Obligations and Delegation in the ODP Enterprise Language

Peter F. Linington
University of Kent
Canterbury, UK
Email: P.F.Linington@kent.ac.uk

Hiroshi Miyazaki
Fujitsu
Japan
Email: miyazaki.hir-02@jp.fujitsu.com

Antonio Vallecillo
University of Malaga
Malaga, Spain
Email: av@lcc.uma.es

***Abstract*—The ODP Enterprise Language is used to describe the organizational objectives and policies that apply to the system to be specified. It also captures constraints associated with the environment in which the system is to be used. Because the enterprise specification is concerned more with organizational issues than technical details of the system, there is considerable emphasis in the language design on obligations and norms, rather than on the declaration of some single rigidly required behaviour. This leads to a requirement for specification techniques that encompass a wide range of behaviour and then identify which behaviour should occur and how exceptions are to be handled; this is more challenging than computational specification, where the specification is essentially a recognizer for correct behaviour and does not define what is to happen if there are violations.**

This paper describes work currently in progress within the International Organization for Standardization (ISO) to extend the Enterprise Language so that it is able to express more directly the necessary obligations and other deontic concepts, such as permissions and prohibitions. The approach being taken is to introduce a new kind of object that reifies the deontic constraints and thereby simplifies the description of the behaviour expected.

Once the basic concepts are in place, they can be used to define a wide range of organizational matters, such as delegation rules and the way communities respond dynamically to changes in their structure.

***Keywords*-open distributed processing; enterprise language; obligations;**

I. Introduction

The Enterprise Language [1] was first introduced in the Reference Model for Open Distributed Processing (RM-ODP) [2]–[6], and forms a key part of the toolkit for the specification of ODP Systems. Separating the organizational objectives, policies and environmental constraints into a separate viewpoint capturing the business processes to be supported creates a firm foundation both for the detailed design of a new system and for the evolution of an existing system. It simplifies system management and reduces the risk of unintended changes being introduced during subsequent maintenance.

However, concentrating on what the system ought to do brings us into the realm of deontic logic. This has a long history, dating back to the seminal work of von Wright [7]; a good recent introduction can be found in [8]. Starting from deontic logic allows us to deal with norms and expectations, or more specifically with obligations to perform specified behaviour, permissions to perform such behaviour and prohibitions of behaviour that might otherwise be feasible. There is a subtle shift going on here. We are moving from a style of specification where the testing of correctness is based on whether the next action taken is always as specified to a somewhat looser assessment regime in which sets of obligations must eventually be discharged but where a participant needing to do so may be juggling many potentially conflicting constraints. There is a shift from concentration purely on behaviour towards concern with system state and the future behaviour that this implies.

However, standard deontic logic has a number of problems. One of these is that its formulae state properties of the system as a whole. It therefore becomes necessary that in a conformant system, an obligation for something to happen implies a permission for it to happen. In an enterprise specification we need to associate deontic constraints with specific objects which act as agents, and which may need to obtain permissions in order to discharge obligations they have undertaken. Something less restrictive is required.

We need, therefore, to be able to deal with the sets of obligations, permissions and prohibitions associated with the various objects in the specification and how these sets evolve, but to be able to express the goal seeking behaviour of the various participants acting as agents. This is done here by representing each deontic assertion by a token object. These objects are held by the parties involved and holding one controls their behaviour. These tokens can also be passed from one party to another. The idea of handling this problem by introducing a reification of the deontic constraints was first suggested as a direction for developing the enterprise language in [9] in 2003 and activity to incorporate it into the ISO standards started in 2007. It is now well advanced.

This paper describes the work that has taken place since then within the International Organization for Standardization (ISO) to allow system specifiers to deal with these issues. It aims to explain the approach taken and the way in which the existing enterprise language is being extended to accommodate it. Further work on concrete notations and formalisms is now ongoing.

The paper is organized as follows. A brief reminder of the nature of the enterprise language is given in section II.

In section III, we set the scene by introducing an example that is based on an e-store selling physical goods, which it has delivered to its customers. Then in section IV the new language concepts are introduced and the basic rules for manipulating them explained, Following this, we look, in section V, at the ways the changes in the allocation of obligations can themselves be subject to the same kind of controls, and, in section VI, at how this can be used to express processes like delegation. Section VII looks at the way these mechanisms can be nested. Further, in section VIII we look at the way exceptional events may lead to reallocation of tokens and thus of responsibilities. Finally, we say a little about related work in section IX and then, in section X there are conclusions and an indication of how the work is to to be progressed.

II. The Enterprise Language

The enterprise language [1] is used to specify the view of a system seen by the owners and policy makers operating within the business environment it serves. The aim is to establish constraints and objectives as a basis for the specification of the system itself.

The language expresses groupings of interested parties and the behaviour they are expected to engage in. It provides for the expression of general design constraints and for mutable policies, such as security or commercial policies, in a way that allows changes in organizational objectives to be expressed and so to be used to guide evolution of the supporting systems and their infrastructure.

The main structuring tool in the language is the concept of a *community*. This expresses the way a group of abstract objects representing different players comes together to achieve some objective. Communities are expressed in terms of their type, also known as a contract, that expresses behaviour in terms of a set of participating community roles. These roles are filled, in a particular community, by specific enterprise objects which are then each constrained to conform to the behaviour defined for their role.

Roles are typed, and objects filling them must be compatible with (that is, subtypes of) these types. Rules can also be expressed to constrain role-filling at the instance level, supporting properties such as static or dynamic separation of duties. In simple communities, roles are filled when the community is instantiated, but more complex situations can be expressed by including definitions of role filling or replacement of community members as part of the community behaviour.

The enterprise language also includes concepts for describing chains of accountability and responsibility so that the basis for system actions can be traced back to the legally accountable principals concerned. Doing so helps to bridge the gap between business and technical processes.

More background to the style of and motivation for the features of the language can be found in [10].

To be of practical use, an abstract language like this needs to be supported by an easily accessible concrete notation. In UML4ODP [16], representations of all the ODP viewpoint languages are defined in a way that makes them easily accessible to architects and designers who are already familiar with UML-based tools.

III. A purchasing scenario

As an example of the type of situation to be met with, consider the *Modern Oil Store*, an online retailer specializing in high quality olive oil. At the beginning of the season, the company makes agreements with a number of primary producers for the supply of their best quality oil. Customers order via the web and the supplier arranges with one of their pool of carriers to deliver the goods direct from the producer. The customer provides credit details with the order, but payment is only due on delivery.

Consider the central piece of this behaviour, starting with a customer order and ending with payment for the goods. This sequence of events is shown in a very simplified form in figure 1.

This starts with an order event, which can only happen if the customer has not been blacklisted. The result of the order is that the supplier is obliged to make the oil available to the customer and the customer is obliged to pay upon delivery. When payment is by credit card, the customer gives the supplier, as part of the order, permission to take the required payment. Once the order has been received, the supplier instructs a carrier to transfer the oil and the producer to make it available. Each of these action leaves the subcontractors with obligations to do their part and the supplier with

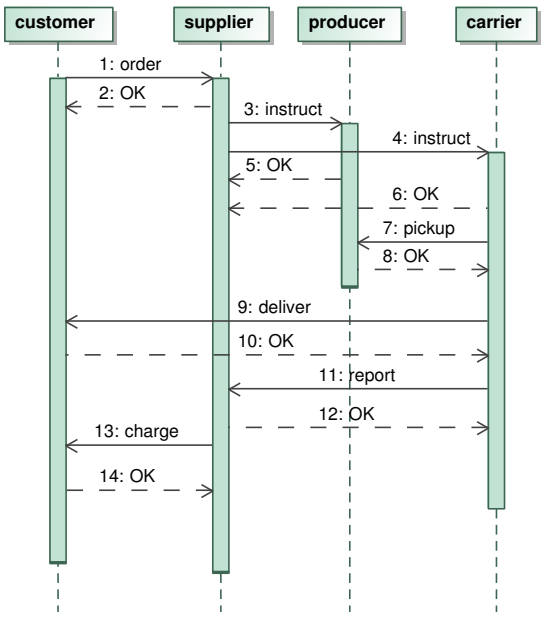

Figure 1. A purchase of olive oil.

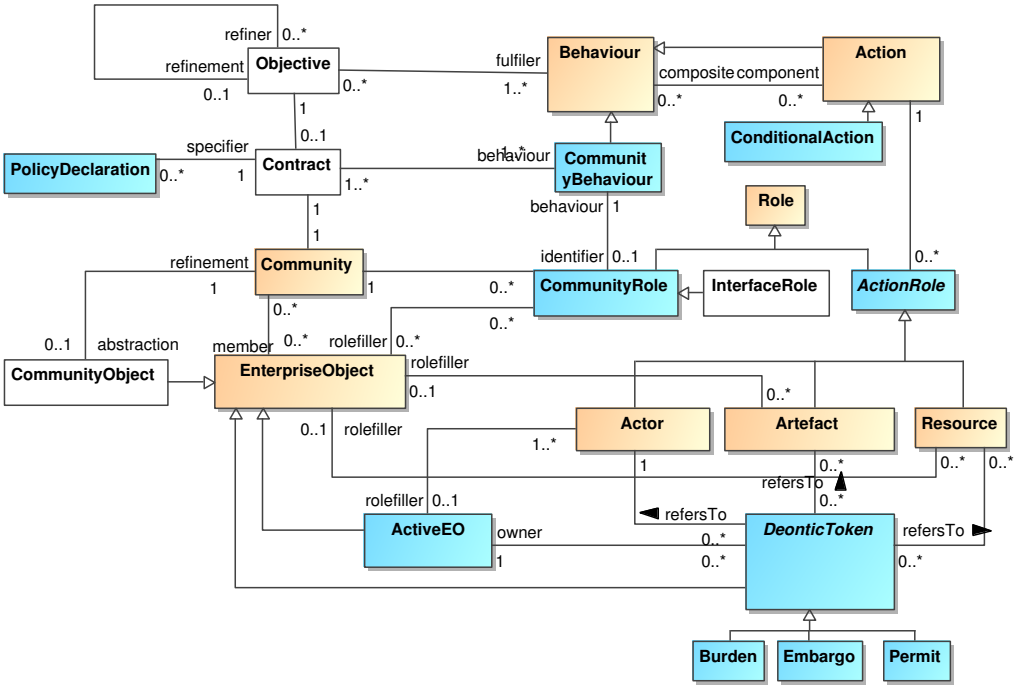

Figure 2. The ODP Enterprise Community Concepts.

obligations to pay for these services and to monitor the correct performance of them by the subcontractors.

The eventual delivery of the oil discharges the supplier's obligation to the customer and the completion of payment discharges the customers obligation to pay. This action also cancels the permission for the supplier to draw upon the customers funds.

IV. Extending the Language

The central structuring concept in the original enterprise language is the community. A community is a configuration of enterprise objects which is formed to achieve a given objective. A community contract is the template from which a community is instantiated, and defines its objective and the behaviour that is to be undertaken to achieve this objective. A number of roles are defined as formal parameters of this contract, which are each filled by one of the various enterprise objects that have come together to form the community. When an enterprise object fills a community role, its behaviour is constrained by that role, as stated in the contract. In our example, the whole olive oil business can be modelled as a community, with roles for customer, supplier, provider and carrier, each filled by an appropriate object to describe a particular transaction.

In a more abstract description, any community can be seen as a whole, forming a single object, which is then called a community object. Such an object can then itself fill roles in some larger-scale communities, resulting in a hierarchical structure of arbitrary depth. Thus an engineer can play a role in a project team, which can then play a role in a development department which is itself in an organization, forming part of an industry sector, and so on. At each stage, a nesting of communities involved can be seen.

Figure 2 and figure 3 show some of the key concepts for expressing this community structure. They also introduce some of the new concepts that control the performance of the community behaviour.

The key extension to the conceptual framework for the enterprise language is the introduction of a new type of enterprise object, called a deontic token. A token has no independent behaviour of its own, but is associated with, or carried by, an enterprise object which does itself display behaviour. The fact that an active enterprise object carries a token modifies the object's behaviour in some way. Note that we introduce specific names for these token objects to distinguish them from the deontic constraint they carry; thus, for example, a burden is a token object carrying an obligation; a burden can therefore be manipulated as an object whilst an obligation cannot. There are three common cases defined; the token may be:

- **a burden**, representing an obligation. The active object carrying a burden must attempt to discharge it either directly, by performing the specified behaviour, or indirectly by performing behaviour that results in

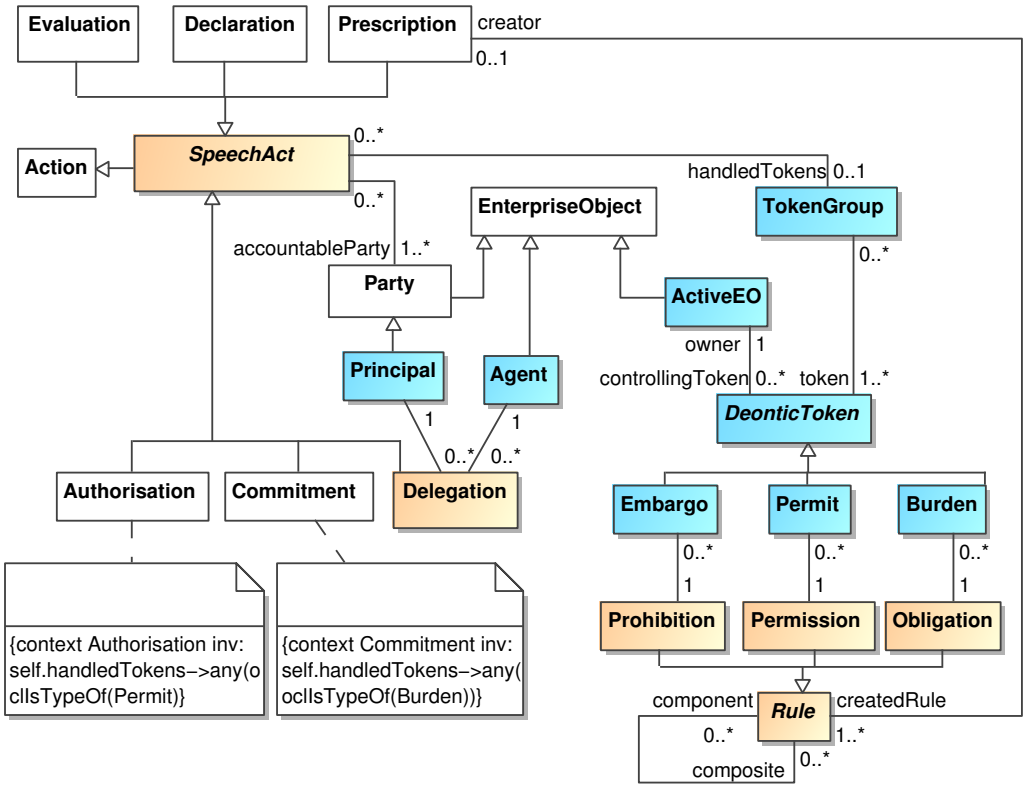

Figure 3. The ODP Enterprise Accountability Concepts.

some other object taking possession of the token and performing the behaviour that actually satisfies the obligation.

- **a permit**, which, when held by an object, makes it able to perform some specified piece of behaviour. An object is obliged not to perform the behaviour unless it already holds the required permit.
- **an embargo**, which, if held, inhibits its holder from performing the specified behaviour.

It should be noted here that these token objects represent modelling constraints, so the association of a token with an active object is part of the description of the situation being modelled, and so is not something a misbehaving object can deny; how this constraint is then policed in an implementation is to be decided later, and is not of concern here.

In general, a token identifies the authority placing the constraint and the behaviour influenced by the constraint, indicating which roles or specific objects are being constrained. Note that the fact that the object affected can be identified either implicitly or explicitly implies that a token may be held initially by an object that is not constrained by it, but later passed on to another object that, because it is identified in the token, becomes constrained when receiving it (see section V).

In fact, the actual definitions are rather more precise than the brief statements given above. We need to specify not just that an active object is involved in some behaviour, but to state how it is to be involved, which requires the specification of its action-roles. The concept of a role is a very broad one, applicable to any situation where a number of objects enter into a configuration in distinct ways. Community-roles were introduced above, but here we need to be concerned not with community-roles, but with action-roles, which are associated with a different aspect of configuration. When an interaction type is defined, we are generally interested in expressing how a number of objects come together to exchange information, and each participant is expected to send or receive some specific items. In other words, each object performs a different action-role, and the action-role expresses what information the role-filling object supplies to or derives from the interaction. Thus, for example, two

objects may perform client and server roles in an interaction, and these action-roles make it clear, in this case, where the initiative lies.

This ability to associate properties of an interaction with its action-roles is particularly important in the enterprise language because interactions often involve more than two participants. Pieces of negotiation between three or four enterprise objects may, for example, be expressed as a single abstract interaction, and the contributions expected from each object will depend on the action-roles they are filling.

So, when we speak of an object's being obliged to perform some behaviour as a result of carrying a burden, we actually mean that the constraint expressed by the burden applies to its ability to perform a stated action-role in the behaviour; involvement in some other action-role will not do. For example, an object may carry a burden to make a payment. It is obvious that this means it is obliged to provide the money in a payment behaviour, and that its being involved in the right type of interaction, but in such a way that means it receives the required sum of money, will not do.

In addition to the idea of tokens, two further definitions are added to express the way these tokens are related to the specification of community behaviour. This behaviour is itself expressed by the recursive composition of actions or sub-behaviours, so a subtype of action, called a conditional action, is introduced to allow the specifier to declare which actions are constrained by the set of available tokens. This makes it possible to see immediately whether a particular piece of behaviour may depend on the set of tokens held by its participants, and so simplifies the analysis of the behaviour.

The other new concept is needed to indicate whether an action will modify the set of tokens held. This is again a subtype of action, and is called a speech act. This name was chosen by analogy with the linguistic concept of speech act, introduced by Austin [11] and developed by Searle [12]; they were motivated by the need to explain how actions like promises or wagers intrinsically change the state of the world, which is clearly relevant to the problem we are solving here.

Finally, another construct is introduced to allow groups of tokens to be manipulated as a whole; we will return to this in section VI.

Let us now return to the scenario outlined in figure 1 to see how these concepts are used. We will examine the performance of individual actions in the specification, but in practice it is expected that much of the detail will be captured in the specification of the corresponding action types so that the statement of the business process is still as simple and uncluttered as in the earlier figure. The type definitions will be expressed in terms of the action roles involved, which will be bound to particular objects when the action is instantiated (that is to say, performed).

All the actions in this example are speech acts, in that

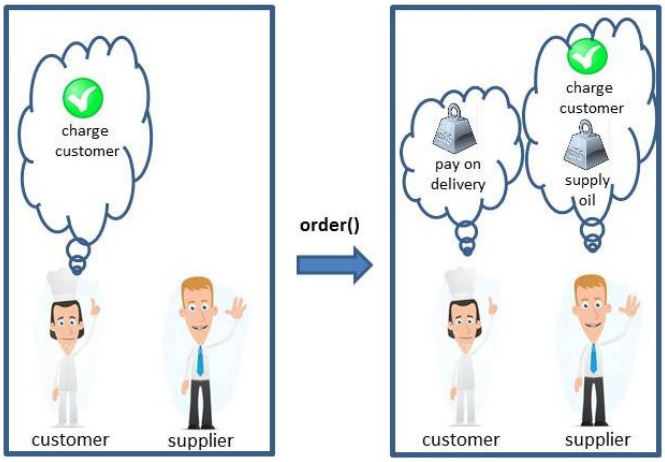

Figure 4. Effects of the order action.

they all modify at least some tokens. In this example, we assume that only some of the actions are conditional; this is a design choice.

The order action is performed first. This is a conditional action, because, although we are assuming that any party could attempt to act as a customer, there are still preconditions to be applied; some potential customers might have been placed on a blacklist, and this is modelled by their carrying an embargo prohibiting their performance of the order action in the role of customer. A customer is also expected to make available a permit for it to be charged on delivery (this is provided before the order action, rather than as a side-effect of it, because of the specific choices the customer needs to make in preparing it).

Order is a speech act, because performing it creates a number of tokens. First, the customer who initiated the action is left holding a burden expressing its obligation to make payment when the requested oil is delivered. The customer will also need to provide a permit giving permission for charges to be made, which is transfered to the supplier by the action. When the supplier accepts the order, completing the action, it is left holding a burden for the obligation to deliver the oil, as well as the charging permit that the customer provided. The way this action changes the sets of tokens held is illustrated in figure 4.

Once the supplier holds a burden, it must decide how best to satisfy it. It cannot, in this case, discharge the obligation directly because it has neither its own stocks of olive oil nor any direct access to the customer. It therefore approaches its goal of providing the oil by having its subcontractors perform each of these two subgoals as separate steps; first, it asks one of the providers to make the required quantity of oil available, and then it asks a suitable carrier to collect this consignment and deliver it to the customer.

The two actions with which the supplier instructs the producer and the carrier are similar. They are not conditional

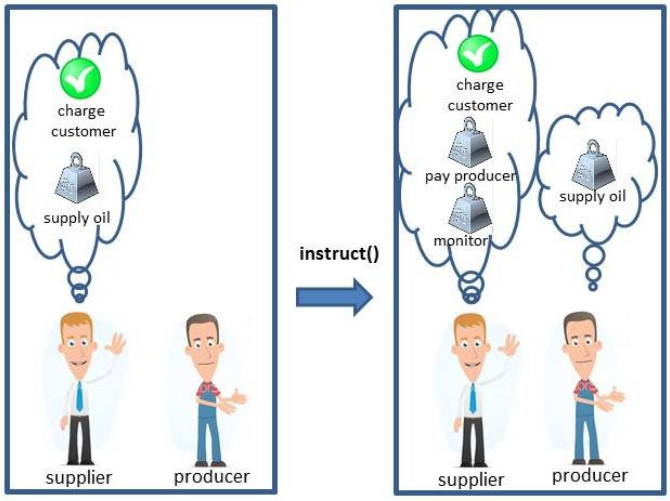

Figure 5. Effects of the instruct action involving the producer.

actions, because we have chosen not to model explicit access control between the supplier and its subcontractors. If the instructions are successful, each leaves the subcontractor with a burden to carry them out and the supplier with a burden to pay for these services. The burden to supply the goods remains with the supplier, but the actions instructing the subcontractors have left the supplier with an expectation that that this burden will be discharged; the supplier must continue to monitor the situation to see that this in fact happens (see figure 5, which shows the supplier interacting with the producer to set up the first subgoal).

When the goods are delivered and this is reported to the supplier, the burden to supply them is discharged. Thus the reporting speech act performed by the carrier deletes the corresponding tokens held by the supplier and the carrier

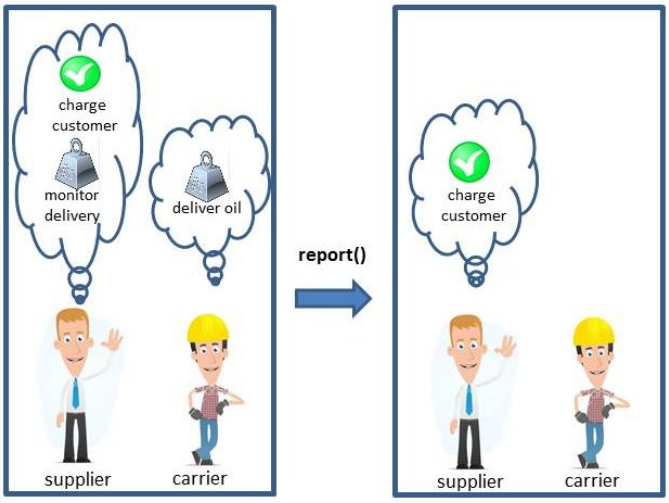

Figure 6. Effects of the report action.

(see figure 6). The burden on the producer was deleted when the pickup action was performed.

Finally, the supplier performs the speech act of charging the customer. This is a conditional action because it can only be performed if the supplier has the necessary permit, which is destroyed as a side effect of its use in the charging action. This action also satisfies and deletes the payment obligation on the customer (see figure 7).

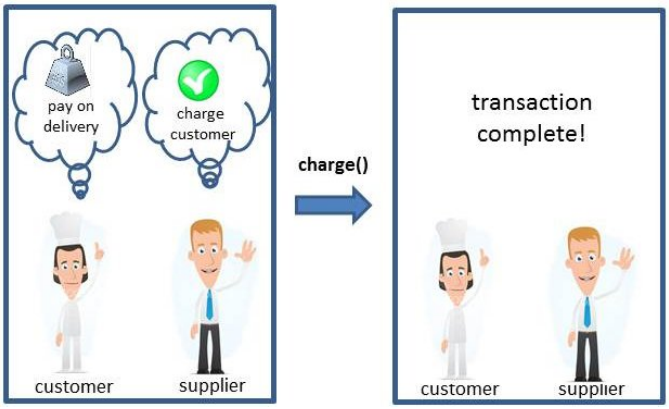

Figure 7. Effects of the charge action.

Note that we have taken a conservative approach here in making the carrier's report explicit and linking the discharge of the supplier's burden to it. Other styles of specification are possible if a trusted carrier, such as a national postal service, is used; we could then omit the report and consider the obligation to be discharged when the carrier is instructed to perform the delivery. However, there would then need to be some separate definition of exceptions and their processing to cover situations where the shipment is lost (see section VIII).

V. The Passing of Tokens

So far we have looked at the simple case where a burden is created associated with one object, which then holds it until the object's behaviour causes the burden to be discharged and so to cease to exist. However, in more complex situations, a token may be passed from object to object.

Since passing a token is itself an interaction, it can, in turn, be controlled by other tokens. Thus there can be tokens that represent the obligation to pass a token, a permission to do so, or a prohibition to prevent it being done. This can lead to the following patterns.

- A source of authority responsible for access control in a system is a factory for permits and itself holds permits allowing it to transfer the permits it creates to any of the other objects in its security domain.
- an object holds an obligation to perform some action A, but needs a permit in order to perform this action. It can

attempt to fulfil its obligation in one of two different ways:

1) it can attempt to acquire the necessary permit to do A from some authority and then perform the action; or
2) it can attempt to acquire a permit that will allow it to pass the burden to an object which has, or can more easily obtain, a permit to perform the action A.

In the example above, we described subcontracting in terms of a speech act that created new burdens for the tasks being assigned to the subcontractors, but it would also have been possible to imagine a different way of describing the business in which the supplier acted as a broker, and passed on the burden to supply oil completely to another supplier who would become responsible for the remainder of the process. This would require that the supplier should have any necessary permits to pass the burden on. It could either have obtained these permits previously, or it could obtain them dynamically as a result of a negotiation triggered by this particular order.

VI. Delegation

From these elements, it is possible to construct descriptions of a wide range of mechanisms for the control and management of enterprise systems. One of these is the specification of a framework for delegation. Delegation can take many forms, and using deontic tokens to express it allows us to distinguish the different variants.

The basic motivation for delegation by some party to another is that the delegator can take on broader responsibility, with greater reliability, if they involve others in helping to perform their duties. Delegation involves passing-on to others responsibility for some tasks, together with permissions to use the resources necessary to carry them out. The recipients usually already have some obligation to accept and act on the delegation.

It is clear that this process involved the passing of permits, burdens and embargos, but a complete description of a delegation scheme needs to state:

- what permits the delegator needs in order to allow it to pass burdens and permits to the delegatee;
- what additional burdens the delegatee is given in consequence of the delegation, in order to allow the delegator to revoke the delegation when necessary and recover associated resources;
- whether or not the delegator can still exercise the permits that have been delegated; in other words, are permits passed or cloned, and is there an embargo that prevents the delegator from exploiting cloned permits while the delegation is in effect;
- what burdens the delegation places on the delegator to monitor and control the activities of the delegatee;
- how the tokens passed during delegations are organized into groups and whether the delegatee can perform further delegation of either the whole group of tokens received or of some subset of it.

The example given above has already illustrated a number of these features in the behaviour of the supplier's subcontractors. This situation involves a fairly loose flavour of delegation, without many shared resources. Delegation within a single organization is likely to be based on greater sharing of resources and more complex pre-existing rules of procedure.

Consider, for example, a service manager who is in charge of a team of customer service engineers. The service manager is responsible for solving customer problems but may not have the technical qualifications, and hence the permissions, to carry out servicing. When new engineers join the team, the manager validates their credentials and passes them permits to perform service actions and to access customer account records. The manage is able to distribute a group of tokens, only some of which he could himself use. He also passes the burden to obey departmental procedures and to satisfy particular customer issues. The engineers accept a burden to act on the manager's instructions when they are appointed.

VII. Tokens, domains and communities

There are two possible styles in which enterprise systems can be specified, depending on the default situation expected. This can be illustrated, for example, by considering how permissions are checked in an access control system. Either actions are allowed by default and specific prohibitions expressed, or actions are forbidden by default and some then explicitly permitted. This decision is made independently for each action, so that, for example, a document read action may be allowed by default and specific accessors blocked, while a write action might be prohibited unless explicitly allowed. The decision on which of these styles to apply is taken by the controlling authority, which establishes the policy for its domain of responsible. Nested domains can each have their own local policy.

In general, we can identify a hierarchical structure of domains representing the way authority is derived, and in the enterprise language this is mirrored by a hierarchy of communities. Real specifications will generally just declare their assumptions about the local sources of authority, rather than attempting to trace them back through the broad abstract communities that might in principle represent the social and legal structures underpinning concepts of ownership and corporate governance. However, such a structure could, in an ideal world, be codified.

Once a source of authority has been identified, this can delegate control of a specific subset of its resources to smaller-scale domains, and in each of these domains the local authority can decide whether the default assumption

is of permission or prohibition. However, this freedom may be constrained by obligations applied when control of the resources is delegated; for example, a sub-domain controller may be constrained to exercise positive authentication to comply with some broader policy.

So far we have described how the behaviour of an object is modified by the tokens it holds. This can be expressed as a direct association of the tokens with an object instance, or a level of indirection can be used. One important case arises in defining the behaviour of communities, where expected behaviour is expressed in terms of the way the community roles interact and these requirements are then reflected in the observable behaviour of the objects filling those roles. One of the jobs a community specification has to do is to show how a community farms out the burdens it holds to its members, and so the behavioural specification can be simplified by associating burdens with roles and then defining the filling of a role as extending any token associations with that role so that the tokens are implicitly associated with the objects filling the roles (see figure 8).

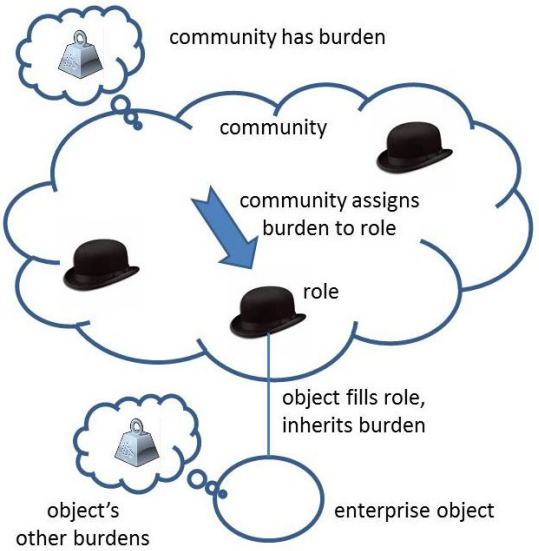

Figure 8. Associating tokens with a role.

It would, of course, be possible to define this in terms of the explicit passing of tokens from the community to its members when it is formed or when a new member enters into a role. However, doing so would add a great deal of additional detail to the specification which can be avoided if the implicit extension of the role associations is used.

Associating tokens with roles does, however, raise some additional questions. What is the status of tokens associated with roles that are not currently filled by any object? This is not a particular problem for permissions or prohibitions, as interactions expected of an empty role cannot happen. Obligations, on the other hand, do present a problem. If the community assigns one of its burdens to an empty role, that obligation will not be discharged. We must therefore define the association of tokens with roles so that they are still present in a more abstract view as being held by the community.

This also provides a starting point for the definition of how tokens should be handled when an object ceases to fill a role, either because it leaves the community or because it ceases to exist. Basically, tokens will revert to the community in these circumstances.

VIII. Exceptional circumstances

This section looks further at the handling of deontic tokens when exceptions occur. These may arise because an object leaves (or is ejected from) some community. What happens, for example, if one of the carriers involved in the olive oil business fails suddenly? It becomes insolvent and as a result ceases to fill its role in the supply community. Suppose it was currently involved in a transaction and had received instructions to collect a batch of oil and deliver it to a customer. The producer had discharged their obligation by making the oil available to the carrier, but the other parties all held tokens related to their various roles.

The exception may be triggered either by the failing carrier issuing a notification of the event, or, if it fails silently, by the supplier observing some timeout. It will do this because it is still holding a burden obliging it to monitor the correct performance of the carrier's duties. In either case, the obligation to the customer reverts to the community as not being handled at a more detailed level, and a rule is triggered which re-assigns it to the supplier. The supplier will then reissue instructions to the producer and to a different carrier, so that the order is filled. Other rules may also be triggered, including the issue of some recompense to the customer for the delay.

The supplier will also initiate a separate thread of activity in which it seeks to recover the undelivered goods from the carrier and to make any applicable claim for breach of contract. Note that this is an area where behaviour is likely to be expressed as interactions involving more than two parties; dispute resolution will often involve abstract interactions involving all the parties involved, together with some adjudicator.

In general, exception handling may be expected to revoke existing token and to issue new ones, either to compensate for failed activity or to seek redress for the damage done. This may be a refinement of some recovery mechanism defined in a general community contract used as a basis for defining the current community.

IX. Relationship to other work

The prime aim of this paper has been to raise awareness of the activities taking place within the standardization bodies. The high level of consensus building and consultation involved in formal standardization takes time, and so many

related ideas have been put forward since this work began within ISO in 2007.

As we indicated in the introduction, approaches based on standard deontic logic are somewhat brittle, lacking the ability to localize constraints by associating them with specific agents acting in particular roles. Within these limitations, much work has been done to categorize different kinds of obligation, and [13] gives a well reasoned example.

Research within a service oriented structure has addressed issues of agency, and [14] gives a typical example. This work lacks the expressive power of the approach presented here because the deontic constraints are not fully reified, which also makes it harder to express specifications in a notation that is already familiar to practitioners; contrast this with the approach here of mapping enterprise concepts to UML standardized in UML4ODP [16].

The closest parallel to the approach taken here is in Belnap's [15] work on STIT (for "see to it that") logic and his associated approach to the modelling of obligations. This work inspired our analysis, although it is difficult to take as a complete basis if we are to retain an object based approach, and so the similarities at a detailed level are limited.

X. Conclusions and future work

This paper has introduced an approach to enterprise specification based on the reification of deontic constraints. The work to add support for this technique to the ODP standards is being carried out in two closely related projects.

The first project is concerned with the addition of the new concepts and rules to the Enterprise Language standard, ISO 15414 [1]. This work is now well under way, and has reached the key stage of issuing the draft of the updated text for ballot by the national members who are represented in the relevant subcommittee — ISO/IEC JTC1 SC7 on Software and Systems Engineering. This draft will need to progress through at least two rounds of balloting before becoming a standard.

In addition to the specific mechanisms described so far, the draft is also expected to contain some non-normative material on the way deontic properties can be formalized. This takes us beyond the use of simple labelled transition systems towards frame-based representations and it is important that potential tool-builders should be aware of the possibilities available.

The second project builds upon the first; while the enterprise language provides an abstract conceptual framework, real specifications need to be expressed in a concrete notation. In ODP, one such representation is defined in the standard ISO 19793 on Use of UML for ODP System Specifications [16], and the second project will update this standard to make it capable of expressing the new concepts. Clearly, it has not been possible to progress very far on notational matters until the abstract framework is stable, so the two projects were deliberately staggered, with serious work on the second starting only now that the details of the abstract language are gaining acceptance.

It is quite common for an abstract specification language to be supported by more than one representation. The enterprise language is no exception, and the work in ISO is looking at the support of the same set of concepts using more than one style. The focus here is on the support of the abstract concepts in BPMN, as representing a commonly use style for business process specification.

If all goes according to plan, this activity is expected to result in published standards in some two to three years, and it is hoped that research on supporting prototype tools will also be carried out during this period.

Acknowledgment

The authors would like to acknowledge the contribution made to the development of ODP by a very large number of experts around the world over many years. However, the responsibility for any errors in the current work rests with the authors alone.

The authors would also like to thank the anonymous referees for their detailed and helpful comments on the first version of this paper.

References

[1] *ISO/IEC IS 15414, Information Technology — Open Distributed Processing — Enterprise Language*, 2006, also published as ITU-T Recommendation X.911.

[2] *ISO/IEC IS 10746-1, Information Technology — Open Distributed Processing — Reference Model: Overview*, 1998, also published as ITU-T Recommendation X.901.

[3] *ISO/IEC IS 10746-2, Information Technology — Open Distributed Processing — Reference Model: Foundations*, 2010, also published as ITU-T Recommendation X.902.

[4] *ISO/IEC IS 10746-3, Information Technology — Open Distributed Processing — Reference Model: Architecture*, 2010, also published as ITU-T Recommendation X.903.

[5] *ISO/IEC IS 10746-4, Information Technology — Open Distributed Processing — Reference Model: Architectural Semantics*, 1998, also published as ITU-T Recommendation X.904.

[6] P. F. Linington, Z. Milosevic, A. Tanaka, and A. Vallecillo, *Building Enterprise Systems with ODP — An Introduction to Open Distributed Processing*. Chapman & Hall/CRC Press, 2012.

[7] G. H. von Wright, "Deontic Logic," *Mind*, vol. 60, pp. 1–15, 1951.

[8] D. Rönnedal, *An Introduction to Deontic Logic*. CreateSpace, 2010.

[9] P. F. Linington and S. Neal, "Using policies in the checking of business to business contracts," in *Proc. 4th IEEE Int. Work. on Policies for Distributed Systems and Networks (POLICY'03)*. Lake Como, Italy: IEEE Computer Society, Jun. 2003, pp. 207–218.

[10] P. F. Linington, Z. Milosevic, and K. Raymond, "Policies in Communities: Extending the ODP Enterprise Viewpoint," in *Proc. 2nd Int. Enterprise Distributed Object Computing Work. (EDOC'98)*, San Diego, USA, Nov. 1998, pp. 14–24.

[11] J. L. Austin, *How to Do Things With Words*. Harvard University Press, 1962, 2nd edition, 1975.

[12] J. Searle, *Speech Acts*. Cambridge University Press, 1969.

[13] G. Governatori and A. Rotolo, "A conceptually rich model of business process compliance," in *Proc. 7th Asia-Pacific Conference on Conceptual Modelling*. Australian Computer Society, 2010, pp. 3–12.

[14] M. P. Singh, A. K. Chopra, and N. Desai, "Commitment-based service-oriented architecture," *Computer*, vol. 42, no. 11, pp. 72–79, Nov. 2009.

[15] N. D. Belnap, M. Perloff, and M. Ming Xu, *Facing the Future: Agents and Choices in Our Indeterminist World*. Oxford University Press, 2001.

[16] *ISO/IEC IS 19793, Information Technology — Open Distributed Processing — Use of UML for ODP System Specifications*, 2008, also published as ITU-T Recommendation X.906.