Reflections on the design and application of the normative specification language eFLINT

L. Thomas van Binsbergen[1][0000−0001−8113−2221]
ltvanbinsbergen@acm.org

Informatics Institute, University of 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 provided by automating 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 of formalising interpretations relying on legal and software expertise.
This paper reflects on the domain-specific language eFLINT developed to experiment with potential solutions based on concepts as modularity and inheritance. 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 (possibly conflicting) requirements. This paper reflects on the current design of the language by recalling various applications and sketches future developments. As such, this paper reports on intermediate results and insights of an ongoing 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. 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 system components to enhance adaptability and transparency. For example, the XACML [48] and ODRL [36] 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. Smart contracts [63, 27] are another example of automating compliance with, in particular, legal contracts. However, both examples are limited from a legal standpoint [66, 15, 41, 44].

A general solution to automating compliance requires, firstly, a formal representation of norms that is sufficiently general to capture norms from a variety of sources, at different levels of abstractions, and with connections between sources made explicit. For example, the GDPR privacy regulation is more abstract than an organisational policy or data sharing agreement affected by it. 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 norms, property checking to gain confidence in the correctness of a norm specification, forward reasoning to determine the effects of actions and the compliance of individual scenarios, and backward reasoning for policy-aware planning. Thirdly, the solution should embrace the fact that legal processes are inherently subjective, e.g., the process for the interpretation (of norms) and the qualification of situations. This observation makes a fully *ex-ante* approach, such as compliance by design, untenable from a legal perspective (although practical in many situations). Instead, solutions are needed in which (potential) violations within the system are embraced with *ex-post* enforcement mechanisms, such as penalties and induced risk, as a backstop to respond to non-compliance. An overview of the challenges related to compliance is given by Hashmi et al. [30]

This paper reports on intermediate findings of an investigation into a general solution to automating compliance based on important software (language) engineering concepts such as modularity and inheritance. This investigation involves the design an implementation of eFLINT, a domain-specific language (DSL) for developing executable specifications of norms [8]. The languages has been applied in various experiments involving the assessment of concrete (historical) cases [8], bounded model checking, run-time compliance with ex-post enforcement, normative multi-agent systems, and generating access control policies from social policies [7], 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 requirements extracted from the goals and applications of the language. This paper reflects on the design choices most relevant to other researchers of (declarative) norm specification languages, normative reasoning, and automating compliance, and suggests future directions.

2 Related work

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 [43, 56, 55, 14] such as Symboleo [62] and InstAL [50]. The Institutional Action Language (InstAL) is a DSL for specifying norms in terms of duties and powers translating to Answer Set Programming for execution. Symboleo and eFLINT are both based on Hohfeld's legal framework [32] with Symboleo being designed specifically for contracts with embedded notions of contract state. LegalRuleML extends RuleML and specifies norms in terms of permissions and obligations through rules, and unlike eFLINT supports defeasibility and negation of facts [3]. Other formal languages for expressing norms are based on deontic logic [31], action logic [39] and defeasible logic [47, 28].

System policies and smart contracts A significant body of work exists concerning the formalisation, analysis and enforcement of *specific* kinds of norms [37] such as policies for access control [48, 36], network policies [2] (e.g. firewall configurations) and (smart) contracts [62, 60, 27, 59, 68]. Instead, eFLINT is designed for describing a wide variety of normative sources such as laws, regulations, (organisational) policies and contracts. The Margrave Policy Analyzer tool[1] supports multiple formalisms to reason about access control policies and firewall configurations with different analyses such as Change-Impact Analysis [23]. A large number of access control models exist [58, 49] of which the most common are Role-Based Access Control (RBAC) [57] and Attribute-Based Access Control (ABAC) [35]. XACML is a popular ABAC language and the XACML architecture [48] is used as a model for systems applying ABAC. The knowledge representation of eFLINT is sufficiently expressive to capture roles and attributes and the eFLINT reasoner can be used as a policy decision and administration point within the XACML architecture [7]. ODRL is an access control language with policies controlling specific actions on (multi-media) assets [36, 54, 66]. The Usage CONtrol (UCON) model introduces the ability to specify conditions that should hold *during* access events [52].

First introduced by Szabo [63] 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). Blockchain applications can support compliance efforts by providing auditability and transparency [53, 61]. Solidity [22] is a popular imperative smart contract language for the Ethereum platform [12, 67]. Marlowe is a declarative DSL for writing smart contracts at a higher level of abstraction for the Cardano platform [60].

Multi-agent systems Multi-agent systems (MASs) are useful 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 [46, 45] 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 [18, 64, 17], such as the B-DOING framework [19] and the BOID architecture [11, 51]. Most similar to our approach is the use of 'ethical governors' in [13] that can be queried for advise. However, as explained later in this paper, with our approach we can also support active notifications sent by the eFLINT reasoner to support forms of ex-post enforcement. A thorough discussion in integrating norms in MAS is provided by [5].

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 ex-

[1] http://www.margrave-tool.org/

ample, Normative Temporal Logic (NTL) is a temporal logic that replaces the path quantifiers of Computation-Tree Logic (CTL) to express obligations and permissions [69]. Temporal Defeasible Logic (TDL) combines defeasibility and temporal logic [29]. The FIEVeL specification language is used to model institutional policies [65] based on *Ordered Many-Sorted First-Order Temporal Logic (OMSFOTL)* using SPIN [34] for model checking. In [4], the connection between coordination problems and norm enforcement is formalised using Linear Temporal Logic (LTL). The norm specification language Revani [40] uses CTL for the specification of properties concerning privacy in particular.

3 Running Example

This section introduces the eFLINT language through an example program related to auctioning. A reference interpreter for the language is available online [6].

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 code fragment introduces the 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 every object to a unique price.

```
Fact     bidder         Identified by String  // actor placing bids
Fact     object         Identified by String  // objects for sale
Fact     price          Identified by Int     // price to pay
Var      display        Identified by object  // the item on display
Fact     bid            Identified by bidder * object * price * int
Function min-price-of   Identified by object * price
```

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 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 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 evaluated in a context in which the fields of a type are bound as variables, e.g. `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 of any other elements of `display` owing to its status as a `Var`.

```
Act start-bidding Related to object Holds when auctioneer(actor)
  Creates display(object)
```

State transitions occur through the execution of actions and assertions[2]. On the contrary, derivation rules are declarative in that they do not trigger transitions.

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` (explained in section 5). The `Conditioned by` clause establishes (extra) conditions[3] 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. The `int` field of `bid` distinguishes bids from the same bidder.

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)
```

A duty-type declaration defines a fact-type with mandatory fields for a duty-holder and a duty-claimant (and below, the additional field `price`) and zero or more violation conditions. A duty raises a violation when it holds true and when one or more violation conditions hold.

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

The treatment of different kinds of legal obligations with duties and open-texture terms such as 'undue delay' are discussed in later sections. In the example, duties to pay and to deliver are created when the bidding on a particular object ends.

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

[2] And also events, differing from actions in that they are not performed by actors.

[3] `Holds when` clauses are disjuntive, `Conditioned by` clauses are conjunctive.

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 `Foreach`. As a result, all instances of `bid` are terminated for which hold that the object referred to in the bid is on display. This is realised by the application of `When` (infix).

At the end of the following scenario, Bob has a duty to pay 140 for the watch.

```
+bidder(Alice). +bidder(Bob). +bidder(Chloe). +auctioneer(David).
start-bidding(David, Watch). // statement executing an action
raise-hand(Alice). raise-hand(Bob). raise-hand(Alice). raise-hand(Chloe).
raise-hand(Bob). end-bidding(David).
```

4 Application

The design of eFLINT has been motivated to assess concrete scenarios for compliance with norms specified in the FLINT language of Van Doesburg [21, 20] (eFLINT abbreviates "executable FLINT") and to achieve this by establishing a connection between Hohfeld's normative concepts [33, 32] and computational concepts. 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. This section describes different application areas in which experiments with eFLINT have been performed and introduces requirements (in **bold**) identified throughout these experiments.

Supporting Formal Interpretation The Calculemus-FLINT framework [21] 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**). The method prescribes to recognise acts, duties, actors and facts based on sentence structure and to fill act-, duty-, and fact-*frames* with text fragments extracted directly from the sentences. Case assement 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 thus amounts to **forward chaining**.

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 underspecified [27]. 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 [10]. 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 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.

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**. The design of FLINT is action-oriented in its focus on acts and 'obligations to act' (duties). Deontic frameworks are obligation-oriented and consider also '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 have different origins such as different sources of norms or different actors invoking powers to modify the normative positions of (other) actors. In these situations, priority mechanisms are required (**norm priorities**) to determine whether a violation has indeed occurred or **ex-post enforcement** mechanisms are needed to respond retroactively to violations.

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 automatic compliance is based on **explicit qualification**.

Assessing *hypothetical* cases is useful to experiment with the consequences of a set of norms as part of, for example, the drafting of policy (**experimentation**). By specifiying large sets of cases, confidence about the inner workings of the policy is obtained, akin to running test-suites for testing software (**testing**).

Multi-Agent Systems We use multi-agent systems to experiment with compliance in regulated systems, data markets in particular. So-called 'normative advisors' are added that reason with norms based on observations communicated to them by their parent or by the environment (**event-based**). Conclusions, for example about permitted actions or violations, are communicated to the parent. Our approach makes it possible to experiment with different models for distributing institutional facts and institutional reasoning and requires multiple eFLINT reasonsers to co-exist (**instantiation**). One can define a system with a single agent monitoring and assessing the behaviour of all market participants. Alternatively, every market participant has its own normative advisor and uses normative reasoning for planning. Such planning activities by agents require the **simulated** execution of hypothetical scenarios or **backward chaining** to reason towards a goal, the latter of which our tools do not yet support.

We define **ex-post enforcement** agents that decide whether 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, for example, which facts prevented an action from being enabled (**explainability**). Similar reports are needed for **ex-ante enforcement** in order for agents to determine what they can do to enable disabled actions.

Bounded Model-Checking The applications described in the previous subsection are about *concrete* (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, effectively reasoning about *abstract* scenarios. 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. This applications also requires the ‘**closed world** assumption’ that establishes that what is not known to be true is considered false. Furthermore, to make bounded model-checking feasible, we concluded it must be possible to reduce the amount of possible transitions and to reduce the set of possible runtime states (**reducible domains**). Future work is to investigate the applicability of combinatorial, property-based testing [25].

Automating Legal Compliance The eFLINT language has been used in experiments to investigate automating compliance in case management systems used by governmental organisations. Prototypes are designed to adapt to changes in norms, including **dynamic updates**, to formalise connections between organisational policy and the laws they implement, and to enforce the rights and duties of both civil servant and citizen. The eFLINT reasoner is instantiated as a running process for every on-going case, or responds to individual requests. As cases develop over time, information about the case is recorded, either internally or externally to the reasoner (**persistence**). This 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 norms change.

These experiments have also reveiled 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 physical actions (such as `raise-hand` and not `place-bid`) are triggerable by users of the application. Another aspect of the service interface is to determine which facts are provided solely by the application. 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**).

In other projects, we are experimenting with the automatic enforcement of (privacy) regulations, consortium agreements and data sharing conditions within data exchange systems. In this context, **transparency**, **explainability**, mechanisms for dispute resolution and **auditability** are of high importance [53, 61]. Data exchange systems are inter-domain applications that require mechanisms

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

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

for cross-domain authentication [26] and consensus. Consensus is needed on the applied norms, relevant policy information, on the transpired events and on the decisions made by the normative reasoner. A project to build an eFLINT to Solidity [22, 67] compiler is ongoing, producing smart contracts serving as reasoners in a blockchain application. The interpretation is recorded as a smart contract and the assessed scenario is recorded as transactions on the ledger.

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.

Knowledge Representation Following well-known declarative languages such as Prolog [16] and Alloy [38], eFLINT allows users to define sets and relations inductively over atomic strings and integers. The expressivity of the representation can be demonstrated by showing how relational algebra or database operations are supported. Relations associate names with fields as attributes in relational algebra and databases. Relations thus represent database tables, predicates in Prolog, or triples in RDF [42]. A projection operator `<EXPR>.<NAME>` is available to refer to components of a tuple. Derivation rules can define unions and joins by applying `Exists`, `Forall` and `Foreach` operators to enumerate over elements of relations. A similar comparison can be made with algebraic datatypes.

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 perform one of its actions that create/terminate actions or duties (**normative concepts**).

As described in the previous section, 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 conjuction with aggregators such as `Count` and `Sum`. 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, whereas of an infinite type, instances that hold in the current knowledge base are enumerated. In the following fragment, the value of `count-all` differs depending on whether the first or second definition of `numbers` is used, whereas the value of `count` does not:

```
Fact number Identified by 1..5. Fact number Identified by Int.
+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)).
```

In both cases, enumeration is finite as an eFLINT knowledge base is finite. Knowledge in eFLINT is concrete rather than symbolic and *unification* is not used. The statement `+rich(person)` in the following code fragment determines that every *currently* known person is rich rather than every conceivable person. As a consequece, 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”. 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.

The semantics of enumeration also effects 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:

```
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))
```

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 for derivation rules, 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 a derivation rule of a type is evaluated (such that `object` is not enumerated). The alternative semantics violates the previous design choice that knowledge bases are concrete and finite as `start-bidding(David,object)` would hold true for all conceivable objects.

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 = ...)` (Section 3), in which arguments for the fields `bidder`, `object` and `price` of the type `bid` are missing, is

Act frame	«assisting with the contracting of a valid marriage»
Actor	[ordinary or priest ...]
Object	[marriage attempt]
Interested Party	[spouses]
...	...
Creating postcondition	[valid marriage]; ...
Terminates postcondition	[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 of priest ...]
  Recipient   [spouses]
  Related to  [marriage attempt]
  Creates     [valid marriage]()
  Terminates  [marriage attempt]()
```

Fig. 1. FLINT and eFLINT specification of an act. Simplified example taken from [20].

interpreted as `bid(int=..., bidder=bidder, object=object, price=price)`. The context determines whether these variables are bound by an (implicit) `Exists` or `Foreach`.

Implicit arguments simplify the translation from FLINT to eFLINT (**interpretation**). Consider the FLINT act-frame on the left-hand side of Figure 1 and the corresponding eFLINT code on the right-hand side. The FLINT frame can be generated from a syntactical 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, which strings represent a pair of spouses, and with Booleans[4] capturing whether there is a valid marriage or marriage attempt. The resulting 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]` but only one valid marriage can be represented, and without a formal connection to the spouses. 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, the relation `[valid marriage]` (and similar for `[marriage attempt]`) can be given a field `[spouses]` such that the fact can be used 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 the expression is evaluated when an action is performed, the variable `[spouses]` is bound to the recipient of the action.

The decision which of the two solutions to apply is argueably 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

[4] 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.

Figure 1 can be extended (using the `#require` directive) by two separate files, each implementing one of the solutions. In fact, both solutions can be applied for both purposes, owing to the way eFLINT services are implemented (**instantiation**).

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 [8], 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 **ex-post enforcement**.

The language was redesigned according to the methodology presented in [9] and implemented using the generic interpreter back-end of [24] that supports exploratory programming by keeping track of execution history. This 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, the `Syncs with` keyword was added for synchronising actions and events, and *instance queries* were added. A Boolean query `?<EXPR>` evaluates an expression to true or false. An instance expression `?-<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 bids on the object currently on display. All unbound variables in an instance query (`bid` in the example) are implicitly bound using `Foreach`. Queries are useful for **testing** and for **ex-ante enforcement**. 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 [48]. 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.

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 clauses referred to as *accumulating* clauses. These clauses are accumulating in the sense that multiple occurrences of the same kind of clause can occur at a type and in that their effects somehow accumulate. For example, an instance is derived if one or more of the derivation rules of the type derives the instance. A duty is violated if (it holds true and) one or more of its violation conditions holds true. All pre-conditions of a type[5] 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 below) are created/terminated/obfuscated.

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. 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). The `Syncs with` keyword was introduced to connect the institutional actions from various interrelated normative documents, such as a sharing agreement and the GDPR regulation example in [7]. This way, normative sources can be interpreted separately at their own **abstraction level** with additional, separate code fragments making any connections explicit.

Experience demonstrated that it is also useful to use eFLINT actions and events to model physical processes and to connect them to institutional processes using `Syncs with`. This way, qualification rules can be written within eFLINT itself and integrating eFLINT as a service becomes easier. A generic algorithm can systematically convert the events of an arbitrary event stream to phrases without making assumptions about the specification with which the reasoner is loaded. Institutional meaning is then assigned to these events within eFLINT (when relevant). Physical and synchronised actions motivated the change that actions always manifest their effects when executed, even when disabled.

For security reasons and to separate concerns, it is beneficial to restrict an eFLINT service to only respond to certain inputs and to make certain types strictly internal. This way, for example, we can disallow any duties from being created or terminated directly. For the running example we might like to say that only the `raise-hand`, `start-bidding` and `end-bidding` actions can be triggered. In future work we intend to add a module-system or **service interface** specification mechanism that makes it possible to hide and expose elements of a specification.

At present, eFLINT does offer 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 be pre-conditions expressed using `Conditioned by` or inherited by synchronising with instances of (other) act-types. The distinction between physical and institutional action strenghtens model checking. By expressing LTL properties that only transition over physical actions (synchronised with institutional actions), we can use model checking properties to check the *conformance* [1] of a physical process with an institutional process.

Open Types In a type declaration, the `Open` and `Closed` modifiers are available to indicate whether the **closed world** assumption holds for this type. The logic for open types is three-valued: True, False or Unknown. The `Obfuscates` clause has been added to set the truth value of an instance to Unknown. The treatment of

[5] `Conditioned by` clauses can be associated with all types, not just act-types.

Unknown affects how applications interact with eFLINT services. The eFLINT reasoner undoes the execution of a phrase when Unknown is encountered and reports the Unknown instance as an exception. Such an exception can be understood as a request to the execution context to provide additional input. The eFLINT implementation accepts additional input alongside a phrase to be executed. Additional input is only used for that particular execution request and has the highest priority[6] when establishing the truth of a fact.

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. 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 a continuation passing style). 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 is used to generate an HTML form with various kinds of elements (e.g. checkboxes and dropdowns) with any available information pre-filled. The additional input component of an execution request can also be used to provide *all* input relevant to requests. This way, the eFLINT reasoner can be used statelessly and **persistence** can be realised with an external database.

6 Discussion

This section reflects on the current design of eFLINT compared to the goals and requirements stated in Section 4.

The development of eFLINT went through various phases, starting with the analyses of historical cases, followed by dynamically evolving cases and system integration. 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 is needed for model checking. In general, sometimes certain assumptions are needed that in a more restrictive variant of the language could be enforced synactically 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.

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 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. Automation is needed to deal with the large amount

[6] 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.

of normative sources and the pace with which they are produced. Intervention by legal experts is needed as interpretation is inherently subjective. Software experts are needed to add computational details (such as relations between fields) not commonly contained in the natural language sources. Provenance, **versioning** and retaining **source references** are important qualities of an interpretation pipeline. For these purposes, meta-data about eFLINT fragments should be maintained by the pipeline as is currently being done for FLINT frames.

In the extended version of this paper, we intend to present a new design of a norm specification language which comprises a computational core language and a legal surface-level language. The goal of this (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 (left-hand side) can be used to express the relation between `raise-hand` and `place-bid` without `Extend` and `Syncs with` (right-hand side):

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

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

Another motivation is to experiment with the computational meaning of alternative **normative concepts** such as obligations and prohibitions. The notion of violation will be removed from the core language and replaced by monitors that flag properties satisfied by triggered transitions or reached states. 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.

7 Conclusion

This paper has reflected on the design of eFLINT based on requirements extracted from experiments in various application areas. The presented discussion serves to benefit researchers working on computational representations of legal concepts, attempting to bridge the gap between legal artefacts and software, automating compliance, and policy-driven systems. The main advantage of the language is the flexibility that permits it to be used in multiple application areas simultaneously such as model checking, checking the compliance of historical, hypothetical and dynamically evolving scenarios, as part of an access control system and to formalise norms from a variety of sources at their own level of abstraction. The flexibility is achieved through certain pragmatic design decisions and by designing the language to be modular and specifications extensible. 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 in the Big Data: Real Time ICT for Logistics program (628.009.014) and the AMdEX Fieldlab project supported by Kansen Voor West EFRO (KVW00309) and the province of Noord-Holland.

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
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. 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
4. 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
5. Balke, T., da Costa Pereira, C., Dignum, F., Lorini, E., Rotolo, A., Vasconcelos, W., Villata, S.: Norms in MAS: Definitions and Related Concepts. In: Andrighetto, 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
6. van Binsbergen, L.T.: The eFLINT project on GitLab. https://gitlab.com/eflint (2020), [Online, accessed 12 October 2022]
7. van Binsbergen, L.T., Kebede, M.G., Baugh, J., van Engers, T., van Vuurden, 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
8. 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
9. 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
10. Breebaart, M.: Juriconnect standaard BWB versie 1.3.1 (Oct 2014)
11. 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
12. Buterin, V.: Ethereum white paper (2018)
13. 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

14. Charalambides, M., Flegkas, P., Pavlou, G., Bandara, A.K., Lupu, E., Russo, A., Dulay, N., Sloman, 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
15. 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). https://doi.org/10.5555/2372366.2372381
16. 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
17. Criado, N., Argente, E., Noriega, P., Botti, V.: Towards a normative BDI architecture for norm compliance. CEUR Workshop Proceedings **627**, 65–81 (2010)
18. 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)
19. 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
20. 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
21. 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)
22. Ethereum: Solidity documentation online. https://solidity.readthedocs.io (2016), [Online, accessed 8 October 2022]
23. 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
24. 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
25. 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
26. 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
27. 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

28. 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
29. 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
30. 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
31. 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
32. 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
33. Hohfeld, W.: Some fundamental legal conceptions as applied in judicial reasoning. Yale Law Journal **23(1)**, 59–64 (1913)
34. Holzmann, G.J.: The SPIN Model Checker - primer and reference manual. Addison-Wesley (2004)
35. 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
36. Iannella, R., Villata, S.: ODRL information model 2.2. W3C Recommendation (2018)
37. 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
38. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006)
39. 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
40. 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
41. 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
42. 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/
43. 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
44. Liu, L., Ho, M., Su, B., Wang, S., Hsu, M., Tseng, Y.J.: PanGPCR: predictions for multiple targets, repurposing and side effects. Bioinform. **37**(8), 1184–1186 (2021). https://doi.org/10.1093/bioinformatics/btaa766
45. 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

46. 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
47. 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
48. OASIS eXtensible Access Control Markup Language (XACML) Technical Committee: eXtensible Access Control Markup Language (XACML) Version 3.0 Plus Errata 01 (July 2017)
49. 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
50. 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
51. 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)
52. 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
53. 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
54. 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
55. 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
56. 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)
57. 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
58. 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
59. Schrans, F., Hails, D., Harkness, A., Drossopoulou, S., Eisenbach, S.: Flint for safer smart contracts. https://arxiv.org/pdf/1904.06534.pdf (2019)

60. 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
61. Shakeri, S., Maccatrozzo, 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
62. 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
63. Szabo, N.: Formalizing and securing relationships on public networks. First Monday **2**(9) (1997). https://doi.org/10.5210/fm.v2i9.548
64. Tufis, M., Ganascia, J.G.: Grafting norms onto the BDI agent model. A Construction Manual for Robots' Ethical Systems. Cognitive Technologies (2015)
65. Viganò, F., Colombetti, M.: Symbolic model checking of institutions. pp. 35–44. ACM Press. https://doi.org/10.1145/1282100.1282109
66. 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
67. Wood, D.: Ethereum: a secure decentralised generalised transaction ledger (2014)
68. 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
69. 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