Software and Systems Modeling (2025) 24:741–754
https://doi.org/10.1007/s10270-024-01243-4

SPECIAL SECTION PAPER

Using DSLs to manage consistency in long-lived enterprise language specifications

Peter Linington[1] · Zoran Milosevic[2] · Akira Tanaka[3] · Igor Dejanović[4]

Received: 11 March 2024 / Revised: 30 October 2024 / Accepted: 9 November 2024 / Published online: 29 November 2024

Abstract
Modern enterprise systems are likely to have a very long life. Their specifications therefore need to employ mechanisms that allow them to evolve during their lifetime; where they exploit generic components, these must be adaptable for use in novel situations. The paper looks at some of the issues that arise from this requirement, and how the exploitation of domain-specific language technologies in the tool-chain can assist in maintaining consistency of the specification as a whole. First, it reviews the final state of the family of standards supporting the ODP Enterprise Language, which is intended to handle this kind of application. In particular, it looks at the way the framework for defining policies can be used to accommodate changing requirements during the lifetime of an evolving system. It also looks at the way the idea of deontic tokens enables factoring out of the management of obligations from the basic behaviour of interacting system components. It then proposes a roadmap for building tools that can be used to unify the constraints from different areas of concern into a single specification. The approach taken is to exploit the power of domain-specific languages (DSLs) to allow designers in the various areas of concern to provide their input in terms natural to them. Finally, it looks at the way this approach promotes the establishment of a robust tool-chain capable of handling the evolution and scalability of enterprise systems. The paper uses a running example from the e-health domain to show how specific areas identified in the e-health standards can lead to language definitions, and so to tooling, that can be used to manage unified, system-wide specifications.

Keywords Open Distributed Processing · Domain Specific Languages · Policies · Deontic Tokens

1 Introduction

Many recent cyber-security and system engineering developments, including real-time analytics and AI system integration efforts, require the precise expression of enterprise structure to ensure the development of reliable, safe and evolvable systems. A robust structure can form the basis for managing consistency, such as in controlling system evolution and for analysis of obligations to help enforce the necessary responsibility and trustworthiness principles.

In general, we consider consistency of a specification to be the absence of logical contradictions deducible from it, or of any behavioural deadlocks or livelocks within it. It is important that the supporting tool-chain should alert users to such inconsistencies as the specification evolves.

This paper builds on the established international standard for Open Distributed Processing [1–3], using it as a framework for specifying our target distributed enterprise. It revisits some of the key features of the ISO ODP enterprise language standard [4, 5, 23, 24], in the light of recent devel-

Communicated by Javier Troya and Alfonso Pierantonio.

✉ Peter Linington
P.F.Linington@kent.ac.uk

Zoran Milosevic
zoran@deontik.com

Akira Tanaka
a.tanaka@view5.co.jp

Igor Dejanović
igord@uns.ac.rs

1 School of Computing, University of Kent, Canterbury CT2 7NF, Kent, UK

2 Deontik, Brisbane, Australia

3 view5 LLC, Yokohama 220-0004, Kanagawa, Japan

4 Faculty of Technical Sciences, University of Novi Sad, Novi Sad 21000, Vojvodina, Serbia

opments in data modelling and science, including machine learning, behaviour-centric modelling, and real-time analytics, while considering the latest in contemporary software engineering and tooling. This includes recent conceptual modelling approaches and established software engineering techniques, such as model-driven development and domain-specific languages (DSLs).

We look, in particular, at two areas within the enterprise language: the use of deontic tokens[1] to separate basic behaviour from the obligations that guide it, and the provision of a clear policy framework to support system evolution, thereby allowing reconfiguration during the system's lifetime; dynamic changes in policy provide explicit support for change to deal with new system requirements.

The ODP Enterprise Language factors out particular non-functional requirements, such as deontic concerns and policy structures; this refactoring can lead to situations in which interpreting the specification involves merging several specification fragments, often each using representations tailored to the various areas of concern.

The paper is motivated by the need to provide a solid, standards-based, framework for the expression of accountability requirements, reinforcing the broader system engineering concerns. This requires, for example, more explicit support for consumer interactions when addressing their personal privacy preferences, such as in terms of informed consent [29], while still relying on the established security standards and practices.

Adopting this new style of working may need an increasing level of automation, enabled by new technologies, such as AI. At the same time, regulatory and legislative changes will need to be accommodated to deliver safe, reliable and trustworthy systems.

The contributions in this paper are organized as follows. Sections 2 and 3 review the ODP concepts of policy and deontic token, and Sect. 4 outlines a roadmap for managing the dependencies between tool specification elements. Section 5 introduces a working example from the healthcare sector. Section 6 then relates the example to the two concepts introduced in Sects. 2 and 3, and Sect. 7 shows how they can be used to organize specifications in a large system. Section 8 introduces the approach taken to provision of a tool-chain, Sect. 9 gives an introduction to domain specific languages, and Sect. 10 gives specifics of the tool-chain and examples of its use. Section 11 introduces some related work. Finally, Sect. 12 gives conclusions and suggests future directions.

2 What is a policy?

Fundamental to the concept of a policy is the recognition, when a system is being designed, that the requirements placed on it will change during its lifetime. Changes may be because of natural evolution of a particular system instance, or because of different instances being part of a general product line. In either case we can distinguish between separate epochs[2] concerned with design, implementation, configuration or adaptation during ongoing use.

In preparation for expected change, the original designer can declare those parts of the specification that are likely to be affected by the change as being policies. A policy will have an initial, or default value, but this can be changed as circumstances dictate. It is not, however, reasonable for the policy to be changed in a completely arbitrary way. In a financial application, for example, we may want to establish a policy which governs how interest is calculated, and be able to use this to take account of changes in tax regulations, but it would not be reasonable to use this flexibility to install code that transfers a part of the holding to the administrator's pension fund. There needs to be some control over what a policy can be set to do.

To deal with these considerations, the ODP concept of policy represents a named point of variation, where required behaviour can be changed. However, the point of variation also has associated with it what is known as a policy envelope that sets bounds on what range of behaviour is actually allowed. Often this will be done by restricting values to be within an enumerated set, but values may be constrained to any behaviours subject to given constraints. These different styles of constraint are both ways of defining the policy envelope.

Other information associated with a policy is its point of application, limiting when and where it can be invoked, and the policy setting behaviour, which controls the circumstances when the policy value can be changed, and by whom. The definitions of the architectural elements involved are in [2–4]. Further discussion of the issues involved here can be found in [25].

In all cases, however, declaring a policy involves the separation of a core design from some specializations of it. These are likely to cover various possible use cases, appropriate in different epochs.

[1] deontic: pertaining to duties or obligations; deontic token: a package of information communicated when updating the deontic state of an object.

[2] In RM-ODP, an epoch is defined to be a phase or period of time during which some set of rules applies; it can be thought of as a generalization of, for example, a design or implementation phase.

3 Deontic Tokens

Another structuring tool provided in the enterprise language is the ability to associate deontic tokens with actions performed. In ODP we identify the set of objects involved in related interactions as forming a community, and these communities provide an organizational context in which such tokens are defined.

The focus here is on actions which have some profound effect on the obligations or permissions associated with parties within the system. Following the usage in linguistics, these actions are called speech acts. This is because initiating one has potentially far-reaching consequences for the state of the system that result directly from the fact that the interaction has happened; they do more than just progress a chain of sequential dialogue steps. A simple example of this would be a purchase action which not only places the responder under an obligation to supply goods, but also leaves the purchaser with an obligation to pay for them (perhaps within 30 days).

We separate the basic behaviour that covers all the things that might possibly happen from the behaviour aimed at achieving specific goals, which are made explicit by adding permissions and obligations. This is modelled by wrapping such obligations or permissions into tokens which are communicated as part of the speech act and thereby change the obligations or capabilities of the participants. Fig. 1 outlines the typical steps in the lifecycle of a deontic token. This example shows the transfer of an access permission, tracking the process through the following stages:

1. at some point the business process specifies that a permission is to be transferred, and an empty token is created.
2. this token is populated by information from the local access manager.
3. the token is made available to the invocation process, which initiates the required speech act.
4. the token becomes part of the speech act payload.
5. the recipient of the speech act schedules local actions to process the token in its local context.
6. this may involve adding the permission to its access management state.
7. in other circumstances, the recipient may simply act as an intermediary, passing on the token without interpreting it.

The advantage of making this division is that it simplifies the business process without needing to be explicit at that point on how a permission was obtained or an obligation is to be discharged. On the other hand, generic obligation handling templates can be set up that can then be applied to a number of pieces of basic dialogue.

Basic behaviour can be considered as expressing all the valid sequences of actions that can happen, and is similar to the definition of a protocol or process state machine. Actions outside the basic behaviour are violations, but not all actions within it are equally desirable. Decoration of the behaviour with deontic tokens allows the expression of preferences and goals.

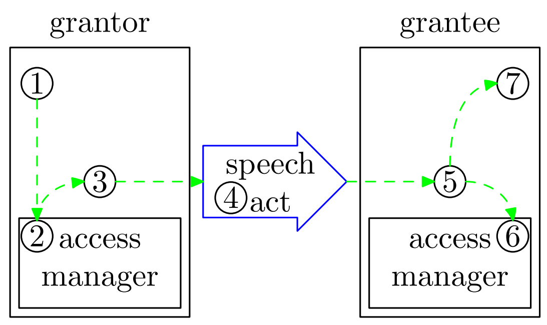

Fig. 1 Stages in the lifecycle of a deontic token

Any of the active objects in a design may hold tokens representing deontic information. Active objects in any role can create tokens called permits, burdens or embargos, and these may be subject to constraints or guards. They represent permissions, obligations and prohibitions, respectively. These tokens can then be passed to other objects as a result of performing speech acts with them.

An active object may also pass on a token it has received, if this better expresses the designer's intent than creating a fresh one, enabling, for example, a delegation pattern. One object holding an obligation to see that some action is performed in the future might arrange that this happens by passing the token expressing its burden on to an agent object. On receiving the token, the agent, by holding it, becomes obliged to see to it that the action is carried out. Thus, holding a deontic token constrains its holder to perform some behaviour that discharges an obligation or exploits a permit. However, there may be many possible behaviours that could do this, and the obligation does not determine exactly which option should be taken (or indeed guarantee that it can ever be discharged satisfactorily).

A token may require more than just that an action be performed eventually. It might, for example, state a time limit before which the action is required. In such cases, the passage of time may cause the violation of this guard expression (as may the results of other behaviours). The definitions of the architectural elements involved are in [2–4].

4 A tool-building roadmap

A system specification draws on many sources, and one of the requirements of the tool-chain is that these sources are merged in a consistent way. By consistency here, we mean logical consistency, so that the resultant merged specification

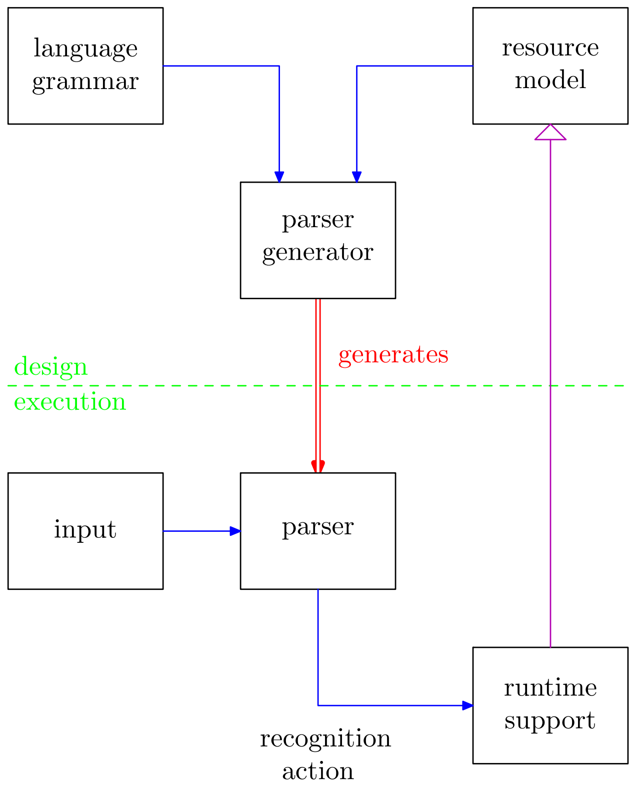

Fig. 2 The role of a language parser. Items above the green horizontal dotted line are involved in the design phase, and those below it are involved in runtime use. The magenta generalization relationship indicates that the runtime support refines the resource model, and the red double arrow indicates that the parser generator creates a parser for runtime use. The blue interactions show how the information needed by the various components is provided

can be subjected to model checking and any contradictions, or behaviour livelocks or deadlocks found. It is then up to the designer to resolve any problems found, since doing so requires knowledge of the design intent. The sources include, for example, broadly accepted standardized frameworks, enterprise structures and session types, policy specifications and control of non-functional aspects of management, such as those concerned with responsibility or trustworthiness. Over a period, any of these may evolve, and the tool support used needs to be structured so as to facilitate this evolution. The introduction of a DSL provides a pattern to manage this structuring.

Any language can be seen as a set of rules for recognizing and interpreting a sequence of tokens (see Fig. 2) and then updating the receiver's state in consequence. Recognition is based on matching the input to the possibilities allowed by the language grammar, but interpreting the utterance matched depends on knowledge of the state of the world. Implementing an interpreter therefore requires knowledge of a model of the resource being manipulated, which is to be updated as a consequence of matching the language elements. When a match is detected, an action must be performed to satisfy the language semantics, and the nature of this action is determined by the association defined in the language specification between language elements and local resources.

These resources can take many forms, but in the context of this paper might, for example, involve a model for a repository of deontic constraints, such as available permissions.

Linkage to this model can take many forms. The simplest is by sharing of a namespace, so that names in the utterance match those in the model, and appropriate actions, such as reporting or changing the state of the resource, are selected based on the type of the language element matched. In an object-oriented model, for example, object accessor or mutator actions might be tied to the position within the utterances, maybe by treating input parameters as object mutators and responses as object accessors. In more complex cases, the grammar can be decorated with instructions for manipulating the resource model in an application specific way.

However, this is not necessarily a one step process. Consider the transfer of deontic tokens introduced above. There are some basic pieces of behaviour making up the business process, themselves defined by a language, in which the tokens are seen as atomic objects. The semantic action on receiving a token is to recognize it as needing storage and eventual further processing of its content. However, this processing can treat the token transparently, without interpreting it.

When the time comes, a further instance of the language pattern is invoked, the nature of which is determined by the token type. This parses the stored token, which involves recognition of the deontic elements and invocation of methods identified in the deontic DSL and defined in the associated deontic runtime support.

This repeated series of evaluations of language elements to unwrap successive semantic facets forms the basis of a roadmap for tool construction proposed here. The separation of different areas of concern is realized by the introduction of fragments drawn from different domain-specific languages. The main steps in the roadmap are then as follows.

1. High-level architectural analysis to identify areas of concern which are largely independent.
2. Creation of corresponding resource models. Where possible these can be existing reusable standard models.
3. Definition of domain-specific languages for the management of each resource model type.
4. Generation of the DSL fragments needed for each aspect of the design (including the necessary semantic rules). These will have as terminal actions invocations of the parts of the runtime system created by refinement of the resource models.
5. Integration with a suitable test harness for model checking and validation.

This paper has first identified two distinct areas of concern related to the evolution of enterprise, namely policy (Sect. 2) and deontic tokens (Sect. 3), which are then used as a basis for the corresponding resource models and related DSLs, the concepts for which are introduced and then elaborated in the following sections.

5 An example of system evolution

As explained above, the flexible design of a complex system requires support for handling changes, both in terms of the variability of design choices supporting the selection of specific design parameters during the system life cycle and in terms of the re-assignment of responsibilities of parties involved, such as during delegation.

In terms of supporting variability of design choices, for example, one of the latest digital health standards, HL7 Fast Health Interoperability Resources (FHIR) [12], provides a good example of a specification that takes account of the need for future tailoring. This standard is part of a family supporting interoperability between loosely federated health support systems. The standard defines common information components ("FHIR Resources") across many different jurisdictions and clinical domains, such as patient, observation, medication and consent, for example, while allowing their tailoring (by constraints or extensions). Tailoring is often needed to reflect the requirements of specific jurisdictions, such as Australia, UK or Japan or any other domains of use, such as supporting the International Patient Summary [13].

How this tailoring is done is set out in an implementation guide (IG), which provides a set of rules about how FHIR Resources should be used to solve a particular problem, with associated documentation to support and clarify the usage. Another example is where FHIR provides an enumeration type with values that can be used in various common circumstances. These are referred to as value sets. For example, the current FHIR Consent Resource [14] includes a set of Regulatory consent policy types from the US and other regions, captured in the Consent.regulatoryBasis attribute of the Consent Resource.

The approach taken by FHIR is an instance of the more generic policy structure described above for supporting variability of design after the specification epoch.

We use the FHIR Consent Resource Framework to provide a working example, adopting the roles of grantor, grantee, consent enforcer and consent manager in defining a community and the supporting idea of *subject* to specify who the consent applies to (in ODP, a community defines how a group of participants should behave in order to achieve their objective). We can then go on to position our deontic language (see below) in relation to it, providing the content to be referenced by the Consent.policyBasis attribute.

There are, of course, many kinds of consent, but, for simplicity, we will concentrate on consent in the service of privacy management.

6 Factoring out obligations

The FHIR consent support also illustrates how obligations can be introduced in a controlled way, as explained in [29]. Consent occurs when one person (the grantee, who receives the consent statement) accepts and voluntarily agrees to the proposal or desires of another (the grantor, who has issued the consent statement). This is the same usage as we find in common speech, and also, in a more restrictive way, in fields such as the law, medicine and research.

In digital health, consent is a record of a healthcare consumer's choices (or choices made on their behalf by a third party); it permits or denies recipients individually, or by virtue of the roles they might take within a given policy context, to perform one or more actions, for specific purposes and periods of time; see [14].

The key deontic concepts here are permissions or prohibitions, which are given by the consumers (acting as grantors) to the recipients (who might be clinicians or researchers). These control their actions directly, while the *policy context* implies a set of background rules to apply to the recipients, such as medico-legal rules or perhaps business rules, all of which are defined as part of a broader consent community. These can not only take the form of obligations or authorizations, but can also involve broader accountability rules such as delegation of obligations or permissions, as can be found, for example, in a principal–agent relationship.

When a patient gives a consent to a clinician to access, for example, the health information data stored in the patient's electronic health repository, this action changes the deontic states of both parties and is thus regarded as a speech act. So, the patient can provide access to all their healthcare data, but restrict access to their mental health data, and this decision must be respected by the clinicians. This can be modelled as a permit to access all their health data, while the prohibitions over access to their mental health data can be modelled as a separate overriding embargo for the mental health data. This is a modelling style which can also be reflected in the policy language syntax adopted. Figure 3 illustrates the transfer of an explicit permission object. These deontic expressions can form the elements of the external policy language (as in the HL7 Consent Resource policyBasis attribute) and would be used to provide the added semantics for this FHIR Resource.

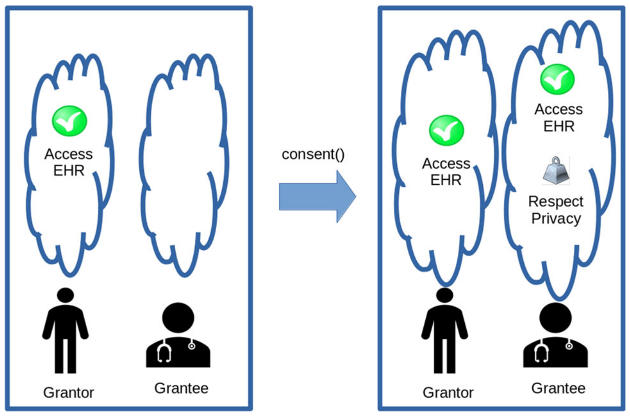

Fig. 3 How deontic elements are modified by performance of a consent speech act

7 Software engineering problems

The two specification features addressed in this paper raise different issues from a software engineering perspective. Handling a specification using policies is largely an exercise in unification, while adding deontic constraints is a matter of coverage and completeness, typically reducing the level of non-determination in the specification.

At any particular point in time, the specification elements can, in principle, be parsed and interpreted to build an overall system model, processing each element in a way consistent with the language in which it is expressed. Thus, information may be drawn from integrated specification tools, or from domain-specific languages matched to different parts of the problem domain. There is a need to map between the namespaces of different elements, taking account of variations in their levels of abstraction, but still producing a unified view.

A major problem arises, however, when policies change, since inconsistencies arise when there are interactions between different communities following diverging policies. It may be possible to control the propagation of new policies so as to avoid short-term inconsistencies, but some form of escalation will be needed when irreconcilable differences come to light.

Indeed, the potential for inconsistencies is one of the motivations for the introduction of the concept of a policy envelope. Knowledge of this envelope can be used to guide the implementation of dynamic checking and reconciliation when policies change. Attempting to change a policy to one which violates the envelope should result in a runtime error. Some gross errors, such as policy name clashes between different source components, should be flagged during integration.

The introduction of deontic tokens, on the other hand, was motivated by a desire to trade the extent of the system model against the level of non-determinism in it. Here the impact of adding more pieces of behaviour without further qualification is much more significant. Consider a system in which there are a number of different pieces of behaviour, perhaps ordering goods, making deliveries or making payments. These can all be described in detail, but each is seen as initiated by some unspecified internal action of one or other of the parties.

Adding deontic constraints implies modelling some sort of preference or goal-seeking behaviour. Thus, making a delivery places an obligation on the purchaser to make payment. The implication of this is that we need to introduce state and behaviour into the system model corresponding to to-do lists and scheduling functions. The behaviour is extended not by rigid succession, but by requirements that actions be performed eventually, although not necessarily immediately. The amount of detail to be modelled has grown significantly.

It is worth noting that considerable care is needed in validating the design process when obligations are involved. The transition from an abstract system specification to an implementation is often described as being represented by a refinement relationship, so that any solution in which the system specification is a valid abstraction of the implementation would be accepted. However, when we consider security, or, more particularly, privacy, this is not enough.

If we consider a design in which privacy is maintained by establishing the encapsulation of the information concerned within an object, accessed only by policed interactions with that object, there is a problem. It should be clear that a refinement which reveals an additional interface to the object, but without the necessary policing, will be valid from a behavioural point of view, but will destroy the privacy guarantees. This is because hiding the backdoor is a valid behavioural abstraction. It is therefore necessary to add constraints that preserve the encapsulation property, going beyond the requirements for behavioural refinement.

8 Language support

Having identified a number of distinct pieces of specification, we need to examine how to structure a tool-chain that is able to relate and unify them. This is already familiar territory, because the ODP framework is inherently viewpoint based, and so a specification using it is normally manifest as a set of viewpoint models linked by a set of pairwise correspondence models.

Tactically, we can approach the relation of these parts in two ways. One option is for the unification to be performed at a language level, deriving a composite grammar and processing all the elements of the specification using it. Alternatively, each element can be processed independently according to its own domain-specific language grammar to construct a set of related metamodels and a set of mapping rules between them. In either case we must consider the handling of both syntax and semantics.

The choice we have taken here is to work with a set of domain specific languages, yielding a set of metamodels. This is motivated, in part, by the wish to simplify the organization of semantic rules and increase modularity. Having each language associated with a minimum set of resource models reduces interdependencies. A number of effective solutions now exist for the support of DSL parsing, but the metamodels these produce have to be linked with rules for semantic interpretation. If we can factor semantic aspects in some way, we can make the additional rules for related behaviour more modular.

Indeed, this is why we introduce the management of deontic aspects in terms of token passing. Making the tokens strongly encapsulated allows their interpretation to be localized with the deontic structures and rules, largely decoupled from the basic behaviour. The fulfilment of obligations is at a different level of detail behaviourally from the policing of interaction protocols.

9 Domain Specific Languages

Domain-specific languages (DSLs), in contrast to general-purpose languages (GPLs), are languages specifically tailored to the particular domain of usage. They incorporate concepts from the domain at hand, providing better expressiveness and the possibility of direct involvement of subject-matter experts in the specification of the solution. Note that, in general, such DSLs can be either textual or graphical based languages, but, for simplicity, we restrict discussion in this paper to textual languages.

Domain-specific languages (DSLs) have been successfully applied over many years [7] to various domains such as rehabilitation [21], cognitive sciences [10], ecology [20], and business contract specification [26].

Domain-specific analysis, verification, optimization, parallelization, and transformation (AVOPT) are challenging, if not impossible, when using general-purpose languages [28]. On the other hand, AVOPT with DSLs is not only possible but also relatively straightforward to implement. This brings about a better user experience and consistency in the specification, which is an important feature for long-term developments.

Compilers and interpreters utilize concepts from the metamodel produced to execute system specifications. In the case of our language, this involves the execution of workflows to ensure that all actions are executed, deontic tokens and policies are handled, and condition and guard constraints are checked and resolved according to the system specification and language semantics, for example by updating of the health record. The workflow is executed by the ODP-EL interpreter which is configured by the system specification written in an ODP-EL DSL and is a part of the broader application implementation.

Subject-matter experts, in collaboration with system developers, use the DSL to generate the system specification. System developers are also responsible for developing parts of the system not covered by the DSL, such as updates of the electronic health record. Such actions are bound via references to the resource models declared in the DSL, as discussed in Sect. 4.

10 From specification to implementation

Section 9 highlighted the benefits of using several DSLs to express different aspects of an enterprise specification and thus support consistency, particularly when controlling system evolution and expressing the dynamics of obligations among active enterprise objects, such as parties to a contract.

10.1 Available tools

We believe the grammar-based tools, built with model-driven support, such as Xtext [35] and textX [11] provide a good foundation for implementing ODP-EL concepts.

Both of these tools include support for expressing DSL grammars in terms of a set of syntax rules which describe valid relationships between the constituent symbols of the language. These can be applied in a way that can be used by a parser to analyse the expressions and ensures their validity. They can also produce metamodels which represent the key concepts and relationships of the grammar.

Xtext is a language workbench component of the Eclipse-based EMF/Ecore tooling family and it integrates tightly with the Java and Ecore typing system. Xtext supports various aspects of language development, including defining the syntax ensuring well-defined structure and enabling efficient parsing, creation of the metamodel from the grammar and integration with the existing Eclipse environment. Considering its grounding as part of Eclipse tooling, Xtext is primarily aimed at a Java environment.

The textX tool started as an Xtext metalanguage implementation in Python, but has evolved further (see [34]). textX is simpler and more lightweight than Xtext, enabling fast DSLs development in Python that is IDE agnostic, provides a fast round-trip from grammar modification to testing and, although made with DSL development in mind, can also be used in different contexts [11]. This includes language support inside IDEs, support for human readable configuration languages, support for a Model-Driven Engineering tool-chain, analysis of legacy source code, and so on.

textX uses a single grammar to construct a parser and a metamodel at runtime—see Sect. 4. The metamodel contains all the information about the target language and a set of supporting Python classes inferred from grammar rules. The parser will parse programs or models written in the new language and will construct a Python object graph; that is, the model conforming to the metamodel. The model can subsequently be used for interpretation or source code generation.

The textX tool has support for error reporting, debugging, and metamodel and model visualization. The current version of the metalanguage, although similar to Xtext's, differs in various places providing some distinctive features [33].

10.2 Development of a DSL

We have been using these tools to check our intuition about the tool-chain requirements when supporting applications with the ODP framework. We began our experiments with textX, attracted by its lightweight approach to DSL development but have later transported the textX created grammar into the Xtext environment, with added benefits of interoperability with the DSL tooling.

We started with the development of deontic concepts, because of the need to implement several cross-organizational use cases, such as those presented in [24] and [29]. Our aim was to develop user-driven syntax to reflect the dynamics of the obligations, permissions and delegations, while maintaining clarity, simplicity and consistency of the expressions. This is particularly important for domain experts concerned with the expression of responsibility and accountability concepts, which are key for both organizational and inter-organizational processes. We then went on to incorporate concepts to support a set of policies.

The textX tooling allowed us rapid prototyping of the DSL in an iterative and incremental manner, focusing only on the concepts required by the use cases in question. This included quick changes in the grammar and syntax as new concepts were required to express more complex rules, entities or patterns.

For example, the use of the concept of event was initially introduced to signify occurrences such as activation of some permits (which were created during instantiation of community roles). In the example below, access_trigger (Fig. 4, line 19), which was originally a simple item, was subsequently replaced with a policy (see Fig. 4, line 49), to support expression of various event types modelled as policy values for such access_trigger events, namely observation_performed and emergency_situation event types. Another example was the need to express more complex rules associated with the permit_valid duration parameter, so that it can be changed from an initial value of 30 days to any value from 1 to 6 months, as stated in its envelope (see Fig. 4, line 56). There are many such examples where simpler rules are changes to more complex rules as a result of ongoing requirements analysis.

```
1  community consent {
2      # includes concepts to support (part of) HL7 FHIR privacy consent
3      objective "Support_patient_privacy_consent_preferences"
4      events {
5          observation_performed, emergency_situation
6      }
7      # artifacts − objects referenced by actions
8      artifact subject {
9          id ID
10         name string
11         birth_date string
12     }
13     role grantor {
14         speech act give_consent (subject)
15             [now − subject.birth_date > legal_age ] {
16             # permission for grantee to read subject's data; triggering
17             # event can exploit permission until permit_valid time
18             AccessEHR: permit on grantee (subject)
19                 triggered by access_trigger [this.time + permit_valid]
20             # obligation for respecting privacy compliance
21             RespectPrivacy: burden on grantee (subject)
22                 discharged by
23                     # This burden is discharged by the timeout only.
24                     # I.e. it doesn't observe any event
25                     [this.time + privacy_valid]
26         }
27     }
28     role grantee {
29         # access_EHR action allowed if a grantee holds AccessEHR permit.
30         action access_EHR(subject) [AccessEHR]
31         action perform_observation (subject)
32             emits observation_performed
33     }
34     role consent_enforcer {
35         monitorRespectPrivacy: burden
36         handleError: burden
37     }
38     role consent_manager {
39         # The actor that manages the consent lifecycle (see FHIR)
40         manageConsentWorkflow: burden
41     }
42     policy legal_age: duration {
43         policy setting by consent_manager
44         initial value 18 years
45         envelope {
46             this >= 16 years
47         }
48     }
49     policy access_trigger: event {
50         policy setting by consent_manager
51         initial value observation_performed
52         envelope {
53             one of (observation_performed, emergency_situation)
54         }
55     }
56     policy permit_valid: duration {
57         policy setting by consent_manager
58         initial value 30 days
59         envelope {
60             from 1 to 6 months
61         }
62     }
63     policy privacy_valid: duration {
64         policy setting by consent_manager
65         initial value 2 years
66         # Envelope references permit_valid policy; it is evaluated
67         # even when the permit_valid is about to be changed.
68         envelope {
69             from 1 to 10 years
70             this >= permit_valid
71         }
72     }
73 }
74
75 party doctor as consent.grantee
76 party patient as consent.grantor
```

Fig. 4 Example of the DSL specification for the consent community

```
Model:
    communities*=Community
    objects*=EnterpriseObject
;
EnterpriseObject:
    ActiveEO | DeonticToken
;
ActiveEO:
    Party | Agent
;
Community:
    'community' name=ID '{'
        'objective' objective=STRING
        events=Events?
        artifacts*=Artifact
        roles*=CommunityRole
        policies*=Policy
    '}'
;
Artifact:
    'artifact' name=ID '{'
        ('parties' parties+=[CommunityRole]['','])?
        properties*=Property
    '}'
;
Action:
    SpeechAct | BasicAction
;
BasicAction:
    'action' name=ID ('(' parameters*=[Artifact][','] ')')?
    guard=Guard?
    ('emits' triggers_event=Event)?
;
SpeechAct:
    'speech' 'act' name=ID ('(' parameters*=[Artifact][','] ')')?
    guard=Guard?
    '{'
        tokens*=DeonticToken
    '}' ('emits' triggers_event=Event)?
;
Events:
    'events' '{'
        events*=Event[',']
    '}'
;
Event: name=ID ('(' artifacts+=[Artifact][','] ')')?;

DeonticToken:
    Burden | Embargo | Permit
;
Embargo:
    name=ID ':'
    'embargo' 'on' action=ID guard=Guard
;
Burden:
    name=ID ':' 'burden' violation?='violation'
        ('on' role=[CommunityRole])?
        ('(' parameters*=[Artifact][','] ')')?
        (guard=Guard)?
        ('triggered' 'by' trigger=Event)?
        ('discharged' 'by' (finish_event=EventExpression)?)?
            (event_guard=Guard)?
;
Permit:
    name=ID ':'
    'permit'
    ('on' role=[CommunityRole])?
    ('(' parameters*=[Artifact][','] ')')?
    (guard=Guard)?
    ('triggered' 'by' trigger=Event)?
    ('expired' 'by' (finish_event=EventExpression)?)?
        (event_guard=Guard)?
;
Guard: '[' condition=Condition ']';

Condition: /\([^)]*\)|[^\]]*/;

EventExpression: op=AndExpression ('&' op=AndExpression)*;
AndExpression: op=PrimaryExpression ('|' op=PrimaryExpression)*;
PrimaryExpression: '(' op=EventExpression ')' | Event;

Comment: /#.*/;
FQN:        ID('.'ID)*;
```

Fig. 5 Example of part of the DSL consent grammar

10.3 An example of usage

A fragment of our deontic language for the e-consent example is shown in Figs. 4 and 5.

Figure 4 shows the consent example as written by a user. Here the resource model is implicit in the community role specifications, with linkage based on name equivalence, as discussed in Sect. 4. It defines a community with roles to be filled by the grantor and grantee. It also declared the supporting management roles for a consent enforcer and manager. These include standing obligations which are created by the current role so do not include *"on..."* and *"discharged by..."* parts normally found in common deontic expressions, because they are never discharged and are implicitly created in association with the current role.

The grantor role declares (in lines 13–27) the speech act that gives consent and the permits and burdens it conveys. The guard declared in line 15 limits applicability based on the subject's age.

The final section of the community specification (lines 42–72) illustrates the declaration of policies applicable to the community. They identify design options related to the likely changes in permissions and obligations associated with the consent actions.

Note that in the *permit valid* policy (lines 56-62), the affected behaviour is implicit in its use in token and action guards. For example, the policy affects the AccessEHR permit and consequently it affects the action access_EHR as this action depends on it.

The last two lines indicate that the roles in this community will each need to be bound to an appropriate party in the enterprise.

Figure 5 shows an illustrative part of the grammar generated to support the deontic aspects of the DSL described above. (The full grammar is too long to include here in its entirety.)

The result of parsing the consent example, using this grammar, is a machine processable structure suitable for interpretation of the design. For example, Fig. 6 shows a visualization of the metamodel it represents.

Comparison of this figure to the metamodel in the ODP-EL shows that there are significant differences. This is one of the consequences of the rapid prototyping approach taken, since there are none of the architectural considerations applied that contributed to the drafting of the original standard. If a different set of use cases had been chosen for the DSL development, a different structure would probably have been produced.

The example given here has illustrated how different specification components can be processed during integration. What we have not shown is the application of consistency checking to the resultant whole. When system evolution takes place, the individual steps will need to be repeated and the

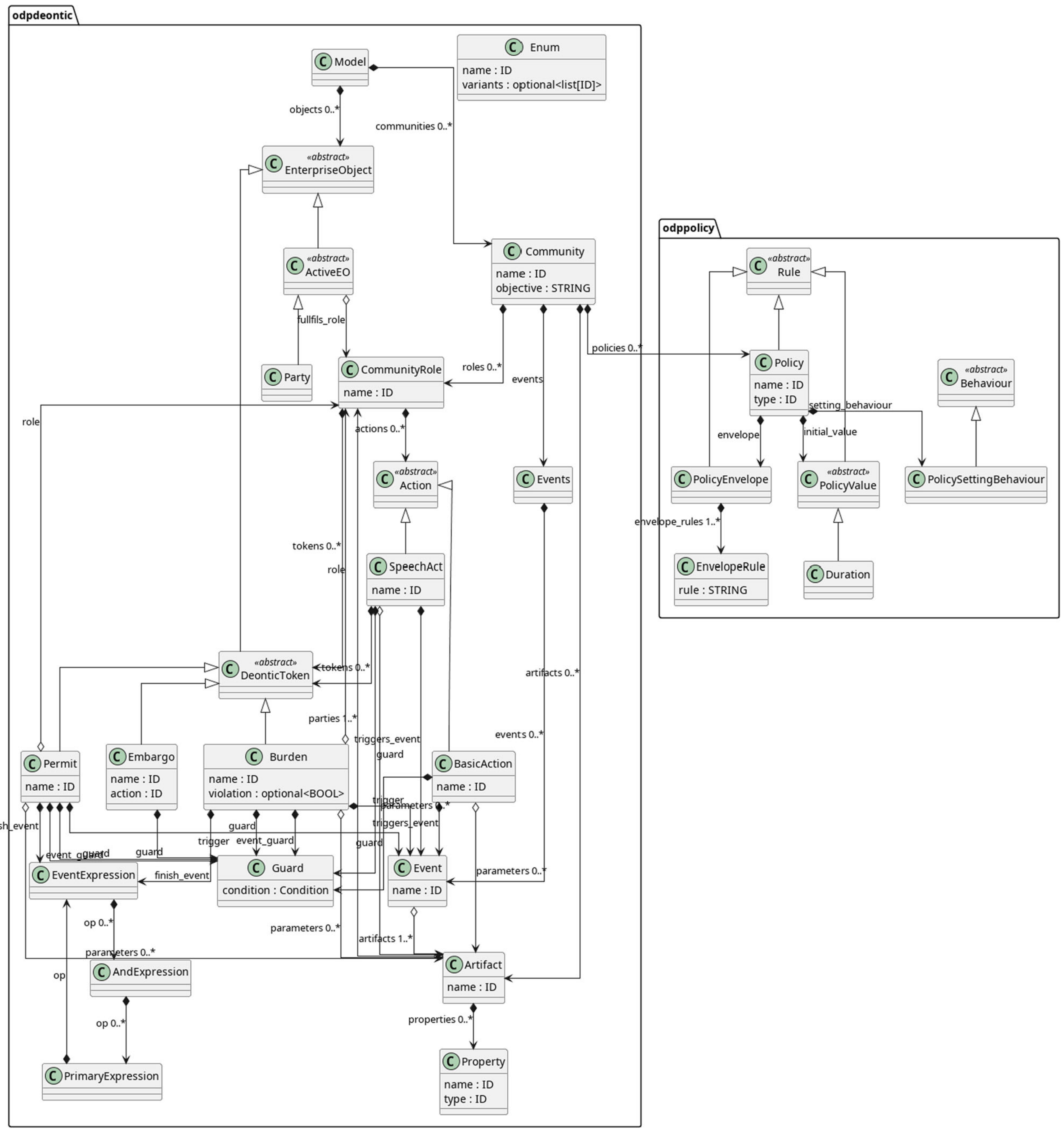

Fig. 6 The resulting metamodel

consistency checking redone. This process will need to be iterated until no problems remain.

11 Related work

We have already summarized the work on ODP that led up to this work, and the digital health standardization work used in our case study. The remainder of this section covers other areas of parallel activity.

11.1 Resource models

There are many individual pieces of resource modelling, but perhaps the most coherent framework supporting this kind of activity is that carried out to define the Unified Foundation Ontology [16]. This has a very broad field of application, but its use to represent legal and governance-related concepts is particularly relevant here. For example, [15] concentrates on an ontology of legal positions and how they can be described using the theory of constitutional rights, as proposed by the philosopher of law Robert Alexy [6]. This in turn extends the system of legal positions proposed by the jurist Wesley Hohfeld that includes eight fundamental legal concepts (right, duty, no-right, privilege, power, liability, disability and immunity) and their correlative positions [19]. These are positions with a counterpart in the same legal relation, such as duty in relation to right (for example, "John's duty to pay his debt to Mary" is "Mary's right that John pay his debt to her".).

The emphasis of the ODP Enterprise Language is more on organizational structure, and associated governance and accountability rules, permitting the tracing of responsibilities across actors in the system.

11.2 Standardization frameworks

There are also various standardization activities aimed at the creation of frameworks for expressing and exchanging information about legal constraints. Of particular interest are the Open Digital Rights Language [31] and the Semantics of Business Vocabulary and Business Rules [32].

11.3 Semantic tokens

We have focused here on deontic tokens, but there are many other uses of tokens in behavioural specifications. In particular, we need to distinguish these from security and authorization tokens. Such tokens may have some deontic properties, or they may be simple key values that enable access but do not allow any reasoning about obligations. A typical example is the OAuth2.0 authorization protocol [30], which uses uninterpreted access tokens to support client access to controlled data.

Both the OAuth2.0 authorization protocol and the FHIR resource models provide mechanisms for controlling the applicability of tokens based on qualification in terms of declaration of the scopes in which they can be applied. This allows access control options to be supported using a user-friendly interface (rather than a granular scope language), which is quite empowering for users since it gives them many options to select in a user interface. However, users can sometimes find this overwhelming, due to the complexity of available choices and the need for familiarity with the scope structure.

Another approach to simplifying user navigation is given in the US federal Trusted Exchange Framework and Common Agreement (TEFCA) [27]. While the expressions of scope provide lower level constraints on data access rules, the new requirements developed by the US TEFCA agreements provide higher-level, organizational constraints in line with the deontic token semantics. The TEFCA agreements specify rules for participants in the health information exchange network defining who is allowed to query data for what purpose and who is required to respond. This model puts significant trust in the participants but also requires monitoring of responses by the TEFCA coordinating authorities to ensure that responses are in line with privacy obligations and other regulatory requirements. TEFCA future agreements environment would significantly benefit from the richness of deontic and accountability rules available from our policy language. This will, in particular, be required in order to support the consumer-oriented principles regarding their data sharing through the networks. These principles are

1. providing permissions (that is, authorization) to consumers to query health data about themselves that may be spread across multiple providers,
2. ensuring "rights to know" for consumers when any provider or other party queries data about themselves; and
3. giving the ability to consumers to configure what is shared about themselves.

11.4 Policy languages

There have been many proposals for policy definition languages, dating back to trailblazing definitions, such as Ponder [9], and a review can be found in [17]. However, the main features introduced in the ODP Enterprise Language that these language definitions lack are the concept of a policy envelope and of the associated need to place constraints that must be satisfied by the policy in use if the system specified is to remain well-behaved. Without such constraints, an inappropriate policy can undermine the system's objectives.

11.5 Co-evolution

Whenever two or more software artefacts are interdependent, such that changes in one necessitate changes in the others to maintain global consistency, we refer to this as coupled software transformation or co-evolution [22]. In the context of domain-specific languages (DSLs), any changes to the grammar (or metamodel) require corresponding updates to all related programs or models to preserve consistency. A comprehensive overview of various approaches to this issue can be found in [18]. Our current DSL-based approach does not support automatic co-evolution, and addressing this limitation is deferred to future work.

12 Conclusions and future directions

We have reviewed two techniques for structuring and managing the evolution of enterprise specifications. We have also shown how the creation of the associated designs can be simplified by defining domain-specific languages tailored to their representation. However, this is just the first step in creating a tool-chain for deploying such designs. The ability to represent and browse the metamodel in use simplifies the communication of the design to implementors and maintainers.

Using the same description, we can foresee tooling to generate representations for deontic tokens and channels for transferring them. The techniques required are well known, dating back to the days of remote procedure call and beyond.

More challenging would be the integration of token passing with components involved in the management and policing of the deontic constraints themselves. One might draw parallels here with scheduling functions to be found in workflow systems. When an obligation is passed to an active object, it becomes part of the body of information to be used in selecting the actions to be initiated in order to discharge the obligation.

The resulting systems are likely to exploit a variety of technologies in addition to language processing. There will be a role, for instance, for model checkers and constraint solvers. There may also be a place for generative AI, although a critical review of its capabilities at present (see, for example, [8]) shows a gulf between mimicry and conscious design.

Other ODP Enterprise Language concepts can help manage change or evolution of the complex enterprise systems, and we plan to support some of these in the future. For example, the concept of a community contract allows for dynamic changes in configuration, by adding new roles, creating new communities or establishing community federations. These would all benefit from the use of deontic tokens, policies and the associated mechanisms for detecting policy conflicts and dealing with their resolution.

We are considering applying deontic and accountability concepts to serve as input to certain guardrails that can apply to behaviour of AI agents, in the context of their interaction with humans in complex enterprise systems. This is of particular value in systems which involve both human and AI agent actors, such as in digital health, which require clear expression of accountability, even in potentially complex chains of delegation across different types of actors.

There are many challenges still to be faced in exploring this vision.

We also believe that there would be value in investigating whether the ODP deontic and accountability concepts can be mapped onto various FHIR Resources, possibly as design patterns over FHIR Resources. This may be of benefit for simpler use cases where the use of DSL may not be necessary.

Acknowledgements The authors would like to acknowledge the major contribution made by Antonio Vallecillo to the development of the ODP standards and to the Enterprise Language in particular. He worked within the standards organization to clarify the concepts relating to policies and to the use of deontic tokens, and his contributions have added greatly to the precision and expressive power of these standards.

References

1. *ISO/IEC IS 10746-1, Information Technology — Open Distributed Processing — Reference Model: Overview*. Also published as ITU-T Recommendation X.901 (1998)
2. *ISO/IEC IS 10746-2, Information Technology — Open Distributed Processing — Reference Model: Foundations*. Also published as ITU-T Recommendation X.902 (2009)
3. *ISO/IEC IS 10746-3, Information Technology — Open Distributed Processing — Reference Model: Architecture*. Also published as ITU-T Recommendation X.903 (2009)
4. *ISO/IEC IS 15414, Information Technology - Open Distributed Processing - Enterprise Language 3rd edn*. Also published as ITU-T Recommendation X.911 (2015)
5. *ISO/IEC IS 19793, Information Technology — Open Distributed Processing — Use of UML for ODP System Specifications*. Also published as ITU-T Recommendation X.906 (2014)
6. Alexy R., Rivers J: A Theory of Constitutional Rights. Oxford University Press (2009)
7. Bettini, L.: Implementing Domain Specific Languages with Xtext and Xtend, 2nd edn. Packt Publishing (2016)
8. Cámara, J., Troya, J., Burgueño, L., Vallecillo, A.: On the assessment of generative AI in modeling tasks: an experience report with chatgpt and uml. Softw. Syst. Model **42**(22), 781–793 (2023). https://doi.org/10.1007/s10270-023-01105-5
9. Damianou, N., Dulay, N., Lupu, E., Sloman, S.: Ponder: A language for specifying security and management policies for distributed systems. the language specification - version 2.2. Technical Report DoC 2000/1, Imperial College of Science Technology and Medicine, Department of Computing, (2000)
10. Dejanović, Igor, Dejanović, Mirjana, Vidaković, Jovana, Nikolić, Siniša: Pyflies: A domain-specific language for designing experiments in psychology. Applied Sciences **11**(17), 27 (2021) https://www.mdpi.com/2076-3417/11/17/7823
11. Dejanović, Igor, Vaderna, Renata, Milosavljević, Gordana, Vuković, Željko: TextX: A python tool for domain-specific languages implementation. Knowledge-Based Systems **115**, 1–4 (2017) http://www.sciencedirect.com/science/article/pii/S0950705116304178
12. *Fast Healthcare Interoperability Resources V5.0.0*, (2023). http://hl7.org/fhir/R5/

13. *Fast Healthcare Interoperability Resources: International Patient Summary*, (2024). http://hl7.org/fhir/uv/ips/ImplementationGuide/hl7.fhir.uv.ips
14. *Fast Healthcare Interoperability Resources: Consent*, (2023). https://build.fhir.org/consent.html
15. Griffo, C.L., Almeida, J.P.A., Guizzardi, G.: *Legal Theories and Judicial Decision-Making: An Ontological Analysis*, volume 330 of *Frontiers in Artificial Intelligence and Applications*, pages 63–76. IOS Press, (2020)
16. Guizzardi, G., Benevides, A.B., Fonseca, C.M., Porello, D., Almeida, T., João, P.A., Sales, P.: UFO: Unified foundational ontology. Appl. Ontol. **17**(1), 167–210 (2022)
17. Han, W., Lei, C.: A survey on policy languages in network and security management. Comput. Netw. **56**(1), 477–489 (2012)
18. Hebig, Regina, Khelladi, Djamel Eddine, Bendraou, Reda: Approaches to co-evolution of metamodels and models: a survey. IEEE Trans. Softw. Eng. **43**(5), 396–414 (2016)
19. Wesley Newcomb Hohfeld: Some fundamental legal conceptions as applied in judicial reasoning. Yale Law J. **23**(1), 16–59 (1913)
20. Johanson, Arne N., Hasselbring, Wilhelm: Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment. Empir. Softw. Eng. **22**(4), 2206–2236 (2016)
21. Kosar, Tomaž, Zhenli, Lu., Mernik, Marjan, Horvat, Marjan, Črepinšek, Matej: A case study on the design and implementation of a platform for hand rehabilitation. Appl. Sci. **11**(1), 389 (2021)
22. Lämmel, R.: Coupled software transformations revisited. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, pages 239–252, (2016)
23. Linington, P. F., Milosevic, Z., Tanaka, A., Vallecillo, A.: *Building Enterprise Systems with ODP: An Introduction to Open Distributed Processing, 1st Edition*. Chapman&Hall/CRC Innovations in Software Engineering and Software Development, (2011)
24. Linington, P. F., Miyazaki, H., Vallecillo, A.: Obligations and Delegation in the ODP Enterprise Language. In *IEEE 16th International Enterprise Distributed Computing conference*, (2012)
25. Linington, P.F.: Policy specification: Meeting changing requirements without breaking the system design contract. In *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)*, (2006). https://doi.org/10.1109/EDOCW.2006.81.
26. Linington, P.F., Milosevic, Z., Cole, J., Gibson, S., Kulkarni, S., Neal, S.: A unified behavioural model and a contract language for extended enterprise. *Data and Knowledge Engineering*, 51(1):5–29, (2004). Contact-driven coordination and collaboration in the Internet context
27. Mandel, Joshua C., Pollak, J.P., Mandl, Kenneth D.: The patient role in a federal national-scale health information exchange. J. Med. Internet Res. **24**(11), e41750 (2022)
28. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. (CSUR) **37**(4), 316–344 (2005). https://doi.org/10.1145/1118890.1118892
29. Milosevic, Z., Pyefinch, F.: Computable Consent - From Regulatory, Legislative, and Organizational Policies to Security Policies. In *Enterprise Design, Operations, and Computing. EDOC 2022. Lecture Notes in Computer Science, vol 13585*, (2022). https://doi.org/10.1007/978-3-031-17604-3_1
30. *The OAuth 2.0 Authorization Framework*, (2018). https://datatracker.ietf.org/doc/html/rfc6749
31. *ODRL Information Model 2.2*, (2018). https://www.w3.org/TR/odrl-model/
32. *Semantics Of Business Vocabulary And Business Rules*, (2019). https://www.omg.org/spec/SBVR/1.5/About-SBVR
33. *Comparing textX to other tools*. http://textx.github.io/textX/latest/about/comparison/
34. *textX grammar*. http://textx.github.io/textX/latest/grammar/
35. *Language Engineering for Everyone!*, (2015). https://eclipse.dev/Xtext/index.html

Peter Linington is Emeritus Professor of Computer Communication at the University of Kent, UK.His research interest cover a range of distributed systems and networking topics, particularlymodelling at the boundary between computational and business process views. He has been activelyinvolved in international standardization of OSI and ODP over a period of forty years.

Zoran Milosevic is a Principal at Deontik, Australia, a boutique consulting business specializing in enterprise systems planning, development, and deployment, particularly in digital health, finance, and digital twins. His research interests encompass enterprise architecture, interoperability, agent AI, and computable policies, while also contributing to HL7, ISO and OMG standards over years.Zoran was the founder of the EDOC (Enterprise Design and Operations Conference), a well-established event attracting industry leaders, researchers, and academics. He is a Senior Member of IEEE, Fellow of the Australian Computer Society, a founding member of the Australasian Institute for Digital Health, and a passionate advocate for the ethical and responsible development of technology.

Akira Tanaka consulting company in Japan, specialized in applying viewpoints and model-based approaches to software development. He has been involved in RM-ODP standardization from its early days. He led the ODP committee of INTAP in Japan, participated frequently in EDOC's WODPEC. He was also active in OMG, including as a contributor to the UML Profile for EAI specification and UML Profile and Metamodel for Services (UPMS) RFP. Further information can be found at http://www.view5.co.jp/e/.

Igor Dejanović is a Professor of Computer Science at the University of Novi Sad, Serbia. His research primarily focuses on Software Language Engineering, withexpertise in meta-modeling, domain-specific languages, and parsing techniques. In addition to his academic work, he runs a boutique consulting business thatapplies his research insights to real-world industry challenges. An activecontributor to the open-source community, Igor maintains several Free/Libre and Open Source Software (FLOSS) projects.