Cooperative Specification via Composition Control

Christopher A. Esterhuyse
Informatics Institute, University of Amsterdam
Amsterdam, Netherlands
c.a.esterhuyse@uva.nl

L. Thomas van Binsbergen
Informatics Institute, University of Amsterdam
Amsterdam, Netherlands
ltvanbinsbergen@acm.org

Abstract

High-level, declarative specification languages are typically highly modular: specifications are comprised of fragments that are themselves meaningful. As such, complex specifications are built from incrementally composed fragments. In a cooperative specification, different fragments are contributed by different agents, usually capturing requirements on different facets of the system. For example, legal regulators and system administrators cooperate to specify the behaviour of a data exchange system. In practice, cooperative specification is difficult, as different contributors' requirements are difficult to elicit, express, and compose.

In this work, we characterise cooperative specification and adopt an approach that leverages language features specifically introduced for controlling specification composition. In our approach, specifications model the domain as usual, but also specify how specifications may change. For example, a legal regulator defines 'consent to process data' and specifies which agents may consent, and which relaxations of the requirement are permitted. We propose and demonstrate generic language extensions that improve composition control in three case study languages: Datalog, Alloy, and eFLINT. We reflect on how these extensions improve composition control, and afford new data exchange scenarios. Finally, we relate our contributions to existing works, and to the greater vision of multi-agent data exchange to the satisfaction of their shared, complex, dynamic requirements.

CCS Concepts: • **Software and its engineering** → *Collaboration in software development*; • **Social and professional topics** → *Computing / technology policy*; • **Theory of computation** → *Program specifications.*

Keywords: Specification Languages, Program Composition, Program Refinement, Data Exchange, Meta-Programming

ACM Reference Format:
Christopher A. Esterhuyse and L. Thomas van Binsbergen. 2024. Cooperative Specification via Composition Control. In *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering (SLE '24), October 20–21, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3687997.3695635

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
SLE '24, October 20–21, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1180-0/24/10
https://doi.org/10.1145/3687997.3695635

1 Introduction

Data exchange systems are large distributed systems, concerned with the controlled exchange and processing of data across organisational boundaries [26, 27, 38]. Each system is driven by the participation of autonomous *agents*. These include organisations involved in the data exchange, or cyber-physical entities driving system behaviour at runtime. When these systems behave counter to agents' requirements or expectations, agents may be harmed. For example, if the medical records of patient Bob are unintentionally broadcasted to international data processors, (the privacy of) Bob is harmed. It is imperative that these systems be carefully controlled to minimise harm. At the very least, this is a fundamental legal requirement in the European Union, for example, as per the EU General Data Protection Regulation (GDPR) [11].

In this work, we take a typical approach to controlling complex systems in general, and data exchange systems in particular; (the desirable behaviour of) the system is *specified* in some formal language, producing an artefact called a *specification*. This approach clarifies a way for agents to agree on their requirements on the system. First, the agents agree on the specification language. Then it suffices for them to agree which specification captures each of their requirements. A sufficiently precise and expressive language affords robust and predictable reasoning about specifications. Finally, the agents reason about the real system itself via its specification, relying on the *enforcement* of the specification, which maintains the compliance of the system to its specification, for example, by automated enforcer agents.

The focus of this work is not enforcement, whose theory and practice is explored in other works. For example, in some cases, non-compliance is prevented ("ex-ante enforcement"), e.g., employing access control [15, 31], model checking [7], or by generating implementations from specifications. In other cases, enforcement is instead continuously or periodically monitored, detected, and corrected ("ex-post enforcement"), e.g., employing usage control [19, 29, 30], runtime verification [3], or process mining [39]. The focus of this work is also not the ways specifications model physical systems or conceptual domains of discourse. For example, many specification languages differ in their fundamentals,

for example, providing different core ontologies from which their specifications are built. For example, both eFLINT [37] and Symboleo [32] propose an essentially relational worldview, but only Symboleo builds specifications using *Contract* as a primitive concept, while eFLINT instead provides *Fact* and *Event* as primitives, which can be instantiated and composed to model contracts. Moreover, multiple semantics may be associated with the same syntax, each with different reasoning capabilities and computational complexities, as is the case for the various profiles of OWL2 [41].

This work focuses on the features of specification languages supporting *cooperative specification*, a process by which cooperating agents systematically develop a shared specification that captures all their requirements. Precisely, the specification is assembled incrementally as agents contribute new parts. The challenges typical to developing specifications are exacerbated by the characteristics of data exchange systems in particular: these systems are subject to complex and changing requirements, as a consequence of the large number and variety in the agents, reflecting the number of participating organisations, and the complexities of their roles in the exchange of data. For example, legal experts formalise institutional entities and their relationships, process orchestrators coordinate the processing and communication of automated workers, while data providers negotiate, refine, and update the conditions on the use of data. For these agents to develop a shared specification, at some point, in some form, the agents must reason about, communicate about, and control each others' contributions to the specification. How are conflicts between requirements identified and resolved? How do agents determine which properties of the specification may be changed?

Motivating Example. Three agents cooperate to develop a shared eFLINT specification of their data exchange system that captures all their requirements. "Administrator" Amy begins by formalising the notion of *processing* as an agent-data relation. The relation has no elements, but Amy intends for this to be changed by subsequent contributions:

```
Fact processing Identified by agent * data.
```

"Barrister" Bob contributes the notion of *consent* (to the processing of data) as an agent-consent relation:

```
Fact consent Identified by agent * processing.
```

"Data-consumer" Dan contributes an assertion of the membership of particular processing and consent elements:

```
Extend Fact processing Derived from
  processing("Dan", "Amy's XRays").
Extend Fact consent Derived from
  consent("Amy", processing("Dan", "Amy's XRays")).
```

Both extensions of Dan's contribution are treated similarly by the semantics of the language. However, the latter extension violates Bob's meta-level requirement that consent somehow only originates at the agent in question; Dan must not grant Amy's consent! The specification fails to sufficiently capture these kinds of meta-level requirements.

Our approach is to leverage specification languages for (homogeneous) *meta*-specification: the specification captures domain concepts as usual, but also, it captures agents' requirements of how the specification is permitted to change. We are inspired, in part, by cooperative programming, where it is commonplace for programs to constrain their extensions; as a simple example, Java class members marked *private* capture meta-level requirements and harden the program against bugs. Prior contributors experience this as control over the specification. Future contributors experience this as insight into which contributions are considered permissible. The homogeneity of this approach enables the robust formalisation of requirements inter-relating domain concepts (like *data*) to meta-level concepts (like *agents* and *specification-contributor*). Subsequently, we evaluate existing languages through this lens, and propose language extensions that improve their suitability to cooperative specification.

Precisely, in this article, we contribute the following:

1. a definition of *composition control* in a specification language, affording reasoning about its suitability to cooperative specification (Section 2),
2. the definition and evaluation of various composition control mechanisms in the Datalog language, which is chosen for its simplicity (Section 3), and
3. the evaluation of the suitability of realistic specification languages Alloy and eFLINT to cooperative specification, before and after language extensions adapted from those shown in Section 3 (Section 4).

Throughout the main sections, we interleave background material and discussion of our findings where it is needed. The remaining sections discuss the relation between our contributions and existing works like programming language features and abstract argumentation frameworks (Section 5), reflect on our findings of our contributions and lay out the most immediately promising future works (Section 6), before we conclude with a summary (Section 7).

2 Definitions up to Composition Control

This section introduces our fundamental concepts and terminology concerning 1. specification languages, 2. the cooperative specification process, and 3. composition control, which makes a language useful for cooperative specification.

2.1 Language Schema

We introduce *language schemas* as abstractions over a wide range of specification languages. In summary, specifications are composable, have determined semantic meanings.

Let a *language schema* be a quintuple $\langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$, whose elements satisfy the following properties. We accompany each with a description of the underlying intuition:

- $\mathcal{P}$ are the *programs*, a formal language, i.e., a set defined inductively by formal grammar rules. Agents write and communicate programs.
- $\mathcal{M}$ are the *models*, another formal language. Agents conceptualise their domain via models.
- $[\![]\!] : \mathcal{P} \to \mathcal{M}$ maps programs to models. Conceptually, this attributes each syntactic program its semantic model. We style the application of $[\![]\!]$ to p as $[\![p]\!]$.
- $\circ : \mathcal{P} \to \mathcal{P} \to \mathcal{P}$ composes syntactic programs.
- $\checkmark : \mathcal{P} \to \mathbb{B}$ recognises the subset of *valid* programs. Intuitively, this characterises a crucial program property like "sensible" or "internally consistent".

In the sequel, we use *model* to refer specifically to the function $[\![]\!]$ and its codomain, $\mathcal{M}$. We use *semantics* to include any reasoning about programs. We distinguish *static* from *dynamic* semantics: only dynamic semantics is defined in terms of a program's model. For example, type-checking a program is a matter of static semantics, while testing the model-equivalence of programs is a matter of dynamic semantics. We use this distinction later to roughly categorise language schemas. For example, Section 3.2.4 gives a static definition of *valid*; hence, the validity of these programs can be checked without the cost of computing their models.

Definition 2.1 captures the concept of refinement, central to our approach (and we henceforth use $\triangleq$ for definitions):

Definition 2.1 (p_1 refined by p_2). $p_1 \triangleright p_2 \triangleq$

$$\checkmark(p_1) \ \wedge \ \checkmark(p_1 \circ p_2) \ \wedge \ [\![p_1]\!] = [\![p_1 \circ p_2]\!]$$

In the context of some language schema, $p_1 \triangleright p_2$ asserts that program p_1 is refined by p_2 (conversely, p_2 refines p_1) such that p_1 can be replaced by the extended $p_1 \circ p_2$ whilst preserving the model and validity of p_1. Refinements are of interest when the extension p_2 restricts future contributions, i.e., serves as a mechanism to control composition.

Demonstration. We demonstrate the use of the language schema with a simple, synthetic example definition of $\mathcal{S}_{\mathbb{N}} \triangleq \langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$. Let the programs ($\mathcal{P}$) be the powerset of natural numbers ($2^{\mathbb{N}}$), i.e., each "program" $p \in \mathcal{P}$ is some set of numbers $p \subseteq \mathbb{N}$. Let the models ($\mathcal{M}$) be natural numbers, and let $[\![p]\!]$ give the maximum in p if it exists, or 0 otherwise. Let program composition ($\circ$) be set-union ($\cup$). And finally, let each $\checkmark(p)$ check that no number in p is divisible by another in p. Although artificial, $\mathcal{S}_{\mathbb{N}}$ demonstrates how we define and reason about language schemas. For example, in the case of $\{5, 6\} \circ p$, validity is preserved by $p \triangleq \{7\}$, but not by $p \triangleq \{3\}$. Furthermore, we can observe that $\{5, 6\} \triangleright \{4\}$.

2.2 Cooperative Specification

We introduce *cooperative specification*: a sequence of agents $\langle a_1, a_2, a_3, ..., a_n \rangle$ take turns to develop a shared specification.

First, the agents agree on the specification language, which we represent as language schema $\langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$. Next, they define program $(((p_1 \circ p_2) \circ p_3) \circ ...) \circ p_n$, where each p_i is contributed by a_i. Note that agents may contribute repeatedly, i.e., no a_i and a_j are necessarily distinct. Often, but not necessarily, each a_i has knowledge of $(((p_1 \circ p_2) \circ p_3) \circ ...) \circ p_i$ at the time p_i is contributed. Equivalently, in turns, agents replace program p with some $p \circ p'$. In either case, the process is constrained by the a priori agreement between agents that composition must always preserve program validity. This can be understood as internalising a minimal valuation of programs, capturing a fundamental Deontic modality: a given contribution is *permitted* iff it preserves validity.

Demonstration. Amy and Bob agree to cooperate in the definition of a shared specification. They agree to use $\mathcal{S}_{\mathbb{N}}$ as their specification language. Amy contributes program $\{5, 6\}$, then Bob contributes $\{13\}$, and finally, Amy contributes $\{11\}$. Consequently, Bob and Amy agree on the shared program $p = \{5, 6, 11, 13\}$, its semantic meaning $[\![p]\!] = 13$, and that they cooperated successfully, i.e., that $\checkmark(p)$ holds.

2.3 Composition Control

We characterise the *composition control* of a given language schema $\langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$ by reasoning about its *control triples*, which we define as the set containing each $\langle p_1, p_2, p_3 \rangle$ where $(p_1 \triangleright p_2) \wedge \checkmark(p_1 \circ p_3) \wedge \neg\checkmark(p_1 \circ p_2 \circ p_3)$. In other words, replacing p_1 with $p_1 \circ p_2$ preserves its model and validity, but removes the validity of the composition with p_3.

Intuitively, each control triple $\langle p_1, p_2, p_3 \rangle$ identifies an opportunity for the agent contributing p_1 to intervene in its composition with p_3, by instead contributing $p_1 \circ p_2$. This formulation separates the concerns of p_1 and p_2: only p_1 directly concerns the semantic model, and only p_2 constrains the validity of the composition with p_3.

Demonstration. The prior language schema $\mathcal{S}_{\mathbb{N}}$ exhibits non-trivial composition control, i.e., it has some control triples. For example, Amy's first contribution exploits control triple $\langle \{6\}, \{5\}, \{10\} \rangle$: Amy's inclusion of 5 did not alter the model (6) but it did constrain Bob's subsequent contribution. (Perhaps the number 10 is somehow undesirable?) Thus, Amy and Bob cooperatively modelled the semantic concept (6) and characterised permitted changes specification, and so, an undesirable change was avoided ($\cup\{10\}$).

3 Composition Control Features in Datalog

This section presents a set of generic, composition control language features. Throughout this section, we use Datalog as the cooperative specification language. We select Datalog in particular, as it strikes a desirable compromise between complexity (affording varied and meaningful specifications) and simplicity (affording succinct definitions and examples).

Grammar Notation. In the remainder of this article, we define several formal languages inductively via grammar rules. Syntactic categories are defined by non-terminal symbols, denoted in italics (like *rule*). We omit the definitions

of common and trivial syntactic categories and grey their symbols (like *letter*). Concrete, terminal tokens are monospaced and brightly coloured (like `statement` and `:-`). Each $:=$ defines a grammar rule, rewriting from left to right, with alternatives separated by |. We use $x?$ to denote an optional x element and ϵ to mark the absence of tokens. We use x^* to denote x-lists (i.e., let $x^* \triangleq \epsilon \mid x\ x^*$). Finally, x^{y+} and x^{y*} denote lists with at least 1 and 0 elements, respectively (i.e., let $x^{y+} \triangleq x\ (y\ x)^*$ and $x^{y*} \triangleq x^{y+}?$). For example, given $(D :=$ `-` $?\ \mathit{digit}^+)$, D denotes signed integers like -23.

3.1 The Datalog Language

Datalog has been discussed in the literature for decades, including a rich exploration of various language extensions such as *weak negation* and *composite objects*; [6] overviews the Datalog language variants, but the original Datalog language is sufficient for our purposes. We expect many readers to be familiar with Datalog to some extent, but we give our own semi-formal account to be clear. Firstly, the following grammar gives the syntax of Datalog programs:

$$\begin{array}{rlrl} \mathit{program} & := \mathit{rule}^{\cdot *} & \mathit{rule} & := \mathit{head}\ (\texttt{:-}\ \mathit{body})? \\ \mathit{head} & := \mathit{atom} & \mathit{atom} & := \mathit{predicate}\ (\texttt{(}\ \mathit{arg}^{,*}\ \texttt{)})? \\ \mathit{body} & := \mathit{atom}^{,*} & \mathit{arg} & := \mathit{variable} \mid \mathit{constant} \\ \mathit{predicate} & := \mathit{constant} & \mathit{variable} & := \mathit{uppercase\ letter}^* \\ & & \mathit{constant} & := \mathit{lowercase\ letter}^* \end{array}$$

For example, the following is a 3-rule program:

```
bad(A) :- no-access(A, D), access(A, D).
no-access(amy, data1).     access(amy, data1).
```

Datalog affords a straightforward operational semantics: rules populate an initially-empty set of *truths* $\subseteq$ *atom* from existing truths to a fixed point. Desirably, Datalog programs also afford a straightforward logical interpretations: rules are implications, bodies are conjunctions of atoms, and atoms are first-order, atomic, Boolean logical variables. For example, the example program above has the model with truths { `no-access(amy, data1)` , `access(amy, data1)` , `bad(amy)` }. Each model determines the outcomes of all conceivable queries by containing all but only those atoms identified as true by the model. For example, `bad(amy)` is true, while `bad(bob)` and `eats(bob, baked, potatoes, 500)` are each false.

Despite its simplicity, Datalog has seen significant application in practice, as it enables complex reasoning.

3.2 Composition Control in Datalog Variants

3.2.1 Datalog with Trivial Validity.

We define language schema $\mathcal{S}_D \triangleq \langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$ to capture Datalog as straightforwardly as possible, as follows:

- $\mathcal{P} \triangleq$ *program*, the syntactic category of Datalog programs, i.e., rule sequences.
- $\mathcal{M}$ is the powerset of Datalog atoms, i.e., atom sets.
- $[\![]\!]$ captures the typical Datalog semantics.
- $\circ$ is given by the usual, syntactic program concatenation, such that the composite program $p_1 \circ p_2$ has the rules of the union of the rules in programs p_1 and p_2.
- $\checkmark$ is a trivial, constant function: each program is valid.

A hallmark of the semantics of many Datalog variants in the literature, is that the semantics disregards the order of program rules. Thus, in $\mathcal{S}_D$, $\circ$ associates and commutes over $[\![]\!]$. This eases the burden on agents cooperating to define composite $p_1 \circ p_2$, as the semantics imposes no ordering on their contributions. However, $\mathcal{S}_D$ affords no meaningful composition control, as all programs are valid.

Demonstration. Agents Amy and Bob cooperate to define composite $p_1 \circ p_2$. By defining p_1, Amy constrains $[\![p_1 \circ p_2]\!]$, but not p_2. For example, if $p_1 \triangleq$ `no-access(amy, data1)`, Amy asserts that `no-access(amy, data1)` $\in [\![p_1 \circ p_2]\!]$. However, Amy cannot control, and Bob cannot infer, which definitions of p_2 Amy considers permissible. Did Amy omit `no-access(bob, data1)` intentionally? Is Amy leaving this addition to Bob, or does Amy not permit its truth? Generally, $\mathcal{S}_D$ programs do not specify how they may be changed. Amy experiences this as a lack of control over Bob. Bob experiences this as a lack of insight into what Amy permits.

3.2.2 Simple Dynamic Validity.

We define $\mathcal{S}'_D$ identically to the prior $\mathcal{S}_D$, except defining validity as follows: $\checkmark(p) \triangleq$ `error` $\notin [\![p]\!]$. This attributes special significance to the atom `error`, which is otherwise entirely ordinary.

$\mathcal{S}'_D$ has many control triples. Consider $\langle p_1, p_2, p_3 \rangle$, where:

$p_1 \triangleq$ `knows(amy, bob).` $\quad p_2 \triangleq$ `error() :- knows(X, dan).`

$p_3 \triangleq$ `knows(amy, dan)`.

Replacing p_1 with $p_1 \circ p_2$ preserves its validity and its model: {`knows(amy, bob)`}. However, compositions including both p_2 and p_3 are invalid. Intuitively, p_2 translates the presence of the undesirable truth `no-access(amy, data1)` to invalidity.

Extrapolating from this example, $\mathcal{S}'_D$ affords a straightforward means of composition control: agents internalise impermissibility of the truth of a as a rule invalidating the program on the condition that a is true. We call this a *dynamic* form of composition control, as a program's (in)validity is entangled with its model. Thus, $\mathcal{S}'_D$ treats Datalog as a (simple) *homogeneous meta-programming* language: Datalog constructs specify the use of Datalog constructs. On the one hand, this approach comes at the cost of complexity; if computing a program's model is costly, determining its validity is costly also. On the other hand, this approach unifies reasoning about domain concepts and specification concepts, ergonomically affording significant expressive power.

The reader may recognise (`error :-...`) rules as mimicking the *constraint rules* common to various logic programming languages and essential in answer-set programming.

Demonstration. Once again, Amy and Bob cooperate to define $p_1 \circ p_2$. Amy defines $p_1 \triangleq$ `no-access(amy, data1).`

`error :- no-access(bob, data1)`, and leaves p_2 for Bob to define. By their shared goal of preserving validity, Bob's definition of p_2 is constrained. Amy experiences this as control over Bob. Bob experiences this as insight into what Amy permits. This time, Amy's contribution makes explicit that atom `no-access(bob, data1)` is intentionally false. However, the power imbalances between agents remain external to the specification. For example, p_2 prohibits the contribution of rule `no-access(bob, data1)` by anyone, and not just Bob.

3.2.3 Dynamic Validity Reflecting Contributors. We define $\mathcal{S}''_D$ as a variant of the prior $\mathcal{S}'_D$ by introducing a static restriction on the contributions each agent can make to the program. Specifically, we force each contribution to reflect the identity of its contributor. Thus, for example, Amy can constrain Bob's contributions by constraining truths that reflect Bob's identity. This conflates two notions of "agent" that are usually separated: agents are meta-level entities contributing and using specifications, but agents are also represented in specifications as domain-level concepts.

Precisely, for each rule r in each program p whose head has atom a as the first parameter, the contributor of p is identified by a, which is variable-free. For example, only Amy can contribute program `no-access(amy, data1)`, while the contributor of program `error() :- no-access(bob, data1)` is unconstrained, because `error` has no first parameter. This partitions rules with parameterised heads over agents; each such rule identifies the agent able to contribute it. However, note that it presents no real loss of expressiveness. For example, with rule `access(amy, bob, data1)`, Amy can still assert that Bob accesses data1, but crucially, not without identifying Amy as the contributor. Finally, observe that some programs are necessarily composed from the contributions of several agents. For example, Amy and Dan must cooperate to create program `welcome(amy, bob). welcome(dan, amy)`.

Demonstration. Amy and Bob cooperate in developing a specification. Amy contributes `no-access(amy, amy, data1). error() :- no-access(X, X, data1)`. This prevents only Bob from restricting Bob's access to data1. This demonstrates a powerful pattern: programs meaningfully internalise agents (like Amy and Bob), their relationships to each other, and their relationships to domain concepts (like data1).

3.2.4 Static Validity via Type-Sealing. Finally, we define $\mathcal{S}'''_D$ as a variant of $\mathcal{S}''_D$ which has a static definition of validity, by introducing a minimal syntactic extension to Datalog. Precisely, let $\mathcal{P} \triangleq program'$, defined as follows:

$$program' \coloneqq phrase^{\cdot *} \qquad agent \coloneqq constant$$
$$phrase \coloneqq \texttt{rule}\ rule \mid \texttt{seal}\ predicate\ \texttt{except by}\ agent^{,+}$$

We call q and c the *type* and *contributor*, respectively, of each rule matching $(q(c, \ldots)\ldots)$. As in $\mathcal{S}''_D$, we require that rules always reflect the identities of the contributors to the shared specification. Validity is defined statically; program p is invalid iff it contains `seal` q `except by` $c_1, c_2, \ldots, c_n$ and a rule of type q with contributor $c \notin \{c_1, c_2, \ldots, c_n\}$. For example, `no-access(amy,bob,data1). seal no-access except by amy. no-access(bob,dan,data1)` is invalid, as witnessed by the last rule breaking the seal on `no-access`. Note that the first rule does not break ("preserves") the seal, as its contributor is Amy, who is excepted from the seal on `no-access`.

For each control triple $\langle p_1, p_2, p_3 \rangle$ of $\mathcal{S}'''_D$, a seal in p_2 is broken by a rule in p_3, or vice-versa. agents can use seals to reason about valid programs in terms of their parts. For example, given $p_1 \triangleq$ (`no-access(bob,bob,X) :- private(X). seal no-access except by bob`), for any p_2, it is certain that $\checkmark(p_1 \circ p_2) \longrightarrow$ `no-access(amy, data1)` $\notin [\![p_1 \circ p_2]\!]$. In this fashion, agents reason about (models of) programs despite their incomplete knowledge of (models of) programs.

4 Realistic Cooperative Specification

This section adapts the approaches to improving composition control from Section 3 to realistic specification languages Alloy (Section 4.2) and eFLINT (Section 4.3).

For each language, we first give an account of the language, simplified for our purposes (as we detail in a dedicated paragraph), and we remark on its current use for (data exchange) system specification. Then, we demonstrate and evaluate its use for cooperative specification. Finally, we define a minor language extension, and then demonstrate and evaluate the improvements to its composition control.

4.1 Running Example Usage Scenario

Here, we present a particular cooperative specification scenario to be used throughout this section. This makes the demonstrations more predictable to the reader, and affords some comparison between the findings of Alloy and eFLINT.

The following three agents cooperate in the development of a shared specification. Each agent is concerned with their own facet of the system and their own requirements:

- **"Administrator" Amy** is concerned with controlling the distributed infrastructure, for example, by defining processing events, and specifying liveness properties.
- **"Barrister" Bob** is a legal expert, and is concerned with enforcing an interpretation of a core part of the GDPR [11]: the lawfulness of data-processing requires the consent of the subject of the processed data (§6.1).
- **"Data-processor" Dan** is a user of the infrastructure, and is concerned with personally processing data.

4.2 Improving Cooperative Alloy Specification

4.2.1 An Account of Alloy. Alloy has seen significant research and application to the modelling and model-checking of software systems [16]. Reflecting its basis on UML and OML [14], Alloy specifications represent data via named relations and constraints over relations expressed in a first-order logic based on Tarski's calculus of relations [35]. Alloy

has seen application to DSL engineering [25] and for the static analysis of complex systems [17]. The Alloy website[1] provides release versions of Alloy tools, tutorials, Alloy reference examples, and a language specification.

The meaning of each Alloy specification p is a set of *satisfying* instances. Each instance prescribes particular members to named N-ary relations. Each instance formalises a satisfactory system configuration, such that Alloy specifications predicate satisfactory configurations. The names and types of the relations are determined by each *signature* in p, which defines an *atomic* relation r, and several *fields* over r and other atomic relations. Here, Alloy affords ease of use by leveraging the expected familiarity of its users with the object-oriented paradigm: each signature corresponds to a record data type. *Facts* explicitly constrain the model in terms of *predicates*, first-order logical formulae over the relations. The *AlloyAnalyzer* tool reports the satisfying instances.

The following defines our simplified Alloy syntax:

$$\begin{aligned}
\mathit{program} &:= \mathit{para}^* & \mathit{para} &:= \mathit{macro} \mid \mathit{factDef} \mid \mathit{sigDef} \\
\mathit{id} &:= \mathit{letter}^+ & \mathit{factDef} &:= \texttt{fact}\ \mathit{id}?\texttt{\{}\ \mathit{pred}\ \texttt{\}} \\
& & \mathit{macro} &:= \texttt{let}\ \mathit{id} = (\mathit{pred} \mid \mathit{rel}) \\
\mathit{sigDef} &:= \texttt{one}?\ \texttt{sig}\ \mathit{id}\texttt{,}^+\ (\texttt{in}\ \mathit{rel})?\texttt{\{}\ \mathit{fieldDef}\texttt{,}^*\ \texttt{\}} \\
\mathit{fieldDef} &:= \mathit{id}\ \texttt{:}\ (\texttt{one} \mid \texttt{set})?\ \mathit{rel} \\
\mathit{rel} &:= \mathit{id} \mid \texttt{none} \mid \texttt{(}\ \mathit{rel}\ \texttt{)} \mid \texttt{*}\ \mathit{rel} \mid \mathit{rel}\ (\texttt{.} \mid \texttt{\&} \mid \texttt{->} \mid \texttt{+})\ \mathit{rel} \\
\mathit{pred} &:= \mathit{id} \mid \texttt{true} \mid \texttt{(}\ \mathit{pred}\ \texttt{)} \mid \texttt{not}\ \mathit{pred} \mid \mathit{pred}\ \texttt{and}\ \mathit{pred} \\
& \quad \mid \texttt{all}\ \mathit{id}\ \texttt{:}\ \mathit{rel}\texttt{\{}\ \mathit{pred}\ \texttt{\}} \mid \texttt{some}\ \mathit{rel} \mid \mathit{rel}\ (\texttt{=} \mid \texttt{in})\ \mathit{rel}
\end{aligned}$$

Given program p, the AlloyAnalyzer computes the model of a given specification: a set of instances, defined by:

$$\begin{aligned}
\mathit{instanceSet} &:= \mathit{instance}\texttt{,}^* & \mathit{instance} &:= \texttt{\{}\ \mathit{atom} \mid \mathit{field}\ \texttt{\}} \\
\mathit{field} &:= \mathit{id}\ \texttt{(}\ \mathit{atom}\texttt{,}^*\ \texttt{)} & \mathit{atom} &:= \mathit{id}\ \mathit{digit}^+
\end{aligned}$$

Demonstration. Consider the following program:

```
sig Agent, Data {} let Processor = Agent
sig State {    step:              set State ,
           process:        Data->Processor ,
           consent: Agent->Data->Processor }
fact allReachableStep { all s:State {s.*step=State}}
fact processingImpliesConsent { all s:State {
                  s.process in Agent.(s.consent) }}
```

The first paragraph (in *para*) introduces atomic relations `Agent` and `Data`. The second paragraph defines `Processor` as an alias of `Agent`; this affords more suggestive human interpretation of the field-relations over agents playing multiple roles. The third paragraph introduces the final atomic relation, `State`, along with three field relations. Note that each field-relation implicitly includes the signature-type itself as the first parameter. For example, each `process` member is an state-data-agent triple. The final paragraphs define facts that explicitly constrain the satisfying instances. For example, `allReachableStep` asserts that each state can reach any other through a transitively-closed `step`.

The AlloyAnalyzer reports that this specification has several satisfying instances. The following shows one example:

```
{ State1, step(State1, State1), Agent1, Agent2,
          Data1, consent(Agent1, Data1, Agent1) }
```

Simplifications. Our simplified Alloy language is a sublanguage of the real Alloy, so it is executable using the real AlloyAnalyzer. We make two sorts of simplifying omission. Firstly, we omit many expression combinators for predicates (like `iff`) and relations (like `:>`). Secondly, we simplify the (user control over) the practical limitations on instance enumeration by omitting paragraphs enabling fine control over the AlloyAnalyzer's search for satisfying instances; in real Alloy, it is common practice to interleave facts with `run` and `check` commands, which guide the search for instances. Moreover, the AlloyAnalyzer always only enumerates instances up to a finite *scope*, which limits the cardinality of atomic relations, such that the enumeration is tractable. Finally, we omit the language generalisations of Alloy version 6, which include temporal operators, and a generalisation of Alloy instances (in a backward-compatible manner) to linear traces. As reflected by the official Alloy tutorials, these novelties can be approximated by modelling traced states as Alloy relations; we take this approach in our Alloy examples.

4.2.2 Cooperative Alloy Specification.

We encode Alloy as straightforwardly as possible in language schema $\mathcal{S}_A$, and reason about the resulting composition control features:

- $\mathcal{P}$ is the set of syntactically-correct Alloy specifications, i.e., $\mathcal{P} \triangleq \mathit{program}$.
- $\mathcal{M}$ is the powerset of instances, i.e., $\mathcal{M} \triangleq \mathit{instanceSet}$.
- $[\![]\!]$ captures the dynamic semantics of Alloy, mapping each program to its satisfying instances.
- $\circ$ concatenates Alloy programs.
- $\checkmark$ recognises Alloy programs which 1. satisfy the static semantics of Alloy: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed, and 2. the model has some satisfying instances.

$\mathcal{S}_A$ affords cooperative specification if $[\![p]\!]$ is interpreted as the potential, concrete system behaviours. $\mathcal{S}_A$ faithfully captures the characteristic declarativeness of Alloy specifications; notably, paragraphs in p interact only indirectly, via the instances $[\![p]\!]$. Precisely, adding signatures multiplies the elements of $[\![p]\!]$, and enables the subsequent addition of new signatures and facts. Adding facts to p removes elements from $[\![p]\!]$. The specification is invalidated once the last satisfying instance is removed.

$\mathcal{S}_A$ has control triples of the form $\langle p_1, p_2, p_3 \rangle$ such that $[\![p_1 \circ p_2]\!]$ and $[\![p_1 \circ p_3]\!]$ are non-empty, but $[\![p_1 \circ p_2 \circ p_3]\!]$ is empty. Generally, facts constrain the satisfying instances independently. For example, if p_2 and p_3 contain only facts,

[1] https://alloytools.org/

then $[\![p_1 \circ p_2 \circ p_3]\!] = [\![p_1 \circ p_2]\!] \cap [\![p_1 \circ p_3]\!]$. Thus, $\mathcal{S}_A$ has some limited use in cooperative specification. Facts afford composition control, but as invalidity is witnessed by no instances, the cause of invalidity is difficult to diagnose.

Demonstration. Agents Amy, Bob, and Dan cooperate to develop a shared specification. Amy contributes first, capturing the fundamental dynamics of data-processing as a `state-step` transition system, capturing processing and consent as changing relations over agents and data:

```
sig Agent, Data {} let Processor = Agent
sig State {    step:              set State ,
           process:        Data->Processor ,
           consent: Agent->Data->Processor }
```

Next, Bob captures a legal requirement: each data has exactly one subject. Moreover, Bob formalises the role of `consent`: only a data subject can consent to its processing. To add new relations over existing `Data`, Bob uses a trick: `D` is introduced with novel fields, and then `D` is unified with `Data`:

```
sig D in Data { subject: one Agent } fact { D=Data }
fact onlySubjectsConsent {
    Agent.(State.consent) in subject }
```

Next, Amy specifies a reasonable liveness property: any conceivable consent is reachable from any state.

```
fact anyConsentAlwaysReachable { all s: State {
        s.*step.consent = Processor->Data->Agent }}
```

Finally, Dan considers a reasonable contribution: Dan and Amy are distinct Agents, and Dan processes some data:

```
one sig Amy, Dan in Agent {} fact {Amy != Dan}
fact danProcesses { some State.process.Dan }
```

To Dan's surprise, this modest contribution invalidates the specification. Unfortunately, the cause is not apparent in the output of the AlloyAnalyzer: no instances are given from which to begin a diagnosis. Moreover, the cause is not intuitive, because it emerges from subtle interactions between several facts. The only recourse is for Dan to analyse the constraints with some external tool, or to consider instances one by one to search for patterns. Here, the underlying cause is not Dan at all, but a subtle conflict between the roles Amy and Bob impose on `consent` as captured in `anyConsentAlwaysReachable` and `onlySubjectsConsent`, respectively. Together, these facts imply that either `State` is empty, or `Agent` is a singleton. Amy and Bob cooperated ineffectively, but this was not captured by the specification. Moreover, the language does not enable Bob (as the legal expert) to overrule Amy in matters concerning consent.

4.2.3 Cooperative Alloy Specification with Static Composition Control.

We define $\mathcal{S}'_A$ as a variant of $\mathcal{S}_A$ that adds the mechanism for static composition control via type-sealing that is presented in Section 3.2.4. Intuitively, Alloy is extended with a new kind of paragraph that expresses constraints on contributions instead of instances.

We define $\mathcal{P} \triangleq \mathit{programExt}$ to introduce *seals*, a new kind of (extended) paragraph, and to annotate all existing paragraphs to reflect the identities of their contributors:

$$\mathit{programExt} \coloneqq \mathit{paraExt}^* \qquad \mathit{agent} \coloneqq \mathit{id}$$
$$\mathit{paraExt} \coloneqq \mathit{agent} : \mathit{para} \mid \texttt{seal}\ \mathit{id}\ \texttt{except by}\ \mathit{agent}$$

$\mathcal{S}'_A$ adopts the definition of $\mathcal{M}$ from $\mathcal{S}_A$ entirely unchanged, and $[\![]\!]$ is adapted: seals are discarded before the model is computed from un-annotated paragraphs as before. We say *paragraph x breaks the seal on i in program p* if p contains an extended paragraph (`seal` i `except by` $a_1, a_2, \ldots, a_n$) and paragraph $(a : x)$ where $a \notin \{a_1, a_2, \ldots, a_n\}$. For example, program (`dan: fact {some Agent} seal Agent except by bob`) is invalid, because Dan's fact breaks the seal on `Agent`. $\mathcal{S}'_A$ refines the definition of validity from $\mathcal{S}_A$ such that programs with broken seals are invalid. As before, programs with empty models are invalid. Thus, the invalidity of $\mathcal{S}'_A$ programs has both a static component (in seals) and dynamic component (in models), which can be evaluated separately.

$\mathcal{S}'_A$ includes the control triples of $\mathcal{S}_A$. Moreover, $\mathcal{S}'_A$ has new control triples matching $\langle p_1, p_2, p_3 \rangle$ in which a seal in p_2 is broken in p_3 or vice-versa. Broken seals are discovered without the need to compute models, and their breakage can be explained to users in straightforward, syntactic terms.

Demonstration. We adapt the prior cooperation between Amy, Bob, and Dan. The contents of contributions are largely unchanged, but each rule is prefixed by $(a :)$ to reflect the identity of its contributor. Also, Bob's contribution includes a new extended statement: `seal consent except by bob`, to prevent other contributions from making inappropriate use of the `consent` relation (intentionally or otherwise).

Consequently, Amy finds that `anyConsentAlwaysReachable` cannot be contributed as Amy intended, as doing so would break Bob's seal on `consent`. Bob experiences this as control over Amy. Amy experiences this as insight into which contributions are (not) permitted by Bob. In general, $\mathcal{S}'_A$ programs are still subject to (unintentional) invalidity that is difficult to diagnose and correct. However, in cases contributors do anticipate undesirable contributions, these can be explicitly internalised in the specification via seals. In these cases, seals explicitly capture and communicate these constraints in a form that is easy to understand and diagnose.

4.3 Improving Cooperative eFLINT Specification

4.3.1 An Account of eFLINT.

eFLINT is a domain-specific specification language suited to formalising a variety of sources of norms [37]. It sees active development and application to the regulation of cyber-physical systems for which compliance to regulatory norms is important, e.g., medical data processing systems. As such, the design of eFLINT reflects an emphasis on modelling via abstractions that connect

normative concepts (like actions and norm-violations) with discrete computational concepts (like state transitions). The language emphasises the extensibility of specifications to reflect the dynamism of norms in practice, for example, to mirror amendments to external legal regulations [36].

An eFLINT specification is a sequence of *statements* whose meaning is 1. the *knowledge base* it denotes, a relational model of the present institutional reality, and 2. the *violations*, marking a subset of knowledge base elements as normatively undesirable. Appending new statements to a specification incrementally (re)constructs the relations to capture a mix of refinements or amendments to the model. Precisely, each knowledge base is represented as a set of data-typed tuples: the union of all relation members. eFLINT distinguishes between various sorts of data-type to maintain the correspondence to legal notions. Notably, both *fact*- and *duty*-types define N-ary relations over institutional entities, but only duties predicate the conditions under which duties are violated as a function of the knowledge base.

The following defines our simplified eFLINT syntax:

program := *stmt*'* *stmt* := *fDef* | *dDef* | *alias*

fDef := (`Fact` *type* *fSig* | `Extend Fact` *type*) *clause**

fSig := `Identified by` *var**+

dDef := (`Duty` *type* *dSig* | `Extend Duty` *type*) *dClause**

dSig := `Holder` *var* `Claimant` *var* (`Related to` *var**+)?

dClause := `Violated when` *boolExpr* | *clause*

clause := `Derived from` *instExpr*'*
| `Conditioned by` *boolExpr*

instExpr := *string* | *var* | `(` `Foreach` *var* `:` *instExpr* `)`
| *instExpr* `Where` *boolExpr*
| *type* `(` *instExpr*'* `)` | *instExpr* `.` *var*

boolExpr := `True` | `Not` *boolExpr* | `Holds` *instExpr*
| *instExpr* (`!=` | `==`) *instExpr*

alias := `Placeholder` *type* `For` *type*

type := *lowercase* (`-` | *letter*)* *var* := *type* *digit**

The meaning of a program is its *model*: relational knowledge bases $\langle k_h, k_v \rangle$, each of whose members are called *instances*. Primitive instances are strings, and complex instances are type-tagged tuples of larger instances. Each t-type instance i *holds* (written $i \in k_h$) iff it is constructed by some t-type derivation rule clause (`Derived from`) and satisfying all t-type condition clauses (`Conditioned by`). Furthermore, i is *violating* (written $i \in k_v$) iff i holds and k_h satisfies any violation condition of i. We call *fSig* and *dSig* statements *extensions* iff they contain `Extends`, and otherwise they are "new" type-definitions which *suppress* all previous extensions or definitions of the same type. Suppressed statements are ignored, effectively "overwritten". Models, knowledge bases (*kBase*) and instances (*inst*) are defined as follows:

model := `hold:` *kBase* `violate:` *kBase*

kBase := `{` *inst*'* `}` *inst* := *string* | *type* `(` *inst*'* `)`

Demonstration. Consider the following program:

```
Fact agent Identified by string.
Fact data  Identified by string
    Derived from data("CatScans"), data("CTScans").
Extend Fact data Derived from data("XRays").
Placeholder subject   For agent.
Placeholder processor For agent.
Fact process Identified by processor * data.
Fact data    Identified by subject * string
    Derived from data(subject("Amy"), "XRays").
Extend Fact data
    Conditioned by subject != subject("Amy").
```

The first two statements introduce `agent` and `data` as unary relations over the inbuilt string-type. The latter also includes a clause deriving two data-instances via simple instance expressions. The third statement adds another clause to `data`, deriving another instance. Next, the placeholder expressions introduce aliases for `agent` which help to suggest the interpretations of relations defined in the remaining statements. Next, `process` is defined as a binary relation over processors (i.e., agents) and data. Then, `data` is re-defined, suppressing the previously `data`-statements; effectively, this replaces all the prior `data`-type instances with a single new one. Note that the previous `process` definition refers to `data`, but is not suppressed. Finally, a (derivation) condition clause is added to `data`, understood to filter `data`-instances from the model. Here, this effectively removes the only data instance.

Simplifications. Our simplified eFLINT language is a sublanguage of the real eFLINT, so it is executable with the real eFLINT interpreter. We make three sorts of simplifying omission. Firstly, we omit eFLINT's *postulation* statements, which add or remove individual instances, overwriting previous, conflicting postulations. Secondly, we omit *events*, *actions*, *invariants*, and *Booleans* which complement *duties* and *facts* in affording the definition of data types which are subtly specialised in syntax to mirror more legal concepts, and semantically in their interactions with violations and postulations. Finally, we omit several instance- and predicate-expression operators, including `Exists` and `Or`. For our purposes, these omitted constructs can be sufficiently approximated by simplified eFLINT, using combinations of facts and duties.

4.4 Cooperative eFLINT Specification

We encode eFLINT as in language schema $\mathcal{S}_e$, and reason about the resulting composition control features:

- $\mathcal{P}$ is the set of syntactically-correct eFLINT programs, i.e., $\mathcal{P} \triangleq$ *program*.
- $\mathcal{M}$ is the set of models, i.e., $\mathcal{M} \triangleq$ *model*.

- $[\![]\!]$ captures the dynamic semantics of eFLINT, mapping each program to its model.
- $\circ$ concatenates programs.
- ✓ recognises programs which 1. satisfy the static semantics of eFLINT: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed, and 2. have no violations in their models.

$\mathcal{S}_e$ affords cooperative specification through the manipulation of violated duties via statements (re)defining and extending fact- and duty-types. This definition of invalidity conflates normative violations (at the domain level) with unsuccessful cooperation (at the meta-level). On the one hand, this affords agents elegantly internalising cross-cutting concerns. This manifests as a wide variety of control triples $\langle p_1, p_2, p_3 \rangle$, including cases in which p_2 contains no statements directly altering duties. On the other hand, agents cannot decouple these notions. Notably, agents cannot successfully cooperate in the specification of a system exhibiting normative violations; this is impractical for modelling real cyber-social systems, where violations cannot be reliably avoided, so they are corrected and worked around instead.

The biggest problem is that $\mathcal{S}_e$ thoroughly empowers later contributors, which earlier contributors experience as superficial composition control. Precisely, the contributor of p_1 has no control over each $[\![p_1 \circ p_2]\!]$. Any derivation clause in p_1 can be effectively removed by a derivation condition in p_2, and any statement in p_1 can be suppressed by a statement in p_2. However, these language features cannot be simply removed, as they are needed for the specification amendments that are desired. The problem is that $\mathcal{S}_e$ insufficiently captures which amendments contributors permit.

Demonstration. Agents Amy, Bob, and Dan cooperate to develop a shared specification. First, Amy makes the following contribution, formalising the processing of data as a process-data relation, where processors are agents, and data is partitioned over subjects. Each subject can have multiple data-instances by identifying them with distinct strings:

```
Fact agent Identified by string.
Placeholder subject   For agent.
Placeholder processor For agent.
Fact data    Identified by subject   * string.
Fact process Identified by processor * data.
```

Next, Bob formalises the legal notion of consent (to process data) as an agent-process relation. Bob also introduces the duty of processors to acquire consent for processing of a subject's data. Each duty instance is formalised as a processor-subject-process triple, specified to be held by the processor, and claimed by the subject. However, Bob does not (yet) specify any cases in which these duties are violated:

```
Placeholder consenter For agent.
Fact consent Identified by consenter * process.
Duty get-consent Holder processor
  Claimant subject Related to process
  Derived from (Foreach process: get-consent(
    process.processor, process.data.subject, process))
  Conditioned by Not(Holds(consent(subject, process))).
```

Amy makes another contribution, introducing the notion that some processes are started.

```
Fact started Identified by process.
```

Next, Bob contributes an extension that defines when the duty to get consent is violated: when the process is started. Bob also amends the definition of `process` to ensure that Bob's prior `get-consent` derivation clause does not "overlook" started processes. In other words, Bob specifies that members of the `started` relation are not only of the `process` data type, but are also members of the `process` relation:

```
Extend Duty get-consent Violated when started(process).
Extend Fact process
    Derived from (Foreach started: started.process).
```

Dan makes the following, final contribution. The first statement is simple: Dan starts to process Amy's X-Ray Data. The other two statements remove Dan's duty to get consent for any data in general, and then (for good measure) grant Dan consent to process any data in general.

```
Extend Fact started Derived from started(process(
    processor("Dan"), data(subject("Amy"), "XRays"))).
Extend Duty get-consent
    Conditioned by processor != agent("Dan").
Extend Fact consent Derived from (Foreach process:
    consent(process.data.subject, process)
      Where process.processor == agent("Dan")).
```

Each of the above contributions preserved the validity of the specification. However, by contributing last, Dan had full control over the model. We conclude that $\mathcal{S}_e$ failed to internalise some of Bob's important, intuitive requirements of the cooperation: 1. each agent is in control of their own consent, and 2. each existing duty to get consent is removed only by getting consent. In this case, Dan violated these requirements even without suppressing any prior statements. Bob experiences this as a lack of control over what Dan contributes. Dan experiences this as a lack of insight into what Bob permits or desires. For example, Bob and Dan each extend a type defined by another agent (Amy and Bob, respectively) with a derivation clause, failing to capture that only the former extension is permissible or desirable.

4.5 Cooperative eFLINT Specification with Hybrid Static/Dynamic Composition Control

We define $\mathcal{S}'_e$ as a variant of $\mathcal{S}_e$ which has improved composition control by adapting a mix of the language features

presented in Sections 3.2.3 and 3.2.4. Intuitively, errors dynamically recognise invalidating instances, while seals statically constrain clauses, e.g., preventing existing errors from being trivialised. Precisely, let $\mathcal{S}'_e = \langle \mathcal{P}, \mathcal{M}, [\![]\!], \circ, \checkmark \rangle$, where:

- Let $\mathcal{P}$ be *program* but with *clause* extended by:

  | `Seal Conditions` | `Erroneous When` *boolExpr*

- Let the models include new *error* instances. Precisely:

  $\mathcal{M} :=$ `hold:` *kBase* `violate:` *kBase* `error:` *kBase*

- Let each $[\![p]\!] \triangleq \langle k_h, k_v, k_e \rangle$ capture the eFLINT semantics (as before) in the holding (k_h) and violating (k_v) instances. Moreover, let k_e be the subset of k_h with an (`Erroneous when` e) clause where e is satisfied by k_h. Finally, let each instance be implicitly identified by an extra parameter with type `string` and variable `contributor`; each instance expression in a statement that is contributed by agent a implicitly constructs instances with a fixed argument identifying a. For example, the following statement is well-typed:

  ```
  Fact f Identified by string Derived from f("a")
    Erroneous when string == contributor.
  ```

- (as before) $\circ$ concatenates programs.
- Let $\checkmark(p)$ iff 1. (as before) it satisfies the static semantics of eFLINT: identifiers are bound by quantifiers or resolve to either identifier- or predicate-definitions, and expressions are well-typed; 2. no statement is suppressed; 3. there is no type t whose *seal is broken in* p: for no $i < j$ and type t, the ith t-statement includes `Seal Conditions` and the jth t-statement includes `Conditioned by`; 4. it has no errors in its model.

Syntactically, $\mathcal{S}'_e$ is an extension of $\mathcal{S}_e$, to support new composition control features. Corresponding programs also have corresponding semantic models with respect to holding and violating instances. However, there are two cases where a program p has different validity under $\mathcal{S}'_e$ and $\mathcal{S}_e$. Firstly, only $\mathcal{S}'_e$ programs are invalidated by suppressed statements. This does not affect which models are expressible, as omitting suppressed statements does not affect the model, but it does prevent later contributors from trivialising earlier contributions. Secondly, both languages can specify violations, but errors in $\mathcal{S}'_e$ model the violations of $\mathcal{S}_e$: they invalidate the specification. Thus, errors and violations are represented and computed similarly in $\mathcal{S}'_e$, but they have different meanings. Notably, this lets contributors successfully cooperate in specifying systems which have normative violations.

The language extensions to $\mathcal{S}'_e$ implement a mix of static and dynamic composition control. Dynamically, errors recognise cases of invalidity as a function of the knowledge base. Statically, sealing clauses prevent subsequent statements adding conditions to types. Thus, seals let agents stop later contributors from arbitrarily removing errors, e.g., by extending types with the clause `Conditioned by Not(True)`.

Demonstration. We reconsider the prior demonstration in which Amy, Bob, and Dan cooperate. The contributions of Amy and Bob proceed exactly as before, with one minor alteration: Bob's first contribution is extended with the following statements, which serve to more completely capture Bob's requirements of the other contributions to the specification. Firstly, `get-consent` is extended to prevent subsequent `Conditioned by` clauses. This ensures each `get-consent` duty is removed *only* by getting consent. Secondly, `consent` invalidates the specification if any consent instance is inferred by a derivation rule not contributed by the consenter:

```
Extend Fact get-consent Seal Conditions.
Extend Fact consent Seal Conditions
  Erroneous when agent(contributor) != consenter.
```

Consequently, Dan's prior contribution would now invalidate the specification. Dan experiences this as insight into the distinction between amendments to type definitions that are and are not permitted. Evidently, Amy permits anyone to add and remove `process`-instances, but `Bob` permits only particular extensions to the definitions of `consent` and `get-consent`. Moreover, agents can diagnose the cause for invalidity in (potential) contributions, because each case of invalidity can be traced to two statements: one that establishes a meta-requirement, and one that violates it. For example, Dan's prior contribution is invalidated in two ways. The static reason is that Dan conditions `get-consent` after Bob specifies not to. Finally, the specification distinguishes between domain-level violations and meta-level invalidity, such that agents can successfully cooperate to describe data processing systems in which data-processors violate the specification by processing data without consent. For example, Dan cannot grant consent on behalf of Amy. However, Dan can process Amy's X-Ray data without consent, even though the agents agree that this represents Dan violating a (domain-level) normative duty to get consent from Amy.

5 Related Work

This section overviews related work and remarks on opportunities for using related works to inform the cooperative specification of data exchange systems in the future.

5.1 Composition Control Mechanisms of General-Purpose Programming Languages

Many long-established general purpose programming languages have language features that are essential for protecting fundamental program abstractions. A conspicuous example is the *visibility*- or *access-modifiers* in languages including Java and C++. These have no impact on the dynamic semantics, but only introduce static errors if identifiers are accessed out of the specified scope. Most obviously, visibility modifiers protect abstractions, thus hardening programs against the accidental introduction of bugs.

The strength of these approaches is their simplicity; the errors they introduce are largely isolated from other semantic concepts, yet reliably connected to particular syntactic constructs that are directly exposed to the programmer. Consequently, they are easily computed and reasoned about. Visibility modifiers directly inspired the notion of sealing presented in Section 3.2.4. They have also inspired other works for similar reasons. For example, [1] adapts them for use in securing object-oriented databases.

Note that programming languages also internalise controls that are more powerful, but come at the cost of being difficult to reason about. For example, enforcing the laws of Haskell type classes requires property-based testing [18] or automatic theorem proving [2]. Nevertheless, these controls offer fruitful ideas for powerful control mechanisms.

Many existing languages and features remain to be investigated for applicability to cooperative specification. For example, we hypothesise that the notion of *mode* in the Mercury language [33] captures a simple, useful, static abstraction. In Mercury, modes prescribe the input/output modalities of arguments in logical predicates, enabling their compilation to efficient, imperative procedures [8].

5.2 Smart Contract Languages

Smart contract languages are oriented around the multi-party definition and use of *smart contracts*, which encode inter-agent powers as cryptographically secured, replicated, executable programs [13, 34]. We are particularly inspired by two characteristics of these languages.

Firstly, smart contract languages are designed around the understanding that shared contracts represent *commitments*; they are meaningful and useful because they are not trivially retracted. For example, contracts justify costly and sensitive work. A major consequence is that, in this context, it makes less sense to rely on destructive *refactorings* to amend specifications. Instead, there is greater emphasis on extensible specifications and anticipating and avoiding conflicts and contention. This view is evident in our notion of cooperative specification, which frames change as an inherently constructive process via specification composition.

Secondly, smart contract languages emphasise the capturing of inter-agent power dynamics. This is achieved by internalising agents in the specifications themselves, such that they may be systematically reasoned about in relation to other domain-level concepts. For example, the DAML smart contract language (presented in an archived article [5]) reflects *signatories* of smart contracts as the *agent* argument of the corresponding contract construct. To some extent, this pattern is even observable in languages targeting a broader notion of contract. For example, eFLINT provides *actor* as an inbuilt type which parameterises each user-defined *action*-type by default [37], affording their relation to domain-level concepts via eFLINT types and instances, as usual.

5.3 Powerful Formalisms and Formal Verification

Many communities develop (understandings of) complex software systems by encoding them in powerfully expressive formalisms, to capture complex requirements and to automate complex reasoning processes. For example, by encoding system behaviour as dependently-typed definitions, the Coq theorem prover can verify complex properties. For example, [12] encodes the Grid Component Model of modular grid computing systems [4] in Coq's vernacular language.

Compared to works in these communities, the specification languages we have considered are conservative in the power of their semantics and the extent of their automation. To some extent, this is incidental, and can be improved; future work can improve the expressivity of Alloy and eFLINT to allow capturing new requirements. However, to some extent, it is advantageous to minimise the expressive power of our specification languages up to the requirements of the application domain. This choice minimises conceptual burden on human users developing specifications, and minimises the cost of automated reasoning activities. Moreover, it leaves room for more powerful languages and systems to assist in the development and reasoning about simple specifications. For example, we see potential in using the (more powerful) Clingo language to partially automate the search for desirable contributions to a cooperative specification expressed in the (less powerful) Datalog language; Clingo is a versatile answer-set solver overviewed in [20].

5.4 Abstract Argumentation Frameworks

Abstract argumentation frameworks in the style of Dung [9] capture logical systems in which different logical conclusions can conflict, precisely, as an argument-argument *attack* relation. The ASPIC+ framework builds new relations (like argument-preorder $\leq$) and properties (like *contrary*) for characterising logical systems [40]. Its use is in enabling reasoning about key "rationality" postulates of particular systems [28], which can be understood as characterising their "usefulness". Works like [24] are related, investigating argument-preferences as a means to resolve conflicts.

These works complement our own; we share the premise of composing declarative specifications whose meaning is robust to the presence of conflicts and changes. Appropriately, our motivations also often overlap. For example, [23] investigates abstract argumentation in the context of conflicts between agent-local desires, and system-wide norms, comparable to our example in section 4.1 of Dan's desire conflicting with Bob's legal norms. We see promise in more thoroughly studying and incorporating the methods and tools of abstract interpretation in the development and application of cooperative specification languages.

6 Reflections and Future Work

In this article, we have demonstrated the extension of the Datalog, Alloy, and eFLINT specification languages to give agents control during cooperative specification without significant impact on the languages' abilities to capture domain concepts. As a result, specifications make clear which specification properties should be preserved by composition. In practical terms, the requirements added by agents reveal which details are intentionally fixed and where collaborators are invited to contribute. We did observe that the benefits of the composition control features vary across designs. For example (as demonstrated with Alloy), static composition control captured requirements in a form that afforded simple diagnosis in terms the agents can understand. Whereas (as demonstrated with eFLINT) dynamic composition control effectively captured requirements inter-relating domain- and meta-level requirements, such as the requirement for contributors not to consent on the behalf of their peers. Which method is most suited depends heavily on the application domain and the language in question.

At present, our contribution is conceptual and has not yet been implemented as an extension to an existing language. Moreover, as a language extension, the features are not immediately available to all potential users (until any extension is widely adopted). The previous sections have shown the value of the discussed languages, eFLINT in particular, for the specification of data exchange systems, yet thorough evaluation using one or more case studies is required.

In the remainder of this section, we lay out additional directions of future work.

Specialised eFLINT Variant. We observe that eFLINT is quite well-suited to the formalisation of data exchange systems through its explicit connection between legal and computational concepts and its constructive approach to deriving information, such that semantic problems are explainable in terms users can understand. This contrasts with our findings for Alloy, whose specifications are more simple and composable, but for which invalidity is generally difficult to diagnose. To bring our approach to practice we intend to formalise eFLINT with composition control features and extend the existing reference implementation accordingly.

Developing Datalog Variants. Like eFLINT, Datalog has a constructive approach to prescribing relations, which lays the groundwork for capturing meta-level requirements whose violations are easily diagnosed and understood. Datalog has been thoroughly researched by many people over many years (as summarised, for example, in [21, 22]) and there are many formalised Datalog variants to choose from.

We see value in exploring (additional) Datalog variants that are specialised for cooperative specification. We propose the development of different languages for exploring the extremes of the static-dynamic spectrum of composition control mechanisms. The static variant focuses entirely on seals, separating meta-level and domain-level concerns to emphasise low-cost reasoning and simple explainability. The dynamic variant focuses entirely on reflection and dynamic (in)validity, emphasising a simple semantics, powerful expressiveness, and the flexibility of programs to change.

Enforcement. Our notion of cooperative specification resembles dynamic enforcement, such as with access control, when cooperative specification is interleaved with the execution of the specified system. Our work was done with this possibility in mind. Also, our notion of meta-specification matches the *policies* of [10], which prescribes the means by which incrementally composed specifications control agent communications and actions. Future work can develop particular runtime systems enforcing particular cooperative specifications realising data exchange scenarios.

7 Conclusion

In this paper, we have presented the challenges that arise in agents cooperating to formalise their requirements of a complex, distributed system. We consider data exchange systems in particular, whose nature exacerbates the usual difficulties in expressing and composing requirements.

We give the problem a technical framing via *cooperative specification*, where agents incrementally compose their contributions to a shared program, careful to preserve its validity. We showed that existing languages Datalog, Alloy, and eFLINT take us part of the way to satisfactory solutions. However, we observe their limitations in capturing important "meta-level" requirements, which concern the ways agents may change the specification itself. Driven by our focus on *composition control* language features, we define minor language extensions that let agents capture more meta-level requirements, and we demonstrate how these enable more successful scenarios of cooperative specification. Future case studies are required to thoroughly evaluate our contributions in the context of data exchange systems.

We identified several opportunities for future work to continue developing specialised cooperative specification languages. We hope to leverage related works to ease the development of the languages and the specifications, to the end of improving the productivity of the specification process, and the fruitfulness of its results. This contributes to the vision of inter-organisational data exchange systems executed by autonomous, cooperating agents that enforce their shared requirements, even when requirements change.

Acknowledgments

This research is funded by projects AMdEX-fieldlab (Kansen Voor West EFRO grant KVW00309), and AMdEX-DMI (Dutch Metropolitan Innovations ecosystem for smart and sustainable cities, made possible by the Nationaal Groeifonds).

References

[1] Mansaf Alam. 2011. Access Specifiers Model for Data Security in Object Oriented Databases. In *Eighth International Conference on Information Technology: New Generations, ITNG 2011, Las Vegas, Nevada, USA, 11-13 April 2011*, Shahram Latifi (Ed.). IEEE Computer Society, 1068–1069. https://doi.org/10.1109/ITNG.2011.191

[2] Andreas Arvidsson, Moa Johansson, and Robin Touche. 2019. Proving Type Class Laws for Haskell. In *Trends in Functional Programming*, David Van Horn and John Hughes (Eds.). Springer International Publishing, Cham, 61–74.

[3] Ezio Bartocci and Yliès Falcone (Eds.). 2018. *Lectures on Runtime Verification - Introductory and Advanced Topics*. Lecture Notes in Computer Science, Vol. 10457. Springer. https://doi.org/10.1007/978-3-319-75632-5

[4] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. 2009. GCM: a grid extension to Fractal for autonomous distributed components. *Ann. des Télécommunications* 64, 1-2 (2009), 5–24. https://doi.org/10.1007/S12243-008-0068-8

[5] Alexander Bernauer, Sofia Faro, Rémy Hämmerle, Martin Huschenbett, Moritz Kiefer, Andreas Lochbihler, Jussi Mäki, Francesco Mazzoli, Simon Meier, Neil Mitchell, et al. 2023. Daml: A smart contract language for securely automating real-world multi-party business workflows. *arXiv preprint arXiv:2303.03749* (2023).

[6] Stefano Ceri, Georg Gottlob, and Letizia Tanca. 1989. What you Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. Knowl. Data Eng.* 1, 1 (1989), 146–166. https://doi.org/10.1109/69.43410

[7] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem (Eds.). 2018. *Handbook of Model Checking*. Springer. https://doi.org/10.1007/978-3-319-10575-8

[8] Tyson Dowd, Fergus Henderson, and Peter Ross. 2001. Compiling Mercury to the .NET Common Language Runtime. In *First International Workshop on Multi-Language Infrastructure and Interoperability, BABEL 2001, Satellite Event of PLI 2001, Firenze, Italy, September 8, 2001 (Electronic Notes in Theoretical Computer Science, Vol. 59)*, Nick Benton and Andrew Kennedy (Eds.). Elsevier, 73–88. https://doi.org/10.1016/S1571-0661(05)80757-1

[9] Phan Minh Dung. 1995. On the Acceptability of Arguments and its Fundamental Role in Nonmonotonic Reasoning, Logic Programming and n-Person Games. *Artif. Intell.* 77, 2 (1995), 321–358. https://doi.org/10.1016/0004-3702(94)00041-X

[10] Christopher A. Esterhuyse, Tim Müller, and L. Thomas van Binsbergen. 2024. JustAct: Actions Universally Justified by Partial Dynamic Policies. In *Formal Techniques for Distributed Objects, Components, and Systems - 44th IFIP WG 6.1 International Conference, FORTE 2024, Held as Part of the 19th International Federated Conference on Distributed Computing Techniques, DisCoTec 2024, Groningen, The Netherlands, June 17-21, 2024, Proceedings (Lecture Notes in Computer Science, Vol. 14678)*, Valentina Castiglioni and Adrian Francalanza (Eds.). Springer, 60–81. https://doi.org/10.1007/978-3-031-62645-6_4

[11] European Commission. 2016. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). https://eur-lex.europa.eu/eli/reg/2016/679/oj

[12] Nuno Gaspar, Ludovic Henrio, and Eric Madelaine. 2014. Bringing Coq into the World of GCM Distributed Applications. *Int. J. Parallel Program.* 42, 4 (2014), 643–662. https://doi.org/10.1007/S10766-013-0264-7

[13] G. Governatori, F. Idelberger, Z. Milosevic, R. Riveret, G. Sartor, and X. Xu. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law* 26, 4 (2018), 377–409. https://doi.org/10.1007/s10506-018-9223-3

[14] Yujing He. 2006. Comparison of the Modeling Languages Alloy and UML. In *Proceedings of the International Conference on Software Engineering Research and Practice & Conference on Programming Languages and Compilers, SERP 2006, Las Vegas, Nevada, USA, June 26-29, 2006, Volume 2*, Hamid R. Arabnia and Hassan Reza (Eds.). CSREA Press, 671–677.

[15] Vincent C. Hu, D. Richard Kuhn, and David F. Ferraiolo. 2015. Attribute-Based Access Control. *Computer* 48, 2 (2015), 85–88. https://doi.org/10.1109/MC.2015.33

[16] Daniel Jackson. 2003. Alloy: A Logical Modelling Language. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings (Lecture Notes in Computer Science, Vol. 2651)*, Didier Bert, Jonathan P. Bowen, Steve King, and Marina Waldén (Eds.). Springer, 1. https://doi.org/10.1007/3-540-44880-2_1

[17] Daniel Jackson. 2019. Alloy: a language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76. https://doi.org/10.1145/3338843

[18] Johan Jeuring, Patrik Jansson, and Cláudio Amaral. 2012. Testing type class laws. *SIGPLAN Not.* 47, 12 (sep 2012), 49–60. https://doi.org/10.1145/2430532.2364514

[19] Christian Jung and Jörg Dörr. 2022. *Data Usage Control*. Springer International Publishing, Cham, 129–146. https://doi.org/10.1007/978-3-030-93975-5_8

[20] Roland Kaminski, Torsten Schaub, and Philipp Wanko. 2017. A Tutorial on Hybrid Answer Set Solving with clingo. In *Reasoning Web. Semantic Interoperability on the Web - 13th International Summer School 2017, London, UK, July 7-11, 2017, Tutorial Lectures (Lecture Notes in Computer Science, Vol. 10370)*, Giovambattista Ianni, Domenico Lembo, Leopoldo E. Bertossi, Wolfgang Faber, Birte Glimm, Georg Gottlob, and Steffen Staab (Eds.). Springer, 167–203. https://doi.org/10.1007/978-3-319-61033-7_6

[21] Bas Ketsman and Paraschos Koutris. 2022. Modern Datalog Engines. *Found. Trends Databases* 12, 1 (2022), 1–68. https://doi.org/10.1561/1900000073

[22] David Maier, K. Tuncay Tekle, Michael Kifer, and David Scott Warren. 2018. Datalog: concepts, history, and outlook. In *Declarative Logic Programming: Theory, Systems, and Applications*, Michael Kifer and Yanhong Annie Liu (Eds.). ACM Books, Vol. 20. ACM / Morgan & Claypool, 3–100. https://doi.org/10.1145/3191315.3191317

[23] Sanjay Modgil and Michael Luck. 2008. Argumentation Based Resolution of Conflicts between Desires and Normative Goals. In *Argumentation in Multi-Agent Systems, Fifth International Workshop, ArgMAS 2008, Estoril, Portugal, May 12, 2008. Revised Selected and Invited Papers (Lecture Notes in Computer Science, Vol. 5384)*, Iyad Rahwan and Pavlos Moraitis (Eds.). Springer, 19–36. https://doi.org/10.1007/978-3-642-00207-6_2

[24] Sanjay Modgil and Henry Prakken. 2011. Revisiting Preferences and Argumentation. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, Toby Walsh (Ed.). IJCAI/AAAI, 1021–1026. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-175

[25] Rodrigo M. L. M. Moreira and Ana C. R. Paiva. 2015. A Novel Approach using Alloy in Domain-specific Language Engineering. In *MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, ESEO, Angers, Loire Valley, France, 9-11 February, 2015*, Slimane Hammoudi, Luís Ferreira Pires, Philippe Desfray, and Joaquim Filipe (Eds.). SciTePress, 157–164. https://doi.org/10.5220/0005228101570164

[26] B. Otto, S. Steinbuss, A. Teuscher, and S Lohmann. 2019. IDS Reference Architecture Model – version 3.0. https://doi.org/10.5281/zenodo.5105529

[27] Boris Otto, Michael ten Hompel, and Stefan Wrobel (Eds.). 2022. *Designing Data Spaces: The Ecosystem Approach to Competitive Advantage*. Springer. https://doi.org/10.1007/978-3-030-93975-5

[28] Henry Prakken and Sanjay Modgil. 2012. Clarifying some misconceptions on the ASPIC$^+$ framework. In *Computational Models of Argument - Proceedings of COMMA 2012, Vienna, Austria, September 10-12, 2012 (Frontiers in Artificial Intelligence and Applications, Vol. 245)*, Bart Verheij, Stefan Szeider, and Stefan Woltran (Eds.). IOS Press, 442–453. https://doi.org/10.3233/978-1-61499-111-3-442

[29] Alexander Pretschner, Manuel Hilty, and David Basin. 2006. Distributed usage control. *Commun. ACM* 49, 9 (sep 2006), 39–44. https://doi.org/10.1145/1151030.1151053

[30] Ravi Sandhu and Jaehong Park. 2003. Usage Control: A Vision for Next Generation Access Control. In *Computer Network Security*, Vladimir Gorodetsky, Leonard Popyack, and Victor Skormin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 17–31. https://doi.org/10.1007/978-3-540-45215-7_2

[31] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. 1996. Role-Based Access Control Models. *Computer* 29, 2 (1996), 38–47. https://doi.org/10.1109/2.485845

[32] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. 2020. Symboleo: Towards a Specification Language for Legal Contracts. In *28th IEEE International Requirements Engineering Conference, RE 2020, Zurich, Switzerland, August 31 - September 4, 2020*, Travis D. Breaux, Andrea Zisman, Samuel Fricker, and Martin Glinz (Eds.). IEEE, 364–369. https://doi.org/10.1109/RE48521.2020.00049

[33] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. 1994. The Implementation of Mercury, an Efficient Purely Declarative Logic Programming Language. In *ILPS 1994, Workshop 4: Implementation Techniques for Logic Programming Languages, Ithaca, New York, USA, November 17, 1994*, Koenraad De Bosschere, Bart Demoen, and Paul Tarau (Eds.). ftp://ftp.elis.rug.ac.be/pub/prolog/ilps94_workshop/somogyi.ps.Z

[34] N. Szabo. 1997. Formalizing and Securing Relationships on Public Networks. *First Monday* 2, 9 (1997). https://doi.org/10.5210/fm.v2i9.548

[35] Alfred Tarski. 1941. On the calculus of relations. *The journal of symbolic logic* 6, 3 (1941), 73–89.

[36] L. Thomas van Binsbergen, Milen G. Kebede, Joshua Baugh, Tom M. van Engers, and Dannis G. van Vuurden. 2021. Dynamic generation of access control policies from social policies. In *The 11th International Conference on Current and Future Trends of Information and Communication Technologies in Healthcare (ICTH-2021), Leuven, Belgium, November 1-4, 2021 (Procedia Computer Science, Vol. 198)*. Elsevier, 140–147. https://doi.org/10.1016/J.PROCS.2021.12.221

[37] L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom M. van Engers. 2020. eFLINT: a Domain-specific Language for Executable Norm Specifications. In *GPCE '20: Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Virtual Event, USA, November 16-17, 2020*, Martin Erwig and Jeff Gray (Eds.). ACM, 124–136. https://doi.org/10.1145/3425898.3426958

[38] L. Thomas van Binsbergen, Merrick Oost-Rosengren, Hayo Schreijer, Freek Dijkstra, and Taco van Dijk. 2024. *AMdEX Reference Architecture – version* . https://doi.org/10.5281/zenodo.10565915

[39] Wil M. P. van der Aalst and Josep Carmona (Eds.). 2022. *Process Mining Handbook*. Lecture Notes in Business Information Processing, Vol. 448. Springer. https://doi.org/10.1007/978-3-031-08848-3

[40] Leendert W. N. van der Torre and Serena Villata. 2014. An ASPIC-based legal argumentation framework for deontic reasoning. In *Computational Models of Argument - Proceedings of COMMA 2014, Atholl Palace Hotel, Scottish Highlands, UK, September 9-12, 2014 (Frontiers in Artificial Intelligence and Applications, Vol. 266)*, Simon Parsons, Nir Oren, Chris Reed, and Federico Cerutti (Eds.). IOS Press, 421–432. https://doi.org/10.3233/978-1-61499-436-7-421

[41] W3C OWL Working Group. 2012. OWL 2 Web Ontology Language Document Overview (Second Edition).

Received 2024-07-01; accepted 2024-08-30