Internet, inter alia) as offered in typical contemporary ICT-infrastructures with substantial added-value to business processes. We argue that interacting (core) business entities located at or endowed with such ICT capabilities should be modelled in a way that separates the composition from the distribution logic so that business interactions can be understood and evolved in a location-transparent way. Our approach is based on a mathematical model that we have recently developed for modelling context-aware interactions. An example from banking is used for illustrating its applicability.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Distributed Programming, Parallel programming. D.2.11 [**Software Architectures**]: Languages – *connectors;* F.1.2 [**Modes of Computation**]: Interactive and reactive computation.

General Terms

Design, Languages, Verification.

Keywords

Evolution, location-awareness, rule-based business modelling, service composition and coordination, software architectures

1. INTRODUCTION

Modern business processes are becoming more and more complex, reflecting the increasing dependency of the economy, and the functioning of the society as a whole, on intricate and volatile intra- and inter- organisational cooperation. On the other hand, business operations are relying more and more on day-to-day advances in Information and (wired/wireless) Communication Technology (ICT). In order to remain competitive, respond to market pressure and attract more customers, companies are pressed to provide ever more sophisticated added-value services. For instance, banks are continuously creating new services or updating existing ones to match the expectations and profiles of their customers, while at the same time supporting more and more advanced channels for day-to-day banking such as ATM, Internet, PDA, Pay-TV, inter alia [23].

This tension between complexity and agility is raising new challenges on the way software needs to support business information systems. It is clear that these challenges transcend by far the capabilities of the software engineering techniques that have been traditionally used for business process development. This is why most business designers are looking for new solutions around workflows [1] and, more recently, web services [34]. As a result, significant standards, techniques and models have been advanced in both directions for modelling and enacting business processes.

However, we argue that the *operational* character of these approaches (even when supported by mathematical models like BizTalk [8,24]) makes it very hard to tackle all the above features adequately. Although it is widely accepted that abstraction and rigor are the preponderant means for tackling levels of multi-dimensional complexity, addressing these requirements equally and coherently, as their nature and expected added-value determine, requires a more declarative approach and semantic modelling primitives that work at a level of abstraction in which the different dimensions can be integrated and reasoned about.

More specifically, on the one hand, current standards lack rich mechanisms like service negotiation, contracting and service communication and coordination as required for flexibility and *dynamic* adaptation and evolution [7] in cross-organisational processes. In addition, despite some progress in semi-automatic derivation of service-oriented business processes from informal business rules [26], the relationship between business rules, their evolution and web-services in general remains largely unexplored.

On the other hand, proposals based on Web services experience serious difficulties in addressing location-awareness as an essential business concern for dealing with multi-channels provided by present day's technology. Web services can be programmed in ways that respond to the need for businesses to operate in different platforms and through different channels (say, banking at an ATM, across the internet, or through a PDA/mobile phone), but Service Description Languages do not provide abstraction mechanisms for modelling the underlying *distribution logic* and the way it adheres and enforces given *business policies*.

The purpose of this paper is to put forward a set of primitives through which distribution concerns can be addressed in service-oriented business modelling. We do so by extending the approach that we have put forward in [6] for addressing the composition logic, i.e. "the way composite services can be constructed for defining processes or workflows that interact with sets of Web Services to achieve certain goals" [11,32].

In section two, we justify the use of a rule-based architectural approach for modelling both the composition and the distribution logic of services, discuss the main assumptions that we make on the way service-oriented development applies to business processes, and present the running example – a simplified banking system. In section three, the coordination primitives that address the composition logic are reviewed and illustrated using the example. In section four, location primitives are presented as the building blocks for the envisaged distribution logic, and illustrated using the same example. In section 5, we present an architecture for modelling and evolving agile and dynamic business processes based on coordination and distribution.

2. MOTIVATION

In this section, we justify why and how we are bringing together concepts and techniques from service-oriented development, rule-based business modelling, software architecture, and context-aware computing.

A rich set of specifications is currently available for software development over service-oriented architectures that include the Business Process Execution Language for Web Services (BPEL4WS or BPEL for short) [9], WS-Coordination [35] and WS-Transaction [36], *inter alia*. A so-called BPEL composition is a business process or workflow that interacts with a fixed set of Web services to achieve a certain goal. A business process is taken as a series of activities involving a given set of partners connected according to given data and control flow requirements. For instance, a *banking process* can be taken to consist of several activities, including specifically:

- Customer identification and authentication.
- Customer execution of banking transactions (deposits, withdrawals, loans, mortgages, etc).
- Customer exit.

Web Services are "self-contained, modular applications that can be described, published, located, and invoked over a network, generally the Web" [34]. They are capable not only of performing business activities on their own, but also to take part in higher-order business transactions by engaging in more or less complex interactions with other Web services.

This approach offers a significant number of advantages. For instance, by being platform-neutral, Web services support the definition of business processes by using existing elementary or complex services, possibly offered by different service providers or extracted from so-called legacy systems. However, even applications developed on the basis of BPEL are still some way from addressing the challenges raised by the need to tackle complexity and agility as identified in the introduction. One of the reasons is that BPEL-style applications are rather unstructured and static. For instance, services are composed in a rather ad hoc and unprincipled manner by simply combining their operations and input and output messages. This makes business processes difficult to evolve. If the business rules under which the process operates change or need to be adjusted, the workflow will have to be revised and additional or modified service interfaces may have to be used for the interconnections.

Recent investigations in business process modelling are shifting the emphasis towards more abstraction through business rule-driven approaches [15,29]. Business rules are understood as "projections of organisations' constraints and declarations of (internal/external) policy/conditions that must be satisfied for doing business" [33]. They specify actions to be taken on the occurrence of particular events, including "state of being" changes concerning individuals, infrastructure, consumables, informational resources, and business activities in general.

Rule-driven approaches offer a number of advantages that are crucial for coping with dynamically evolving complex business processes. They support the specification of business models *independently* of the specific processes that happen to be running at any one instant. They focus on more primary requirements and address business domain descriptions in a declarative rather than operational way. For all these reasons, they are generally more apt to support *evolution*.

The exploitation of these potentials for achieving new degrees of dynamism and abstraction in Web Services composition remains largely unexplored. An exception is the recent work by Papazoglou et al. [24]. In this approach, starting from a very general specification, the composition is scheduled, constructed and finally executed with the assistance of business rules judiciously classified in a repository. Besides basic elements such as events, conditions, and messages, this classification includes rules dealing with the activity flows, the data required for their composition and the constraints to be respected. The direct construction and subsequent execution of the composition from the business rules is performed in terms of XML-like descriptions. However, the approach does not address the distribution dimension.

Our contribution follows in this path and aims to enhance the potential of service-oriented architectures by developing semantic primitives that raise the level of abstraction and capture rule-based business modelling. In the approach that we have in mind, each business activity is a dynamic entity that is put together, at runtime, from a number of self-contained applications (services) that need to be located and invoked over a distribution network. The way these services are brought together and invoked, what is sometimes called "orchestration", must follow given business rules as set by the organisation. For instance, it is clear that a withdrawal activity is subject to the business rules that apply to the specific customer and account involved as business entities. Depending on the nature of the account and of the customer, certain constraints may apply that determine if, for a given amount, the withdrawal is authorized and, if so, what operations of the bank itself need to be executed.

More specifically, our approach aims to capture business rules at an interaction level so that dynamic adaptation of services and cross-organisational service cooperation can be intrinsically and explicitly supported (composition logic). For this purpose, we adopt techniques akin to those that have been developed for Software Architecture [3]. We propose to capture as a *connector* any business rule dealing with intra- and inter-organisation cooperation. On the one hand, as modelling primitives, architectural connectors can be made to describe business service compositions in a declarative way as shown through the rule-based approach proposed in [5]. On the other hand, as shown in [22] architectural approaches support dynamic evolution as required for agility and reconfigurability.

In our approach, the mechanisms that are required for regulating the relationships, functioning and cooperation of services are externalised from business rules in terms of semantic primitives that we call coordination laws. These describe composition mechanisms in terms of event-condition-action (ECA) rules that can be superposed dynamically on stable core business entities. Superposition [16] is non-intrusive on the code that implements the services. Therefore, business architectures can be dynamically

evolved, as volatile business rules change or new cross-organisational links come into force, while ensuring compliance to core business invariants.

However, business activities depend on business channels and networks in ways that are orthogonal to the interactions that business relationships impose. For instance, depending on the location where the banking process is requested, identification and authentication can consist of:

(1) A simple "hello" when the request is made by the customer in person at the desk of the branch where the account has been held for 20 years.

(2) The presentation of a personal identity document if the clerk has only recently joined that branch or if the customer at a different branch makes the request.

(3) A complex transaction involving debit cards and codes if the request is made at an ATM (not necessarily by the customer).

(4) A collection of passwords, security codes and predetermined personal questions if the request is made through the Internet (again, not necessarily by the customer).

In terms of a service-oriented architecture, this means that the way composite services need to be constructed should obey not only a composition logic derived from coordination concerns, but also a distribution logic derived from location concerns. Indeed, location-awareness is common to business channels (e.g. ATM, Branch, Pay-TV), mobile devices (e.g. PDA), internet-based facilities/software, and sensors, inter alia. The presence and quality of communication with other partners as well as the ability to migrate or move to other locations are among the crucial features that need to be taken into account at the level of this distribution logic.

Notice that, by location, we do not mean necessarily the space of addresses typically used in the Web. In the literature, service-oriented modelling is almost always instantiated to "Web Services", i.e. "software that can process XML documents it receives through some combination of transport and application protocols" [31]. Such services need to be located and invoked over the Web using addresses and referencing mechanisms that identify where services can be found using a given protocol like TCP or HTTP.

We have already motivated that this is a rather low-level view of what can be called the "service-oriented paradigm", which we would like to explore from the point of view of business process modelling in the architectural approach that we motivated. In particular, we would like to distance ourselves from both the XML-centred view of information exchange, and the Web-oriented notions of location and reference protocols. Our proposal is to work on a space in which locations correspond to business entities and channels organised according to a given business domain. Therefore, we do not work with a fixed notion of location at all. We propose that, as part of business modelling, the notion of location and distribution network that best applies to the business domain be specified in abstract terms through data sorts and operations.

To the best of our knowledge there is no *conceptual modelling* approach that addresses location-awareness in business processes in the sense that we have motivated, except for the work in [2], one of our main sources of inspiration. This work invokes the notion of "channel" for addressing location-awareness. It is, altogether, rather "operational", not as declarative as we wish ours to be, because it uses state machines as a modelling tool. It does not cope with the evolutionary side either, and it has not been integrated within an architectural approach that provides explicit connectors that can handle location-dependency aspects.

This is why, in what concerns the distribution logic that captures the dependency on the business channels and networks, we propose an approach based on explicit connectors that we call location laws. As with coordination laws, these connectors can be superposed dynamically and evolved independently of the other business aspects, allowing systems to self-adapt or be adapted to changes that occur at the distribution level without interfering with the core business policies.

The semantics of our distribution logic builds on our recent work around CommUnity, a formal approach that we have been developing for architectural description [14]. CommUnity includes primitives that capture distribution and mobility aspects [4,19]. The whole approach has a mathematical semantics defined over Category Theory [13]. We borrow in particular the notion of space of mobility (location structure) and corresponding contexts with the "be-in-touch" and "reach" relationships as preconditions for communication and mobility. These ingredients are then combined in a new format for condition-action rules that model the way service composition depends on the properties of the current context.

3. COORDINATION CONCERNS

Coordination primitives, as we have been promoting in recent work [5], provide a clean separation between the modelling of the computations performed by stable core entities on the running business configuration to ensure the functionality of basic business services, and the mechanisms that reflect how the (intra- or cross-organisational) interactions between these business services should be coordinated according to given business rules.

The emphasis is, therefore, on the aspects that subsume what in the literature has become known as the "Service Composition Layer" of Service-Oriented Architectures [27], i.e. the level at which business processes can be put together from elementary services. We aim for the level at which so-called business protocols and processes [11] are addressed. What we have in mind is the definition of processes or workflows that interact with sets of Web services to achieve certain goals in terms of abstract service descriptions, separated from specific deployments. In our approach, such interactions are captured using the concepts of coordination laws and interfaces. In terms of architecture description languages, these correspond to connector types and roles. In terms of business modelling, they capture business rules that regulate and compose required and provided services by the core entities that instantiate the roles.

This view addresses the emphasis put by BPEL [9] on the definition of service compositions in terms of processes that interact with partners that are external to the composition itself and identified only in terms of abstract interfaces. Indeed, it is particularly important that we are able to separate the definition of the "composition logic" from the run-time composition of specific services as part of a process that is being executed to fulfil a specific business goal. We address the former in terms of "coordination laws" that capture the business rules according to which complex business activities are put together from more basic services. The purpose of this section is to focus on the coordination model that we adopt for composing abstract services according to business rules.

In fact, in our approach, we go one step further and assign partners not to the business process as a whole but to the activities that are performed as part of the process. This recognises the fact that the partners involved in one activity may be different from those in another activity within the same process. Moreover, it may not be possible to pre-determine which partners will become involved in a given activity as this may depend on what has happened in the process so far.

The abstract description of the services that are partners in a given business is made in terms of what in [5] we called *coordination interface*. For instance, as a business activity, a withdrawal involves both an account and a customer regardless of the way the withdrawal is requested, if by the customer proper or anyone else. The purpose of the identification activity is, precisely, to determine the business entity that is involved in the business activity. Hence, in the case of a withdrawal, two coordination interfaces are required: one catering for the account service through which the debit needs to be performed, and the other for the customer service that is invoked as a result of the identification and authentication activity.

Note that these are "business" partners, not software components that offer operations as in object-oriented approaches. We fully support the distinction made in [31] between Web-services and distributed objects. In this paper, we are in no way concerned with the way services are programmed and deployed. For us, an account is not a software component that instantiates an object class. An account is understood as a business service, a unit of organisation around which a number of operations are grouped together to fulfil certain goals.

Such business partners are not units of execution either. A customer does not perform a withdrawal by calling the account to execute a debit. It is the composition logic, as captured by a coordination law, that dictates that a debit, as an operation of the account service, needs to be invoked whenever a customer issues a request for a withdrawal, say at an ATM through some combination of keys and buttons. The debit is to be located according to the account as a business entity, not as a software component that stores the code of the debit operation.

The trigger/reaction mode of coordination that our approach supports requires that each coordination interface identifies which *events* produced at execution time are required to be detected as triggers for the process to react, and which *operations* must be made available for the reaction to superpose the required effects. Notice that this separation is supported, for instance, in BPEL processes, by distinguishing between different kinds of actions (e.g. synchronous request/response or asynchronous one-way operation) that implement interactions among the process and its partners. Indeed, in BPEL, this separation occurs at a lower level of abstraction and has to be set in a pre-defined, static way. The same applies to the identification of the exchange of messages that such modes of interaction may require between the partners involved: in WSDL, each operation/event in our sense is a sequence of input and output messages.

The two composition interfaces that we have in mind can be described as follows:

```
coordination interface CW-CI
partner type   CUSTOMER
operations     owns(a:ACCOUNT):Boolean
events         withdraw(n:money; a:ACCOUNT)
end interface
```

```
coordination interface AW-CI
partner type   ACCOUNT
operations
    balance():money
    debit(a:money)   post balance() = old balance()-a
end interface
```

Each interface identifies the type of the partner that it models. A coordination interface does not identify a specific instance of this type, just the operations and events that partner instances are required to make available. Notice how the properties of the operations that are required are specified in an abstract way in terms of pre- and post-conditions.

This type should be specified in terms of a sort of business identities and functions that can relate the partner to other business entities as required by the application domain. For instance, the sort *ACCOUNT* should be provided with a function *bank* of type *BANK* identifying the bank in which it resides, again as business entity, not as a software component. To be more precise, as discussed in section 5, *a:ACCOUNT* may identify a service that is running as part of a bigger service *bank(a):BANK*. That is, we are not necessarily committed to creating a new independent service upon instantiation of a coordination interface: we may bind the interface to a running service that will take the instance of the interface as a sub-service. In this way, we may cater for situations in which the bank, as an organisation, runs a separate service for each account, one single (complex) service for all accounts, one single (huge) service for the whole bank, and so on.

Another important requirement for the intended composition logic is that the activity, as a composite service itself, should be described only on the basis of the interfaces and the data and control flow aspects that the coordination mechanisms put in place to ensure the underlying business goal. This is what, in BPEL, would be called the "state and logic" necessary for coordinating the interactions between the process and the partners. This "composition logic" can be described in terms of what we call a coordination law [5]:

```
coordination law SW-CL
partners     acco:AW-CI; cust:CW-CI
rules
when  cust.withdraw(n,acco)
      with acco.balance() ≥ n &
           cust.owns(acco)
      do   acco.debit(n)
end law
```

Besides identifying the coordination interfaces, a coordination law specifies the rules that define the behaviour of the service. Such coordination rules are of the form:

```
when  event
      with condition
      do   set of operation invocations
```

Each coordination rule identifies, under the "when" clause, a trigger to which the contracts that instantiate the law will react – a request by the customer for a withdrawal in the case at hand. The trigger can be just an event observed directly over one of the partners or a more complex condition built from one or more events. Under the "with" clause, we include conditions (guards) that should be observed for the reaction to be performed. If any of the conditions fails, the reaction is not performed and the occurrence of the trigger fails. Failure is handled through whatever mechanisms are provided by the language used for deployment. See [9] for explicit handling of faults within BPEL.

The reaction to be performed by the composite service is identified under the "do" clause as a set of elementary activities. This set may include calls to operations provided by one or more of the partners as well as actions that are internal to the "composition logic" of the business activity itself. The whole interaction is handled as a single transaction, i.e. it consists of an atomic event in the sense that the trigger reports a success only if all the actions identified in the reaction execute successfully and the conditions identified under the "with" clause are satisfied. Details on transaction protocols for web-service interactions can be found in [36].

In what concerns the language in which the reactions are defined, we normally use an abstract notation for defining the synchronisation set as above. This is important for bringing to a more abstract modelling level the definitions of business processes that recent languages for "orchestration" like BizTalk [8] promote, in terms of algebras and models for concurrency. Our opinion and experience is that the architectural modelling level at which we promote the representation of business interactions makes it easier to bridge the gap from the more organisational high-level goals and policies that dictate how business should be run to the choice of particular control and synchronisation structures that can make specific processes run.

The externalisation of this composition logic in a coordination law is decisive for supporting the required agility in terms of dynamic business evolution. The fact that the conditions on which an account may be debited by its owners are not hard-coded in the operations made available by the account, make it possible for these conditions to be changed without interfering with the deployment of these services. For instance, in order to offer a VIP-withdrawal in which a given credit limit is allowed, we just have to change the composition logic as modelled by the coordination rule; the basic debit operation does not need to be changed.

```
coordination law VIPW-CL
partners       acco:AW-CI; cust:cCW-CI
rules
when  cust.withdraw(n,acco)
      with acco.balance()+cust.credit()≥ n  &
           cust.owns(acco)
      do   if  acco.balance()≥ n
               then acco.debit(n)
               else acco.debit(1.01*n)
end law
```

Notice that a different partner is now required to play the role of the customer: we need a service that offers an operation for obtaining the credit limit currently assigned to the customer:

```
coordination interface cCW-CI
partner type   CUSTOMER
operations     owns(a:ACCOUNT):Boolean
               credit():money
events         withdraw(n:money; a:ACCOUNT)
end interface
```

Coordination interfaces can be hierarchically organised so as to facilitate location and binding of specific concrete services. We leave such matters to a subsequent paper.

4. LOCATION CONCERNS

This section puts forward the concepts and constructions that we are developing for addressing location-awareness in service-oriented business modelling. As emphasized in the introduction, our purpose is to provide elements for a "distribution logic" that can capture the way service composition needs to take into account properties of the underlying business channels and communication infrastructure. Just like coordination mechanisms that separate service functionality from the "composition logic", which we illustrated in the previous section, we want to define location primitives that can externalise the way business activities depend on properties of the distribution topology over which services are composed. The properties that we address in the paper are:

(1) The communication status, i.e. the presence, absence, or quality of the communication link between locations where given services are executing but require data to be exchanged and synchronisation protocols to be observed as part of the composition logic.

(2) The ability to continue the execution of an activity at another location, which requires the new location to be reachable from the present one for the execution context to be moved.

For this purpose, we capitalise on the work developed around CommUnity [19]. Although, for simplicity, we will not address this specific aspect in depth, the space of locations can be defined by the user as an abstract data type with a sort *loc* and functions that capture the properties of the notion of location that are suitable for the application domain at hand. This is because, typically, different kinds of applications require different notions of location. When a specific notion of location is fixed, as for instance in Ambients [10], modelling a different space of mobility requires the encoding of a different notion of location, which can be cumbersome and interfere with other aspects. Two observables capture location awareness as discussed above: communication is handled through BT:*set(Loc)* and movement/reachability through REACH:$Loc \times Loc$.

As we did for the composition logic through coordination laws, location laws are the means through which we model the distribution logic of a given business domain. Whereas coordination laws interconnect partners that are meaningful for the underlying composition logic, e.g. customers and accounts in the case of the withdrawal, the partners involved in location laws derive from the distribution logic and, therefore reflect business channels like ATMs, bank branches, etc.

That is to say, for the distribution logic of a withdrawal, what is important is not if the customer has a VIP-contract with the account, but whether the ATM at which the request for the withdrawal is made has enough cash in store and is in touch with the branch in which the account is held. The composition logic will determine whether the withdrawal can proceed according to the relationship that exists between the customer and the account, whereas the distribution law will determine how much money can be given according to the context in which the transaction is being made (cash available at the ATM and status of the communication between the ATM and the branch).

Just like with coordination laws, locations laws are associated with business activities within a process, not with the process as a whole. This is because we want to allow for business entities to change location during the process. For instance, we may well envisage an instantiation of the banking process in which the customer is a mobile entity that starts the process and performs some activities through a PDA while driving to the bank where, upon arrival, he continues by performing other activities until he eventually finishes the process over the internet in his office where he needed to retrieve information that he was lacking at the bank. The modelling of this kind of mobility within a business process is still under active research and will not be further discussed in the paper. See [4,19] for the mathematical domain over which we are defining these aspects and early insights on how to use them.

Requirements on the location of the distribution partner is an obligatory feature in every location interface. These requirements consist in the definition of the type of the location as a subtype of *loc*, including any relevant functions and properties. For instance, if a location is required to handle high-precision calculations, its type needs to be such that, upon instantiation, service operations are executed on hardware that complies with the required properties. Security requirements may be reflected in other properties and functions on the data that is transmitted.

```
location interface ATMW-LI
location type ATM
operations
    default(),cash():money
    acco():ACCOUNT
    give(n:money) post cash() = old cash()-n
events     withdraw(n:money)
end interface
```

The event that is being required is self-evident and, as we shall see in the next section, refers to the business activity for which we have already defined coordination laws. When this interface is instantiated, this event can be refined in many different ways depending on the actual machine at which the business activity is being performed: the pressing of a button in the keyboard, the filling of a menu on the screen, etc. The parameter of the event will also need to be provided on instantiation.

The ATM is required to make available two services: the amount of cash available inside the machine and the default maximum amount that the machine gives if there is no connection to the account. The ATM service is also required to make available the number of the account that is currently being serviced. This data will have been stored upon identification through the ATM card. We will see in the next section that location (and coordination) interfaces are instantiated in run-time to services that may be running, i.e. instantiation does not mean creation of a service. In the case at hand, the instance of ATM will be the service that will have been running when the ATM was "switched on" and that will have accepted and authenticated the card involved in the first activity of the specific banking process at stake.

The location interface that applies to the bank is as follows:

```
location interface rBANKW-LI
location type BANK
operations internal(n:money; a:ACCOUNT)
           maxatm(a:ACCOUNT): money
end interface
```

That is, the bank is required to be make available, for every account, the maximum amount that can be debited from an ATM, as well as accommodate executions of withdrawals internally. This is because we want to be able to move withdrawals to the bank when they are requested at the ATM and there is no communication between the two locations.

These two location interfaces are brought together in the location law that defines the distribution logic of the withdrawal activity when performed at an ATM:

```
location law ATMW-LL
locations  bank: rBANKW-LI; atm: ATMW-LI
rules
when  atm.withdraw(n) &
      BT(atm,bank)
      with n ≤ bank.maxatm(atm.acco()) &
           n ≤ atm.cash()
      do   atm.give(n)
when  atm.withdraw(n) & ¬BT(atm,bank)&
           REACH(atm,bank)
      let  N=min(atm.default(),n) in
      with N ≤ atm.cash()
      do   atm.give(N)
      mv   bank.internal(N, atm.acco())
end law
```

As in coordination laws, location laws declare a number of partners (called locations) and their interfaces. The ECA rules that we use for describing the distribution logic in location laws differ from the ones used in coordination laws because the composition logic does not require the communication and reachability status to be taken into account. On the contrary, in location laws, we need to take into account the properties of the context in which the trigger occurs, the condition needs to be evaluated, and the action needs to be performed.

Indeed, as neither the presence nor the quality of communication can be taken for granted in location-aware business components, we have to take explicit account of the communication status between any involved interfaces using their locations. For instance, depending on whether given locations are in touch, either a full composition of operations is performed across all locations involved thus synchronising the services in execution at these locations, or just a composition of the operations available at the location where the trigger is perceived can be performed.

This dependency is made explicit through the use of BT. In the location law, two different rules are considered depending on whether the two locations are in touch when the request for the withdrawal is detected. Notice that the distinction is made at the level of the trigger (the event of the ECA), not the guard (condition). This is because each case needs to be treated differently, in particular through different guards: when BT holds, the guard concerns upholding the maximum withdrawal permitted by the bank at an ATM whereas, when BT does not hold, it is the maximum allowed by the ATM itself that needs to be upheld.

The fact that two locations are not "in touch" (BT) does not mean that one cannot be reached from the other (REACH). Reachability allows for mobility of services, namely for service execution to be moved to other locations as an instance of another service. In the case that concerns us, even in the absence of communication with the bank, ATMs can provide a limited amount of cash as long as there is a protocol with the bank for remote/delayed transmission of the corresponding withdrawal. The operations that continue the execution of the activity at a different location are declared under *mv* whereas those that are executed locally are identified under *do* as usual.

Notice that what is being moved for execution at the bank concerns a full withdrawal service, not the elementary debit operation that we discussed in the previous section. Indeed, the required service needs to be executed in the right context, which means taking into account the coordination and location rules that apply, internally at the bank, to that specific client and account. The way the service is moved from the ATM to the bank is left unspecified: it should be handled at the level of the definition of the location types, namely the topology of movement that applies. In the case of current Web services, these are rather trivial situations as reachability is, once again, handled at the level of network addresses. In our example, this movement can be just the storage of a request until communication becomes available (lightweight mobility), or the print out of instructions that are delivered in hand at the bank and executed on arrival at the end of the day (strong

snail mobility), just to name a few and stress that we are modelling services that are not necessarily deployed over the Web!

As discussed in the next section, the transaction to be executed may involve whatever operations are required by the composition logic through the coordination rules that react to the same trigger. Indeed, the location rules above are not concerned with the contracts that the customer has with the bank with respect to withdrawals from the specific account that is involved as a partner, just as the coordination rules discussed in the previous section were not concerned with distribution. This separation of concerns is, precisely, what the paper aims to explain.

Before we discuss the integration of separately modelled concerns, consider a few more examples that illustrate other situations. For instance, consider the situation in which the request for the withdrawal is made at a branch of the bank, although not necessarily the one in which the account is held. We still need two location interfaces because two locations are involved:

```
location interface BRW-LI
location type BANK
operations
    cash():money
    give(n:money) post cash() = old cash()-n
events     withdraw(n:money; a:ACCOUNT)
end interface

location interface BANKW-LI
location type BANK
end interface
```

In this case, nothing is required of the bank location that concerns the distribution logic; only the coordination rules will apply as discussed in the next section. This becomes evident in the location law itself:

```
location law BRW-LL
locations  bank: BANKW-LI; branch: BRW-LI
rules
when  branch.withdraw(n,a) &
      BT(branch,bank)
      with n ≤ branch.cash()
      do   branch.give(n)
end law
```

In this case, there is no location rule for the situation in which the branch is not in touch with the "bank", i.e. with the location in which the account is held. This means that, in those circumstances, the request for the withdrawal is not recognised, i.e. does not constitute a trigger (the clerk at the branch just says "sorry: the system is down again"...)

Consider now a different business activity – identification. At an ATM, two locations are involved: the ATM itself and the card.

```
location interface ATMId-LI
location type ATM
operations
    acco():ACCOUNT;
    cust():CUSTOMER;
    accept(c:CARD) post acco()=ac(c) & cust()=ct(c)
events     enter(n:PIN)
end interface

location interface CARD-LI
location type CARD
operations attempts():nat
           code():PIN
           reject post attempts() = old attempts()+1
           accept post attempts()=0
end interface
```

The interface for the ATM detects the entering of a pin number. as an event. As elementary services, it involves the acceptance of a card, which implies retrieving from the card the identities of the account and the customer. This is done through operations *ac:CARD→ACCOUNT* and *ct:CARD→CUSTOMER* available at the level of the data types provided as part of the underlying business model. On the side of the card, elementary operations handle attempts at guessing the code that is stored.

The corresponding location law is pretty intuitive:

```
location law ATMId-LL
locations  atm: ATMId-LI; card: CARD-LI
rules
when  enter(n) &
      BT(atm,card)
      with card.attempts() ≤ 3
      do   if n = card.code()
           then card.accept() &
                atm.accept(card)
           else card.reject()
end law
```

Notice that, in this case, BT means that the ATM is able to recognise the card and, hence, "communicate" with it, namely to extract information from it as done through the action *accept*. If the card is not recognised, then the trigger is not recognised either and the evaluation of the guard is not even attempted.

5. INTEGRATION OF CONCERNS

So far we proposed a set of semantic primitives through which we can separate two different concerns in business modelling: the coordination mechanisms that should be put in place to compose services (composition logic or layer) and the location-aware aspects that handle the dependency on the business channels across which services are distributed (distribution logic or layer).

This separation of concerns seems to be rather intuitive. As a business activity, a withdrawal from a bank account should involve a number of partners that execute required services in a coordinated way, i.e. according to certain logic, regardless of where they are located. For instance, the use of a credit facility is part of a business contract between the customer and the bank regardless of the channel through which withdrawals are made. Likewise, the limitations that the absence of communication between an ATM and a bank imposes on the activity is independent of the existence of a credit allowance.

This is why it is important to support this separation of concerns at the level of business modelling. On the one hand, each dimension can be refined independently of the other. On the other hand, changes in one dimension can be done without interfering with decisions made in the other.

Being able to model these concerns separately does not mean that they are independent. The way a business activity is performed within a process system emerges from the coordination and location laws that jointly apply to that activity. In this section, we discuss this mechanism of emergence, i.e. we are concerned with the away both concerns get integrated in a model of the business activity as it ends up being executed.

As an example, consider the withdrawal once again. At run-time, the way the withdrawal is processed is determined not by independent partners and locations but by *located partners*: for instance, *cust@atm* and *acco@bank*. That is, both coordination and location interfaces need to be instantiated by the same run-time services. In particular, because the ATM component identifies a

customer and an account, we have *cust=atm.cust()* and *acco=atm.acco()*, i.e. a single customer service and a single account service. This makes it clear that the business partner that is involved in the activity is not necessarily the person standing in front of the ATM but the customer identified in the card.

To be more precise, the instantiation of the coordination and location laws means binding the coordination and location interfaces to services that are running on the current system configuration. Hence, in the case of a withdrawal, we will have services running: one that binds *cust* and *atm*; the other binds *acco* and *bank*.

As already mentioned, these services are not necessarily disjoint or independent, and they are not necessarily created upon instantiation. For instance, as discussed in section 3, *acco* may be a service running autonomously within *bank*. On the other hand, the ATM service *atm* will have started when the ATM was switched on; when the binding of the location interface *ATMW-LI* takes place, it will have a context in which *atm.acco()* and *atm.cust()* will hold the identities of the account and customer to which the withdrawal applies. This is because, through the location law *ATMId-LL*, this data will have been retrieved from the card during the identification activity. Moreover, the binding also establishes that the value of *cust.owns(acco)* is *true*. Notice that, at a branch, the binding of *cust* would not necessarily establish this equality: in the case of the ATM, it is the use of the card that authenticates the pair *(cust.acco)*. This is another reason in support of making business processes location-aware.

The way a process activity like a withdrawal interacts with these services in described in the coordination and location rules according to the events that are detected in the run-time configuration. For instance, the event that triggers the withdrawal business activity instantiates as *atm.withdraw(n)* in the location interface and *cust.withdraw(n,acco)* in the coordination interface Assuming that the coordination law that is active in the run-time configuration is *SW-CL* (see section 3), the occurrence of the event is subject to the following rules:

```
when  cust.withdraw(n,acco)
      with acco.balance() ≥ n &
           cust.owns(acco)
      do   acco.debit(n)

when  atm.withdraw(n) & BT(atm,bank)
      with n ≤ bank.maxatm(atm.acco()) &
           n ≤ atm.cash()
      do   atm.give(n)

when  atm.withdraw(n) & ¬BT(atm,bank)& REACH(atm,bank)
      let  N=min(atm.default(),n) in
      with N ≤ atm.cash()
      do   atm.give(N)
      mv   bank.internal(N,atm.acco())
```

The joint execution of ECA rules that we have in mind, as formalised in [14], takes the conjunction of the guards and the parallel composition of the actions (i.e. the union of the corresponding synchronisation sets) when BT holds. When the located partners are not in touch, i.e. cannot communicate, the coordination rules do not apply. As a result, the rules according to which a withdrawal is performed are:

```
when  atm.withdraw(n) & BT(atm,bank)
      with n ≤ acco.balance() &
           n ≤ bank.maxatm(acco) &
           n ≤ atm.cash()
      do   atm.give(n)  &
           acco.debit(n)

when  atm.withdraw(n) & ¬BT(atm,bank)& REACH(atm,bank)
      let  N=min(atm.default(),n) in
      with N ≤ atm.cash()
      do   atm.give(N)
      mv   bank.internal(N,acco)
```

That is, when the ATM is in communication with the bank, the withdrawal is performed according to the coordination rule of a standard withdrawal and the location rule of the ATM. Notice, however, that *cust.owns(acco)* holds as a result of the binding and, hence, was omitted from the "with" condition. The need for communication is obvious in the guard condition, which requires the balance of the account to be checked and the action, which requires the account to be debited. In the case of the joint execution of the guard, BT is necessary to ensure synchronous, atomic execution of the reaction. Notice that synchronous execution does not involve REACH because the service is not being moved from one location to another: both services are executed, each in its location, but atomically, which is what requires communication. Naturally, this semantics requires a proper distributed transaction management system to be in place. See [20] for transaction protocols in the scope of Web services.

Summarising, as claimed in section 2, our approach is activity-oriented in the sense that, for each activity within a business process, we identify which are the location and coordination concerns that apply to the business entities involved, and how they are put together to enforce the business process logic (e.g. the activity ordering). In general, there is a 0-N correspondence between each business process activity and coordination / location laws. That is, depending on the semantics of each activity, we may have no coordination laws (which is the case of identification in the example) or one or more coordination laws (case of withdrawals); and the same for location laws.

We have to emphasize that, depending on the business entities involved in a specific activity, not every law applies at each configuration. Determining which laws should apply and, for those that apply, how the business entities instantiate the interfaces (location and coordination), and how the corresponding instantiated coordination and location laws bind the entities together with contracts, is out of the scope of this paper. See [5,6] for configuration management primitives that apply to coordination laws. In what concerns location laws, we are now developing similar configuration primitives.

6. CONCLUDING REMARKS

In this paper, we discussed a service-oriented architectural-based approach that addresses current challenges in modern business process modelling for reflecting dynamic cross- and intra-organisational interactions as well as dependencies on the business channels and networks over which organisations operate. Our approach is inspired in the rich set of specifications that is currently available for software development over Web services, i.e. "software that can process XML documents it receives through some combination of transport and application protocols" [31]. Languages and techniques as made available by BPEL4WS [9], WS-Coordination [35] and WS-Transaction [36], *inter alia*, remain too close on this narrow view of services that need to be located and invoked over the Web using addresses and referencing mechanisms that identify where services can be found using a given protocol like TCP or HTTP. As a consequence, they offer little support to the higher-levels of abstraction in which business rules and organisational infrastructures need to be modelled.

This is why we decided to distance ourselves from both the XML-centred view of information exchange, and the Web-oriented notions of location and reference protocols. Our proposal addresses a rule-based approach to business modelling and addresses a space in which locations correspond to business entities and channels organised according to a given organisational communication and distribution network.

The semantic primitives that we proposed for business modelling capture structural features of architectural connectors in separating concerns and addressing business rules as first-class entities. Following our approach, the aspects that relate to the way business rules determine how the services involved in a business activity need to be orchestrated fall under what we call "coordination laws". These are semantic primitives that are used for modelling the "service composition layer" of service-oriented architectures or, for short, their "composition logic".

In what concerns the "distribution logic" that captures the dependency on the business channels and networks (e.g. properties of the computational platform and communication network, mobility of devices/sensors, inter alia), we proposed a similar approach based on explicit connectors we called location laws. As with coordination laws, these connectors can be superposed dynamically and evolved independently of the other business aspects, allowing systems to self-adapt or be adapted to changes that occur at the distribution level without interfering with the core business policies.

The semantics of both the composition and distribution logic, and of coordination and location laws, builds on recent work around CommUnity, a formal approach that we have been developing for architectural description [14]. CommUnity includes primitives that capture distribution and mobility aspects [19], and explicitly separate between components computation, coordination and distribution/mobility. Besides recently forwarded operational semantics—including graph transformations, Tile and rewriting logic—the main strength of CommUnity lies in its logic of interactions, which is based on Category Theory [13]. CommUnity is also endowed with a software tool for editing, simulating and validating distributed software architectures. Extensions of CommUnity towards context-aware computing are now being explored that will further enrich this architectural approach.

We are currently working on more case studies in order to consolidate and validate this service-oriented architectural approach. We are also collaborating with ATX Software, the IT company with whom we developed the Coordination primitives, on the methodological aspects of location laws; one of our main goals is to develop a deeper understanding and classification of business rules so that semi-automatic derivation of coordination and location laws can be ultimately achieved. In this sense, the work forwarded in [26] on classifying Web Services-oriented rules could be a significant input for us. Last but not least, extensions to modelling languages like the UML with coordination and distribution laws are also being investigated at Leicester.

Acknowledgements

N..Aoumeur was supported by the European Commission through the contract IST-2001-32747 (AGILE: Architectures for Mobility). C.Oliveira was supported by Fundação para a Ciência e Tecnologia, Portugal, through the PhD Scholarship SFRH/BD/6241/2001, and the European Science Foundation through the Scientific Network RELEASE. The authors would like to thank P.Kosiuczenko and A.Lopes for many insights and suggestions on the work reported in this paper.

7. REFERENCES

1. W.Aalst, A.T.Hofstede and M.Weske, "Business Process Management: A Survey", in *International Conference on Business Process Management (BPM 2003),* LNCS 2678, Springer 2003, 1-12.
2. L.Abom, "Frameworking RM-ODP in Banking", in A.M.Cordeiro and H.Kilov (eds) *WOODPECKER 2001*, ICEIS Press 2001.
3. R.Allen and D.Garlan, "A Formal Basis for Architectural Connectors", *ACM TOSEM*, 6(3), 1997, 213-249.
4. L.Andrade, J.L.Fiadeiro, A.Lopes and M.Wermelinger, "Coordination for Distributed Business Systems", in *Information Systems for a Connected Society*, J.Eder, R.Mittermeir and B.Pernici (eds), University of Maribor Press 2003, 27-37.
5. L.F.Andrade and J.L.Fiadeiro, "Service-Oriented Business and System Specification: Beyond Object-orientation", in H.Kilov and K.Baclwaski (eds), *Practical Foundations of Business and System Specifications*, Kluwer Academic Publishers 2003, 1-23.
6. L.F.Andrade and J.L.Fiadeiro, "Composition Contracts for Service Interaction", *Journal of Universal Computer Science*, in print.
7. A Baina, S. Tata, and K. Benali, "A Model for Process Service Interaction", in *International Conference on Business Process Management (BPM 2003),* LNCS 2678, Springer 2003, 261-275.
8. BizTalk Orchestration – a new technology for orchestrating business interactions, Microsoft Research 2000.
9. Business Process Execution Language for Web Services, version 1.1, May 2003, IBM
10. L.Cardelli and A.Gordon, "Mobile Ambients", in Nivat (ed), *FoSSACs'98*, LNCS 1378, 140-155, Springer-, 1998.
11. F.Curbera, R.Khalaf, N.Mukhi, S.Tai and S.Weerewarana, "The Next Step in Web Services", in [27], 41-47.
12. T.Elrad, R.Filman and A.Bader (Guest editors). Special Issue on Aspect Oriented Programming. *Communications of the ACM* 44(10) 2001.
13. J.L.Fiadeiro, *Categories for Software Engineering*, Springer 2004.
14. J.L.Fiadeiro, A.Lopes and M.Wermelinger, "A Mathematical Semantics for Architectural Connectors", in *Generic Programming*, R.Backhouse and J.Gibbons (eds), LNCS 2793, Springer 2003, 190-234.
15. P.Kardasis and P.Loucopoulos, "Expressing and Organising Business Rules", *Information and Software Technology*, in press.
16. S.Katz, "A Superimposition Control Construct for Distributed Systems", *ACM TOPLAS* 15(2), 1993, 337-356.
17. Z.Kleppe, J.Warmer and W.Bast, *MDA Explained: The Model Driven Architecture--Practice and Promise*, Addison-Wesley 2003.
18. A.Lindsay, D.Downs and K.Dunn, "Business Processes – attempts to find a definition", *Information and Software Technology* 45(1):1015-1019, 2003.
19. A.Lopes and J.L.Fiadeiro, "On how Distribution and Mobility interfere with Coordination", in *Recent Trends in Alge-*

braic Development Techniques, M.Wirsing, D.Pattinson, R.Hennicker (eds), LNCS 2755, Springer 2003, 343-358.

20. M.Little, "Transactions and Web Services", in [27], 49-54.
21. P.Loucopoulos, "The S3 (Strategy-Service-Support) Framework for Business Process Modelling", in *CAiSE Workshops – Information Systems for a Connected Society*, CEUR Workshop Proceedings vol. 75, Technical University of Aachen (RWTH), 2003.
22. J.Magee and J.Kramer, "Dynamic Structure in Software Architectures", in *4th Symp. on Foundations of Software Engineering*, ACM Press 1996, 3-14.
23. A.Maurino, B.Pernici and F.Schreiber, "Adaptive Channel Behavior in Financial Information Systems", in *CAiSE Workshops – Information Systems for a Connected Society*, CEUR Workshop Proceedings vol 75, Technical University of Aachen (RWTH), 2003.
24. G.Meredith and S.Bjorg, "Contracts and Types", in [27], 41-47.
25. B.Orrinsi, J.Yang, and M.Papazoglou, "A Framework for Business Rule Driven Web Service Composition", in *Proc. of Conceptual Modeling for Novel Application Domains*, LNCS 2814 Springer 2003, 52-64.
26. B.Orrinsi, J.Yang, and M.Papazoglou, "A Framework for Business Rule Driven Web Service Composition", in *Proc. of Conceptual Modeling for Novel Application Domains*, LNCS 2814 Springer 2003, 52-64.
27. M.Papazoglou and D.Georgakopoulos (guest editors), Special Issue on Service-Oriented Computing, *Communications of the ACM* 46(10), 2003.
28. G.-C.Roman, C.Julien and J.Payton, "A Formal Treatment of Context-Awareness", *Proc. FASE 2004*, LNCS 2984, 12-36, Springer-Verlag, 2004
29. D.Rosca and C.Wild, "Towards a Flexible Deployment of Business Rules", *Expert Systems with Applications* 23:385--394, 2002.
30. M.Shaw, "Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status", in D.A. Lamb (Ed.), *Studies of Software Design,* LNCS 1078, Springer 1996.
31. W.Vogel, "Web Services Are Not Distributed Objects", *IEEE Internet Computing* 2003.
32. J.Yang, "Web Service Componentization", in [27], 35-40.
33. W.Wan-Kadir and P.Loucopoulos, "Relating Evolving Business Rules to Software Design", *Journal of Systems Architecture*, 2003.
34. *Web Services architecture overview – the next stage of evolution for e-business,* September 2000, http://www.ibm.com/developerworks/web/library/w-ovr/
35. *Web Services Coordination, version 1.0,* http://www.ibm.com/developerworks/web/library/ws-coor/
36. *Web Services Transaction, version 1.0,* http://www.ibm.com/developerworks/web/library/ws-transpec/