31 AUTOMATING PARTNER SELECTION FOR A VIRTUAL ORGANIZATION

Tomasz Janowski♦, Liu Yonghe* and Huang Biqing*
The United Nations University, Macau

Establishing a virtual organization requires not only being able to identify individual member enterprises but also prescribe the roles they should play in the organization. This becomes a difficult task when selection is not only based on static competencies but also on how the enterprise is able to dynamically deliver services to its clients, within the area of its competence. In this paper we present a simple model which explains the behavior of a virtual organization in terms of the services it can offer to and receive from its environment, by means of the services by its members and service-based interactions between them. We also describe how the model could be used to define and automate the problem of partner selection: produce an architecture (members and their roles) to deliver the required service.

INTRODUCTION

Establishing a virtual organization requires the basic infrastructure for communication and information sharing between individual enterprises to be already in place, not unlike for other business applications or distributed computing at large. While this level of support is necessary it is also insufficient; the added value comes from the higher-level functions able to address specific problems facing a virtual organization. One of such problems is partner selection.

Instead of relying only on the known companies we would like to actively seek information about potential partners on the Internet, taking into account not only the static "competence" of an enterprise but how it is able to dynamically deliver services to its clients, within the area of its competence. The following problems exist when we use for this purpose existing Internet search tools: (1) it is not clear which keywords we should use to describe what we look for, (2) the search is likely to produce an abundance of largely irrelevant and out-of-date data, (3) it is unclear

♦ The United Nations University, International Institute for Software Technology, P.O. Box 3058, Macau, e-mail: tj@iist.unu.edu
* From the CIMS Research and Engineering Center at Tsinghua University, Beijing, China

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35577-1_37

L. M. Camarinha-Matos et al. (eds.), *Infrastructures for Virtual Enterprises*

which criteria we should apply to such data and (4) the data gives little indication as to the roles different partners could play in the organization. Relying on the general-purpose support, here an Internet search engine means we also have to rely on human assistance to interpret the low-level results. In this paper we aim to study partner selection as an infrastructure support for a virtual organization, reducing the role of human users to decision-making, given possible partners and the roles they could play in the organization, instead of interpreting raw data.

In order for this support to be possible we have to state very clearly what is the goal of the organization, the roles of its members contributing towards this goal and possible interaction between them. Not only description of the goals should be machine-readable but carry enough semantic information to allow software to make decisions without asking for human assistance, to compare what the organization needs and what the potential member has to offer. We adopt a model that explains the behavior of a virtual organization in terms of the services it can offer to and receive from its environment, by means of the services by individual members and service-based interactions between them. The concept of a service includes a range of possible interactions between the entities in the marketplace - suppliers, manufacturers, retailers, logistics providers, and stockholders - all potential members of the organization. For example: delivery of a certain number of sub-products from the supplier to the manufacturer (supplier is a service provider, manufacturer is a service consumer), transport of products between the manufacturer and the retailer (provided by a transportation company), marketing of a product for the customers living in a certain area (provided by a marketing company) etc. We can describe the services atomically, built them from some other services and compare services with an ordering which decides if one service is at least as "good" as another one. We treat all entities in the marketplace as both providers and consumers of services, often at the same time: a service describes what the entity requires from its environment and what it can deliver when it receives what it needs. This conditional delivery of services provides an external view on individual entities, which is all we need to know for partner selection.

A virtual organization delivers its services using the services offered by its members and service-based interactions between them. Partner selection starts with a service the organization should deliver to its customers. If we can find an entity in the marketplace which can solely deliver this service then the search is finished. In general, however, the entity will only deliver the service if provided with some other services. Then we continue looking for the entities which are able to deliver those auxiliary services, and so on. With each iteration we build the links between the entities in the marketplace (supply-chains using the product analogy), deciding on the members of the virtual organization and the roles they should play (services). The search is finished when the organization is self-contained and does not require any more services from its environment, or on purpose decides to rely on the environment to deliver some of them. Such external services can be determined at the start or during the process of selection. The result is a possible architecture (members and their roles) of the virtual organization to deliver the required service.

The rest of the paper is organized as follows. Section 2 presents the service-based model for a virtual organization. Section 3 illustrates the model by a simple example. Section 4 applies the model to formulate the problem of partner selection. Section 5 provides discussion, comparisons and directions for our future work.

SERVICE-BASED MODEL

We introduce in this section a service-based model for a virtual organization. The aim is to formulate the problem of partner selection. The model is given in a formal notation of RAISE: Rigorous Approach to Industrial Software Engineering (RAISE, 1992) (RAISE, 1995). In the sequel we introduce different elements of the model: services, service-providers, service-based interactions and service composition.

Service

A service represents what the entity in the marketplace requires for carrying out its business activities and what such activities can deliver. At this stage we are not interested in the details of a service, therefore we introduce an abstract type `Service` which values represent possible services. In RSL, a type represents a collection of values together with some operations on them.

```
type
  Service
```

In addition to describing services we would like to be able to match them: one service represents what is needed by an entity in the marketplace and another service represents what some other entity can provide. How to decide if those two services match? This may differ between cases, but in general we should be able to provide an ordering between services which represents this matching: if $s \leq t$ then t is matching the service required by s. We require that the service ordering is reflexive (any service satisfies itself), anti-symmetric (it two services satisfy each other then they are equal) and transitive (if one service satisfies the second and the second satisfies the third then also the first satisfies the third). We define the ordering as a Boolean function $\leq$ on the pairs of services, as below.

```
value
  ≤ : Service × Service → Bool
axiom
  ∀ s : Service • s ≤ s,
  ∀ s, t : Service • s ≤ t ∧ t ≤ s ⇒ s = t,
  ∀ s, t, u : Service • s ≤ t ∧ t ≤ u ⇒ s ≤ u
```

We also assume two special values of the type `Service` representing the upper and the lower bounds of the service ordering: `top` and `bottom`. `bottom` is the least demanding service which can be satisfied by any service whatsoever, `top` is the most demanding service that is only satisfied by itself.

```
value
  top : Service • (∀ s : Service • s ≤ top),
  bottom : Service • (∀ s : Service • bottom ≤ s)
```

Service Provider

We treat each entity in the marketplace as a possible service provider. We are not interested in the details of how the delivery of a service takes place, so we define a provider as a value of an abstract type `Provider`.

```
type
  Provider
```

Values of this type will represent service providers in a particular state, which state can change over time. In general, the entity can only provide a given service conditionally, given that it can receive some other services from the environment. Function `provides` represents static provision of services. It takes three arguments: the input service, the provider and the output service, and returns `true` or `false` depending if the provider is able to deliver an output service when given the input service. We constrain this function with respect to the ordering on services. One axiom says that any provider can deliver an output service given an input service which is at least as strong. In this situation no new service is produced, the entity delivers the service produced elsewhere without any change. Another axiom allows strengthening the input service and weakening the output service.

```
value
  provides: Service × Provider × Service → Bool
axiom
  (∀ p : Provider, s, t : Service •
    s ≥ t ⇒ provides(s,p,t) ∧
    provides(s,p,t) ⇒ (∀ s',t' : Service • s ≤ s' ∧ t' ≤ t ⇒ provides(s',p,t'))
  )
```

Some special cases of this function include: `provides(s,p,s)`, given `s` as input `p` can produce `s` itself; `provides(top,p,s)`, given `top` as input `p` can deliver any service `s`; `provides(bottom,p,s)`, `p` can deliver `s` on its own. Function `provides` takes into account only static provisions for services, for a given state of the provider. But this state can change when services are delivered. The function `trans` carries out this change of state. It takes the same arguments as function `provides` but returns the new provider. The function is partial in the sense that the result is only meaningful for those arguments for which `provides` returns `true` i.e. the delivery of a service is possible in a given state.

```
value
  trans: Service × Provider × Service ~→ Provider
  trans(s,p,t) as p'
    post true
    pre provides(s,p,t)
```

Service-Based Interactions

Given the definitions in the last section it is possible that the provider can itself deliver services which support some of its other services, becoming also a service consumer. This represents the level of being self-sufficient for the service provider, beyond which it certainly needs input from its environment. This maximum of the service delivery is represented by the function `provides_star`: if `provides_star(s,p,t)` then either `provides(s,p,t)` directly or there exists an intermediate service `s'` which can be output by `p` after receiving `s` and such that `t` can be delivered in the new state `trans(s,p,s')`.

```
value
  provides_star: Service × Provider × Service ~→ Bool
  provides_star(s,p,t) ≡
    provides(s,p,t) ∨
    (∃ s':Service • provides_star(s,p,s') ∧ provides_star(s',trans(s,p,s'),t))
```

In general, a service produced by one provider will be consumed by another. Suppose we have a set of service providers and we would like to describe how they behave together, by delivering services individually and also interacting with each other. We first introduce the function `virtual` which given a set of service providers produces a single one.

```
value
  virtual: Provider-set → Provider
```

Instead of defining this function explicitly, we define implicitly the behavior of the resulting service provider in terms of the functions `provides` and `trans`. The first axiom says that any service which can be delivered by a single provider can be also delivered by the virtual provider, i.e. if `provides(s,p,t)` then also `provides(s,virtual(ps),t)` given that `p` ∈ `ps`. Only the state of this one provider will change, i.e. `trans(s,virtual(ps),t)` equals the result of `virtual` applied to `ps` where instead of `p` we have `trans(s,p,t)`.

```
axiom
  (∀ ps:Provider-set, p,p':Provider, s,t:Service •
    p ∈ ps ∧ provides(s,p,t) ⇒
    provides(s,virtual(ps),t) ∧
    trans(s,virtual(ps),t)=virtual(ps \ {p} ∪ {trans(s,p,t)})
  )
```

The second axiom includes the possibility for two providers to actually interact. If one can deliver a service which another can consume, then the interaction between them becomes possible. Their states will change simultaneously, taken non-deterministically for each one but not affecting the rest.

axiom
(∀ ps:Provider-**set**, p,q:Provider, s,t:Service •
(∃ s':Service •
p ∈ ps ∧ q ∈ ps ∧ provides(s,p,s') ∧ provides(s',q,t)
⇒ provides(s,virtual(ps),t) ∧
trans(s,virtual(ps),t)=virtual(ps \ {p,q} ∪ { trans(s,p,s'), trans(s',q,t) })
)
)

We treat a virtual organization as a set of interacting service providers. Function `virtual` represents the abstracted behavior of the organization which on outside appears like a single provider. One obvious advantage is to minimize the number of modeling concepts. But more important is allowing for different levels of abstraction to describe the organization: on the concrete level the organization is a set of interacting service providers, on the abstract level it is an entity which can interact with its environment by delivering/consuming services. This in turn allows us to justify design of a virtual organization with respect to its abstract specification, using refinement. We say that provider `q` refines provider `p`, `refines(p,q)`, iff `provides(s,p,t)` for any `s` and `t` implies `provides(s',q,t')` for an `s'` which is not stronger than `s` and `t'` which is not weaker than `t`, i.e. `q` can deliver at least all services that `p` can while it requires no more from the environment. This property must be also preserved under state-changes, i.e. the new states again refine each other: `refine(trans(s,p,t),trans(s',q,t'))`.

value
refine: Provider × Provider → **Bool**
refine(p,q) ≡
(∀ s,t:Service • provides(s,p,t) ⇒
(∃ s',t':Service •
s' ≤ s ∧ t ≤ t' ∧ provides(s',q,t') ∧ refine(trans(s,p,t),trans(s',q,t'))))
)

Given the required behavior described by the provider `p`, the set `ps` of providers correctly implement this behavior iff `refine(p,virtual(ps))`.

Composition of Services

In this section we extend the model introduced so far by putting more structure on the type `Service`. We do so by service composition: concurrent, alternative and sequential, all defined as binary functions:

value
con,alt,seq: Service × Service → Service

Composition `con(s,t)` describes that `s` and `t` are present simultaneously, `alt(s,t)` that only one of `s` or `t` is present but we don't know which one,

`seq(s,t)` that service `t` is delivered after `s`. We introduce some axioms which express requirements imposed by those interpretations, by equality over services and also using functions `provides` and `trans`.

Consider concurrent composition. The order in which we compose services concurrently is irrelevant, i.e. `con` is both commutative and associative. This justifies an extension of `con` into `ext_con` which takes a set of services instead of just two; the result for an empty set of services is `bottom`. Composing concurrently `s` with `bottom` gives `s`. Also `con` gives a stronger service than those described by its components, i.e. `s` $\leq$ `con(s,t)` and `t` $\leq$ `con(s,t)`.

axiom
(∀ s,t,u:Service •
s ≤ con(s,t) ∧ con(bottom,s) = s ∧
con(s,t)=con(t,s) ∧ con(s,con(t,u))=con(con(s,t),u)
)
value
ext_con: Service-**set** → Service

If `p` can deliver concurrently `t1 and t2` then it should also be able to deliver them individually, for the same input. But not the other way round: despite being able to deliver `t1` and `t2` individually, `p` may not be able to deliver them simultaneously. This becomes possible when we have separate entities that can simultaneously consume/deliver services, as in the virtual organization. In particular if `provides(s1,p1,t1)` and `provides(s2,p2,t2)` and if `p1` and `p2` are different then the composition of `p1` and `p2` can deliver concurrently `con(t1,t2)` when provided concurrently with `con(s1,s2)`.

axiom
(∀ p:Provider, s,t1,t2:Service •
provides(s,p,con(t1,t2)) ⇒ provides(s,p,t1) ∧ provides(s,p,t2)
),
(∀ p1,p2:Provider, ps:Provider-**set**, s1,s2,t1,t2 •
p1 ∈ ps ∧ provides(s1,p1,t1) ∧
p2 ∈ ps ∧ provides(s2,p2,t2) ∧ p1 ≠ p2 ⇒
provides(con(s1,s2),virtual(ps),con(t1,t2))
)

Concurrent services make also possible interactions between more than two providers: if `p` can deliver `t` given concurrently `con(s1,s2)`, if `p1` and `p2` can deliver separately `s1` and `s2` given respectively `t1` and `t2`, then their composition can deliver `t` given concurrently `con(t1,t2)`.

Consider alternative composition. Like concurrency, the order in which we compose services is irrelevant, i.e. `alt` is both commutative and associative. Unlike concurrency, `s` composed with itself gives `s`, i.e. `alt` is indepotent. Also concurrency gives a service which is equal or stronger than both of its components

while alternative composition gives a service which is weaker (or equal); we are not sure which of the two services will be eventually chosen.

axiom
(∀ s,t,u:Service •
alt(s,t) ≤ s ∧ alt(s,s) = s ∧
alt(s,t)=alt(t,s) ∧ alt(s,alt(t,u))=alt(alt(s,t),u)
)
value
ext_alt: Service-**set** → Service

Because `alt(s1,s2)` is equal or weaker than both `s1` and `s2`, if `p` can deliver `t` given alternatively `s1` or `s2` then it can also deliver `t` given only `s1` or only `s2`. The opposite also holds: if `p1` can deliver `t` given `s1` and `p2` can deliver `t` given `s2` then `p1` and `p2` can deliver `t` given alternatively `s1` or `s2`.

axiom
(∀ p:Provider, s,t1,t2:Service •
provides(alt(s1,s2),p,t) ⇒ provides(s1,p,t) ∧ provides(s2,p,t)
),
(∀ p1,p2:Provider, ps:Provider-set, s1,s2,t •
p1 ∈ ps ∧ provides(s1,p1,t) ∧ p2 ∈ ps ∧ provides(s2,p2,t) ⇒
provides(alt(s1,s2),virtual(ps),t)
)

Consider sequential composition. Service `seq(s,t)` is stronger than `s` itself (`s≤seq(s,t)`) and composition is associative but we cannot change the order in which `s` and `t` are delivered. Sequential delivery is inherently possible for every service provider, normally defined jointly by the functions `provides` and `trans`. Here we can also make it explicitly part of the service definition: if `p` can deliver `t1` given `s1` and in the new state deliver `t2` given `s2` then it can also deliver sequentially `seq(t1,t2)` given sequentially `seq(s1,s2)`.

axiom
(∀ s,t,u:Service • s ≤ seq(s,t) ∧ seq(s,seq(t,u))=seq(seq(s,t),u)),
(∀ p:Provider, s1,t1,s2,t2: Service •
provides(s1,p,t1) ∧ provides(s2,trans(s1,p,t1),t2) ⇒
provides(seq(s1,s2),p,seq(t1,t2))
)

Like concurrency, sequential composition makes possible the interactions between more than two service providers. In particular, different services that are part of sequential composition can be satisfied (consumed) independently by separate providers. This way the links between service providers extend over time, involving entities that are able to change their states and change their behavior accordingly.

MANUFACTURING EXAMPLE

Suppose a service means delivery of a certain quantity of a given product. We introduce the abstract type `Product` which values represent different products and a function `bill` from `Product` to maps from `Product` to `Nat` to represent how we can obtain products from sub-products. This map represents all sub-products of a given product and their quantities: how many items we need to assemble a single item of the final product. We constrain function `bill` by requiring that the quantity of each sub-product is non-zero and no product is a sub-product of itself.

```
type
  Product
value
  bill: Product → (Product m Nat) •
    (∀ p,q:Product • ~issub(p,p) ∧ q ∈ dom bill(p) ⇒ bill(p)(q)>0),
  issub: Product × Product → Bool
  issub(q,p) ≡ q ∈ dom bill(p) ∨ (∃ r:Product • issub(q,r) ∧ issub(r,p))
```

Let `Nat1` represent positive natural numbers. We define the type `Service` to include two kinds of values: `bottom` (no product is delivered) and `fin(p,n)` (n ≥ 1 items of product `p` are delivered). By underscore we mean that `Service` can also contain other values, to be defined later. The ordering `s≤t` is always true if `s=bottom`. If `s=fin(p,n)` then `s≤t` provided `t=fin(p,m)` for the same `p` and `m≥n`. It is easy to see that this definition satisfies the axioms of the partial order.

```
type
  Service == bottom | fin(prod:Product,val:Nat1) | _
value
  ≤ : Service × Service → Bool
  s ≤ t ≡
    case (s,t) of
      (bottom,_) → true,
      (fin(p,n),fin(q,m)) → p=q ∧ n≤m,
      _ → false
    end
```

Suppose a service provider contains the stock for various products and is able to manufacture products from sub-products. We introduce two functions on the type `Provider`: `shop` returns the set of products that can be manufactured on the shopfloor and `stock` returns the number of items of a given product on stock.

```
value
  shop: Provider → Product-set
  stock: Provider × Product → Nat
```

We also consider some operations on this model: `store` increments the stock for a given product, `deliver` decrements the stock and `manufacture` increments the stock for the product and decrements the stocks from all its sub-products, according to the bill. All take two arguments (service and provider) and accordingly change the state of the provider. Function `manufacture` takes two preconditions: the product can be manufactured and the stock contains all required sub-products.

```
value
  store, deliver, manufacture : Service × Provider ~→ Provider
```

Function `provides` takes the input service, provider and the output service and decides if the service can be delivered: either the output is `bottom` or the stock is at least equal the quantity required by the output service or we can manufacture the missing part from existing sub-products. Function `trans` carries out the corresponding state-change: stores the result of the input service and delivers the output service (which may involve manufacturing the missing part of products).

```
value
  provides: Service × Provider × Service → Bool
  provides(s,p,t) ≡
    t=bottom ∨ stock(store(s,p),prod(t)) ≥ val(t) ∨ prod(t) ∈ shop(p) ∧
    (∀ q : P.Product • q ∈ dom bill(prod(t)) ⇒
      stock(store(s,p),q) ≥ bill(prod(t))(q)*(val(t)-stock(store(s,p),prod(t)))
    ),
  trans: Service × Provider × Service ~→ Provider
  trans(s,p,t) ≡
    if t=bottom then store(s,p)
    else if stock(store(s,p),prod(t)) ≥ val(t) then deliver(t,store(s,p))
         else let p'=store(s,p), u=fin(prod(t),val(t)-stock(p',prod(t)))
              in deliver(t,manufacture(u,p')) end
         end
    end
```

Based on those functions we define function `virtual` which produces the behavior of the virtual organization by summing up the sets of products which can be manufactured on the shopfloor and the stocks for all products. The resulting function indeed satisfies the axioms expressing what services (product and quantity) can be delivered by the virtual organization and how the state will change as a result.

```
value
  virtual: Provider-set → Provider
axiom
  (∀ p:Provider, ps:Provider-set •
    shop(virtual(ps ∪ {p})) = shop(virtual(ps)) ∪ shop(p) ∧
    (∀ q:Product • stock(virtual(ps ∪ {p}),q) = stock(virtual(ps),q) + stock(p,q))
  )
```

AUTOMATING PARTNER SELECTION

Based on the model in Section 2 we can formulate the problem of partner selection. The problem takes three inputs: the service we want to deliver, the set of providers we can choose from and the service we wish those providers can rely upon; this service is assumed to be provided by the environment. Both services may in fact represent sets of services, put together by means of concurrent or other composition. The output determines which of those providers can deliver the required service while relying on the assumed service. In addition to selecting service providers we wish to determine the roles they should play in the delivery of the output service.

The simplest case is a single provider in the set able to solely deliver the required service when given the input service. Function `can_provide_1` decides if such a provider exists. Function `provider_1` returns any such provider. In this simple case the organization consists of only a single entity, which role is simply to deliver the output service when given the input service.

```
value
  can_provide_1: Service × Provider-set × Service → Bool
  can_provide_1(s,ps,t) ≡
    (∃ p:Provider • p ∈ ps ∧ provides(s,p,t)),
  provider_1: Service × Provider-set × Service ~→ Provider
  provider_1(s,ps,t) as p
    post p ∈ ps ∧ provides(s,p,t) pre can_provide_1(s,ps,t)
```

However, selection of a single entity that can deliver the required service on its own is not always possible. More often the combination of two or more entities can do the job which a single entity cannot. Below we consider the case of two providers. Function `can_provide_2` decides if we can find two providers in the set that together can deliver the required service. Function `provider_2` returns those two providers and a service which "links" them together. The first provider delivers this service when given the input service, the second delivers the output service while consuming the intermediate service. Their roles are set very precisely.

```
value
  can_provide_2: Service × Provider-set × Service → Bool
  can_provide_2(s,ps,t) ≡
    (∃ p1,p2:Provider, u:Service •
      {p1,p2} ⊆ ps ∧ provides(s,p1,u) ∧ provides(u,p2,t)),
  provider_2: Service × Provider-set × Service ~→ Provider × Service ×
Provider
  provider_2(s,ps,t) as (p1,u,p2)
    post {p1,p2} ⊆ ps ∧ provides(s,p1,u) ∧ provides(u,p2,t)
    pre can_provide_2(s,ps,t)
```

The resulting organization consists of exactly two entities. One step further is to consider a chain of providers such that the first provider consumes the input service, the last delivers the output service, and from the second onwards, every provider consumes the service the previous one delivers. Function `can_provide_chain` decides if such a chain exists and `provider_chain` returns the list of providers with corresponding output services. For instance `<(p1,u),(p2,t)>` is a two-member chain where `p1` delivers `u` given `s` and `p2` delivers `t` given `u`. We do not exclude that `p1` and `p2` denote the same provider, but this requires to take into account state-changes. The last service is always the final required service.

```
type
  Chain = (Provider × Service)-list
value
  can_provide_chain: Service × Provider-set × Service → Bool
  can_provide_chain(s,ps,t) ≡
    (∃ p:Provider, u:Service • p ∈ ps ∧ provides(s,p,u) ∧
      (t=u ∨ can_provide_chain(u,ps \ {p} ∪ {trans(s,p,u)},t))),
  provider_chain: Service × Provider-set × Service ~→ Chain
  provider_chain(s,ps,t) as pc post
    let (p,u)=hd pc in
      p ∈ ps ∧ provides(s,p,u) ∧
      if tl pc = <..> then t=u
      else tl pc = provider_chain(u,ps \ {p} ∪ {trans(s,p,u)},t) end
    end pre can_provide_chain(s,ps,t)
```

The resulting virtual organization consists in general of several entities that are linked sequentially. More complex configurations are also possible, say in the form of a tree which decides that a given provider receives input partly from the environment (if weaker than the given service) and partly from the providers in the set, possibly several of them. Each one can in turn rely on the inputs received partly from the environment and partly from other, one or more, providers. We can decide in the tree structure if the input service combines component services concurrently, sequentially or alternatively, with corresponding constraints on a single provider occurring in more than one place in the tree. For instance, if the provider occurs several times in a single branch then we should take into account the fact that its state will change, from the leaf towards the root. But the tree structure is not very appropriate if we are to allow the same provider to occur concurrently in several sub-trees, having to capture an arbitrary interleaving of its state-changes. A kind of graph structure, similar to transition systems, would have to be used instead. We plan to investigate such issues in a companion paper.

More generally, we may like to derive some rules to follow for carrying out partner selection. Suppose `t` is the output service, `s` the input service and `ps` is the set of providers we can choose from. We could make the whole selection task easier if we reduce this problem into the one where we receive stronger input, say $s \leq s'$, and are required to produce weaker output, say $t' \leq t$. This is described by the

functions `stronger_input` and `weaker_output` below. We could also combine those functions together.

```
value
  stronger_input: Service × Provider-set × Service ~→ Service × Provider
  stronger_input(s,ps,t) as (u,p)
    post p ∈ ps ∧ provides(s,p,u) ∧ s≤u
    pre (∃ p:Provider, u:Service • p ∈ ps ∧ provides(s,p,u) ∧ s≤u),
  weaker_output: Service × Provider-set × Service ~→ Provider × Service
  weaker_output(s,ps,t) as (p,u)
    post p ∈ ps ∧ provides(u,p,t) ∧ u≤t
    pre (∃ p:Provider, u:Service • p ∈ ps ∧ provides(u,p,t) ∧ u≤t)
```

Another methods of reduction is to look at the structure of the input/output service to see if we can decompose the problem into a number of simpler sub-problems. In particular, if the input service is built with alternative composition, say `s=alt(s1,s2)`, then the problem for `s` and `t` reduces into two problems: for `s1` and `t`, and `s2` and `t`. Solving one of those problems is enough to solve the original problem because `alt(s1,s2)` ≤ `s2` as well as `alt(s1,s2)` ≤ `s2`, i.e. both assumptions are stronger than the original assumption.

```
value
  decompose_in: Service × Provider-set × Service ~→ Service
  decompose_in(s,ps,t) as s'
    post let ext_alt(ss)=s
         in s' ∈ ss ∧ provides(s',virtual(ps),t) end
    pre (∃ ss:Service-set, s':Service •
          s=ext_alt(ss) ∧ s' ∈ ss ∧ provides(s',virtual(ps),t))
```

Even more important is to be able to decompose output which is given with concurrent composition, say `t=con(t1,t2)`. Then the problem for `s` and `t` reduces into two problems: for `s` and `t1`, and `s` and `t2`. Unlike before, we have to solve both problems, not just one of them. Moreover, we must do so with disjoint sets of providers. This is described by the function `decompose_out`.

```
value
  decompose_out: Service × Provider-set × Service ~→ (Service m~ Provider-
set)
  decompose_out(s,ps,t) as sm post
    let ext_con(ts)=t in
      ts = dom(sm) ∧
      (∀ u:Service • u ∈ ts ⇒ sm(u) ⊆ ps ∧ provides(s,virtual(sm(u)),u)) ∧
      (∀ u1,u2:Service • {u1,u2} ⊆ ts ∧ u1 ≠ u2 ⇒ sm(u1) ∩ sm(u2)={})
    end pre ...
```

CONCLUSIONS

In this paper we considered the problem of partner selection for a virtual organization, in particular how the problem could be automated. We decided not to place any particular assumptions about the kind of business the organization is involved with, be it manufacturing, logistics, marketing etc. We also decided that competence is on its own insufficient for selecting a partner, but this selection should be rather based on the dynamic provision for services the partner can offer to its environment. Finally, we decided that before we can come with any implementation for this problem, we should be able to first define and formalize it. Underlying such decisions is what we consider to be crucial for a systematic treatment of the issues surrounding the concept of a virtual organization: abstraction. We would like to be able to tackle such issues in the general setting, then only instantiate them to concrete situations, rather than consider them for specific kinds of businesses or even particular enterprises. We believe this is part of the general question to what extent we can treat a virtual organization (with no added adjectives) as the topic of analytical, not only empirical or technological, study.

The need for abstraction is particularly acute for formulating and solving the problem of partner selection, where different partners may be involved in various kinds of business activities and offer each other a variety of services. Heterogeneity in this case is part of life. In order to overcome such differences as well as factor out necessary similarities, we presented a model that represents business activities as services and business entities as service providers. A service received a formal definition with an ordering to allow matching services: one service is provided, another one is required. A service provider is given a formal operational semantics describing its changing ability to conditionally deliver services to its environment (which includes customers as well as other service providers). The state of the provider can change over time, i.e. with every delivery of a service. Not unlike its members, a virtual organization is treated as a service provider who delivers its services by means of the services by its members and service-based interactions between them. With this simple conceptual model we were able to formalize the problem of partner selection for a virtual organization starting from the services the organization is required to provide to its environment (its business goal) and the services it is allowed to rely upon (its assumptions). Selection was carried out by searching for those providers who are able to fulfill the goal under given assumptions. When formulating such goals and assumptions we could use service composition: concurrent (services provided simultaneously), sequential (provide one service, then another one) or alternative (one of two services, chosen non-deterministically). The model provides not only the possibility to actively seek partners through advanced technology, but also the mechanism to dynamically organize the structure within the virtual organization. It describes the behavior of individual members and their interactions. Unlike a traditional organizational view of an enterprise that concentrates on the internal operations, we took a cross-enterprise view of how enterprises can effectively cooperate at an abstract level.

The concept of a virtual organization (Davidow and Malone, 1992) (Browne, 1995) (Camarinha-Matos and Afsarmanesh, 1997) proved to be of the lasting

interest, evolving from the early ideas of Computer Integrated Manufacturing (Teicholz and Orr, 1987) and Enterprise Integration (Vernadat, 1996), with several representative approaches to enterprise modeling like e.g. GRAI (Zanettin et al, 1989), CIMOSA (AMICE, 1993), PERA (Williams, 1994) or GERAM (GERAM, 1998). We have seen in recent years several conferences dedicated, wholly or partly to virtual organizations, e.g. (Afsarmanesh et al, 1998) (Schonsleben and Buchel, 1998) (Bernus and Nemes, 1996). We have also seen several projects, notably PRODNET (Garita et al, 1997) (Production Planning and Management in an Extended Enterprise) and NIIIP (National Industry Information Infrastructure Protocol). We believe the work presented here complements existing results about virtual organizations. Most published results assume that the virtual organization already exists, while we concentrate on its creation. Often the main criterion considered for choosing a partner is its competence, a rather static attribute. We consider instead how the partner is able to dynamically deliver services to its clients, which is more adequate for the connected, on-line business. Most of the published results concentrate on the enabling technologies, paying less attention to the organizational and management issues. We believe such issues are critical for the success of the virtual organization. We also understand technology today is changing very rapidly, but the concept (virtual organization) and the problems related to it will remain. They are simply of the lasting interest and value. We seek to understand those problems independently from the technology used today, which is one of the reasons we chose to present the model for a virtual organization formally; the need for a more formal treatment of enterprise models have been observed before (Nemes et al, 1996) (Gruninger and Fox, 1996). Another reason is the need for abstraction. This work extends the model for a virtual manufacturing organization in (Janowski et al, 1998a) and its semantics in (Janowski et al, 1998b).

This paper presents work in progress. There are several directions we plan to continue our study. We plan to further develop the model of services, with new methods of composition (e.g. recursive definition of a service which repeats over time), quantitative comparison of services (based on the levels of attributes like price, reliability, servicing etc.), probabilities assigned to the services composed alternatively (one delivered with probability `a`, another one with probability `1-a`) etc. We plan to further develop the model of service providers with various methods to compose them together (so far we only considered providers as members of a set), for example: `(s/t).p` denotes a provider which first delivers service `s` given `t` and then behaves like the provider `p`; `p+q` behaves like either `p` or `q` depending on which service assumptions are satisfied first; `p>q` two providers acting concurrently but also interacting by `p` providing services to `q`; `p<q` two providers acting concurrently but also interacting by `p` receiving services from `q`; `p\s` behaves like `p` but not delivers any `s'` which is stronger of equal `s`; `p/t` behaves like `p` but not accepts any `t'` weaker than or equal `t`; etc. Similar like in process algebras (Hoare, 1985) (Milner, 1989), this should provide for an expressive architecture-description language for a virtual organization. We plan to extend semantics of service providers to allow for non-deterministic choice of the future state by means of function `trans` which returns the set of providers, not a single one. We carried out in this paper an example from manufacturing to illustrate the model, where a service denotes delivery of a given quantity of products and a service provider is a model of

a manufacturing enterprise. We plan to build more examples with different service models, say representing logistics (a service is a pair of locations in the communication graph), marketing (extending (Janowski et al, 1998c)) but also more detailed manufacturing models which take into account delivery time, price etc. Finally, we plan to build prototype applications for demonstration of the service-based model for a virtual organization on the Internet.

REFERENCES

1. Afsarmanesh H, Camarinha-Matos L and Marik V, editors. Proceedings of the 3rd IEEE/IFIP Conference on Information Technology for Balanced Automation Systems in Manufacturing. Chapman and Hall, 1998.
2. AMICE. CIMOSA: Open System Architecture for CIM. Springer Verlag, 1993.
3. Bernus P and Nemes L, editors. Modelling and Methodologies for Enterprise Integration. IFIP, Chapman and Hall, 1996.
4. Browne J. The Extended Enterprise: Manufacturing and The Value Chain. In Camarinha-Matos L and Afsarmanesh H, editors, BASYS95. Chapman and Hall, 1995.
5. Camarinha-Matos L. and Afsarmanesh H. Virtual Enterprise: Life Cycle Supporting Tools and Technologies. In Handbook of Life Cycle Engineering: Concepts, Tools and Techniques, Chapman and Hall, 1997.
6. Davidow W.H. and Malone M.S. The Virtual Corporation: Structuring and Revitalizing the Corporation for the 21st Century. Harper Collins, New York, 1992.
7. Garita C, Camarinha-Matos L, Afsarmanesh H and Lima C. Towards an Architecture for Virtual Enterprise. In Proceedings of the 2nd World Congress on Intelligent Manufacturing Processes and Systems, 1997.
8. Gruninger M. and Fox M.S. The Logic of Enterprise Modelling. In Modelling and Methodologies for Enterprise Integration, IFIP, 1996.
9. Hoare C.A.R. Communicating Sequential Processes. Prentice Hall International, 1985.
10. IFIP-IFAC Task Force. GERAM: Generalised Enterprise Reference Architecture and Methodology. June 1998.
11. Janowski T, Lugo GG and Zheng H. Composing Enterprise Models: The Extended and The Virtual Enterprise. In 3rd IFIP/IEEE Conference on Information Technology for Balanced Automation Systems in Manufacturing, Chapman and Hall, 1998.
12. Janowski T, Zheng H and Lugo GG. Market-Driven Symbolic Execution of Models of Manufacturing Enterprises. In 2nd IEEE International Conference on Formal Engineering Methods, IEEE Computer Press, 1998.
13. Janowski T, Atienza RV and Lugo GG. Integrating Enterprise Models and Models for Marketing Analysis. In 2nd IFIP Conference on Design of Information Infrastructure Systems for Manufacturing, Chapman and Hall, 1998.
14. Milner R. Communication and Concurrency. Prentice Hall, 1989.
15. Nemes L, Bernus P and Morris R. The Meaning of an Enterprise Model. In Modelling and Methodologies for Enterprise Integration, 1996.
16. The RAISE Language Group. The RAISE Specification Language. BCS Practitioner Series. Prentice Hall, 1992.
17. The RAISE Method Group. The RAISE Development Method. BCS Practitioner Series. Prentice Hall, 1995.
18. Schonsleben P and Buchel A, editors. Proceedings of the IFIP International Working Conference on Organizing the Extended Enterprise. Chapman and Hall, 1998.
19. Teicholz E and Orr J. Computer Integrated Manufacturing Handbook. McGraw-Gill, 1987.
20. Vernadat F. Enterprise Modelling and Integration. Chapman and Hall, 1996.
21. Williams T. The Purdue Enterprise Reference Architecture. Computers in Industry, 24(2), 1994.
22. Zanettin M, Roboam M and Pun L. GRAI: Integrated Methodology to Analyse and Design Manufacturing Systems. Computer Integrated Manufacturing Systems, 2(2):82--98, 1989.