

# Reflections on the design, applications and implementations of the normative specification language eFLINT

L. Thomas van Binsbergen<sup>a</sup>, Christopher A. Esterhuyse<sup>a</sup>, Tim Müller<sup>a</sup>

<sup>a</sup>*Informatics Institute, University of Amsterdam, Science Park 900, 1098 XH, Amsterdam, The Netherlands*

## Abstract

Checking the compliance of software against laws, regulations and contracts is increasingly important and costly as the embedding of software into societal practices is getting more pervasive. Moreover, the digitalised services provided by governmental organisations and companies are governed by an increasing amount of laws and regulations, requiring highly adaptable compliance practices. A potential solution is to automate compliance using software. However, automating compliance is difficult for various reasons. Legal practices involve subjective processes such as interpretation and qualification. New laws and regulations come into effect regularly and laws and regulations, as well as their interpretations, are subjected to constant revision. In addition, computational reasoning with laws requires a cross-disciplinary process involving both legal and software expertise.

This paper reflects on the domain-specific software language eFLINT developed to experiment with novel solutions. The language combines declarative and procedural elements to reason about situations and scenarios respectively, explicates and formalises connections between legal concepts and computational concepts, and is designed to automate compliance checks both before, during and after a software system runs. The various goals and applications areas for the language give rise to (conflicting) requirements. This paper reflects on the current design of the language by recalling various applications, the requirements they imposed, and subsequent design decisions. As such, this paper reports on results and insights of an investigation that can benefit language developers within the field of automated compliance.

## 1. Introduction

Laws, regulations, contracts, and other *social policies* serve to regulate social systems by establishing *norms* that determine how actors within the system are expected to behave. A wide range of automated techniques is available to aid in the enforcement of such social policies within software systems. In distributed software systems, *system policies* are widespread as means to regulate the behaviour of system components, separating the description of the policy from the implementation of the components to enhance adaptability and transparency. For example, the XACML [83] and ODRL [67] languages are designed to regulate access to resources in a unified manner through access control policies that can easily be reconfigured when relevant social policies change. As such, access control policies help to partially automate compliance of software with social policies. Smart contracts [106, 54, 92] are another example of automating compliance with, in particular, legal contracts. However, both examples are limited from a general legal standpoint in that they are too static (smart contracts) [78] or not sufficiently expressive (access control) [113, 31, 72].

A general solution to automating compliance that is tenable from the legal perspective needs to address several requirements. Firstly, a formal representation of policies is required that is sufficiently general to capture policies from a variety of sources and at different levels of abstraction. For example, the GDPR privacy regulation is more abstract than an organisational policy or data sharing agreement. Secondly, the representation should enable different kinds of reasoning such as conformance checking to ensure the implemented (business) process can only behave in compliance with policies (see [59] for definitions of compliance and conformance), property checking (e.g., model checking [33]) to gain confidence in the correctness of a policy specification, simulation to determine the effects of actions and the compliance of individual (hypothetical) scenarios, and search and optimisation for policy-aware planning. Thirdly, the solution should embrace the fact that legal processes such as interpretation and qualification are inherently subjective, and that compliance depends on (in)actions by users. These observations make a fully *ex-ante* approach untenable from a legal perspective (although practical in many situations). This is due to the fact that: (a) the same legal norm may be interpreted differently by different stakeholders, (b) conflicting legal norms may simultaneously apply to a case, (c) norms may be interpreted differently and situations may be qualified different depending on the specifics of a case, and

\*Corresponding author

Email addresses: ltvansbergen@acm.org (L. Thomas van Binsbergen), c.a.estherhuyse@uva.nl (Christopher A. Esterhuyse), t.muller@uva.nl (Tim Müller)

(d) legal obligations may depend on actions by users that cannot be automatically enforced (an actor may decide not to comply with an obligation). Solutions are needed in which violations within the system can occur and be observed, with *ex-post* enforcement mechanisms, such as penalties and compensations, as a backstop to respond to non-compliance. An overview of the challenges related to compliance is given by Hashmi et al. [60]

This paper reports on intermediate findings of an investigation into a general solution to automating compliance. Central to the investigation is the design and implementation of eFLINT, a domain-specific language (DSL) for developing executable specifications of norms [16]. The language supports various types of reasoning and covers different types of ex-ante and ex-post enforcement. The design is based on important software (language) engineering concepts such as modularity and inheritance and combines concepts from the declarative, logical, and imperative programming paradigms. The languages has been applied in various experiments involving the assessment of concrete (historical) cases [16], bounded model checking [44], runtime compliance with ex-post enforcement [77], normative multi-agent systems [87], and generating access control policies from social policies [15], among others. Over time, the language has evolved to become more flexible and suitable for use within these various application areas. In this paper, we reflect on the design of the language using (sometimes conflicting) requirements extracted from the goals and applications of the language. We reflect on the design choices relevant to other researchers of norm specification languages, normative reasoning, and automating compliance, and suggests future directions.

The paper is organised as follows. Section 2 discusses work related to various aspects of norm representation and enforcement. Section 3 introduces the eFLINT language through a running example. Section 4 recounts the motivations for the eFLINT language and describes various applications of the language and the requirements they imposed on the language’s design. Section 5 describes in which ways the language evolved and analyses specific design decisions that have been made to address the requirements, revealing various trade-offs. Section 6 motivates and compares the two most prevalent implementations, including a performance comparison. Section 7 reflects on the main limitations of the current design and suggests future directions. Section 8 concludes.

## 2. Related work

In this section we describe work related to automating compliance and compare eFLINT to existing languages.

*Norm representation.* Several software languages and logics exist to formalise and reason with norms from various sources and within different application areas. Compared to other languages, eFLINT is most similar to languages

based on the Event Calculus [76, 96, 95, 30] such as Symboleo [104] and InstAL [85]. The Institutional Action Language (InstAL) is a DSL for specifying norms in terms of duties and powers translating to Answer Set Programming (ASP) for execution. In [44], eFLINT has been given a semantics based on ASP and in Sections 5.5 and 6 we reflect on the use of ASP. Symboleo and eFLINT are both based on Hohfeld’s legal framework [63] that emphasises normative relations between actors centred around the normative concepts of power and duty. Where eFLINT is designed to specify norms from a wide range of legal documents, Symboleo is designed specifically for contracts, embedding notions of contract state in the language. LegalRuleML extends RuleML and specifies norms in terms of permissions and obligations through rules, and like eFLINT supports defeasibility and negation of facts [6].

Other formal languages for expressing norms are directly based on (modal) logic such as deontic logic [62], action logic [70] and defeasible logic [82, 55]. An important aspect of eFLINT, compared to the Deontic alternatives, is that the language is action-based and supports the legal concept of power – the ability to change the normative positions of (other) actors. The benefit of the action-based approach is that checking the compliance of software systems is simplified because software systems are inherently action-based. These features enable eFLINT for various types of applications supporting dynamic (online) or static (offline) compliance-checking.

*Access and Usage Control.* A significant body of work exists concerning the formalisation, analysis and enforcement of *specific* kinds of norms [68] such as policies for access control [83, 67], network policies [2] (e.g. firewall configurations) and (smart) contracts [104, 102, 54, 100, 115, 3, 109]. Instead, eFLINT is designed for specifying a wide variety of norms such as from laws, regulations, (organisational) policies and contracts. The Margrave Policy Analyzer tool<sup>1</sup> supports multiple formalisms to reason about access control policies and firewall configurations with different analyses such as Change-Impact Analysis [47]. Many access control models exist [98, 84]. The most common are Attribute-Based Access Control (ABAC) [66], Role-Based Access Control (RBAC) [97], and Purpose-Based Access Control (PBAC) [25]. XACML is a popular ABAC language and the XACML architecture [83] is widespread as a model for systems integrating ABAC or other forms of access control. ODRL is an access control language with policies controlling specific actions on (multi-media) assets [67, 93, 113]. The Usage CONtrol (UCON) model introduces the ability to specify conditions that should hold *during* access events [88].

The knowledge representation of eFLINT is sufficiently expressive to capture roles, attributes and purposes and the eFLINT reasoner can be used as a policy decision

---

<sup>1</sup><http://www.margrave-tool.org/>

point within the XACML architecture [17, 15]. And unlike ODRL<sup>2</sup>, the eFLINT language definition is shipped with semantics [44], albeit not yet standardised. Future work is to investigate whether the discrete nature of eFLINT events is a limiting factor to apply usage control.

*Smart Contracts.* Another specific type of system policy is a *smart contract*. First introduced by Szabo [106] for the exchange of digital assets, smart contracts are now most popular for their usage as scripts executing ‘transactions’ recorded on a distributed ledger (the blockchain). Owing to their execution model, smart contracts can guarantee the exchange of digital assets proceeds according to predefined rules encoded in the smart contracts (including restitutions if actors do not uphold their end of the agreement). With the distributed ledger, blockchain applications can support compliance efforts by providing auditability and transparency [90, 103, 109]. Solidity [46] is a popular imperative smart contract language for the Ethereum platform [24, 114]. Marlowe is a declarative DSL for writing smart contracts at a higher level of abstraction for the Cardano platform [102]. Contracts specified in the Symboleo language can be executed as smart contracts on the Ethereum platform through a translation [92].

Early experiments suggest a translation from eFLINT to Solidity is feasible and useful, although we have not yet demonstrated this conclusively. In [43], we developed an abstract framework for enforcing and communicating policies between actors. The framework guarantees that actors agree which actions are justified by policies, establishing a sound basis for accounting and auditing. Implementations can instantiate the framework using distributed ledgers, but other implementations are feasible.

*Multi-agent systems.* We have used multi-agent systems (MASs) to experiment with the integration of eFLINT in (running) systems organised according to different architectures. We have used the Belief, Desire, and Intention (BDI) agents of [80, 79] to simulate applications, adding one or more eFLINT reasoners as so-called normative advisors. Other approaches extend BDI agents by adding normative concepts as reasoning capabilities to agents [37, 108, 35], such as the B-DOING framework [38] and the BOID architecture [22, 86]. Most similar to our approach is the use of ‘ethical governors’ in [27] that can be queried for advise. However, with our approach we can also support active notifications sent by the eFLINT reasoner to support forms of ex-post enforcement. A thorough discussion on integrating norms in MAS is provided by [10].

*Model checking.* In the context of (bounded) model checking, properties expressed in temporal logics are used to express norms or to reason with norms. For example, Normative Temporal Logic (NTL) is a temporal logic that

replaces the path quantifiers of Computation-Tree Logic (CTL) to express obligations and permissions [116]. Temporal Defeasible Logic (TDL) combines defeasibility and temporal logic [56]. The FIEVeL specification language is used to model institutional policies [112] based on *Ordered Many-Sorted First-Order Temporal Logic (OMSFOTL)* using SPIN [65] for model checking. In [7], the connection between coordination problems and norm enforcement is formalised using Linear Temporal Logic (LTL). The norm specification language Revani [71] uses CTL for the specification of properties concerning privacy in particular. Symboleo contracts can also be verified with LTL-properties [89] using the nuXmv model checker [28].

We have conducted experiments with a variant of LTL specific to eFLINT to specify eFLINT properties [36]. Separately we have used ASP to search for counter-example scenarios that disprove specified properties [44], in both cases realising bounded model checking [12].

*Other Types of Enforcement.* Methods to enforce norms within software systems can be categorised as static or dynamic and as ex-ante or ex-post. Static, ex-ante methods attempt to ensure software is compliant prior to its execution as part of its design. Examples are privacy-by-design [75], policy simulations [48], model-driven engineering [99], and model checking [58, 57]. These approaches are less suitable in situations where it is difficult to determine all legal requirements a priori or when legal requirements are likely to change. Dynamic, ex-ante methods are used to prevent non-compliance at runtime and are generally more easily adapted to changing legal requirements. For example, access and usage control policies (discussed above) can be modified ‘on the fly’ without having to redesign the system.

Both types of ex-ante methods cannot fully guarantee compliance when multiple, potentially conflicting, sources of norms are applied or when compliance is the result of the inaction of users. In these cases, ex-post enforcement mechanisms are needed to check for compliance after events have occurred, potentially responding to any observed violations. Dynamic, ex-post methods monitor events within the system without necessary attempting to prevent violations. Violations are observed and potentially prevented from recurring through adaptations, resolving conflicts, or applying penalties and compensations to users. Runtime verification is well established as a technique for runtime monitoring and violation detection [11, 29] and MAPE-K is a well known method for creating systems that can adapt to feedback [73, 23]. Static, ex-post methods involve the processing of event logs [91, 107, 110] and checking the conformance of implemented process models with respect to models of compliant behaviour [1, 59].

We conclude that a norm specification language aiming to automate compliance in general cases should applicable in all four types of enforcement. The applications of eFLINT discussed in Section 4 show the types of enforce-

---

<sup>2</sup>A working group is specifying semantics for ODRL here: <https://w3c.github.io/odrl/formal-semantics/>.

ment supported by the language, covering various ex-ante and ex-post methods.

### 3. Running Example

This section introduces the eFLINT language through an example program related to auctioning. The program is introduced incrementally with code fragments building on top of each other in the order they are given. The reference interpreter for the language is available online [14].

An eFLINT program consists of type-declarations forming a specification, statements describing a scenario, and queries. The type-declarations introduce sets and relations, each with a domain whose instances receive a truth-assignment in a knowledge base, indicating whether the instance is an element of the corresponding set or relation.

The following *type declarations* introduce the initially empty sets `bidder`, `object`, `price`, and `display` and the relations `bid` and `min-price-of`. The set `display` is declared with `Var`, indicating at most one instance can hold true for this type, i.e. `display` acts as a variable to which a (single) object is assigned (if any). Similarly, the relation `min-price-of` is ‘functional’, i.e. it maps objects to a unique price as a mathematical (partial) function.

```
Fact      object    Identified by String
Fact      price     Identified by Int
Var       display   Identified by object
Function min-price-of Identified by object * price
Fact      bid       Identified by bidder * object * price * int
```

The following *statements* define the function by asserting certain instances.

```
+min-price-of(Watch, 100).
+min-price-of(Clock, 200).
+min-price-of(Painting, 400).
```

Derivation rules can be added to type declarations to infer truth assignments from knowledge about (other) facts:

```
Extend Fact object Derived from min-price-of.object
Extend Fact price  Derived from min-price-of.price
                  ,bid.price
```

The `Extend` keyword adds clauses to an existing declaration. The last line above states the price of both every minimal price and of every bid is an element of `price`. The results of multiple derivation rules accumulate through set union. A derivation rule can also be written as a Boolean expression using `Holds when`. The Boolean expression is evaluated in a context in which the fields of a type are bound as variables, i.e.. `bidder` and `price` below.

```
Var highest-bid Identified by bidder * price Holds when
  (Exists bid: bid.bidder == bidder && bid.price == price
   && bid.object == display.object
   && (Forall bid': bid'.price <= price
        When bid'.object == display.object))
```

This clause assumes that the fields (and also `bid`) can be enumerated to determine the truth of the expression for each combination of instances of these types.

Act-types are fact-types with instances – referred to as actions – that can be *performed*. Special clauses determine the effects of performing actions. The act-type below has

two fields, the implicit field `actor` and an `object`. An instance of the type holds true when its actor is recognised as an `auctioneer`. A performed action that does not hold true raises a violation, but still has its effects. The effect is the assertion of the instance `display(object)`, as well as the implicit termination<sup>3</sup> of any other elements of `display` owing to its status as a `Var`.

```
Var auctioneer // responsible for displaying objects
Act start-bidding Related to object
      Holds when auctioneer(actor)
      Creates display(object)
```

State transitions occur through the execution of actions and assertions<sup>4</sup>. On the contrary, derivation rules are declarative in that they do not trigger transitions. Instead, they can be understood to ‘close’ a knowledge base.

The actor of an action can also be named explicitly:

```
Act place-bid Actor bidder Related to object, price
      Holds when bidder
      Conditioned by display(object), price >
      Max(Foreach bid: bid.price When bid.object == object)
      Creates bid(int =
      Count(Foreach bid: bid When bid.object == object)))
```

This act-type declaration uses a form of constructor application with implicit arguments to create an instance of `bid`. The `int` field of `bid` distinguishes bids from the same bidder. The implicit arguments to `bid` – namely `bidder`, `object`, `price` – are copied directly from the arguments to `place-bid`. A detailed explanation of constructor application with implicit propagation is provided in section 5. The `Conditioned by` clause establishes (extra) conditions<sup>5</sup> for instances to be enabled. In this example, bids are only allowed on displayed objects and must involve a price higher than any previous bid.

A *physical* action, as opposed to the previous *institutional* actions, is always enabled and only raises violations when it synchronises with an institutional action raising a violation. The act-type declared below intuitively captures the *qualification* of the action of raising one’s hand at an auction as placing a bid on the item currently on display with a price higher than the previous bid.

```
Physical raise-hand Syncs with place-bid
      bidder = actor, object = object, price =
      min-price-of.price + 10*
      Count(Foreach bid: bid When bid.object == object))
      When bidder(actor) && display(object) &&
      (min-price-of.object == object)
```

The consequence of synchronising an action A with an action B is that A inherits all the pre- and post-conditions of B, effectively triggering B whenever A is triggered. The ‘synchronises with’ relation is directional: in the above, `raise-hand` is not triggered if `place-bid` were to be triggered. The relation is transitive and potentially cyclic<sup>6</sup>.

<sup>3</sup>The dual of creation in our terminology.

<sup>4</sup>And also events, differing from actions in that they are not performed by actors.

<sup>5</sup>`Holds when` and `Derived from` clauses are disjunctive. `Conditioned by` clauses are conjunctive.

<sup>6</sup>Cycles are easy to detect and resolve as semantically the transitively closed set of synchronised actions is simultaneously triggered.

A duty-type declaration defines a fact-type with mandatory fields for a duty-holder and a duty-claimant to establish a duty-claim relation in the Hohfeldian legal framework, optional additional fields (such as `price` below) and zero or more violation conditions. A duty raises a violation when it holds true and when one or more of its violation conditions hold.

```
Bool undue-payment-delay
Duty payment-duty Holder bidder Claimant auctioneer
  Related to price
  Violated when undue-payment-delay()
```

The encoding of different kinds of legal obligations as duties and the concept of open-texture terms such as ‘undue delay’ are discussed in Section 5. In the example, the duty to pay is created when the bidding on a particular object is ended by the auctioneer.

```
Act end-bidding Holds when auctioneer(actor)
Creates payment-duty(bidder=highest-bid.bidder
  ,price=highest-bid.price)
Terminates display.object
  ,bid When bid.object == display.object
```

The `Terminates` clauses refers to the variables `display` and `bid`. As they are not fields of the act, the variables are implicitly bound by an occurrence of `For each`. As a result, all instances of `display` are terminated (at most one) and all instances of `bid` are terminated for which hold that the object referred to in the bid is on display. This condition on the termination of `bid` is realised by the application of `When` (infix). Both effects of `Terminates` are considered to occur simultaneously, i.e. the `Terminates` expressions are evaluated in the same context, in which the object is still on display. The same is true for `Creates` and `Terminates` clauses and, when in conflict, the creation of a fact trumps the termination of the same fact.

A scenario is a sequence of statements, including assertions of creation (+) and termination (-) and action triggers (without prefix). At the end of the following scenario, Bob wins the auction and has a duty to pay 140 for the watch because Bob is the last to raise his hand and because all but the first bids raise the minimum price of a 100 with increments of 10 each. The existence of the duty is confirmed by the `query` (prefixed by ?) at the end of the fragment holding true.

```
+bidder(Alice). +bidder(Bob). +bidder(Chloe).
+auctioneer(David).
start-bidding(David, Watch). // action trigger
raise-hand(Alice). raise-hand(Bob). raise-hand(Alice).
raise-hand(Chloe). raise-hand(Bob).
end-bidding(David).
?payment-duty(Bob, David, 140). // evaluates to True
```

## 4. Application

This section describes different applications of eFLINT and introduces requirements (in **bold**) identified in these application areas. An overview of the requirements is given in Table 1. Note that not all requirements are (fully) met by the eFLINT design. Section 5 reflects on the design choices made to (attempt to) satisfy the requirements.

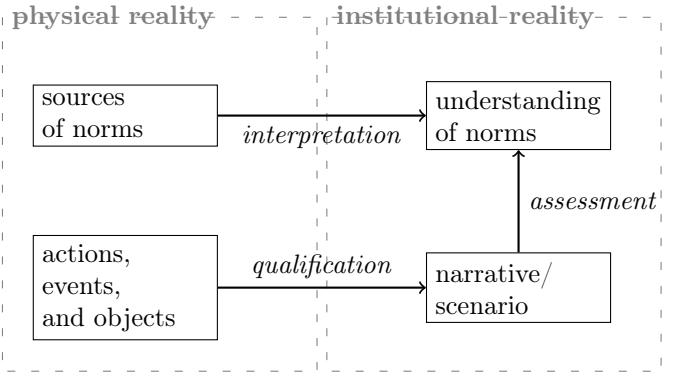


Figure 1: Schematic overview of the processes of interpretation, qualification and assessment as taken from [16] and based on Searle [101].

### 4.1. Supporting Formal Interpretation

The design of eFLINT was initially motivated to assess concrete scenarios for compliance with norms specified in the FLINT language of Van Doesburg [40, 39] (eFLINT abbreviates “executable FLINT”) and to achieve this by establishing a connection between Hohfeld’s normative concepts [64, 63] and computational concepts [16]. The design and reference implementation of eFLINT were created as a vehicle for experimenting with a language that can give computational meaning to normative concepts and that can be applied in a diverse set of application areas. These goals and application areas have resulted in various extensions and modifications to the language.

The Calculemus-FLINT framework [40] provides a method for interpreting sources of norms to produce a structured, formal **interpretation** and assessing a particular case for compliance against the formalised interpretation (**assessment**, see Figure 1). The method suggests to recognise acts, duties, actors and facts based on sentence structure and to fill act-, duty-, and fact-*frames* (type-declarations in eFLINT) with text fragments extracted directly from the sentences (an example is given in Figure 2 of Section 5). Case assessment has been a key motivation behind the development of eFLINT. Both eFLINT and FLINT associate pre- and post-conditions with acts and violation-conditions with duties. Together these conditions determine the outcome of a case, i.e. the performed actions/events, their effects, and any violations. Determining the compliance of a scenario amounts to **forward chaining** across derivation rules and transitions.

The FLINT language is intended to be used by legal experts (**domain-users**). A legal expert interpreting a source of norms can apply their trained, yet subjective expertise to resolve possible ambiguities. From a legal perspective, it is important that subjectivity is part of the interpretation process, as normative texts are meant to be applied in varying contexts, circumstances and cases. In particular, **open-texture terms** such as ‘undue delay’ are deliberately under-specified [54]. The precise interpretation of an open-texture term is to be determined within

the specific context in which the norms are applied. For this reason, the norm specification language should be able to delay the interpretation of certain parts until the application context is known (**specialisation**).

Analysing a case based on a formal interpretation increases **transparency** and can aid the process of resolving disputes caused by differences in interpretation. To improve transparency further, FLINT **source references** between the frames of an interpretation and the original source locations using the JuriConnect standard [21]. The ability to easily switch between alternative interpretations is needed when assessing a case or resolving disputes. However, alternative interpretations may only differ in small, subtle details about, for example, an individual article. For this reason, interpretations should be developed modularly as a collection of (possibly small) fragments, with fragments being easy to replace (**modularity, versioning**). We conclude that specifications should reflect the level of abstraction provided by the original source (**abstraction level**) and specifications should be **extensible** so that further details can be specified later.

#### 4.2. Executable Normative Concepts

Another goal in the development of eFLINT has been to find a (computational) core language that can be used to give operational semantics to various **normative concepts**. As an internal language, eFLINT could support various user-facing languages, including FLINT. The design of FLINT is action-oriented in its focus on acts and ‘obligations to act’ (duties). To support a wider range of user-facing languages, additional normative concepts need to be mapped to computational concepts. Deontic frameworks also consider ‘obligations of fact’, i.e. the obligation to achieve or preserve a certain state in the world. Moreover, a prohibition to act is not necessarily equivalent to the negation of a permission to act. This is the case when permissions and prohibitions are assigned by different actors (with different levels of authority), originate from different sources of norms (with different precedence), or differ in specificity (with the most specific norm typically having precedence). In these situations, priority mechanisms are required (**norm priorities**) to determine whether a violation has indeed occurred. Neither FLINT nor eFLINT has such mechanisms at present.

If conflicts are not fully resolved ex-ante, **ex-post enforcement** mechanisms are needed to respond retroactively to violations. For example, an arbiter may decide, based on priorities, whether a particular violation is indeed actionable in a given situation. Ex-post enforcement mechanisms are also needed in cases where the compliance of a system depends on the (in)action of its users. For example, a user may decide not to respond to a duty.

Disputes also arise due to the subjectivity of qualification, the process by which observations about the world get normative meaning. For this reason, it is important that compliance is based on **explicit qualification** (see Figure 1). As with interpretation, qualification rules can

be made available for transparency and can be reused and versioned.

The mechanisms described above can be used to assess *historical* cases individually, or collectively to determine the compliance of a system as a whole (e.g., as part of an audit). Assessing *hypothetical* cases is useful to experiment with the consequences of a set of norms as part of, for example, the drafting of policy (**simulation, experimentation**). By specifying large sets of cases, confidence about the inner workings of the policy is obtained, akin to running test-suites for testing software (**testing**).

#### 4.3. Automating Governance

The eFLINT language has been used in experiments to investigate automating governance in case management systems used by governmental organisations, e.g., for deciding on permits or tax reduction requests. In these systems, relevant laws are concretised by the policies of the governmental organisation (**specialisation**). We developed prototypes do demonstrate the ability to adapt to changes in norms (**dynamic updates** to law or organisational policy) and to enforce the rights, powers, and duties of both civil servant and citizen [105]. In these case management systems, the eFLINT reasoner is **instantiated** as a running process once for every on-going case or once in total, responding to individual requests generated for what should be clearly distinguishable cases. In both models, information about the case is recorded as cases develop over time. This information is recorded statefully within the reasoner or within an external knowledge base (**persistence**). The latter requires mechanisms to provide facts about a case to the reasoner (**case input**), to record facts established by the reasoner (**case output**) and to apply **data migrations** when changes in norms affect the fields in type-declarations. In most practical applications, eFLINT is used statelessly, relying on external knowledge bases and migration mechanisms.

These experiments have also revealed the important of specifying a **service interface** between the application and the reasoner, for example to determine which actions, events and facts of an eFLINT specification are triggerable by the application and which are internal. For example, one can say that all (and only) the physical actions (such as `raise-hand` and not `place-bid` in the running example) are triggerable by users of the application. Another aspect of the service interface is to determine which facts are provided solely by the application. In this case it is important to be able to distinguish the absence of a fact from falsity. We can therefore not (always) make the closed world assumption (**open world**). Furthermore, the instances of a type can not always be enumerated beforehand as, for example, new citizens register and reasoning is needed with numerical values and dates (**infinite domains**).

#### 4.4. Distributed Systems

In other projects, we are experimenting with the automatic enforcement of (privacy) regulations, consortium

agreements and data sharing conditions within distributed data exchange systems [17, 4, 5]. In such distributed systems, compliance can be checked and enforced dynamically by reasoning services listening and responding to messages (**event-based**), including the planning of tasks by an orchestrator [45]. As inter-domain applications, data exchange systems require mechanisms for cross-domain authentication [53] and consensus. Consensus is needed on the applied norms (e.g. which interpretation of the GDPR is applied), relevant policy information (e.g., whether a particular user is affiliated with a member of a consortium), on the transpired events (e.g., whether data was accessed), and on the decisions made by the normative reasoner (e.g., whether authorisation was given). Methods to communicate about policies, achieve consensus on (the validity of) policy, and for accountable decision making are reported in [42, 43]. In this context, **transparency**, **explainability**, and **auditability** are of high importance [90, 103]. To support transparency and auditability, we have experimented with an eFLINT to Solidity [46, 114] compiler producing smart contracts running in a blockchain application. The interpretation is recorded as a smart contract and the assessed scenario is recorded on the ledger.

We developed multi-agent systems to experiment with different architectures and setups for automating compliance in distributed systems. In these experiments, so-called ‘normative advisors’ are agents that reason with norms based on observations communicated to them by other agents or by the environment [87]. Our approach makes it possible to experiment with different models for distributing institutional facts and institutional reasoning and requires multiple eFLINT reasoners to co-exist (**instantiation**). One can define a system with a single agent monitoring and assessing the behaviour of all agents in the system. Alternatively, every agent can have its own normative advisor and use normative reasoning for planning. Such planning activities by agents can benefit from **simulation** of hypothetical scenarios and **backward chaining** to reason backwards from a goal to the current state.

These experiments also included agents for dynamic, **ex-post enforcement** that decide whether and how to respond to observed violations. These agents may be fully autonomous services, or may require a ‘human in the loop’. In both cases, the enforcement agent should receive a detailed report about the nature of the violation and how the violation was established containing, e.g., which facts prevented an action from being enabled (**explainability**). Similar reports are useful with **ex-ante enforcement**, enabling agents to determine what they can do to enable actions they desire to perform.

#### 4.5. Bounded Model-Checking

The applications described in the previous subsection are about *concrete* (actual, historical, or hypothetical) scenarios and sometimes involve the enumeration of a large set of alternative scenarios, e.g., when testing or running

simulations. In other work we are applying bounded model-checking to attempt verify safety and liveness properties of eFLINT specifications. In these experiments we are reasoning about *abstract* scenarios, using, for example, linear temporal logic formulae to describe scenarios by mixing concrete details and abstract placeholders. To apply bounded model-checking, eFLINT specifications must have **finite domains** such that in every runtime state, only a finite amount of transitions are possible. Finite domains are also required by the aforementioned application of (automated) simulation to perform planning in agents. Bounded model-checking also requires the ‘**closed world assumption**’ that establishes that what is not known to be true is considered false. Alternatively, a three-valued logic (true, false, and unknown) may be considered, at the risk of significantly complicating the reasoning semantics.

Ideally, model-checking can be used in combination with the aforementioned applications, i.e., to verify certain properties before deploying a specification in a case management or data exchange system, without having to maintain multiple versions of the specification. To support this, we concluded it must be possible to reduce the amount of possible transitions and the set of possible runtime states (**reducible domains**) by *extending* a specification. This way, the same specification can be used both by an extension for model-checking and by an extension for deployment.

Other approaches to verification are relevant as well. For example, we wish to investigate the applicability of combinatorial, property-based testing [52].

## 5. Design

This section summarises the most important design choices, modifications and extensions made in order to (better) meet the requirements identified in the previous section and laid out and categorised in Table 1.

### 5.1. Knowledge Representation

**Types.** Following well-known declarative languages such as Prolog [34] and Alloy [69], eFLINT allows users to define sets and relations inductively over atomic strings and integers. Relations associate names with fields, comparable to attributes in relational algebra and databases. Relations in eFLINT are thus algebraic products, akin to database tables, predicates in Prolog, or triples in RDF [74]. The projection operator `<EXPR>. <NAME>` is available to refer to a specific field of an instance, i.e., comparable to projection/selection in relational algebra. Derivation rules can define unions and joins by applying `Exists`, `Forall` and `Foreach` operators to enumerate over elements of relations. In many programming languages, users rely on algebraic sums – the dual of products – to represent the concept of heterogeneous collections, e.g., a set of mixed strings and integers. But instead, like other relational languages (e.g., Alloy), heterogeneity is expressed in eFLINT by using the

Table 1: The various requirements identified in section 4 laid out and categorised.

| <i>legal</i>                  | <i>reasoning</i>  | <i>services</i>   | <i>usability</i>  |
|-------------------------------|-------------------|-------------------|-------------------|
| <b>ex-ante enforcement</b>    | forward chaining  | service interface | domain-users      |
| <b>ex-post enforcement</b>    | backward chaining | case input        | source references |
| <b>interpretation</b>         | norm priorities   | case output       | explainability    |
| <b>explicit qualification</b> | closed world      | persistence       | versioning        |
| <b>normative concepts</b>     | open world        | dynamic updates   | testing           |
| <b>assessment</b>             | infinite domains  | event-based       | modularity        |
| <b>abstraction level</b>      | finite domains    | instantiation     | extensible        |
| <b>auditability</b>           | reducible domains | specialisation    | experimentation   |
| <b>open-texture terms</b>     | simulation        | data migrations   | transparency      |

same facts in relations of different types. For example, `citizen-of(person("Amy"))`, `country("Germany")`) and `member-of(person("Amy"), club("Chess", elo(500)))` treat people as heterogenous collections: mixes of club memberships and national citizenships.

Act-types and duty-types behave as (relational) fact-types in how instances are represented, constructed, created and terminated. This design choice has made it simple to represent the legal concept of power: to change the normative positions of (other) actors, an actor can be assigned actions that create or terminate actions or duties (**normative concepts**).

*Enumerating Domains.* As described in Section 4, eFLINT is intended for reasoning about both historical and dynamically evolving scenarios. The latter relies on types with **infinite domains** whereas in the former the ‘domain of discourse’ is known a priori and is inherently **finite**. The `Foreach`, `Forall` and `Exists` operators enumerate instances of one or more types as part of their application. The `Foreach` operator is used in conjunction with aggregators such as `Count` and `Sum` and in the post-conditions of actions (e.g., `Creates`). Operators `Forall` and `Exists` effectively generalise binary conjunction and disjunction (respectively) over lists of Booleans. A pragmatic design decision has been made to give semantics to these operators depending on the domains of the types they enumerate: the instances of a finite type are all enumerated, independent of whether they hold true or not, whereas of an infinite type, only instances that hold in the current knowledge base are enumerated. By design, enumeration will therefore always terminate. In the former case, termination is guaranteed because the domain is finite and only finitely many instances exist. In the latter case, termination is guaranteed because knowledge bases are concrete and finite by design (discussed below). The difference is subtle, impactful, and may result in surprising, difficult to diagnose specification errors where expected facts are simply absent without a trace. The advantage of the approach, however, is that specifications can be reused across applications with and without a finite domain. As an example, the value of `count-all` in the following fragment differs depending on whether the first or second definition of `numbers` is used. With the first definition of `number`, all

numbers 1 to 5 will be counted. With the second definition, only numbers 1, 3, and 5 will be counted.

```

Fact number Identified by 1..5. //finite domain
Fact number Identified by Int. //infinite domain
+number(1). +number(3). +number(5).
Var count-all Identified by Int Derived from
  Count (Foreach number: number).
Var count Identified by Int Derived from
  Count (Foreach number: number When Holds(number)).

```

The definition `count` has the `When` clause to ensure only numbers that hold are counted irrespective of whether `number` was defined with an infinite domain.

Knowledge bases in eFLINT are finite because instances of types are concrete rather than symbolic, i.e., instances are fully instantiated<sup>7</sup> and do not contain placeholders. The statement `+rich(person)` in the following code fragment determines that every *currently* known person is rich rather than every conceivable, earlier known, or later known person. As a consequence, Chloe is not considered rich at any moment in the execution of:

```

Fact person.
+person(Alice). +person(Bob).
+rich(person). +person(Chloe).

```

The statement `+rich(person)` is to be interpreted as the imperative “for each known person, assert they are rich”, rather than the declarative “every person is rich”. Note that in applications in which the set of known persons (or more generally: the domain of discourse) does not change, both the declarative and imperative interpretation are valid. The decision to maintain concrete facts has been made to ensure knowledge bases are easy to understand for legal experts (**explainability**, **domain-users**) but has not been thoroughly evaluated with legal experts.

The semantics of enumeration also affects derivation rules. Consider the action `start-bidding` of the running example. In the following code fragment, the three derivation rules associated with the action are equivalent. The first can be seen as syntactic sugar for the second and the second is fully explicated by the third.

```

Act start-bidding Related to object
  Holds when auctioneer(actor)
  Derived from
    start-bidding(actor,object) When auctioneer(actor)
  Derived from (Foreach actor, object:
    start-bidding(actor,object) When auctioneer(actor))

```

<sup>7</sup>Reasoning does not require unification as in Prolog.

The query `?Holds(start-bidding(David, Vase))` fails in the running example as `object(Vase)` is not an instance enumerated by the `Foreach`. That the definition of `object` affects the first derivation rule is counter-intuitive, given that the rule does not mention `object`. In an alternative semantics we considered for `Holds when` clauses, the field names of the type could be bound to the fields of a given instance of the type (e.g., `actor` bound to `David` and `object` bound to `Vase`) when determining whether that instance holds true according to the Boolean expression of the clause. In this case, `object` is not enumerated as the variable is already bound and the expression holds true as `David` is an auctioneer. The alternative semantics conflicts with the previous design choice that knowledge bases are concrete and finite as `start-bidding(David, object)` would hold true for all conceivable objects. The alternative semantics has been considered and implemented, but has been discarded for this reason.

*Restricting Domains.* To dynamically restrict the domain of a type, eFLINT allows writing `When <bool-expr>` after an `Identified by` or `Related to` clause. The semantics of this construct is to restrict the domain of a type to only those instances that are syntactically valid and satisfy the given Boolean expression as a constraint. The following (simplified) example (taken from [16]) determines that all instances of `collect-personal-data` must satisfy the condition `subject-of(subject, data)`.

```
Act collect-personal-data
Actor controller Recipient subject
Related to data, processor, purpose
When subject-of(subject, data)
...
```

Combinations of `controller`, `subject`, `data`, `processor`, and `purpose`, for which the constraint does not hold are thus not members of the type `collect-personal-data` and are therefore not enumerated (e.g., by `Foreach` or `Forall`). This feature was considered crucial in the first versions of the interpreter which behaved like a simulator for exploring scenarios rather than a Read-Eval-Print-Loop (REPL) [20]. The interpreter presented users with a finite list of triggerable actions and events for exploration by enumerating the declared act- and event-types (**simulation, experimentation**). For each triggerable action or event, the interpreter showed whether the action or event is compliant or violating when triggered in the current state, following the semantics of the language that distinguishes between *physical ability* – those actions that can be performed – and compliant behaviour – those actions that may be, or ought to be, performed. The feature to restrict domains serves to constrain physical ability separate from compliant behaviour, with the practical consequence that the list of triggerable actions/events presented to the user can be reduced and made more manageable. This feature lost its importance in the shift towards applications of eFLINT with automated reasoning and dynamically evolving scenarios (although the interpreter still presents the list of triggerable actions and events upon

request). Further note that the feature essentially gave eFLINT a data-dependent type system, complicating its semantics. For these reasons, the feature is somewhat controversial, not often used any more, and also not part of the latest implementation (see Section 6.2).

*Implicit Arguments.* Constructor application is written as the name of a type followed by zero or more arguments that may also be left implicit. In a constructor application with implicit arguments, any missing arguments are implicitly the name of the missing field. For example, the constructor application `bid(int = ...)` of the running example (Section 3), in which arguments for the fields `bidder`, `object` and `price` are missing, is interpreted as `bid(int = ..., bidder = bidder, object = object, price = price)`. The context determines whether the introduced variable references are already bound or are implicitly bound by an `Exists` or `Foreach`.

Implicit arguments simplify the translation from FLINT to eFLINT, which is useful during **interpretation**. Consider the FLINT act-frame on the top of Figure 2 and the corresponding eFLINT code on the bottom. The FLINT frame can be generated from a syntactic analysis of the original natural language text, in this case the Code of Canon Law on Catholic marriage. The eFLINT code can be automatically generated from the FLINT frame with facts capturing which strings represent an ordinary or priest, a pair of spouses, and with Booleans<sup>8</sup> capturing whether there is a valid marriage or marriage attempt. The resulting eFLINT code is internally consistent and executable. However, there are some issues with the code. For example, multiple (pairs of) spouses can be represented as instances of the type `[spouses]` (**instantiation**) but only one valid marriage can be represented, and without a formal connection to the spouses, as `[valid marriage]` is of Boolean type. This problem can be fixed in at least two ways by redefining some of the types (**specialisation**). Firstly, `[spouses]` can be defined using the `Var` keyword, encoding an assumption that reasoning involves at most one pair of spouses. Secondly, `[valid marriage]` (and `[marriage attempt]`) can be defined as a set with a field `[spouses]` to determine which pair of spouses are married:

```
Fact [valid marriage] Identified by [spouses]
```

The original act-type definition is still valid with either solution, owing to constructor application with implicit arguments. In the second solution, the `Creates` expression is equivalent to `[valid marriage]([spouses])`. When an action is performed and the expression is evaluated, `[spouses]` is bound to the recipient of the action. The support for implicit arguments and quantifiers in eFLINT thus ensures the direct translation from FLINT to eFLINT yields something executable although extensions may be

---

<sup>8</sup>Keyword `Bool` defines a type with only one value: the empty tuple. The type behaves like a Boolean variable as the value is either present or absent in the defined set.

|                                 |  |
|---------------------------------|--|
| <i>Act frame</i>                | «assisting with the contracting of a valid marriage» |
| <i>Actor</i>                    | [ordinary or priest ...]                             |
| <i>Object</i>                   | [marriage attempt]                                   |
| <i>Interested Party</i>         | [spouses]  |
| ...                             | ...  |
| <i>Creating postcondition</i>   | [valid marriage]; ...                                |
| <i>Terminates postcondition</i> | [marriage attempt]                                   |

```

Fact [ordinary or priest ...]
Fact [spouses]
Bool [marriage attempt]
Bool [valid marriage]
Act <<assisting with the contracting
of a valid marriage>>
  Actor [ordinary or priest ...]
  Recipient [spouses]
  Related to [marriage attempt]
  Creates [valid marriage]()
  Terminates [marriage attempt]()

```

Figure 2: A FLINT and eFLINT specification of the same act. Simplified example taken from [39].

required to obtain the desired semantics specific to the use case.

The decision which of the two solutions to apply is arguably the decision for a software expert to make (rather than a legal expert) as the decision is influenced by the design of the software system in which the eFLINT code is applied. If the code is applied to reason about individual, historical cases, then the first solution is effective. If the code is integrated in a system in which multiple cases are managed, then the second solution might be preferred. An advantage of the design of eFLINT is that both solutions can co-exist: the original definition in Figure 2 can be extended (using the `#require` or `#include` directive) by two separate files, each implementing one of the solutions.

A third solution is to rely on the method by which the eFLINT specification is **instantiated** as a service. To reason with multiple cases, each case can have a dedicated service, separately instantiating one of the two extensions (both could work) with the correct spouses.

## 5.2. Exploratory Programming

One of the most impactful decisions has been to adopt the incremental programming style of languages such as Python in which a program is a sequence of individually valid program fragments. In the original eFLINT paper [16], a strict separation is maintained between type declarations, domain refinements (**reducible domains**), initial state declaration and scenarios. The strict separation was removed by introducing *phrases*, each of which is either a sequence of type declarations and type extensions, a query, a statement, or a sequence of phrases. This is a conservative extension in the sense that the old separation can still be made, if desired. The incremental style of programming makes specifications inherently **extensible**, enables **dynamic updates** and the delayed specification of **open-texture terms**, greatly increases **modularity**, and simplifies debugging and **testing**. The change

was motivated by the desire to assess dynamically evolving scenarios and interpretations and is a natural fit with **event-based** applications. Events raised within the system (e.g. the “raise hand” button has been pressed) can be converted to eFLINT phrases as **case input** and the conclusions can cause new events being raised (e.g. that there is a new highest bidder) as **case output**, enabling feedback and **ex-post enforcement**.

The language was redesigned in [15] according to the methodology presented in [20] and implemented using the generic interpreter back-end of [49] supporting exploratory programming by keeping track of execution history. Exploratory programming enables revisiting previous states for **experimentation** and **simulation**. The execution history contains valuable information to retrace causes and effects, enhancing **explainability**, **transparency** and **auditability**. As part of this redesign, several other changes were made to the language. The **Extend** keyword was added for writing type extensions and the **Syncs with** keyword was added for synchronising actions and events. The **Syncs with** keyword was introduced to connect the institutional actions from various interrelated normative documents, such as a data sharing agreement between hospitals and the GDPR privacy regulation (example taken from [15]). As a result, normative sources can be interpreted separately at their own **abstraction level** with synchronisation clauses and derivation rules as additional, separate code fragments making any connections explicit.

To support type extensions more naturally, a distinction has been introduced between *domain-related* clauses of a type, such as **Identified by** and **Related to**, and all other clauses. As part of the distinction, all other kinds of clauses are allowed to occur multiple times at a type and their effects somehow accumulate depending on the kind of clause. These clauses are therefore referred to as *accumulating clauses*. For example, an instance of a type is derived if one or more of the **Derived from** or **Holds when** clauses of the type derives the instance. A duty is violated if (it holds true and) one or more of its **Violated when** clauses holds true. All pre-conditions (**Conditioned by**) of a type<sup>9</sup> must hold true for an instance of the type to be enabled. And all the instances computed for the expressions of all **Creates**, **Terminates**, and **Obfuscates** (see Section 5.4) are created/terminated/obfuscated.

As a special case, a fact-type with a domain-related clause **Identified by Int** or **Identified by String** can have additional **Identified by** clauses that mention concrete instances of (the same) domain, such as **Identified by "Hello", "World"** and **1..5**. In these cases, the infinite domain **String** or **Int** is replaced by the accumulation (union) of all the concrete values mentioned by the other **Identified by** clauses. This feature makes it easier to extend an existing specification in order to close the domain of discourse, e.g., for model checking or simulations.

<sup>9</sup>**Conditioned by** clauses can be associated with all types.

In the latest implementation, the `Domain` keyword is used to replace this specific usage of `Identified by`.

*Queries.* Where a Boolean query `?<EXPR>` evaluates an expression to true or false, an instance query `?-<EXPR>` evaluates an expression to zero, one or more instances of a type. For example, the instance query `?-bid When display(bid.object)` evaluates to all the bids on the object currently on display. All unbound variables in an instance query (`bid` in the example) are implicitly bound using `ForEach`. The eFLINT interpreter can run in ‘test mode’ during which the interpreter only outputs information about Boolean queries that fail, effectively treating them as assertions (**testing**). An eFLINT service can respond to Boolean queries to give or deny access to resources, akin to a policy decision point in the XACML architecture [83]. Alternatively, an instance query can be used to obtain permissions of a certain kind for translation into, for example, user rights in a database server. Both are examples of **ex-ante enforcement**.

### 5.3. Service Interactions

In dynamically unfolding cases, e.g. when monitoring the compliance of a running system, some process is required to automatically convert observations (events) into the steps of a scenario (**qualification**). In initial experiments with multi-agent systems, we relied on the host language to write such conversion rules, or we hard-coded qualifications into applications. A strict separation was maintained between physical processes (external to eFLINT) and institutional processes (internal to eFLINT). However, experience demonstrated that it is also useful to use `Syncs with`, together with derivation rules, for expressing qualification rules. Doing so, eFLINT actions and events model a physical process as well as an institutional process and the two are connected via `Syncs with` clauses for action- and event-types and derivation rules for fact-types. Such qualification rules can then be adapted within eFLINT without modifying the encoding of the physical process. This feature lets the specifier change their mind about the horizon between internal and external events with minimal disturbance to existing internals. Moreover, this approach to qualification makes it easier to integrate eFLINT as a service within (existing) software. A generic algorithm can systematically convert the events of an arbitrary event stream to valid eFLINT phrases without making assumptions about the specification with which the reasoner is loaded. Institutional meaning is then assigned to these events within eFLINT.

Physical and synchronised actions motivated a design change such that actions always manifest their effects when executed, even when disabled, relying on **ex-post enforcement** mechanisms to respond to any raised violations. This change makes it possible to apply eFLINT both in synchronous and asynchronous settings – e.g., as part of runtime verification – relying on queries for the

former case. (A keyword to `Force` effects was deprecated by this change.)

For security reasons and to separate concerns, it is beneficial to restrict an eFLINT service to only respond to certain inputs and to make other types strictly internal. This way, for example, we can disallow any duties from being created or terminated directly, but instead only through institutional actions (powers). For the running example we might like to say that only the `raise-hand`, `start-bidding` and `end-bidding` actions can be triggered. This requires extending the language with a module-system or `service interface` specification mechanism to make it possible to hide and expose elements of a specification. Neither has been realised at the time of writing.

At present, eFLINT offers the `Physical` keyword as an alternative to `Act` to distinguish between physical and institutional actions. Whether an instance of a physical act holds true is determined only by pre-conditions expressed using `Conditioned by` or inherited by synchronising with instances of (other) act-types. The conditions of a physical action can thus be seen as determining the ‘physical ability’ of performing an action, whereas the conditions (and derivation rules) of institutional actions determine permissions (rights) and powers. The distinction between physical and institutional action strengthens model checking. By expressing LTL properties that only transition over physical actions (synchronised with institutional actions), we can use model checking properties to search for traces of physical behaviour that trigger violations in the institutional model before the physical process runs, enabling a strong form of **ex-ante enforcement** (defined as ‘correctness checking’ in [59]).

### 5.4. Open Types

In a type declaration, the `Open` and `Closed` modifiers are available to indicate whether the **closed world** assumption holds for the declared type. By default, a type is closed. The logic for open types is three-valued: True, False or Unknown. An instance is True or False if and only if it has been explicitly assigned a truth value using `Creates`, `Terminates`, `+<EXPR>`, `-<EXPR>`, or when it is derived<sup>10</sup> as True (see also Section 5.5 on negation as failure). All other instances are assigned Unknown. A closed type can be seen as a special case of an open type for which Unknown is also interpreted as False, i.e. an instance of a closed type not explicitly assigned True or False is implicitly assigned False. The `Obfuscates` clause has been added to the language to set the truth value of an instance to Unknown and can be used in all positions where `Creates` and `Terminates` can be used.

The open types of a specification can be seen as parameters of the specification capturing knowledge not modelled by the specification and only available within the

---

<sup>10</sup>Note that eFLINT does not have derivation rules with negative conclusions but does support negative antecedents in rules.

execution context in which the specification is applied as a service. The declarations of open types thus communicate information about the interface between the service and the execution context, i.e. that the execution context is expected to provide information in order to successfully apply the service. As a rule of thumb, one can declare those (fact-)types as open whose instances have assignments that are not determined or modified by the specification itself. During development of a specification, these instances are recognised as those that need to be asserted True or False in a scenario prior to statements of the scenario (e.g. for testing purposes). In the running example, these are `bidder`, `auctioneer`, and `min-price-of`.

```
Open Function min-price-of Identified by object * price
Open Fact    bidder      Identified by String
Open Var     auctioneer Identified by String
```

Note that the information captured by open types may be subject to change during execution. For example, it may be reasonable to assume additional bidders are added, whereas perhaps the minimum price of an object is fixed once set. A mechanism has been added to the language to provide ‘additional input’ as part of a request (such as a query) to instantiate certain types for the duration of this request only. Additional input is only used for that particular execution request and has the highest priority<sup>11</sup> when establishing the truth of a fact. This mechanism can help to ensure the correctness of the information provided as part the request by ignoring previously provided instances of types. (If the input is to be provided once and for all, the specification or scenario can be extended to provide the information and the additional input mechanism need not be used.)

The treatment of Unknown affects how applications interact with eFLINT services. The main design decisions are what happens when evaluation requires the determination whether an Unknown instance holds true and when evaluation requires the enumeration of the instances of an open type with an infinite domain. The execution of a phrase is interrupted when Unknown is encountered anywhere True or False is expected and reports the Unknown instance as an exception. The execution of a phrase is also interrupted when the instances of an open type need to be enumerated (e.g. to evaluate a `Foreach` or `Forall` expression) and the domain of the type is infinite. If the domain of the type was finite instead, the instances of the domain are enumerated (although possibly resulting in Unknown instance exceptions.) If the type were closed instead, all the True instances for the type available in the knowledge are enumerated, assuming all other instances are False. However, for open types with an infinite domain, the truth of all instances not recorded in the knowledge base is Unknown and it is therefore unknown which of these require

enumeration. In this case, the open type itself is reported as part of an exception alongside the interruption.

Both types of exceptions can be understood as a request to the execution context to provide additional input. The first kind of exception can be resolved by the execution context providing a truth assignment to the reported instance. The second kind of exception can be resolved by the execution context providing the instances within the domain of the reported type that are to be enumerated.

In the API interface of our interpreter implementation, the additional input is provided as a list of truth assignments to instances. When the instances of an open type with infinite domain are to be enumerated, the instances of the type to which True has been assigned in the additional input are enumerated. By simultaneously specifying enumeration and truth assignment, this implementation choice helps prevent Unknown instance exceptions that might occur during the enumeration. The additional input can thus be expressed using the existing syntax for assertion of the form `-<EXPR>` and `+<EXPR>` as exemplified below.

```
+min-price-of(Watch, 100).
+min-price-of(Clock, 200).
+min-price-of(Painting, 400).
+user(Alice). +user(Bob). +user(Chloe). +user(David).
+auctioneer(David).
```

Note that this syntax is not sufficient to distinguish between submitting the empty set of assignments for a type as additional input or submitting no set at all. In our implementation, the interpretation that no set is submitted for that type is chosen. This decision follows the assumption that users have no need to define closed types without instances.

The additional input mechanism can be used to provide information alongside a request to execute a phrase in order to prevent the exceptions from occurring. The additional input may also be provided in direct response to a raised exception in a conversational style, resolving the exception after it occurs. The temporal gap between the two executions – the one without and the one with the required additional input – raises the question whether the state prior to the first execution should be preserved. To this end, it may be beneficial to offer a mechanism to pause and continue phrase execution after missing input has been made available. In one of our multi-agent experiments, an agent responds to a missing information exception by contacting another agent to obtain the information, e.g. to confirm whether a certain certificate is valid. In another application, open types are used to generate input forms for users to fill. This application asks the eFLINT reasoner for all open types in the specification. The response contains information about the types, including their domain and whether any truth values have been assigned to instances. The response is used to generate an HTML form with various kinds of elements (e.g. checkboxes and dropdowns) with any available information pre-filled. The information to generate the form is obtained via a static analysis that (over-)approximates the input required for

---

<sup>11</sup>Additional input facts have a higher priority than facts created and terminated by actions/events which in turn have a higher priority than derived facts.

a particular (sequence of) phrase(s). The form can also be produced more optimistically, yielding only a subset of the form elements that *may* be enough to evaluate the given phrases. Heuristics can be used to select this subset with trade-offs such as between the amount of information required and the difficulty to obtain that information.

The additional input mechanism can also be used to provide *all* input relevant to requests. This way, the interpreter can be used statelessly and **persistence** can be realised with an external database. This persistence mechanism can also be used to recover the state from before a request was made that raised an exception.

During our experiments, two motivations for closing a type previously declared as open have been encountered. Firstly, and as argued before, certain static analyses may require a finite domain of discourse when applied and may require the closed world assumption (for all types). Secondly, a specification with open types may be extended to specialise towards a particular application. An open-type such as **bidder** may be refined such that, in this example, knowledge about valid bidders is now included in the specification. For example, within a particular application it may be valid to say that “all users except the auctioneer are valid bidders”, expressed as follows (the declaration of **bidder** overrides the one from Section 3):

```
Open Fact user Identified by String
Closed Fact bidder Identified by String
    Derived from user When Not(auctioneer(user))
```

One of the newly introduced types is then typically an open type, such as **user** in the example above. With this extension, **bidder** can be considered ‘internal’ and **user** ‘external’.

### 5.5. Reasoning by Default

Many prior examples have demonstrated the use of eFLINT’s rich expression language to denote (complex) relationships and predicates. These expressions play many roles in the language. For example, adding **Extend Fact object Derived from min-price-of.object** closes the knowledge base under a derivation rule, such that (explicitly) adding **min-price-of** relationships implicitly also adds their objects. Crucially, these expressions can quantify and check the instances already in the current knowledge base. Reconsider the definition of **place-bid**, whose **Holds when** and **Conditioned by** clauses together specify which instances of **place-bid** hold, as a complex function of existing bidders and objects on display.

```
Act place-bid Actor bidder Related to object, price
Holds when bidder
Conditioned by display(object), price >
Max(Foreach bid: bid.price When bid.object == object)
Creates bid(int =
Count(Foreach bid: bid When bid.object == object))
```

The intended semantics of these quantifications and checks is *reasoning by default*: instances hold if they were created or derived by some clause, and otherwise, by default, they do not hold. This imposes a conspicuous asymmetry between truth (holding) and falsity (not holding): only

truths are witnessed, by creations or derivations. And how can we be certain that at any given moment, all truths have been witnessed to definitively evaluate occurrences of **Not**, **Foreach ... When ... Count ... When ... etc.**?

The literature on logic and logic programming extensively explores reasoning by default (also called *negation as failure*) [32]. Its definition of falsity as the complement of truth is simple and powerful. eFLINT users enjoy that **Holds(bid)** has a simple Boolean evaluation in any context. But, as with many logic programming languages before, reasoning about eFLINT is complicated by the existence of syntactically valid specifications which have no single logical interpretation. To illustrate the issue: which bidders should be ready, given the following specification?

```
Fact ready Identified by bidder
Holds when Not(ready(bidder)).
```

In general, the issue arises from any case where truth is cyclically dependent on its own falsity. These dependencies can be more subtle. For example, in the following, it may not be obvious that the following is contradictory, defining minimum prices as being lower than themselves. This issue can be corrected by replacing  $\leq$  with  $<$ , altering its meaning to specify that no (different) assignments of value to the same object may be larger.

```
Function min-price-of Identified by object * price
Derived from bid Where
Not(Exists min-price-of':
min-price-of'.object == bid.object
&& min-price-of'.price <= bid.price).
```

Specifications without these troublesome cyclic dependencies are called *stratified* because, intuitively, facts can be categorised into ordered strata, such that the facts depend only on facts in preceding strata, necessarily having no cycles. Several approximations of this ideal notion of ‘stratification’ have been explored [94], because it is generally costly to accurately recognise and exploit.

The original work on eFLINT did not consider the subtleties of reasoning by default. For many years, the interpreter implemented a ‘greedy’ algorithm, which escaped any cycles by overriding falsity with truth as necessary. This solution sufficed to enable years of experimentation and application. Many problems were naturally expressed by stratified specifications, without even trying. Consequently, the interpreter usually behaved exactly as expected. Otherwise, exceptions were usually discovered quickly, and could be worked around by just enough refactoring until the surprises (and underlying cycles) vanished. But the reasoning algorithm was found to exhibit another, even more subtle undesirable characteristic: users could not predict how it would break the symmetry between mutually exclusive truths [44]. For example, nothing in the following specification helps the reader to predict exactly which bidder becomes the only auctioneer by default.

```
Extend Fact auctioneer Derived from bidder
When Not(Exists auctioneer': auctioneer' != bidder).
```

As with cyclic dependencies, minor refactoring usually suffices to break all symmetries. However, in complex speci-

fications, it can be very difficult to recognise the problem and to diagnose the symmetry or cycle that is to blame.

In recent years, as eFLINT started being applied to new problems, including larger-scale and real-world problems, there was a need to fix its semantics and implementation of reasoning by default. In [44], we adapted the *answer set* semantics (also called the *stable-model* semantics) [51] for eFLINT. Intuitively, it enumerates all possible outcomes from all acyclic interleavings of reasoning steps. Desirably, stratified specifications have the same unique interpretations as before. But unstratified specifications are characterised by having no interpretation (or ‘stable model’ or ‘answer’), or many distinct interpretations. Either case recognises the presence of problems in the specification, which users can then address. Section 6 expands on this new implementation, showing also how its new answer-set semantics enabled a new path to model-checking eFLINT via existing answer-set solving tools.

Separate research efforts are ongoing, exploring the *well-founded semantics* [111] as an alternative to the answer-set semantics (e.g., see Seaso in [41]). These semantics are closely related, but they differ in a crucial detail: like the original, greedy semantics, the well-founded semantics ensures that every specification has a unique logical interpretation, and like the answer-set semantics, facts’ values preserve all specified constraints on truth and falsity via eFLINT’s enumerated and checked instances. The trick is that the well-founded semantics ‘isolates’ troublesome facts, by assigning them a third ‘unspecified’ value, marking the presence of a cycle or the breaking of a symmetry if another value was assigned instead. In practice, this protects truth and falsity from logical symmetries and cycles in unrelated facts. This makes unstratified specifications more useful, and their underlying problems easier for users to diagnose. Desirably, the well-founded semantics and answer-set semantics align in an intuitive way: truth and falsities in the former have the same value in *all* interpretations of the latter [111]. That is, the well-founded semantics can be understood to ‘summarise’ the answer-set semantics. Work remains to adapt this semantics to eFLINT and to determine which semantics for reasoning by default is most suitable.

### 5.6. Sequential and Parallel Composition of Phrases

In the beginning of this section it was mentioned that a single phrase can be comprised of a sequence of type declarations/extensions or a sequence of phrases. Defining or extending multiple types as part of a single phrase is useful as these declarations are processed simultaneously, thus accepting mutually recursive definitions. (Otherwise there is no way to order your mutually recursive definitions such that every type is declared before it is used.) The effects of the declarations thus occur in parallel with no observable intermediate steps to the user. (Except when ‘reasoning by default’ as discussed in Section 5.5.)

A sequence of phrases, on the other hand, is processed phrase-by-phrase, with the possible effects of the phrases

manifesting in that order. This is relevant particularly for queries and statements. After all, the result of a query and the effects an action/event are context-sensitive, i.e., are dependent on the knowledge base used to evaluate the query/action/event.

There was no mechanism to trigger multiple actions/events simultaneously prior to the introduction of the `Syncs with` clause. Experience with the clause resulted in the desire to also add a mechanism to execute arbitrary phrases in parallel. After this conservative extension, the syntax of an eFLINT program is a sequence of top-level fragments, each followed by a full stop, where a top-level fragment is either a phrase as defined before or a set of phrases (written between braces and separated by full stops). Semantically, top-level fragments are executed in sequence, with each encountered set of phrases having its phrases executed in parallel as one step in the top-level sequence. The execution of a set of phrases first distils and processes (in parallel) all the declarations in the set (such that new types are in scope) and then executes all the statements and queries in the set (in parallel). For consistency, all the queries and statements in a set of phrases are evaluated in the context of the knowledge base from before the evaluation of the queries and statements but from after the processing of the declarations in the set.

## 6. Implementation

This section discusses several aspects related to the implementation efforts of the eFLINT language, focussing primarily on the reference interpreter (in Haskell) and the pipeline converting eFLINT to Clingo (implemented in Rust). The motivations behind the two implementations are discussed and performance comparisons are made.

### 6.1. Haskell Reference Interpreter

As recalled in Section 4, the first efforts to design and implement eFLINT were primarily aimed at making FLINT executable. The Haskell implementation of eFLINT [13] thus served as a tool to experiment with the semantics of the language and to rapidly prototype with alternative definitions. The decision was made to implement eFLINT as a definitional interpreter, thus simultaneously defining and implementing the language. The main requirements for the implementation were the simplicity to make adjustments and the readability of the code. To meet these requirements, the evaluation functions central to the interpreter are implemented via a principled usage of well-established Haskell monads: the Reader monad for dealing with bindings and scope, the State monad for state transitions, and the Writer monad for output and exceptions. The resulting definitions are therefore somewhat comparable to a big-step style structural operational semantics and, for the well-acquainted, concisely convey a lot of information about the implemented semantics. The grammar combinators of [18, 19] are used to define the concrete syntax of the language. The result is a syntax definition that

resembles abstract syntax quite closely whilst defining a (GLL) parser directly. These and other implementation choices promoted the requirements, generally doing so at the cost of performance.

For these initial purposes, the interpreter was sufficiently self-documenting. However, as the user-base started to grow, other demands were being placed on the (tool) support for the language. Firstly, additional forms of documentation were added. Jupyter notebooks were developed to give a gentle and high-level introduction of the main language features, curated examples were taken from experiments and use cases, and an ever-growing set of unit tests help testing and documenting expected behaviour of specific language features and corner cases.

Secondly, additional front-ends and means to interact with the interpreter were needed, resulting in REPL and API front-ends replacing the initial simulator front-end. The REPL front-end replaces the simulator in the most direct sense and on top of the usual REPL interactions offers meta-commands for exploring (alternative) scenarios in a step-by-step fashion. The API front-end has been used to develop several wrappers for accessing the interpreter as a solver or reasoner, e.g., in Python, Java, and Scala for actor-oriented programming within the AKKA framework. Especially for users of the API front-end (and the users of the software they build), the poor performance of the interpreter renders it impractical beyond basic examples in cases where performance matters. For other users, the interpreter serves as a reference implementation to verify their own experimental implementation of the language. For this reason, we decided to maintain the original requirements for the Haskell interpreter, keeping it as a reference interpreter, and instead seeking for alternative, more performant implementations (with the most successful being reported in the subsequent subsections).

Thirdly, with more users, eFLINT was exposed to additional scrutiny and over time it was repeatedly shown that the documentation was not always complete. Specifically, the Jupyter notebooks and curated examples are too high-level for those seeking a detailed understanding of the language. The test suite provides further details but was repeatedly found to be incomplete and extended as new bugs or corner-cases were discovered. Most problematically, in some cases the interpreter was discovered to exhibit behaviour that was not expected or desired upon closer inspection. These cases occurred when implementation choices in the interpreter were not made consciously but rather followed naturally from the way the code was organised or other features were implemented. From these observations we concluded that a complete formal definition of the semantics of the language is needed, triggering the work on a translation from eFLINT to the Clingo language reported in [44] and the following subsections.

Specifically, this translation and resulting implementation addresses the following points at once:

- (a) The eFLINT semantics is formally specified to fully

document the language’s features in all their details. This point is addressed by specifying a translation from eFLINT to (a subset of) Clingo, inheriting from Clingo’s semantics in several regards.

- (b) The translation addresses the Haskell interpreter’s problems with reasoning by default, opting for the stable model semantics of Clingo.
- (c) The runtime performance of eFLINT code is significantly improved, especially in regards to the evaluation of derivation rules, benefitting from the years of development and optimisation efforts that went into the Clingo solver. Clingo’s superior performance has been demonstrated at international competitions [26]. Performance experiments comparing the two implementations of eFLINT are reported in Section 6.3.
- (d) The new implementation supports model-checking and searching for eFLINT scenarios satisfying given criteria. The new capabilities are inherited from the answer-set semantics in general and the Clingo solver in particular.

## 6.2. Compilation to Clingo

The goal of [44] for the translational semantics was to re-define eFLINT to align with its definition in [15, 16] and the existing interpreter implementation [13] as much as possible except for specific deviations (e.g., relating to reasoning by default). The translation captures the logic of states and transitions (comparable to the Event Calculus) and of built-in predicates (keywords) such as `Enabled`, `Violated`, and `Creates`. The logic of derivation is not part of the translation and is instead externalised to Clingo.

In what follows, we detail the differences between the two implementations (the ‘original’ and ‘new’ implementations of eFLINT, hereafter) and we remark which differences are incidental (e.g., short-term implementation shortcuts) and intentional (e.g., to support new use cases). The reader is referred to [44] for a more comprehensive discussion on Clingo and the eFLINT to Clingo translation. The full source code of the new interpreter, the corresponding documentation, and the full suite of experimental correctness and performance tests are available at [81].

Figure 3 illustrates the translation of a simple eFLINT fact-type definition to Clingo. Rule 1 shows the translation of the `Derived from` clause. The only work for the translation is resolving subtle differences between how eFLINT expressions and Clingo rules encode information. For example, eFLINT expressions can be arbitrarily nested, but Clingo rule conditions must be sequenced. Rule 2 shows how more subtle aspects of the specification are made explicit. This rule defines how instances of `controls` are enumerated (see the input `Identified by`): instances of `controls` that hold true can be enumerated (as the type has an infinite domain, in this case). Rule 3 is not input specific, showing how the output also includes an explicit encoding of the eFLINT semantics in general, which

```

Fact controls Identified by user * dataset
  Derived from (Foreach dataset:
    controls(user("Admin"),dataset)
    Where Not(Exists user: user != user("Admin")
      && controls(user,dataset)))

```

... is translated to ...

```

% 1: Encoding the input clause (Derived from ...)
in((derived,controls(user("Admin"),dataset(A))),S) :-
  state(S) ; not 0 < #count{ user(B) : #true ,not
    user(B) = user("Admin")
  ,in((holds,controls(user(B),dataset(A))),S)
  ,in((enum,user(B)),S) } ; in((enum,dataset(A)),S).

% 2: Encoding the set of quantifiable controls instances
in((enum,controls(user(A),dataset(B))),S) :- state(S) ;
  in((holds,controls(user(A),dataset(B))),S).

% 3: Encoding the semantics of derivation vs. holding
in((holds,I),S) :- in((derived,I),S) ;
  not in((suppressed,I),S) ; not in((terminated,I),S).

% (omitted: other rules encoding the eFLINT semantics,
  relating "terminates", "in", "holds", "enum", etc.)

```

Figure 3: An example translating eFLINT (top) to Clingo (bottom).

eFLINT programmers may take for granted. Here, it logically relates an arbitrary instance `I` being derived and holding in an arbitrary state `S`, but only if it is not suppressed or terminated.

*Robust Reasoning by Default.* Via Clingo, the new eFLINT interpreter implements the *answer-set* or *stable-model* semantics. Consequently, the tool detects and rejects any inputs that express logical contradictions (as zero stable models) or any symmetries, i.e., ‘ambiguities’ (as two or more stable models). Users understand these as fatal problems in their specifications, and can use the tool’s feedback to diagnose the underlying problem. For example, a subtle contradiction in the following definition of the `min-price-of` function results in zero stable models. From this, the user understands that a logical contradiction is present. Some scrutiny reveals that the condition for deriving the minimum price of the vase is contradictory; the condition `Not(Exists min-price-of': ...)` is satisfied *unless* the derivation rule is applied. (The issue can be resolved by replacing `<=` with `<`, breaking the cycle.)

```

Function min-price-of Identified by object * price
  Derived from bid Where
    Not(Exists min-price-of':
      min-price-of'.object == bid.object
      && min-price-of'.price <= bid.price).
+bid(object("Vase"), price(200)).

```

At present, users must diagnose any underlying problems themselves, guided only by Clingo’s enumerated answers. Work continues to ease this burden by producing a helpful diagnosis explaining the Clingo reasoner’s output.

In [44], we showed how the translation to Clingo revealed a subtle bug in the Haskell interpreter implementation, which caused the Haskell interpreter to apply spurious derivation rules in a small subset of the eFLINT specifications with unique stable models, i.e., a subset of accepted specifications. This demonstrates the difficulty

of correctly implementing eFLINT’s notion of derivation, further supporting its delegation to a thoroughly tested (e.g., Clingo) reasoner tool, rather than re-implemented from scratch.

*Simplified Language Features.* The re-definition of eFLINT posed an opportunity to simplify, standardise, and deduplicate the language features that had accumulated over the years. We list these changes here for completeness:

- (a) `When` clauses can no longer follow `Identified by` or `Related to` to restrict (infinite) domains, determining that only instances that satisfy the Boolean expression following the `When` are considered members of the defined type (see also the discussion on this construct at the end of Section 5.1). `When` may still be used to restrict instance expressions wherever they occur. The removed usage may be approximated in `Conditioned by` clauses to conditionally suppress which instances are derived.
- (b) The symmetric dual clauses of `Create`, `Terminate` and `Obfuscate` (e.g., `Created by`) were removed, as they added no expressive power, and no significant convenience, at their cost of making clauses of interest more difficult for users to find in specifications.
- (c) The preconditions on `Enabled` no longer propagate between actions via `Syncs with`. This dramatically simplifies the translation to Clingo and its reasoning about each action in each state. The original semantics is recoverable by (speculatively) performing the action and checking if a violation results. This usage prevails in practice anyway, where eFLINT reasoners are used to check *hypothetical* scenarios for compliance *before* they are reflected in reality. Such speculative execution is empowered by the ‘search’ feature discussed in Section 6.4.
- (d) Originally, `Identified by` had two usages: listing (the types of) fields and listing the finite instances of a type. The latter usage instead uses the new keyword `Domain`, to reduce confusion.

The translation to Clingo has been supported by various (internal) helper definitions, such as a `let`-construct, that could potentially be beneficial to users. This re-definition of the language thus also creates the opportunity to extend the syntax of the language without much effort.

*Unimplemented Incremental Reasoning Features.* In Section 5 it was discussed that the Haskell-based interpreter adopted incremental programming features, supporting its use in cases where user input is interleaved with reasoning and output. Firstly, it raises exceptions to request user input when evaluating open-typed instances. Secondly, it accepts the interleaving of extensions to the specification and to the scenario.

At present, the Clingo-based interpreter supports neither of these features. The specification and scenario are input first, and evaluated second. Work continues to support these features by re-introducing a layer that maintains a mutable intermediate state. The hope is to exploit Clingo’s more advanced features and configuration interface [50], such that the Clingo solver maintains the intermediate state as much as possible, resulting in better efficiency in these cases. Failing that, the new interpreter’s translation layer can re-introduce the state-bookkeeping of the Haskell interpreter. In the meantime, users can approximate these features by maintaining an eFLINT specification as the mutable state themselves, and interleaving their mutations with calls to the Clingo-based interpreter.

### 6.3. Differences in Runtime Performance

Even in cases of identical input-output behaviour, the two interpreters exhibit different performance at runtime. This subsection reports on performance experiments comparing the two eFLINT implementations. As we shall see, the new interpreter’s strength is in cases with complex, yet structured reasoning in each scenario step, and its weakness is reasoning about simple scenarios that span very long periods of time. Generally, the Clingo-based interpreter greatly outperforms the original, and we consider the few cases of slowdown to be acceptable.

*Experimental Setup.* Figures 4 to 8 plot the results of an illustrative selection of a larger set of experiments available as supplementary material [81]. Per plot, each measurement is the median of 10 runs on a machine with the following specifications: { processor: Intel core i9 9th generation (16 hardware threads), memory: 32GB RAM DDR4, storage: NVMe hard drive }. The experiments compare both interpreters’ single-threaded performance. In the Haskell-interpreter’s case, this is the nature of its implementation. In the Clingo-interpreter’s case, Clingo is configured to run in single-threaded mode, as cursory experiments found that it achieves no speedup from multi-threading for inputs with just one stable model, which is the case for all accepted eFLINT specifications. Whether preprocessing of the input or reconfiguration of Clingo enables speedup from parallelism remains to be explored in future work.

*Discussion of Experiments.* Figure 4 evaluates the performance of the interpreters given an input with many (combinations of) facts, but where only one derivation step is possible at a time. For example, in any scenario, only  $x(50)$  is derivable just after  $x(51)$  is derived. The interpreters exhibit similar response curves: runtime grows superlinearly with the number of derivations, presumably as a consequence of paying the price of storing and traversing larger knowledge bases. However, the new interpreter starts faster, and slows down more gradually. The net result is that, compared to the original interpreter, the new interpreter begins with a  $\times 2$  speedup, which grows to  $\times 25$  for the most costly inputs. We have no explanation for

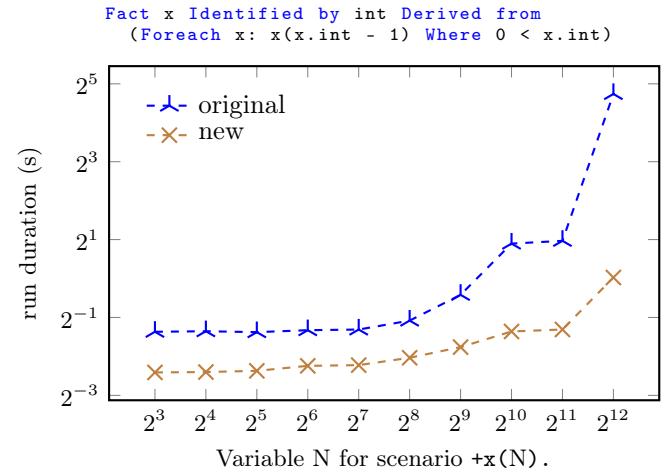


Figure 4: Speed in unfolding long derivation chains.

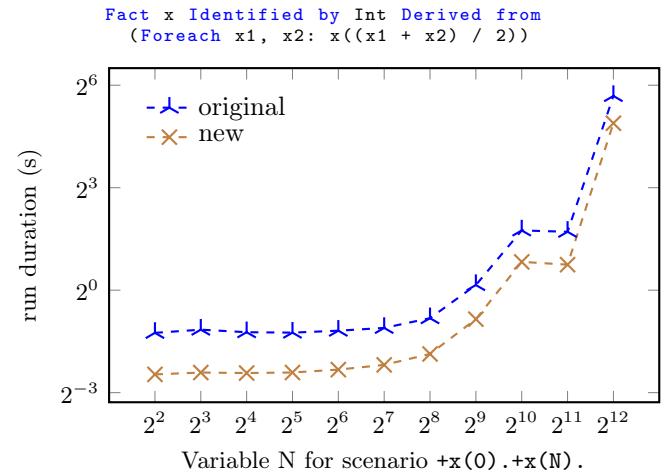


Figure 5: Speed in reasoning with substantial integer arithmetic.

```

Fact x Identified by Int Derived from
(Foreach y: y.x1), (Foreach y: y.x2)
(Foreach y: y.x3), (Foreach x: x(x - 1) Where 0 < x)
Fact y Identified by x1 * x2 * x3 Derived from
(Foreach x: y(x,x,x)),
(Foreach x1, x2, x3: y(x1,x2,x3)
  Where (x1 == x2 || x2 != x3)
    && (Exists y: x2 < y.x1))

```

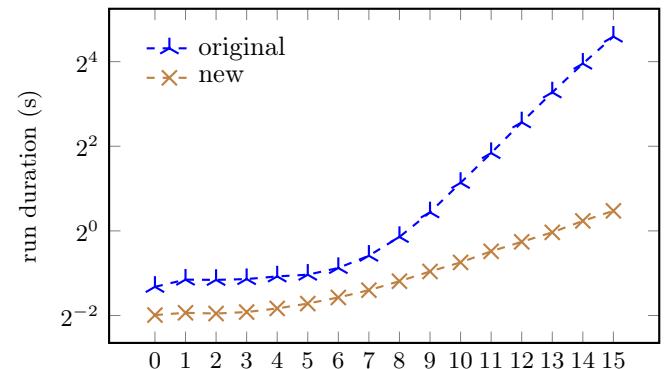


Figure 6: Speed in reasoning in cases requiring the enumeration, combination, and comparison of many instances.

$N = 2^{11}$  representing a significant outlier, but we note that it was reliably reproduced by both interpreters.

Figure 5 demonstrated a case where the new interpreter maintained approximately  $\times 2$  speedup across all test scenarios. We expect that the cost of the division operator ( $\backslash$ ) predominates, but more testing is required.

Figure 6 shows the results of an experiment expected to emphasize the strengths of Clingo: the interpreter must quantify, combine, and compare many terms. This specification causes runtime cost to scale more sharply with the independent variable, but the speedup of the new interpreter over the original are similar to that shown in Figure 4: the simplest scenario induces  $2\times$  speedup, and the largest scales up to  $17\times$  speedup.

Figure 7 shows a case where the original, Haskell-based implementation is expected to excel: minimal reasoning is spread over a long scenario. Intuitively, the original interpreter was optimised around the ‘chronological’ characteristic of eFLINT scenarios and states: reasoning about each state  $N$  depends only on the actions and derivations in states  $N - 1$  and  $N$ , respectively. The Haskell interpreter reasons sequentially, wasting no time considering unrelated states. By comparison, Clingo threatens to reason naively, because the entire eFLINT specification and scenario are encoded as a monolithic Clingo ruleset, and we cannot assume that Clingo recognises and exploits its subtle chronological property. Indeed, we find that the new interpreter’s speedup peaks with 94% at  $N = 2^6$ , waning to 82% speedup at  $N = 2^{10}$ . However, clearly, the new interpreter overall still outperforms the original interpreter by a significant margin, and this experiment suggests that its performance advantage will only be lost in scenarios far longer than we expect to encounter in practice.

Finally, Figure 8 shows a case where the original interpreter outperformed the new one. This is a carefully crafted encoding of the *prime sieving problem*: enumerate all prime numbers to a given maximum [8]. Here, we expect Clingo’s heuristics for selecting rules to fail. Both interpreters then use a similar solution: naively enumerate integer pairs and check if they are factors. Then performance depends on the order the integers are enumerated. Evidently, the Haskell interpreter gets lucky more often. But this outcome was sensitive to the encoding of the problem. For example, chaining `addleq` via `Derived from` instead of `Syncs with` slowed down the Haskell interpreter. More experimentation is required to precisely explain the interpreters’ sensitivity to such particulars of the specification. We conclude that the Clingo-based reasoner benefits from specifications with more structured reasoning, where Clingo’s heuristics speed up its reasoning.

#### 6.4. Generalisation to Scenario Searching

In the typical usage, eFLINT interpreter play out the specified consequences of a given scenario. However, the interpreter’s basis in the answer-set semantics affords a powerful new usage: the user inputs (a) an eFLINT specification, (b) a definition of the search space, and (c) a

```
Fact x Identified by Int Derived from
  (Foreach y: y.x1), (Foreach y: y.x2)
  (Foreach y: y.x3), (Foreach x: x(x - 1) Where 0 < x)
Fact y Identified by x1 * x2 * x3 Derived from
  (Foreach x: y(x,x,x)),
  (Foreach x1, x2, x3: y(x1,x2,x3)
    Where (x1 == x2 || x2 != x3)
      && (Exists y: x2 < y.x1))
```

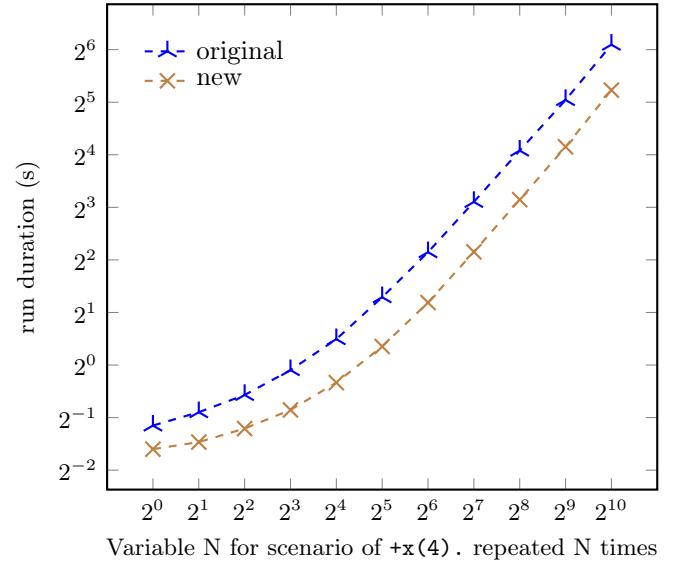


Figure 7: Speed in reasoning about the same, moderate reasoning work at each step of scenarios with varying lengths.

```
Fact prime Identified by Int
  Derived from (Foreach int: prime(int))
    Where Not(Exists int1, int2:
      1 Where int1 * int2 == int))
Event addleq Related to int Creates int
  Syncs with addleq(int - 1) Where 2 < int
```

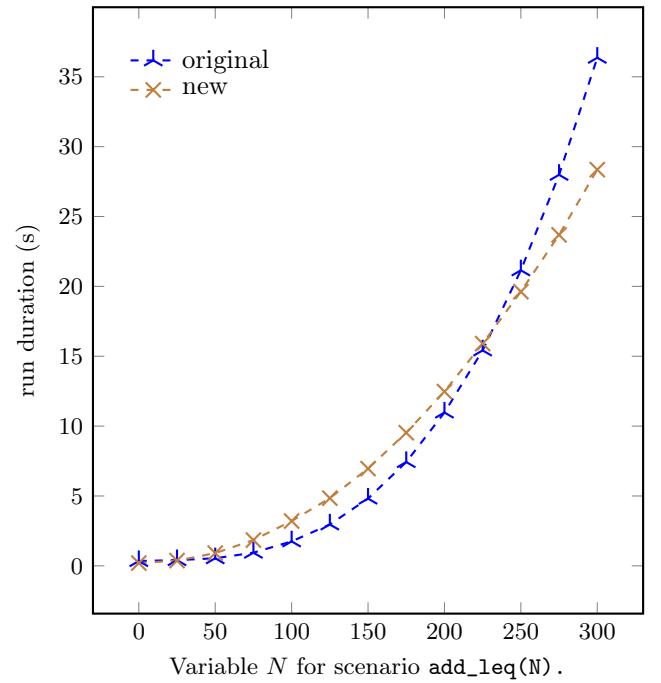


Figure 8: Speed in enumerating all prime numbers up to a maximum.

search criterion recognising (un)desirable eFLINT scenarios. Then the tool outputs scenarios within the search space that satisfy the criterion. We distinguish two kinds of application of this new usage in practice:

- (a) The tool *recommends* a scenario that reaches a given goal, by enumerating examples reaching the goal.
- (b) The tool *model-checks* a given scenario property, by enumerating counterexamples to the property.

The user’s search problem is encoded as Clingo rules alongside those translated from the eFLINT input.

Consider the following example. The *breadth* of the search space is limited by constraining the transitions out of each state: exactly one enabled `raise_hand` action is chosen to be performed. Note that *chosen* is distinguished from *triggered*, allowing for cases where one chosen action indirectly triggers several in synchrony, via `Syncs with`.

```
% Choose exactly one enabled raise_hand per state.
1 = { choose(I,S) : in((enabled,I), S),
      I = raise_hand(Actor) } :- state(S); state(S + 1).
in((trigger,I), S) :- choose(I,S). % perform all chosen
```

The *depth* of the search space is limited to (continuous) scenarios up to length 1000, letting the search terminate.

```
{ state(S) } :- S = 1..1000. % choose final state S.
state(S) :- 1 <= S ; state(S + 1). % close range 1...S.
```

The search tree is rooted by initialising particular bidders, auctioneers, and auctions in state 1. This step can be generalised to searches for extensions of a given  $N$ -state scenario by initialising the first  $N$  states in this manner.

```
in(( create,
     bidder("Amy")),1).
in(( create,
     bidder("Bob")),1).
in(( create,
     auctioneer("Dan")),1).
in(( create,min_price_of(object("Vase"),price(200))),1).
in((trigger,
     start_bidding(object("Vase"))),1).
```

Finally, the following defines the search criterion: each scenario must witness a violation of the property of interest: two bidders never offer the same price for the same object.

```
:- counterexample. % constraint: this must be true.
counterexample :- in((holds, bid(X,Obj,Price)), _);
                X != Y ; in((holds, bid(Y,Obj,Price)), _).
```

When all the above rules are input to the Clingo reasoner alongside the (translated) specification itself, the Clingo solver acts as a model-checker: it searches for counterexamples to our property of interest. The property `holds` (within the search space) if and only the answer set is empty, because no counterexamples were found.

This approach to expressing and solving scenario search problems taps into the full power of Clingo’s answer-set solving capabilities. However, encoding these search problems requires interfacing with the Clingo-encoded output of the eFLINT specification translation. This is not ideal because, as can be seen above, our eFLINT encodings in Clingo are verbose and intricate. In the future, we hope to build an abstraction atop Clingo for users to express their scenario search problems and criteria. The challenge is to balance the expressive power of the full Clingo language, with simplicity and alignment with the abstractions that

eFLINT already provides. In this, it is prudent to draw from the literature, such as [9, 61], concerning problems elegantly encoded in the Clingo language and efficiently solved by the tool. Initial experiments with an eFLINT-specific version of LTL and bounded-model checking are reported here [36].

## 7. Discussion and Future Work

This section reflects on the overall development process and discusses the main avenues of future work we perceive.

The development of eFLINT went through phases, starting with the analyses of historical cases and simulation, followed by dynamically evolving cases, model checking and system integration. Applications involving eFLINT have been developed by BSc, MSc and PhD students, researchers and practitioners at various technological readiness levels<sup>12</sup> (TRL), with most applications validating concepts ‘in the lab’ ( $\leq$ TRL3) or performing demonstrations with stakeholders in realistic test environments ( $\leq$ TRL6). A notable exception is the usage of eFLINT as part of an operational service for data governance within the Dutch Metropolitan Innovations ecosystem, a national ecosystem for data-driven innovation. The ecosystem uses eFLINT for formalising and reasoning with the ecosystem’s regulation, the GDPR privacy regulation, and data sharing agreements. The formalised regulations and agreements are used for producing (ex-ante) authorisations to data transactions in a form of purpose-based access control, for (ex-post) transaction monitoring as part of auditing, and for giving of attestations as part of certification.

Pragmatic design decisions have been made to ensure the language is sufficiently flexible to be used for various kinds of applications. However, a flexible solution is not always the most user-friendly solution. For example, in some applications a strict separation between specification (type declarations) and scenarios (statements) is useful. And the closed world assumption and finite domains are needed for model checking. In general, to use the language in a specific context, certain constraints need to be *assumed* that in a more restrictive variant of the language could be *enforced* syntactically or using static analyses.

At present, the language is not sufficiently friendly for legal experts to use without software expertise. Experience with computer science students does suggest that the language is relatively easy to learn for those with programming experience, especially declarative programming experience. Both language implementations are also lacking a mechanism to report for true (or false) facts how they came to be true (or false). Ideally, such a report contains the sequence of rules and statements that were applied to render the fact true (or false). Such an **explainability** feature is difficult to implement due to the use of negation and the subtle interplay between derivation rules and procedural

---

<sup>12</sup>We follow the definition of TRL by the European Committee.

rules (e.g., `Creates` and `Terminates`). In Section 5 we did note the use of an execution graph to maintain execution history in the original interpreter as an essential, yet less fine-grained, contributor to explainability.

An open question still being investigated is how the interpretation process can best be supported. Our current best effort involves a pipeline in which natural language processing (NLP) produces FLINT frames from a normative source, a legal/domain expert completes and refines the FLINT frames, the FLINT frames are automatically translated to eFLINT, and a software expert completes and refines the eFLINT code. Software expertise is needed, in particular, to refine the eFLINT code in order to formalise the connection between the eFLINT specification and the software environment in which the specification will integrate, e.g., via instrumentation, scripting or API adaptation. Software expertise is also needed to add computational details not easily derived from the natural language sources, such as quantification, and the relations between fields and variables to express constraints on the possible substitutions of variables. FLINT specifications are easier to develop by legal experts as they are textual, tabular, and not executable (only machine-readable). Both languages are currently lacking an explicit mechanism for conflict resolution between rules and normative positions, e.g., based on a hierarchy between authorities or precedence between normative sources (**norm priorities**, discussed in Section 4.2).

Automation of the interpretation process is needed to deal with the large amount of normative sources and the pace with which they are produced. In on-going investigations we are experimenting with the use of code synthesis techniques and generative AI on top of previous efforts using NLP. However, intervention by legal experts is needed as interpretation is inherently subjective (e.g., human-in-the-loop or human-on-the-loop). Provenance, **versioning** and retaining **source references** are important qualities of an interpretation pipeline for traceability and accountability. For these purposes, meta-data about eFLINT fragments should be maintained by the pipeline as is currently only being done for FLINT frames.

As possible future work, we envision a new design of a norm specification language which comprises a computational core language and one or more legal surface-level languages. The experiences gained during the development of eFLINT, reported in this paper, make it possible to identify a minimal computational core that is still completed with respect to the identified requirements. The motivation behind such a (re-)design is twofold. One motivation is improving usability for legal and domain experts by offering a more tailored experience. For example, a keyword such as `Syncs with` should be hidden as we should not require legal experts to think about transition system semantics. A rule-based syntax (top, in the fragment below) can be used to express the relation between `raise-hand` and `place-bid` without `Extend` and `Syncs with` (bottom) as follows:

```
raise-hand(bidder) QualifiesAs
  place-bid(bidder, display.object, price)
  Where price = ...
...
... is equivalent to ...

Extend raise-hand Syncs with place-bid
  (bidder=actor, object=display.object, price = ...)
```

Another motivation is to experiment with the computational meaning of additional **normative concepts** such as obligations and prohibitions. The notion of violation can be replaced in the core language by monitors that flag properties satisfied by triggered transitions or reached states (see the work on Seaso for inspiration [41]). The connection between the legal surface-language and the core language can then establish whether flagged properties are interpreted as violations or, perhaps, as desirable events based on additional, normative information. For example, if a prohibited action is performed, this event may not constitute a violation of the action is explicitly permitted by a (higher) authority (a form of **prioritisation**).

## 8. Conclusion

This paper has reflected on the design and implementation of eFLINT based on requirements extracted from experiments in various application areas. The presented discussion serves to benefit researchers working on specification languages in general and computational representation and execution of legal concepts in particular. The main advantage of the present language is the flexibility that permits it to be used in multiple application areas simultaneously such as model checking, checking historical, hypothetical and dynamically evolving scenarios, and performing access and usage control. Based on the Hohfeldian framework for legal reasoning, the language enables the formalisation of norms from a wide variety of sources. And through specific language constructs, normative sources can be formalised at their own level of abstraction, reused, and interconnected. The flexibility is achieved through certain pragmatic design decisions and by embedding software engineering principles such as modularity, extensibility, and inheritance. The main future developments are to increase the usability of the language, in particular for domain experts, by experimenting with new ways of interacting with the language, including alternative surface-level languages, static analyses, user interfaces and development environments in general.

### Acknowledgments

This work has been executed as part of the SSPDDP project supported by NWO (628.009.014), the AMDEX Fieldlab project supported by Kansen Voor West EFRO (KVV00309) and the province of Noord-Holland, and the DMI ecosystem supported by the National Growth Fund.

Special thanks go to all the students, researchers, and domain experts that have provided feedback over the years.

## References

- [1] van der Aalst, W.M.P.: Conformance Checking, pp. 191–213. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). [https://doi.org/10.1007/978-3-642-19345-3\\_7](https://doi.org/10.1007/978-3-642-19345-3_7)
- [2] Al-Shaer, E.S., Hamed, H.H.: Modeling and management of firewall policies. *IEEE Transactions on Network and Service Management* **1**(1), 2–10 (2004). <https://doi.org/10.1109/TNSM.2004.4623689>
- [3] Alfuhaid, S., Anda, A.A., Amyot, D., Roveri, M., Mylopoulos, J.: Symboleoac: An access control model for legal contracts. In: Paja, E., Zdravkovic, J., Kavaklı, E., Stirna, J. (eds.) *The Practice of Enterprise Modeling*. pp. 227–243. Springer Nature Switzerland, Cham (2025). [https://doi.org/10.1007/978-3-031-77908-4\\_14](https://doi.org/10.1007/978-3-031-77908-4_14)
- [4] Alsayed Kassem, J., Allaart, C., Amiri, S., Kebede, M., Müller, T., Turner, R., Belloum, A., van Binsbergen, L.T., Grunwald, P., van Halteren, A., Grosso, P., de Laat, C., Klous, S.: Building a Digital Health Twin for Personalized Intervention: The EPI Project. In: Haverkort, B.R., de Jongste, A., van Kuilenburg, P., Vromans, R.D. (eds.) *Commit2Data*. Open Access Series in Informatics (OASIcs), vol. 124, pp. 2:1–2:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2024). <https://doi.org/10.4230/OASIcs.Commit2Data.2>, <https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Commit2Data.2>
- [5] Alsayed Kassem, J., Müller, T., Esterhuyse, C.A., Kebede, M.G., Osseyran, A., Grosso, P.: The epi framework: A data privacy by design framework to support healthcare use cases. *Future Generation Computer Systems* **165**, 107550 (2025). <https://doi.org/https://doi.org/10.1016/j.future.2024.107550>, <https://www.sciencedirect.com/science/article/pii/S0167739X24005144>
- [6] Athan, T., Governatori, G., Palmirani, M., Paschke, A., Wyner, A.: LegalRuleML: Design Principles and Foundations, pp. 151–188. Springer International Publishing, Cham (2015). [https://doi.org/10.1007/978-3-319-21768-0\\_6](https://doi.org/10.1007/978-3-319-21768-0_6)
- [7] Aştefănoaei, L., de Boer, F., Dastani, M., Meyer, J.J.: On the Semantics and Verification of Normative Multi-Agent Systems **15**(13), 2629–2652. <https://doi.org/10.3217/jucs-015-13-2629>
- [8] Bahig, H.M., Hazber, M.A.G., Al-Utaibi, K.A., Nassr, D.I., Bahig, H.M.: Efficient sequential and parallel prime sieve algorithms. *Symmetry* **14**(12), 2527 (2022). <https://doi.org/10.3390/SYM14122527>, <https://doi.org/10.3390/sym14122527>
- [9] Balduccini, M., Barborak, M., Ferrucci, D.A.: Pushing the limits of clingo’s incremental grounding and solving capabilities in practical applications. *Algorithms* **16**(3), 169 (2023). <https://doi.org/10.3390/A16030169>, <https://doi.org/10.3390/a16030169>
- [10] Balke, T., da Costa Pereira, C., Dignum, F., Lorini, E., Rotolo, A., Vasconcelos, W., Villata, S.: Norms in MAS: Definitions and Related Concepts. In: Andrijghetto, G., Governatori, G., Noriega, P., van der Torre, L.W.N. (eds.) *Normative Multi-Agent Systems*, vol. 4, pp. 1–31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik. <https://doi.org/10.4230/DFU.VOL4.12111.1>
- [11] Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to Runtime Verification. In: Bartocci, E., Falcone, Y. (eds.) *Lectures on Runtime Verification*, vol. 10457, pp. 1–33. Springer International Publishing, Cham (2018). [https://doi.org/10.1007/978-3-319-75632-5\\_1](https://doi.org/10.1007/978-3-319-75632-5_1), [http://link.springer.com/10.1007/978-3-319-75632-5\\_1](http://link.springer.com/10.1007/978-3-319-75632-5_1), series Title: Lecture Notes in Computer Science
- [12] Biere, A.: Bounded Model Checking. In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, vol. 336, pp. 739–764. IOS Press, 2 edn. <https://doi.org/10.3233/faia201002>
- [13] van Binsbergen, L.T.: Haskell prototype implementation of the eflint language. <https://gitlab.com/eflint/haskell-implementation> (2020), [Online, accessed 23 October 2025]
- [14] van Binsbergen, L.T.: The eFLINT project on GitLab. <https://gitlab.com/eflint> (2020), [Online, accessed 23 October 2025]
- [15] van Binsbergen, L.T., Kebede, M.G., Baugh, J., van Engers, T., van Vuurdelen, D.G.: Dynamic generation of access control policies from social policies. *Procedia Computer Science* **198**, 140–147 (January 2022). <https://doi.org/10.1016/j.procs.2021.12.221>
- [16] van Binsbergen, L.T., Liu, L., van Doesburg, R., van Engers, T.: eFLINT: A Domain-Specific Language for Executable Norm Specifications. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. pp. 124–136. GPCE 2020, ACM (2020). <https://doi.org/10.1145/3425898.3426958>
- [17] van Binsbergen, L.T., Oost-Rosengren, M., Schreijer, H., Dijkstra, F., van Dijk, T.: AMdEX Reference Architecture – version 1.0.0. Zenodo (2 2024). <https://doi.org/10.5281/zenodo.10565915>

- [18] van Binsbergen, L.T., Scott, E., Johnstone, A.: GLL parsing with flexible combinators. In: Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering. SLE 2018, ACM (2018). <https://doi.org/10.1145/3276604.3276618>
- [19] van Binsbergen, L.T., Scott, E., Johnstone, A.: Purely functional gll parsing. Journal of Computer Languages (2020). <https://doi.org/10.1016/j.cola.2020.100945>
- [20] van Binsbergen, L.T., Verano Merino, M., Jeanjean, P., van der Storm, T., Combemale, B., Barais, O.: A Principled Approach to REPL Interpreters, pp. 84–100. ACM (2020). <https://doi.org/10.1145/3426428.3426917>
- [21] Breebaart, M.: Juriconnect standaard BWB versie 1.3.1 (Oct 2014)
- [22] Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., van der Torre, L.: The BOID Architecture - Conflicts Between Beliefs, Obligations, Intentions and Desires. In: In Proceedings of the Fifth International Conference on Autonomous Agents. pp. 9–16. ACM Press (2001). <https://doi.org/10.1145/375735.375766>
- [23] Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg (2009). [https://doi.org/10.1007/978-3-642-02161-9\\_3](https://doi.org/10.1007/978-3-642-02161-9_3)
- [24] Buterin, V.: Ethereum white paper (2018)
- [25] Byun, J.W., Bertino, E., Li, N.: Purpose based access control of complex data for privacy protection. In: Proceedings of the tenth ACM symposium on Access control models and technologies. pp. 102–110 (2005). <https://doi.org/10.1145/1063979.1063998>
- [26] Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: Design and results of the fifth answer set programming competition. Artif. Intell. **231**, 151–181 (2016)
- [27] Cardoso, R.C., Ferrando, A., Dennis, L.A., Fisher, M.: Implementing Ethical Governors in BDI. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 22–41. Springer International Publishing, Cham (2022). [https://doi.org/10.1007/978-3-030-97457-2\\_2](https://doi.org/10.1007/978-3-030-97457-2_2)
- [28] Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv Symbolic Model Checker. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 334–342. Lecture Notes in Computer Science, Springer International Publishing. [https://doi.org/10.1007/978-3-319-08867-9\\_22](https://doi.org/10.1007/978-3-319-08867-9_22)
- [29] Ceci, M., Sannier, N., Abualhaija, S., Shin, D., Bianculli, D., Halling, M.: Toward Automated Compliance Checking of Fund Activities Using Runtime Verification Techniques. In: Proceedings of the 1st IEEE/ACM Workshop on Software Engineering Challenges in Financial Firms. pp. 19–20. ACM, Lisbon Portugal (Apr 2024). <https://doi.org/10.1145/3643665.3648045>
- [30] Charalambides, M., Flegkas, P., Pavlou, G., Bandara, A.K., Lupu, E., Russo, A., Dulay, N., Słoman, M., Rubio-Loyola, J.: Policy conflict analysis for quality of service management. In: 6th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), 6–8 June 2005, Stockholm, Sweden. pp. 99–108. IEEE (2005). <https://doi.org/10.1109/POLICY.2005.23>
- [31] Chowdhury, O., Chen, H., Niu, J., Li, N., Bertino, E.: On XACML’s Adequacy to Specify and to Enforce HIPAA. In: Gunter, C.A., Peterson, Z.N.J. (eds.) 3rd USENIX Workshop on Health Security and Privacy. HealthSec ’12, USENIX Association (2012)
- [32] Clark, K.L.: Negation as failure. In: Gallaire, H., Minker, J. (eds.) Logic and Data Bases, Symposium on Logic and Data Bases, Centre d’études et de recherches de Toulouse, France, 1977. pp. 293–322. Advances in Data Base Theory, Plenum Press, New York (1977). [https://doi.org/10.1007/978-1-4684-3384-5\\_11](https://doi.org/10.1007/978-1-4684-3384-5_11)
- [33] Clarke, E.M.: Model Checking. MIT Press, 2 edn.
- [34] Colmerauer, A., Roussel, P.: The Birth of Prolog, p. 331–367. Association for Computing Machinery, New York, NY, USA (1996). <https://doi.org/10.1145/234286.1057820>
- [35] Criado, N., Argente, E., Noriega, P., Botti, V.: Towards a normative BDI architecture for norm compliance. CEUR Workshop Proceedings **627**, 65–81 (2010)
- [36] De Geus, F.W.: Model Checking Normative Systems (2022)
- [37] Deljoo, A., van Engers, T., van Doesburg, R., Gommans, L., de Laat, C.: A Normative Agent-based Model for Sharing Data in Secure Trustworthy Digital Market Places. Proceedings of the 10th International Conference on Agents and Artificial Intelligence (April), 290–296 (2018)

- [38] Dignum, F., Kinny, D., Sonenberg, L.: Motivational attitudes of agents: On desires, obligations, and norms. Lecture Notes in Computer Science **2296**(Section 2), 83 (2002). [https://doi.org/10.1007/3-540-45941-3\\_9](https://doi.org/10.1007/3-540-45941-3_9)
- [39] van Doesburg, R., van Engers, T.: The false, the former, and the parish priest. In: Proceedings of the Seventeenth International Conference on Artificial Intelligence and Law. pp. 194–198. ICAIL 2019, ACM (2019). <https://doi.org/10.1145/3322640.3326718>
- [40] van Doesburg, R., van der Storm, T., van Engers, T.: CALCULEMUS: Towards a formal language for the interpretation of normative systems. In: AI4J Workshop at ECAI 2016. pp. 73–77. AI4J 2016 (2016)
- [41] Esterhuyse, C.A.: Specification-centric Multi Agent Systems. Ph.D. thesis, University of Amsterdam (2025)
- [42] Esterhuyse, C.A., van Binsbergen, L.T.: Cooperative specification via composition control. In: Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering. p. 2–15. SLE ’24, Association for Computing Machinery, New York, NY, USA (2024). <https://doi.org/10.1145/3687997.3695635>
- [43] Esterhuyse, C.A., Müller, T., van Binsbergen, L.T.: JustAct: Actions Universally Justified by Partial Dynamic Policies. In: Castiglioni, V., Fracalanza, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems. pp. 60–81. Springer Nature Switzerland, Cham (2024). [https://doi.org/10.1007/978-3-031-62645-6\\_4](https://doi.org/10.1007/978-3-031-62645-6_4)
- [44] Esterhuyse, C.A., Müller, T., van Binsbergen, L.T.: A stable model semantics for eflint norm specifications and model checking scenarios. In: Proceedings of the 24th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. p. 80–93. GPCE ’25, Association for Computing Machinery, New York, NY, USA (2025). <https://doi.org/10.1145/3742876.3742882>, the accompanying source code and experimental results are available at [81].
- [45] Esterhuyse, C.A., Müller, T., Van Binsbergen, L.T., Belloum, A.S.Z.: Exploring the enforcement of private, dynamic policies on medical workflow execution. In: 2022 IEEE 18th International Conference on e-Science (e-Science). pp. 481–486 (2022). <https://doi.org/10.1109/eScience55777.2022.00086>
- [46] Ethereum: Solidity documentation online. <https://solidity.readthedocs.io> (2016), [Online, accessed 23 October 2025]
- [47] Fisler, K., Krishnamurthi, S., Meyerovich, L., Tschantz, M.: Verification and Change-Impact Analysis of Access-Control Policies. In: Proceedings of the 27th International Conference on Software Engineering. pp. 196–205. ICSE 2005, Association for Computing Machinery, New York, NY, USA (2005). <https://doi.org/10.1145/1062455.1062502>
- [48] Fratrič, P., Holzenberger, N., Amariles, D.R.: Rules2lab: from prolog knowledge-base, to learning agents, to norm engineering. In: Collier, R., Ricci, A., Nallur, V., Burattini, S., Omicini, A. (eds.) Multi-Agent Systems. pp. 274–282. Springer Nature Switzerland, Cham (2025). [https://doi.org/10.1007/978-3-031-93930-3\\_16](https://doi.org/10.1007/978-3-031-93930-3_16)
- [49] Frölich, D., van Binsbergen, L.T.: A Generic Back-End for Exploratory Programming. In: The 22nd International Symposium on Trends in Functional Programming (TFP 2021). LNCS, vol. 12834. Springer (2021). [https://doi.org/10.1007/978-3-030-83978-9\\_2](https://doi.org/10.1007/978-3-030-83978-9_2)
- [50] Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo. Theory Pract. Log. Program. **19**(1), 27–82 (2019)
- [51] Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, August 15–19, 1988 (2 Volumes). pp. 1070–1080. MIT Press (1988)
- [52] Goldstein, H., Hughes, J., Lampropoulos, L., Pierce, B.C.: Do Judge a Test by its Cover. In: Yoshida, N. (ed.) Programming Languages and Systems. pp. 264–291. Springer International Publishing (2021). [https://doi.org/10.1007/978-3-030-72019-3\\_10](https://doi.org/10.1007/978-3-030-72019-3_10)
- [53] Gommans, L., Xu, L., Demchenko, Y., Wan, A., Cristea, M., Meijer, R., de Laat, C.: Multi-domain lightpath authorization, using tokens. Future Generation Computer Systems **25**(2), 153–160 (2009). <https://doi.org/10.1016/j.future.2008.07.013>
- [54] Governatori, G., Idelberger, F., Milosevic, Z., Riveret, R., Sartor, G., Xu, X.: On legal contracts, imperative and declarative smart contracts, and blockchain systems. Artificial Intelligence and Law **26**(4), 377–409 (2018). <https://doi.org/10.1007/s10506-018-9223-3>
- [55] Governatori, G., Maher, M., Antoniou, G., Billington, D.: Argumentation Semantics for Defeasible Logic. Journal of Logic and Computation **14**(5), 675–702 (10 2004). <https://doi.org/10.1093/logcom/14.5.675>

- [56] Governatori, G., Rotolo, A.: Changing legal systems: Legal abrogations and annulments in Defeasible Logic **18**(1), 157–194. <https://doi.org/10.1093/jigpal/jzp075>
- [57] Groefsema, H., van Beest, N.: Supporting business process variability through declarative process families. *Computers in Industry* **159-160**, 104107 (2024). <https://doi.org/10.1016/j.compind.2024.104107>
- [58] Groefsema, H., van Beest, N.: Design-time compliance of service compositions in dynamic service environments. In: 2015 IEEE 8th International Conference on Service-Oriented Computing and Applications (SOCA). pp. 108–115 (2015). <https://doi.org/10.1109/SOCA.2015.14>
- [59] Groefsema, H., Beest, N.R.T.P.v., Governatori, G.: On the use of the conformance and compliance keywords during verification of business processes. In: Di Ciccio, C., Dijkman, R., del Río Ortega, A., Rinderle-Ma, S. (eds.) Business Process Management Forum. pp. 21–37. Springer International Publishing, Cham (2022). [https://doi.org/10.1007/978-3-031-16171-1\\_2](https://doi.org/10.1007/978-3-031-16171-1_2)
- [60] Hashmi, M., Governatori, G., Lam, H., Wynn, M.T.: Are we done with business process compliance: state of the art and challenges ahead. *Knowl. Inf. Syst.* **57**(1), 79–133 (2018). <https://doi.org/10.1007/s10115-017-1142-1>
- [61] Havur, G., Cabanillas, C., Polleres, A.: Benchmarking answer set programming systems for resource allocation in business processes. *Expert Syst. Appl.* **205**, 117599 (2022). <https://doi.org/10.1016/J.ESWA.2022.117599>, <https://doi.org/10.1016/j.eswa.2022.117599>
- [62] Herrestad, H.: Norms and formalization. In: Proceedings of the 3th International Conference on Artificial Intelligence and Law. pp. 175–184. ICAIL 1993, ACM (1993). <https://doi.org/10.1145/112646.112667>
- [63] Hohfeld, W.N.: Fundamental legal conceptions as applied in judicial reasoning. *The Yale Law Journal* **26**(8), 710–770 (1917). <https://doi.org/10.2307/786270>
- [64] Hohfeld, W.: Some fundamental legal conceptions as applied in judicial reasoning. *Yale Law Journal* **23**(1), 59–64 (1913)
- [65] Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
- [66] Hu, V.C., Kuhn, D.R., Ferraiolo, D.F.: Attribute-based access control. *Computer* **48**(2), 85–88 (2015). <https://doi.org/10.1109/MC.2015.33>
- [67] Iannella, R., Villata, S.: ODRL information model 2.2. W3C Recommendation (2018)
- [68] Jabal, A., Davari, M., Bertino, E., Makaya, C., Calo, S., Verma, D., Russo, A., Williams, C.: Methods and tools for policy analysis. *ACM Computing Surveys* **51**(6) (2019). <https://doi.org/10.1145/3295749>, <https://doi.org/10.1145/3295749>
- [69] Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
- [70] Jones, A., Sergot, M.: A Formal Characterisation of Institutionalised Power. *Logic Journal of the IGPL* **4**(3), 427–443 (06 1996). <https://doi.org/10.1093/jigpal/4.3.427>
- [71] Kafalý, O., Ajmeri, N., Singh, M.P.: Revani: Revising and Verifying Normative Specifications for Privacy **31**(5), 8–15. <https://doi.org/10.1109/MIS.2016.89>
- [72] Kebede, M.G., Sileno, G., Van Engers, T.: A Critical Reflection on ODRL. In: Rodríguez-Doncel, V., Palmirani, M., Araszkiewicz, M., Casanovas, P., Pagallo, U., Sartor, G. (eds.) AI Approaches to the Complexity of Legal Systems XI-XII. pp. 48–61. Springer International Publishing, Cham (2021). [https://doi.org/10.1007/978-3-030-89811-3\\_4](https://doi.org/10.1007/978-3-030-89811-3_4)
- [73] Kephart, J., Chess, D.: The vision of autonomic computing. *Computer* **36**(1), 41–50 (2003). <https://doi.org/10.1109/MC.2003.1160055>
- [74] Klyne, G., Carroll, J.J.: Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C Recommendation (2004), <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>
- [75] Kosenkov, O., Zabardast, E., Fucci, D., Mendez, D., Unterkalmsteiner, M.: Privacy by design: Aligning gdpr and software engineering specifications with a requirements engineering approach. *Information and Software Technology* **190**, 107946 (2026). <https://doi.org/https://doi.org/10.1016/j.infsof.2025.107946>, <https://www.sciencedirect.com/science/article/pii/S095058492500285X>
- [76] Kowalski, R., Sergot, M.: A logic-based calculus of events. *New Generation Computing* **4**(1), 67–95 (1986). <https://doi.org/10.1007/BF03037383>, <https://doi.org/10.1007/BF03037383>
- [77] Liu, L.C., Parizi, M.M., van Binsbergen, L.T., van Engers, T.: Regulatory services to automate compliance with ex-post enforcement. In: Proceedings of AI Approaches to the Complexity of Legal Systems (AICOL) 2023. Springer (2024)

- [78] Liu, L., Sileno, G., van Engers, T.M.: Digital enforceable contracts (DEC): making smart contracts smarter **334**, 235–238 (2020). <https://doi.org/10.3233/FAIA200872>, <https://doi.org/10.3233/FAIA200872>
- [79] Mohajeri Parizi, M., Sileno, G., van Engers, T.: Seamless integration and testing for mas engineering. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 254–272. Springer International Publishing, Cham (2022). [https://doi.org/10.1007/978-3-030-97457-2\\_15](https://doi.org/10.1007/978-3-030-97457-2_15)
- [80] Mohajeri Parizi, M., Sileno, G., van Engers, T., Klous, S.: Run, agent, run! architecture and benchmarking of actor-based agents. In: Proceedings of the 10th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control. p. 11–20. AGERE 2020, Association for Computing Machinery, New York, NY, USA (2020). <https://doi.org/10.1145/3427760.3428339>
- [81] Müller, T., Esterhuyse, C.A., van Binsbergen, L.T.: Source code and experimental data for a translatfrom eflint normative specifications to clingo answer-set programs (2025). <https://doi.org/10.5281/zenodo.15188960>, <https://doi.org/10.5281/zenodo.15188959>, this URL and DOI refer to the most recent version of the artefact, from where the URL and DOI of each (fixed) version is accessible. Version1 has DOI 10.5281/zenodo.15188959, and version2 has DOI 10.5281/zenodo.15470286. The latter is the most recent at time of writing and is reflected by both [44] and this article.
- [82] Nute, D.: Defeasible logic. In: Bartenstein, O., Geske, U., Hannebauer, M., Yoshie, O. (eds.) Web Knowledge Management and Decision Support. pp. 151–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2003). <https://doi.org/10.1007/3-540-36524-9>
- [83] OASIS eXtensible Access Control Markup Language (XACML) Technical Committee: eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01 (July 2017)
- [84] Osborn, S.L.: Mandatory access control and role-based access control revisited. In: Youman, C.E., Coyne, E.J., Jaeger, T. (eds.) Proceedings of the Second Workshop on Role-Based Access Control, RBAC 1997, November 6–7, 1997. pp. 31–40. ACM (1997). <https://doi.org/10.1145/266741.266751>
- [85] Padget, J., Elakehal, E., Li, T., De Vos, M.: InstAL: An Institutional Action Language, Law, Governance and Technology Series, vol. 30, p. 101. Springer Verlag (2016). [https://doi.org/10.1007/978-3-319-33570-4\\_6](https://doi.org/10.1007/978-3-319-33570-4_6)
- [86] Pandžić, S., Broersen, J., Aarts, H.: BOID\*: Autonomous goal deliberation through abduction. p. 1019–1027. AAMAS ’22, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2022)
- [87] Parizi, M.M., van Binsbergen, L.T., Sileno, G., van Engers, T.: A modular architecture for integrating normative advisors in mas. In: Baumeister, D., Rothe, J. (eds.) Multi-Agent Systems. pp. 312–329. Springer International Publishing, Cham (2022)
- [88] Park, J., Sandhu, R.: Towards usage control models: beyond traditional access control. In: Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies. p. 57–64. SACMAT ’02, Association for Computing Machinery (2002). <https://doi.org/10.1145/507711.507722>
- [89] Parvizimosaed, A., Roveri, M., Rasti, A., Anda, A.A., Alfuhaid, S., Amyot, D., Logrippo, L., Mylopoulos, J.: SymboleoPC: checking properties of legal contracts. Softw. Syst. Model. **24**(4), 1093–1126 (2025). <https://doi.org/10.1007/S10270-024-01180-2>, <https://doi.org/10.1007/s10270-024-01180-2>
- [90] Prinz, W., Rose, T., Urbach, N.: Blockchain Technology and International Data Spaces, pp. 165–180. Springer International Publishing, Cham (2022). [https://doi.org/10.1007/978-3-030-93975-5\\_10](https://doi.org/10.1007/978-3-030-93975-5_10)
- [91] Ramezani, E., Fahland, D., van der Aalst, W.M.P.: Where did i misbehave? diagnostic information in compliance checking. In: Barros, A., Gal, A., Kindler, E. (eds.) Business Process Management. pp. 262–278. Springer Berlin Heidelberg, Berlin, Heidelberg (2012). [https://doi.org/10.1007/978-3-642-32885-5\\_21](https://doi.org/10.1007/978-3-642-32885-5_21)
- [92] Rasti, A., Anda, A.A., Alfuhaid, S., Parvizimosaed, A., Amyot, D., Roveri, M., Logrippo, L., Mylopoulos, J.: Automated generation of smart contract code from legal contract specifications with Symboleo2SC. Softw. Syst. Model. **24**(4), 1127–1156 (2025). <https://doi.org/10.1007/S10270-024-01187-9>
- [93] Rodríguez-Doncel, V., Villata, S., Gómez-Pérez, A.: A dataset of RDF licenses. In: Hoekstra, R. (ed.) Legal Knowledge and Information Systems - JURIX 2014: The Twenty-Seventh Annual Conference, Jagiellonian University, Krakow, Poland, 10–12 December 2014. Frontiers in Artificial Intelligence and Applications, vol. 271, pp. 187–188. IOS Press (2014). <https://doi.org/10.3233/978-1-61499-468-8-187>
- [94] Ross, K.A.: Modular stratification and magic sets for DATALOG programs with negation.

- In: Rosenkrantz, D.J., Sagiv, Y. (eds.) Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA. pp. 161–171. ACM Press (1990). <https://doi.org/10.1145/298514.298558>
- [95] Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specifications. In: Stuckey, P. (ed.) Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings. LNCS, vol. 2401, pp. 22–37. Springer (2002). [https://doi.org/10.1007/3-540-45619-8\\_3](https://doi.org/10.1007/3-540-45619-8_3)
- [96] Sadri, F., Kowalski, R.: Variants of the event calculus. In: Sterling, L. (ed.) Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995. pp. 67–81. MIT Press (1995)
- [97] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29**(2), 38–47 (1996). <https://doi.org/10.1109/2.485845>
- [98] Sandhu, R.S., Munawer, Q.: How to do discretionary access control using roles. In: Youman, C.E., Jaeger, T. (eds.) Proceedings of the Third ACM Workshop on Role-Based Access Control, RBAC 1998, October 22-23, 1998. pp. 47–54. ACM (1998). <https://doi.org/10.1145/286884.286893>
- [99] Schmidt, D.: Guest editor’s introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006). <https://doi.org/10.1109/MC.2006.58>
- [100] Schrans, F., Hails, D., Harkness, A., Drossopoulou, S., Eisenbach, S.: Flint for safer smart contracts. <https://arxiv.org/pdf/1904.06534.pdf> (2019)
- [101] Searle, J.: The construction of social reality. Penguin Books (1996)
- [102] Seijas, P., Nemish, A., Smith, D., Thompson, S.: Marlowe: implementing and analysing financial contracts on blockchain. In: Workshop on Trusted Smart Contracts (Financial Cryptography 2020) (2 2020). [https://doi.org/10.1007/978-3-030-54455-3\\_35](https://doi.org/10.1007/978-3-030-54455-3_35)
- [103] Shakeri, S., Maccatrazzo, V., Veen, L., Bakhshi, R., Gommans, L., de Laat, C., Grosso, P.: Modeling and matching digital data marketplace policies. In: 2019 15th International Conference on eScience (eScience). pp. 570–577 (2019). <https://doi.org/10.1109/eScience.2019.00078>
- [104] Sharifi, S., Parvizimosaed, A., Amyot, D., Logrippo, L., Mylopoulos, J.: Symboleo: Towards a Specification Language for Legal Contracts. In: Breaux, T.D., Zisman, A., Fricker, S., Glinz, M. (eds.) 28th IEEE International Requirements Engineering Conference, RE 2020, August 31 - September 4, 2020. pp. 364–369. IEEE (2020). <https://doi.org/10.1109/RE48521.2020.00049>
- [105] Steketee, M., Verheijen, N., van Binsbergen, L.T.: Managing Administrative Law Cases using an Adaptable Model-driven Norm-enforcing Tool. Proceedings of AI4Access2Justice at JURIX’24 (2024)
- [106] Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). <https://doi.org/10.5210/fm.v2i9.548>
- [107] Taghiabadi, E.R., Gromov, V., Fahland, D., van der Aalst, W.P.: Compliance checking of data-aware and resource-aware compliance requirements. In: Meersman, R., Panetto, H., Dillon, T., Misikoff, M., Liu, L., Pastor, O., Cuzzocrea, A., Sellis, T. (eds.) On the Move to Meaningful Internet Systems: OTM 2014 Conferences. pp. 237–257. Springer Berlin Heidelberg, Berlin, Heidelberg (2014). [https://doi.org/10.1007/978-3-662-45563-0\\_14](https://doi.org/10.1007/978-3-662-45563-0_14)
- [108] Tufis, M., Ganascia, J.G.: Grafting norms onto the BDI agent model. A Construction Manual for Robots’ Ethical Systems. Cognitive Technologies (2015)
- [109] Tuler De Oliveira, M., Reis, L.H.A., Verginadis, Y., Mattos, D.M.F., Olabarriaga, S.D.: SmartAccess: Attribute-Based Access Control System for Medical Records Based on Smart Contracts. IEEE Access **10**, 117836–117854 (2022). <https://doi.org/10.1109/ACCESS.2022.3217201>
- [110] van der Aalst, W.: Process mining: discovery, conformance and enhancement of business processes. Springer, Germany (2011). <https://doi.org/10.1007/978-3-642-19345-3>
- [111] Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM **38**(3), 619–649 (Jul 1991). <https://doi.org/10.1145/116825.116838>
- [112] Viganò, F., Colombetti, M.: Symbolic model checking of institutions. pp. 35–44. ACM Press. <https://doi.org/10.1145/1282100.1282109>
- [113] Vos, M.D., Kirrane, S., Padget, J.A., Satoh, K.: ODRL Policy Modelling and Compliance Checking. In: Fodor, P., Montali, M., Calvanese, D., Roman, D. (eds.) Rules and Reasoning - Third International Joint Conference, RuleML+RR 2019, Bolzano,

- Italy, September 16-19, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11784, pp. 36–51. Springer (2019). [https://doi.org/10.1007/978-3-030-31095-0\\_3](https://doi.org/10.1007/978-3-030-31095-0_3)
- [114] Wood, D.: Ethereum: a secure decentralised generalised transaction ledger (2014)
- [115] Wöhrer, M., Zdun, U.: Domain specific language for smart contract development. In: 2020 IEEE International Conference on Blockchain and Cryptocurrency (ICBC). pp. 1–9. IEEE (2020). <https://doi.org/10.1109/ICBC48266.2020.9169399>
- [116] Ågotnes, T., van der Hoek, W., Rodríguez-Aguilar, J.A., Sierra, C., Wooldridge, M.: A Temporal Logic of Normative Systems. In: Makinson, D., Malinowski, J., Wansing, H. (eds.) Towards Mathematical Philosophy: Papers from the Studia Logica Conference Trends in Logic IV, pp. 69–106. Springer Netherlands. [https://doi.org/10.1007/978-1-4020-9084-4\\_5](https://doi.org/10.1007/978-1-4020-9084-4_5)