eFLINT: a Domain-Specific Language for Executable Norm Specifications

L. Thomas van Binsbergen
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
ltvanbinsbergen@acm.org

Lu-Chi Liu
University of Amsterdam
Amsterdam, The Netherlands
l.liu@uva.nl

Robert van Doesburg
Leibniz Institute, University of Amsterdam / TNO
Amsterdam, The Netherlands
robertvandoesburg@uva.nl

Tom van Engers
Leibniz Institute, University of Amsterdam / TNO
Amsterdam, The Netherlands
vanengers@uva.nl

ABSTRACT

Software systems that share potentially sensitive data are subjected to laws, regulations, policies and/or contracts. The monitoring, control and enforcement processes applied to these systems are currently to a large extent manual, which we rather automate by embedding the processes as dedicated and adaptable software services in order to improve efficiency and effectiveness. This approach requires such *regulatory services* to be closely aligned with a formal description of the relevant norms.

This paper presents eFLINT, a domain-specific language developed for formalizing norms. The theoretical foundations of the language are found in transition systems and in Hohfeld's framework of legal fundamental conceptions. The language can be used to formalize norms from a large variety of sources. The resulting specifications are executable and support several forms of reasoning such as automatic case assessment, manual exploration and simulation. Moreover, the specifications can be used to develop regulatory services for several types of monitoring, control and enforcement. The language is explained through an example, formalizing articles 6(1)(a) and 16 of the General Data Protection Regulation (GDPR). A prototype implementation and formal definition of eFLINT are provided as supplementary material.

CCS CONCEPTS

• **Computing methodologies** → **Model development and analysis**; • **Security and privacy** → *Information flow control.*

KEYWORDS

normative modeling, domain-specific language, policy enforcement, GDPR, executable specifications

ACM Reference Format:
L. Thomas van Binsbergen, Lu-Chi Liu, Robert van Doesburg, and Tom van Engers. 2020. eFLINT: a Domain-Specific Language for Executable Norm Specifications. In *Proceedings of GPCE.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 MOTIVATION

Governmental institutions provide services to citizens and companies that are primarily defined in laws and regulations. However, in practice there is often no clear connection between the software systems that support or provide these services and the laws and regulations that govern them. Similarly, business processes are subjected to laws and (inter)national regulations as well as internal policies, branch-wide codes and contracts. In both government and business, a direct connection between a software's implementation and the norms that govern the software's operations is highly desirable. A direct connection makes the software easier to validate and increases the software's maintainability with respect to following changes in regulations and policies. Moreover, with a direct connection it is possible to explain the actions taken within a software system to stakeholders in terms of the relevant norms. Our approach is to automate the required monitoring, control and enforcement processes, that currently are predominantly manual processes, as dedicated and adaptable regulatory services. To improve trust, the regulatory services are based on formal specifications of the relevant norms that can be verified in isolation.

GPCE, Generatic Programming: Concepts & Experiences, 2020
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

This paper presents eFLINT, a domain-specific language (DSL) for formalizing norms as executable specifications. Compared to existing languages, eFLINT is novel in several respects and is most similar to languages based on the event calculus such as Symboleo [28] and InstAL [20]. A significant body of work exists concerning the formalization, analysis and enforcement of specific kinds of norms [14] such as policies for access control [29], network policies [1] (e.g. firewall configurations) and contracts [27, 28]. Instead, eFLINT is designed for describing a wide variety of normative sources such as laws, regulations, policies and contracts. Other formal languages for expressing norms are based on deontic logics [12], action logic [15], defeasible logic [10, 18]. Some of these languages are not suited to capture some important aspects of norms such as the actors bound by the norms and the activities regulated by these norms. An important aspect of eFLINT is that the language is action-based and that the normative positions of actors are derived from the actions they can perform (permissions) or are expected to perform (duties) at a given moment in time. Moreover, the language supports the legal concept of power – the ability to grant (or remove) permissions or duties of (other) actors. The benefit of the action-based approach is that checking the compliance of a scenario or software implementation is simplified because scenarios and software implementations are inherently action-based. Together, these features enable eFLINT for various types of applications requiring online or offline compliance-checking, monitoring, traceability and explainability.

This paper contributes by presenting eFLINT, discussing its use in a variety of applications, reflecting on its design and placing it in a wider context. The language is introduced through an example in Section 3. In section 4 we explain how eFLINT is used for offline and online compliance checking. Section 5 formalizes the parts of articles 6 and 16 of the GDPR (General Data Protection Regulation) that relate to 'consent' and the 'right to rectification' as a case study. After a reflection on the features and design of the language, the language is compared to relevant alternatives in Section 6.

2 LEGAL FOUNDATIONS

In this section we summarize the normative theory that underpins eFLINT. The theory is explained in reference to legal case analysis, involving the processes of *interpretation*, *qualification* and *assessment*, visualized in Figure 1. The diagram distinguishes between physical reality (left-hand side) and the institutional reality of Searle's social theory [26] (right-hand side) as the reality in which actors interact physically with objects and each other on the one hand, and certain abstractions over that physical reality on the other hand.

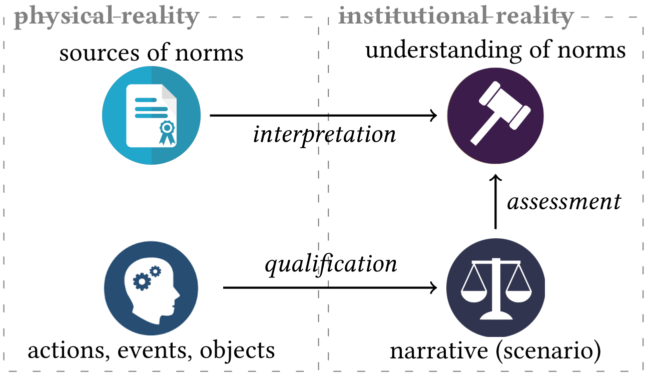


Figure 1: Schematic overview of the processes of interpretation, qualification and assessment.

The institutional abstractions are twofold: a general understanding of the norms relevant to a case (top right) and the understanding of the case itself as a narrative (bottom right, henceforth called scenario): a series of actions, events and observations that may or may not be compliant.

The process of assessment determines whether a particular scenario is compliant with a particular understanding of the norms. In order to assess a scenario, it is first necessary to interpret the sources of norms one deems relevant to the case (top left), such as legal documents, regulations and policy descriptions. The process of qualification attributes institutional meaning to certain actions performed by actors, events and objects (bottom left). Such qualification is always context-bound which makes the qualification process a subtle interplay between the observations and the interpreted norms applicable to those observations. For example, the action of raising one's arm is typically qualified as "requesting permission to ask a question" in a classroom but as "placing a bid" during an auction.

The normative aspect of eFLINT is based on the legal[1] framework constructed by Hohfeld for analyzing courthouse activities [13]. The first core aspect of Hohfeld's framework is the observation that the 'normative position' of an individual, such as the individual being deemed to having a 'duty' or 'power', is always with respect to another individual. For example, if person X has the duty to do A, then there is a person Y having a 'claim' to A being done (and benefiting from A). In this example, X and Y are said to be in a *duty-claim* relation with respect to action A. The second core aspect of Hohfeld's framework is the explicit consideration of change caused by an actor exercising a 'power' (performing an action), possibly having an impact on the actor with the

[1] Although Hohfeld developed the framework for the analysis of courthouse activities, we apply its concepts more generally and speak of "normative positions" rather than "legal positions" and "normative relations" rather than "legal relations" in this paper.

correlative ‘liability’ position. For example, if X and Y are in a *power-liability* relation with respect to A, then X has the power to do A and Y is liable to – in the sense of ‘being bound to’ – the effects of A.

eFLINT sets itself apart from other formal languages for norm specifications by integrating both core aspects of Hohfeld’s framework, i.e. describing ‘normative relations’ rather than individual positions and allowing normative relations to change over time by the effects of actions and events.

3 LANGUAGE OVERVIEW

The dual nature of institutional reality is reflected in the design of eFLINT. An interpretation[2] is formalized as a collection of type declarations. A scenario is formalized as a series of statements. The statements describe a trace in the transition system induced by the declarations of act-types and event-types. Figure 2 gives the (simplified) abstract syntax of eFLINT specifications as sequences of type declarations. The operators that form Boolean expressions and instance expressions[3] have been omitted; they will be introduced alongside the case study in Section 5. An example specification is given by the listings in Figure 4. The example captures the norm that a child has the power to ask a legal parent (i.e. a natural or adoptive parent) for help with their homework, resulting in a duty for the parent to help. Types can be redefined by subsequent type declarations. This is convenient from a perspective of reuse: a generic interpretation can be used by several applications by letting each application specialize certain types to the domain of the application.

Figure 3 gives the abstract syntax of scripts as sequences of statements and queries. An example script is given by the listings of Figure 5. In the example scenario, Alice is a natural parent of Bob, Bob asks Alice for help, but Alice only helps when the homework is already due, causing the `help-with-homework` duty to be violated. For automatic case assessment, it is convenient to use four input files: a file containing the type-declarations of a generic model, a file re-declaring types according to a concrete or specialized domain, a file containing initialization statements and a file that contains the statements of a scenario. This separation across files is practical as there is a conceptual one-to-many relation between these files (in the order listed above). For example, many scenarios can start from the same initial state.

$$\begin{array}{rlll}
x \in & \textbf{type\_ids} & ::= & \ldots \\
s \in & \textbf{strings} & ::= & \ldots \\
z \in & \mathbb{Z} & ::= & \{0, 1, \ldots, -1, -2, \ldots\} \\
i \in & \textbf{instance\_exprs} & ::= & \ldots \\
b \in & \textbf{boolean\_exprs} & ::= & \ldots
\end{array}$$

$$\begin{array}{rlll}
\delta \in & \textbf{domains} & ::= & \textbf{strings} \\
 & & \mid & \mathit{string\_set}(s_1, \ldots, s_n) \\
 & & \mid & \mathbb{Z} \\
 & & \mid & \mathit{int\_set}(z_1, \ldots, z_n) \\
 & & \mid & \mathit{product}(x_1, \ldots, x_n) \\
 & & \mid & \ldots \\
\mathit{fdc} \in & \textbf{fact\_decls} & ::= & \mathit{fdecl}(d, \delta, b_0^?) \\
\mathit{adc} \in & \textbf{act\_decls} & ::= & \mathit{adecl}(d_0, d_1, d_2, d^*, b^*, c^*, b_0^?) \\
\mathit{edc} \in & \textbf{event\_decls} & ::= & \mathit{edecl}(d_0, b^?, c^*, b_0^?) \\
\mathit{ddc} \in & \textbf{duty\_decls} & ::= & \mathit{ddecl}(d_0, d_1, d_2, d^*, b^?, b_0^?) \\
c \in & \textbf{posts} & ::= & \mathit{create}(i) \mid \mathit{terminate}(i) \\
\mathit{dc} \in & \textbf{decls} & ::= & \mathit{fdc} \mid \mathit{adc} \mid \mathit{edc} \mid \mathit{ddc} \mid \ldots \\
 & \textbf{specifications} & ::= & \mathit{dc}^*
\end{array}$$

Figure 2: Abstract syntax of eFLINT specifications.

$$\begin{array}{rlll}
 & \textbf{elems} & ::= & s \mid z \mid \mathit{tuple}(v_1, \ldots, v_n) \mid \ldots \\
v \in & \textbf{instances} & = & \textbf{elems} \times \textbf{type\_ids} \\
\sigma \in & \textbf{configs} & = & \mathcal{P}(\textbf{instances}) \\
t \in & \textbf{stmts} & ::= & \mathit{create}(i) \mid \mathit{terminate}(i) \mid \mathit{trigger}(i) \\
q \in & \textbf{queries} & ::= & \mathit{query}(b) \\
 & \textbf{scripts} & ::= & (\ t \mid q\ )^*
\end{array}$$

Figure 3: Abstract syntax of eFLINT scenarios.

The declaration of a type determines the set of values (instances) of that type, inherited either from an atomic type (e.g. strings or integers) or a composite record-type (represented as tuples in the abstract syntax). Record-types define relations over concepts (when they have two or more fields, e.g. `natural-parent`) or establish predicates over a concept (when they have one field, e.g. `homework-due`). An institutional model of the physical world *at a particular moment in time* is represented by the values, referred to as a *facts*, deemed to hold true at that moment[4]. Sets of facts are the configurations in the transition system induced by the specification. If a record of type `homework-due` is in configuration σ, then the person at the child field of the record is deemed to having their homework due in the context captured by σ. The accuracy of this fact depends on the accuracy of the qualifications that caused this instance to hold, e.g. the `+homework-due(Bob)` statement in Figure 5.

Type declarations have an optional derivation clause (**`Holds when`**). A derivation clause is a Boolean expression[5] ($b_0^?$ in the abstract syntax) that computes which instances of the type hold true in a given configuration. The derivation clause of `legal-parent` determines that, for every parent P and child C,

[2] We use “interpretation” also for the result of the process of interpretation.

[3] An instance expression computes instances of declared types.

[4] The facts of eFLINT are the fluents of the event calculus, see Section 6.

[5] This is a simplification. Derivation clauses are fully explained in Section 5.

```
Fact person Identified by String
Placeholder parent    For person
Placeholder child     For person
Fact natural-parent   Identified by parent * child
Fact adoptive-parent  Identified by parent * child
Fact legal-parent     Identified by parent * child
  Holds when adoptive-parent(parent,child)
          || natural-parent(parent,child)
Act ask-for-help
  Actor      child
  Recipient  parent
  Creates    help-with-homework(parent,child)
  Holds when legal-parent(parent,child)
Fact homework-due Identified by child
Duty help-with-homework
  Holder        parent
  Claimant      child
  Violated when homework-due(child)
Act help
  Actor      parent
  Recipient  child
  Terminates help-with-homework(parent,child)
  Holds when help-with-homework(parent,child)
```

```
Fact person Identified by Alice, Bob, Chloe, David
```

Figure 4: Type declarations capturing (normative) concepts (top) and a domain of discourse (bottom).

```
+natural-parent(Alice, Bob).
+adoptive-parent(Chloe, David).
```

```
ask-for-help(Bob, Alice).
+homework-due(Bob).  // homework deadline passed
?Violated(help-with-homework(Alice,Bob)).
help(Alice,Bob).
```

Figure 5: A script consisting of an initial state (top) and a scenario with a query (bottom). Duty `help-with-homework` is violated after homework is due.

`legal-parent(P,C)` holds true if and only if `adoptive-parent(P,C)` or `natural-parent(P,C)` holds true. A fact-type is either a 'derived fact' or 'postulated fact', depending on whether it is declared with a derivation clause. To ensure consistency, only postulated facts can be created or terminated by statements.

act-, event- and duty-type declarations. Act-, event- and duty-types are fact-types with additional meaning. An act-type declaration consists of a *performing actor-type* (**`Actor`**), a *recipient actor-type* (**`Recipient`**) and optional further related types (**`Related to`**). An action[6] – an instance of an act-type – is a record value with a field for each of the associated types (a $tuple(v_1, \dots, v_n)$). An act-type declaration also associates zero or more pre-conditions and post-conditions with the act-type. An action A is enabled in σ if A is in σ and if its pre-conditions evaluate to true in σ. If A is enabled in σ, then the performing actor X and the recipient actor Y are in a power-liability relation with respect to A. The post-conditions associated with an act-type determine the effects its instances have when executed by a statement. The effects are to create or terminate facts, thus giving rise to a new configuration (and possibly updated normative relations).

The institutional view on the world can also change by physical actions for which there is no institutional counterpart. For example, there is no direct institutional counterpart to 'changing address' in the GDPR even though relocations influence the accuracy of personal data. Moreover, the world also changes due to natural events such as earthquakes, fires and the passage of time. For these reasons, eFLINT distinguishes between actions and events. An event-type declaration (not in the example) is essentially an act-type declaration without a performing actor or recipient actor.

A duty-type declaration contains the type of the *duty holder*, the type of the *claimant* and optional further related types. A duty – an instance of a duty-type – is a record value with a field for each of the types. A duty-type declaration has zero or more violation-conditions. If a duty holds true in configuration σ, then the holder and claimant of the duty are in a duty-claim relation in σ. The holder of a duty is in violation of the duty (and the claimant has a valid claim) according to σ if the duty holds true in σ and if one of the violation-condition holds true in σ. To avoid violating a duty, an actor must perform an action that terminates the duty (e.g. `help`) before one of the violation conditions holds.

transitions and compliance. In summary, a set of facts forms a configuration representing an institutional view on the physical world at a particular moment in time. For any given configuration it is possible to determine the normative relations between actors. Actions and events change configurations when executed, potentially modifying the normative relations between actors. Since actions and duties are also facts, an actor may have the power to assign duties or to grant powers to others. The post-conditions of action- and event-types give rise to a transition system. The execution of an action or event triggers a transition by removing and/or inserting facts from/to the current configuration. For example, the statement `ask-for-help(Bob,Alice)` ($trigger(i)$ in the abstract syntax) executes the action `ask-for-help(Bob,Alice)`, causing the creation of a duty for Alice. Individual facts can also be created and terminated ($create(i)$ and $terminate(i)$

[6] Instances of act-types are institutional actions. There is not necessarily an institutional actions for every physical action or vice versa.

in the abstract syntax). For example, `+homework-due(Bob)` creates the fact that Bob's homework is due (termination is written with a minus symbol). The statements of a script form a trace (sequence of transitions) in the transition system. A query (e.g. `?`**`Violated`**`(help-with-homework(Alice,Bob))`) is a Boolean expression that is evaluated in the context of the current configuration. The language also supports invariants, essentially queries that are performed whenever a transition gives rise to a new configuration.

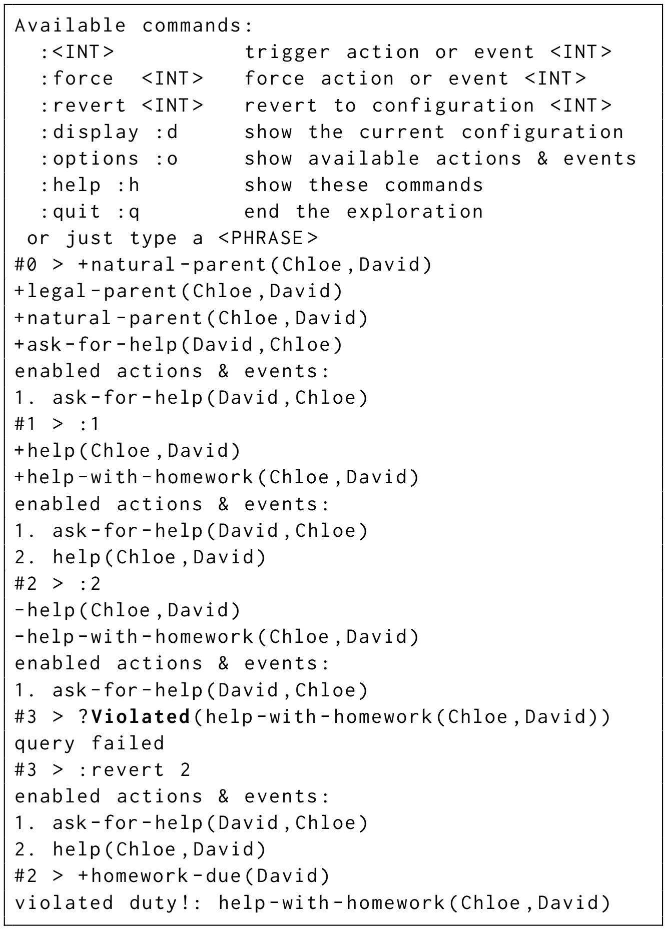
A trace may be *action-compliant* and/or *duty-compliant*. A trace is action-compliant if every transition on the trace is labeled with an event or action that is enabled in the source configuration of the transition. A trace is duty-compliant if no duties are violated in any of its configurations. These notions are independent: a trace can be action-compliant, duty-compliant, action- and duty-compliant or neither. Assessing a scenario is deciding whether it produces an action- and duty-compliant trace, given an initial configuration. The example scenario is action-compliant but not duty-compliant as a violation occurs after the second statement.

A trace records the normative positions and relations of all actors as they evolve over time and therefore provides sufficient information to determine important details about violations such as when they occurred and which actors were responsible. This a crucial aspect: by recording traces, eFLINT makes it possible to reproduce the entire decision making process and to explain the decisions that have been made.

4 IMPLEMENTATION

The previous section explained eFLINT informally through an example. A formal syntax, operational and static semantics are available in the supplementary material of this paper. The supplementary material also provides a prototype implementation of in Haskell. The details of the expression language used by the eFLINT implementation have been omitted in the previous section and are discussed alongside the GDPR case study in the next section. The transition system semantics of eFLINT depends only on the expression language in that there are Boolean expressions and instance expressions. Alternative expression languages can thus be used by alternative implementations, e.g. using objects rather than records to structure data. Similarly, our implementation has integers and strings as atomic values, but other types of atoms, such as floating points, are easily added. This section gives an overview of the different applications supported by the eFLINT implementation at the time of writing.

automated assessment. One of the executables of the implementation receives a specification and script and determines whether the scenario in the script is action- and duty-compliant and whether all the queries are successful. The output is either a list of violations and failed queries or a JSON object representation of the produced trace. The JSON output has been used to develop a simple web-interface. Besides editing, the interface can be used to analyze traces by inspecting the contents of, and the changes to, configurations. The web-interface has been used in a MSc-level course on 'Policy Making and Rule Governance', with user-feedback feeding directly into the design of the language. Automatic assessment is an important tool during the development of eFLINT specifications as it facilitates testing and debugging. Once a specification has been adopted, the primary purpose of automatic assessment is to analyze concrete cases that have observed or hypothetical cases that might arise. Both are crucial, not only in the development of the specification, but also as feedback to lawmakers and policymakers. As part of future work we intend to add model checking to our implementation, expanding the set of tools through which confidence in the correctness of a specification is obtained. eFLINT already supports safety properties in the form of invariant declarations.



```
Available commands:
  :<INT>          trigger action or event <INT>
  :force  <INT>   force action or event <INT>
  :revert <INT>   revert to configuration <INT>
  :display :d     show the current configuration
  :options :o     show available actions & events
  :help :h        show these commands
  :quit :q        end the exploration
 or just type a <PHRASE>
#0 > +natural-parent(Chloe,David)
+legal-parent(Chloe,David)
+natural-parent(Chloe,David)
+ask-for-help(David,Chloe)
enabled actions & events:
1. ask-for-help(David,Chloe)
#1 > :1
+help(Chloe,David)
+help-with-homework(Chloe,David)
enabled actions & events:
1. ask-for-help(David,Chloe)
2. help(Chloe,David)
#2 > :2
-help(Chloe,David)
-help-with-homework(Chloe,David)
enabled actions & events:
1. ask-for-help(David,Chloe)
#3 > ?Violated(help-with-homework(Chloe,David))
query failed
#3 > :revert 2
enabled actions & events:
1. ask-for-help(David,Chloe)
2. help(Chloe,David)
#2 > +homework-due(David)
violated duty!: help-with-homework(Chloe,David)
```

Figure 6: Example interaction with the eFLINT REPL.

exploration. To further support the aforementioned use cases, the eFLINT implementation also supports manually exploring the transition system induced by a specification. The tool can run as a Read-Eval-Print Loop (REPL), loaded with a specification and a script producing an initial state. At the top-level, the REPL accepts declarations, queries and statements, which can be mixed freely and produce immediate feedback. It is also possible to delete or re-declare types, enabling on-the-fly updates to the specifications. After every statement, the REPL reports violations of action- or duty-compliance, changes to the configuration and any invariants that were not upheld. The user can backtrack to a previously visited configuration to explore an alternative route in the scenario. An example interaction with the REPL is shown in Figure 6. The REPL is loaded with the specification of Figure 4. The interaction shows Chloe helping David in the first explored branch. After backtracking, another branch is explored in which homework is due before Chloe has helped.

During the first interaction in Figure 6, the REPL responds with the information that the fact `legal-parent(Chloe,David)` has been added to the configuration. This fact is derived from the fact `natural-parent(Chloe,David)` postulated by the user. In order to make this derivation, the implementation has enumerated all possible instances of `legal-parent` and evaluated the derivation clause for each. Enumerating all instances of types is possible when working with a small[7], finite domain. However, when checking the compliance of a running system, an application discussed below, an open-ended domain is typically required. The ability to redefine and specialize types in eFLINT enables us to reuse specifications across applications in which some require a finite and others an open-ended domain. In the example of Figure 4, an open-ended domain (the declaration of `person` initially does not list its instances) is replaced by a finite domain. Reusing specifications in both types of applications is also made possible by the pragmatic design choice to give different behavior to the enumeration operator (**`Foreach`**, introduced in the next section) depending on whether it enumerates instances of a finite or infinite type. In the latter case, the operator only enumerates the instances of the type that hold true in the current configuration. Note that a domain is finite if all of the atomic types have finite sets of instances, because then, by induction, all record types are finite too.

normative actors. The back-end of the REPL also forms the basis of a TCP server. The server is loaded with a specification and waits for incoming declarations, statements and queries on a given port. The server is used to integrate eFLINT specifications in arbitrary software systems. To develop and experiment with regulatory services for enforcement, we use the Akka framework[8] for actor-oriented programming in Scala. Actor-oriented programming can be used to develop or model complex, distributed systems. The components of a software system are implemented as actors, with message-passing as the only form of communication between them.

Central to our approach is the notion of a 'normative actor' that administers an eFLINT specification. A normative actor is created with one or more specification files and starts its own server instance. The first file contains the type declarations of a high-level norm specification. The additional files provide more declarations, specific to the domain, and may redefine some types towards a specialized domain.

A survey of various software architectures that incorporate policy enforcement mechanisms [22, 23, 33, 34] has revealed at least the four types of enforcement listed below Our implementation of normative actors in Scala enables these four types of enforcement.

- *Ex-ante enforcement of permissions*: ensuring that an actor has permission to execute a particular action before it is performed and blocking the action if there is no permission
- *Ex-ante enforcement of positive duties*: informing actors of their duties to perform certain actions
- *Ex-post enforcement of violations of prohibitions*: applying some form of resolution to the observation that an action has been performed which was not enabled
- *Ex-post enforcement of violated duties*: applying some form of resolution to a violated duty

A normative actor responds to eFLINT statements and queries received by as messages. The response to a query is simply whether the query holds true. Actors can send queries to normative actors to let the responses guide their behavior. For example, an actor can ensure action-compliance by checking whether an action is enabled before performing it.

Receiving a statement causes the normative actor to update its internal state (a configuration) by executing the statement. The resulting transition might impact other actors in the system and they will be informed by the normative actor accordingly. If a duty is created or violated by the transition, the holder and claimant of the duty are informed. Similarly, if an action is enabled by the transition, the performing and recipient actor of the action are informed. If the transition was triggered by a disabled action, the performing and recipient actor of the action are informed of this violation. The actors receiving such messages can react in several meaningful ways. For example, the claimant of a violated duty might have the power to notify an authority that, in turn, has the power to place a penalty on the holder of the violated duty. Another common use case is for the claimant of a new (but not yet violated) duty to

[7] In order not to suffer from combinatorial explosion.

[8] https://akka.io

start a timer that runs out when the claimant thinks the duty should have been fulfilled (terminated). The example of Figure 4 can be extended with a teacher that places the `complete-homework` duty on children. When the timer expires, the claimant (teacher) sends a message to the normative actor to communicate this observation (in the form of a fact, e.g. `homework-due`) possibly causing duties to be violated (e.g. the duty `complete-homework`, but perhaps also `help-with-homework`). In our experiments with GDPR, the event `rectification-delay`, creating the fact `undue-rectification-delay`, is an event triggered by the use of a timer.

The actor sending a statement to a normative actor may be a neutral observer and, for example, not the performer or recipient of an action. The normative actor responds to the observer with a summary of the effects of the transition, similar to the output produced by the command-line REPL. Normative actors can thus be used in a variety of ways.

A system can have one or more monitoring actors making qualifications based on the observed communication between other actors. These qualifications are sent to normative actors that instantiate the norms considered by the monitoring actor. In this case, the communications of normative actors might be restricted to monitoring actors only.

In a multi-agent system (MAS), normative actors can be internalized by agents, playing the role of a (moral, social, or legal) conscience. An agent communicates with its internal normative actors to possibly update its beliefs, desires and intentions. Every normative actor of an agent embodies the particular interpretation of a set of norms adopted by the agent. In this case, the communications of normative actors should be restricted to their encompassing agent.

5 CASE STUDY: GDPR

The eFLINT specification developed in this section captures the following aspects of the GDPR [19]: the requirement to receive a data subject's consent prior to data processing and the subject's 'right to rectification'. The purpose of this section is to dive deeper into the details of the language, such as its expression language, and to demonstrate its expressivity in connection to a realistic case. The presented code snippets form the GDPR component of a larger case study regarding the Know Your Customer (KYC) requirements placed on financial institutions. This case study is performed in collaboration with ABN AMRO and ING. In order to improve the accuracy of customer risk assessment, the banks are willing to share customer data under certain conditions. The banks wish to keep certain data secret, e.g. if the data provides a competitive advantage. Moreover, the banks want to demonstrate compliance with KYC and GDPR requirements. The goal of the wider case study is to experiment with architectures for a data sharing system, incorporating formalizations of the internal policies, sharing agreements, guidelines and regulations that govern the system in order to demonstrate compliance. As an academic case, the study is particularly interesting in that it requires reasoning about multiple norm specifications, each with their own ontologies of concepts. Moreover, satisfying a duty in one policy might cause a violation in another, demonstrating the need for priorities between norms.

The KYC case consists of three eFLINT specifications of less than a 100 lines of code for which one or more normative actors (see previous section) are created. Besides the GDPR specification, the case involves an internal policy specification and a sharing agreement. Every bank has their own specialization of the internal policy. The eFLINT specifications are kept small, formalizing only specific rules and norms to focus on their interaction at the level of policy design, the level of component behavior and the level of component implementation. The system consists of actors representing banks, employees and clients. Each bank communicates with a normative actor loaded with GDPR and a normative actor loaded with the bank's internal policy. The former notifies banks (controllers) and clients (subjects) about (violated) duties and actions. The latter notifies banks and employees. The banks also communicate with a single normative actor that embodies the sharing agreement.

5.1 Concept definitions

The following code fragment below captures the GDPR concepts 'subject' (a natural person) and 'data'.

```
Fact subject
Fact data
Fact subject-of    Identified by subject * data
```

A fact-type declaration without an **`Identified by`** clause defaults to **`Identified by String`**.

expressions. Consider the following expression.

```
(Exists subject: subject-of(subject,data))
```

Expressions are literals, variables or operators and constructors applied to other expressions. Type names can occur as variables in expressions, such as `subject` and `data` above. Type names can also occur as constructors in expressions, for example `subject-of` above. There is no ambiguity between a type name occurring as a variable or as a constructor because only constructors are followed by (zero or more) formal arguments within parentheses.

Constructor application can be written in two styles. The first style – familiar from functional and logic programming – requires as many arguments as the number of fields of the constructed record and the arguments must be written in the same order as the fields are written in the type-declaration. An example is `subject-of(subject,data)`. In the second style, field names are explicitly mentioned. For example, in `subject-of(subject=subject,data=data)` the name `subject` occurs as a field name on the left-hand side of the equal symbol and as a variable on the right-hand side. In this style, formal arguments can be written in any order and can also be omitted. If a formal argument for field x is omitted, then it defaults to $x = x$. The constructor application of this example can thus be written as `subject-of()`. If the variable `subject-of` is bound to a record, then `subject-of.subject` evaluates to the value of the field `subject` of that record (projection).

accumulators. The example expression shows that eFLINT is based on first-order logic with existential and universal quantification via the `Exists` and `Forall` operators. The inner expression of a quantifier, appearing behind the colon, must be a Boolean expression. However, the sub-expression `subject-of(subject,data)` of the example is an instance expression when taken out of context. The static semantics of eFLINT rewrites this expression to the Boolean expression `Holds(subject-of(subject,data))`. The `Holds` operator checks whether the instance computed by its operand holds true in the current configuration.

The example is expression is equivalent to the following:

```
Or(Foreach subject: subject-of(subject,data))
```

The semantics of `Foreach` are to evaluate its inner expression multiple times, each time binding its binders (the comma-separated variables before the colon) to a different combination of instances of their respective types. For example, when the above expression is evaluated, the inner expression is evaluated once for every possible binding of the variable `subject` to an instance of the type `subject`. The behavior of `Foreach` differs depending on whether all its binders refer to types with finite numbers of instances. If this is the case, then the binders are bound to all possible combinations of instances of their types. If one or more of the types does not have a finite amount of instances, only the instances of these types that hold true in the current configuration are enumerated instead. As mentioned in the previous section, this design decision has been made to simultaneously accommodate applications with finite and open-ended domains.

The `Foreach` operator is non-deterministic in that it computes multiple values. However, non-deterministic expressions are only allowed in certain places, such as in the post-conditions of actions and as the operand of an accumulator. An accumulator is an operator that reduces a sequence of values to a single result (thus turning a non-deterministic expression in a deterministic one). The `Or` accumulator evaluates to `true` if any of its inputs is `true`. Similarly, `And` evaluates to `true` if all its inputs are `true`. Occurrences of `Exists` and `Forall` desugar to an application of `Foreach` inside an application of `Or` or `And` respectively. Other examples of accumulators are `Count` and `Sum` for counting instances and summing integers.

derivation clauses. The example expression was taken from the following fact-type declaration with a derivation clause.

```
Fact personal-data Identified by data Holds when
  (Exists subject: subject-of(subject,data))
```

The derivation clause determines that data is personal data if it has a subject, which closely resembles the definition of "personal data" in Article 4(1) of the GDPR.

Derivation clauses come in two forms: a `Holds when` clause with a Boolean expression or a `Derived from` clause with an instance expression. The instance expression of a `Derived from` clause produces all the instances of the type that are deemed to hold true by the clause. Any variables not explicitly bound by occurrences of `Exists`, `Forall` or `Foreach` in this instance expression are implicitly bound by an outermost `Foreach`. A `Holds when` clause is syntactic sugar for a `Derived from` clause. The above fragment is equivalent to the following:

```
Fact personal-data Identified by data Derived from
 (Foreach data: personal-data() When
  (Exists subject: subject-of(subject, data)))
```

The `When` operator is used in non-deterministic expressions to filter out unwanted results. The operator evaluates to the result of its first operand, but only if its second operand evaluates to `true`.

In general, a `Holds when` clause with Boolean expression `[t]` for a fact-type `[x]` with fields `[a]`, `[b]` and `[c]` desugars to `Derived from (Foreach [a],[b],[c]: [x]() When [t])`. A `Holds when` clause can therefore only be part of a fact-type declaration with a record type. The desugaring shows that the expression of a `Holds when` clause is evaluated in the context of a record instance to determine whether that instance holds true. In the example, this is the instance `personal-data(data = data)` for some instance of `data`.

5.2 Consent

This subsection formalizes Article 6(1)(b) on consent [19]. The following fragment defines the related concepts of data controller, data processor, purpose and consent.

```
Fact controller
Fact processor
Fact processes Identified by
  processor * data * controller * purpose
Fact purpose
Fact consent Identified by
  subject * controller * purpose
Fact accurate-for-purpose Identified by
  data * purpose
```

A controller is a legal entity collecting and processing the data of a data subject. A processor is a legal entity storing or processing data on behalf of a controller. An important aspect of the GDPR is that a controller communicates the purpose for which it is collecting and processing data and that the subject gives explicit consent to processing[9] the data for that purpose. If the record of type `consent` that contains subject S, controller C and purpose R holds true in a configuration, then this means that S has given consent to C to collect and process their data for purpose R. If the record of type `processes` that contains processor P, controller C, data D and purpose R holds true in a configuration, then this means that P processes the data D on behalf of C for the purpose R.

The presented formulation abstracts over operations on data by capturing all changes to data as replacements. Every instance of data has at most one subject and at most one instance of data is considered to be accurate for a particular purpose. These requirements have been formalized as invariants. As discussed in Section 4, our implementation runs every invariant as a query after every transition and reports an error if the invariant's expression does not hold true. Invariants are thus useful to find inconsistencies in (applications of) specifications. The latter requirement is formalized by the following invariant declaration.

```
Invariant accuracy-for-purpose :
 (Forall subject,purpose,data,data': data == data'
         When subject-of()
           && subject-of(data=data')
           && accurate-for-purpose()
           && accurate-for-purpose(data=data'))
```

When the postal address of a bank's client changes, the currently held postal address is no longer accurate for the purpose of building a KYC client profile. However, there is no definition of an action for clients (subjects) to change their addresses because relocation is not a GDPR notion. Instead, an instance of the `data-change` event (defined below) is triggered when a change in address is observed. In our wider case study, a behavioral model captures the process of communication between clients and bank employees. In this model, when an employee receives updated information from a client, the `data-change` event is triggered by sending a message to the normative actor that administers GDPR compliance on behalf of the employee's bank.

The event `data-change` captures arbitrary changes to data.

```
Event data-change
 Related to data, new-data, purpose
  When data != new-data
    && subject-of() && subject-of(data = new-data)
 Terminates accurate-for-purpose(data,purpose)
 Creates    accurate-for-purpose(new-data,purpose)
 Holds when accurate-for-purpose(data,purpose)
```

```
Placeholder new-data For data
```

The `When` clause introduces an instance constraint (omitted from the abstract syntax in Section 3), establishing a connection between the fields of the declared type. In this example, the constraint determines that the new data is indeed new and has the same subject as the old data. An instance constraint keeps the `Foreach` operator from enumerating 'invalid' instances of the declared type.

The definition of `data-change` refers to multiple instances of `data`. To support multiple such references, variables can have 'decoration' in the form of integer numbers or prime characters at the end of their name (e.g. `data1`, `data2`, and `data'`). The user can also add their own variable names with a `Placeholder` declaration. In the fragment above, the name `new-data` is introduced as a placeholder for instances of the fact-type `data`. Note that the constructor application `subject-of()` has `data` as an implicit argument because this is the name of one of its fields.

The act-type declaration of `give-consent`, shown below, describes the power of subjects to giving consent to controllers. The `Related to` clause reflects that consent is given for a specific purpose.

```
Act give-consent
  Actor subject
  Recipient controller
  Related to purpose
  Conditioned by !consent()
  Creates consent()
```

The (pre- and post-) conditions of an act-type are evaluated in an environment that binds the field names of the type. For example, to determine the effects of the instance `give-consent(Alice,Bank,ClientProfile)`, the variable `subject` is bound to `Alice`, the variable `controller` is bound to `Bank` and the variable `purpose` is bound to `ClientProfile`. These bindings are used as the (implicit) arguments to the constructor application `consent()`. The pre-condition of `give-consent` prevents repeated execution of the same instance of `give-consent` (although one can argue that the power to give consent is unconditioned).

The act-type `collect-personal-data` given below determines that consent must have been given by the subject for the purpose for which the data is being collected and that the data must be accurate for this purpose. If, in physical reality, multiple processors process the collected data, then the institutional action `collect-personal-data` is to be executed multiple times, with different `processor` arguments.

```
Act collect-personal-data
  Actor controller
  Recipient subject
  Related to data, processor, purpose
    When subject-of()
  Conditioned by consent() &&
    accurate-for-purpose()
```

[9] Consent is one of several grounds for lawfully processing personal data.

```
  Creates processes()
```

5.3 The right to rectification

Article 16 of the GDPR states [19]:

> *The data subject shall have the right to obtain from the controller without undue delay the rectification of inaccurate personal data concerning him or her. [...]*

The interpretation we formalize contains:

(1) the power for the data subject to demand rectification when the data processed by the controller is inaccurate for the purpose it is processed
(2) a duty held by the controller to rectify the data without undue delay when the subject demands it
(3) the controller is relieved of this duty once their processors start using the subject's accurate data instead

The fragment below formalizes (1) as an act-type.

```
Act demand-rectification
 Actor subject
 Recipient controller
 Related to purpose
 Conditioned by (Exists data, processor:
  subject-of() && processes() &&
    !accurate-for-purpose())
 Creates rectification-duty()
```

The action `demand-rectification` is enabled for a subject if there is a processor that, on behalf of the controller, processes inaccurate personal data of the subject. The effect of the action is to place the duty (point (2) above) on the controller.

The duty is defined by the following duty-type declaration.

```
Duty rectification-duty
  Holder controller
  Claimant subject
  Related to purpose
  Violated when undue-rectification-delay()
```

```
Fact undue-rectification-delay Identified by
  controller * purpose * subject
```

Since accuracy depends on purpose, the duty-claim relation between controllers and subjects is related to a purpose.

Like the conditions of act-types, a violation condition is evaluated in an environment binding the fields of the types. The `rectification-duty` is violated whenever there is undue delay in between the demand for rectification and the rectification taking place. The usage of the term "undue delay" in the article leaves room for discussion (deliberately). In other words, potential cases of delay are subject to qualification. The event `rectification-delay`, defined below, can be triggered to indicate that the duty has been violated because of an undue delay. The event is only available when the corresponding `rectification-duty` is active.

```
Event rectification-delay
 Related to controller, purpose, subject
 Creates    undue-rectification-delay()
 Holds when rectification-duty()
        && !undue-rectification-delay()
```

The controller is relieved of the `rectification-duty` when its processors start using accurate data for the given `purpose`.

```
Act rectify-personal-data
 Actor processor
 Recipient subject
 Related to controller, purpose
 Conditioned by processes() When subject-of()
 Terminates
  processes() When subject-of(),
  rectification-duty() When last-inaccurate(),
  undue-rectification-delay() When
    last-inaccurate()
 Creates
  processes() When subject-of() &&
    accurate-for-purpose()
 Holds when rectification-duty()
```

The pre-condition and the first (terminating) post-condition of `rectify-personal-data` have implicit references to the unbound variable `data`. In pre-conditions, unbound variables are existentially quantified implicitly. The pre-condition thus states that the processor must process some personal data of `subject` on behalf of `controller`.

Unbound variables in post-conditions are bound by an implicit `Foreach`. A single post-condition can thus evaluate to zero or more instances that will all be created or terminated when executed. The first terminating post-condition of `rectify-personal-data` terminates all personal data of the subject that the processor is processing on behalf of the controller. The second and third terminating post-condition terminate the `rectification-duty`, held by the given controller for the given subject and purpose, and remove the observation that there has been undue delay for that duty. This happens only if `processor` is the last processor to hold inaccurate data about `subject` with respect to `purpose`. If a post-condition attempts to terminate a fact that does not hold, then this is simply ignored and the execution of the action proceeds normally. This design choice is symmetric with the decision that a configuration is a *set* of facts, i.e. that there are no duplicates of facts. A creating post-condition that attempts to add a fact that already holds true simply has no effect (besides the other facts that it might add). The creating post-condition of `rectify-personal-data` creates the fact that represents that accurate data about the subject is processed. This holds for at most one instance of `data`, because of the assumption mentioned earlier that for every subject and purpose at most one instance of `accurate-for-purpose` holds true.

The predicate that determines whether a processor is the last processor to hold inaccurate data for a given subject and purpose is defined below.

```
Fact last-inaccurate Identified by
    processor * controller * subject * purpose
  Holds when (Forall processor', data :
   processor == processor'
      When  processes(processor=processor')
        &&  subject-of()
        && !accurate-for-purpose())
```

The presented interpretation of the right to rectification appears to make it impossible for controllers to terminate their own `rectification-duty`. However, note that the specification captures an institutional perspective and does not describe physical behavior. From the perspective of the GDPR it is irrelevant how a processor decides whether to rectify the data. For example, the processor may desire to adhere to a contract made with the controller. In our KYC case, this is the sharing agreement, which is formalized separately.

5.4 Reflections

The GDPR case study presented in this section demonstrates that eFLINT can be used to formalize, rather concisely, norms described in significant, real-world regulations. Although not shown, the formalization can be used to assess concrete cases and can be used to reason about the compliance of running systems. In future work we intend to give a comprehensive account of a generic data-sharing architecture that involves normative actors for regulatory services, showing in particular how multiple eFLINT models co-exist and how eFLINT is used to enforce compliance of software with respect to multiple regulations, policies and contracts.

In the design of eFLINT, focus has been on the possibility to simultaneously use specifications in isolation – typically with a finite domain – as well as in in running systems – typically with an open-ended domain. To this end, the language has a REPL-oriented design, supporting manual exploration and enabling external systems to trigger actions and events. This motivation guided other design choices as well, e.g. the semantics of **`Foreach`** and the ability to redefine types to form specialized domains. For example, the GDPR specification does not give structure to the `data` concept. As part of the KYC experiments, `data` is redefined and concretized to be about customer profiles with attributes such as country code and address. Crucially, this change does not demand changes to any of the other types; the specification remains valid.

Omitting formal arguments in constructor applications has significantly improved the brevity of the GDPR specifications. Moreover, by hiding the structure of data, clauses of type declarations read almost like natural text and less like programming instructions. For example, the pre-condition **`Conditioned by`** `consent()` reads more natural than **`Conditioned by`** `consent(subject,controller,purpose)`. The downside is the effort required to reproduce the full constructor applications whenever a detailed reading of the code is necessary. This effort can be greatly reduced with IDE support, e.g. by showing the declaration of a type when holding the cursor over a constructor.

The distinction between derived facts and postulated facts is a crucial feature of eFLINT. Derivation clauses effectively describe logical implications of the kind that are common in logical programming. However, the antecedents and consequents are not limited to Boolean formulas. For example, in the following code, the type `wealth` always has exactly one instance corresponding to the sum of all people's savings.

```
Fact person
Fact balance    Identified by Int
Fact balance-of Identified by person * balance
Fact wealth     Identified by Int Derived from
  Sum(Foreach balance-of : balance-of.balance)
```

An important aspect of postulated facts is that their validity is established externally. In the current implementation, facts are proactively provided by an external system. However, the implementation can be extended to support on-demand request. Determining the validity of a fact can then be delegated to a relevant authority, e.g. a health-care provider confirming the validity of a receipt to supports an insurance claim.

Further reflections are provided in comparison to related work.

6 RELATED WORK

The aspects of norms automated by software are roughly divided into structural aspects – such as production, instantiation and publication – and dynamic aspects – such as implementation, modification, termination, monitoring and enforcement. On the structural side, standards have been developed for the digital representation of contracts (e.g. Oasis eContracts [6]) and for referring to sources of law (e.g. MetaLex, Akomo Ntoso, Juriconnect and ECLI). Type-declarations in eFLINT can easily be extended with references to sources, e.g. using MetaLex [2], making it possible to relate components of traces in the transition system to sources, further enhancing explainability. Natural language processing can assist with the interpretation process, introducing the necessary structure to allow contracts to be analyzed by software [4, 31]. However, legal experts and policy makers are not all programmers, and certain information required to compute with norms is not present in sources, e.g. the structure of, and relations between, values. This observation has motivated and influenced the development of eFLINT.

The computational theory underneath eFLINT is most similar to event calculus [16] and its simplified variants [24]. In the variants of [5, 21], time is represented as a sequence of (distinct) time points. At each time point, certain *fluents* are stated to hold true by the $HoldsAt(f, t)$ predicate (with f a fluent and t a time point). Events initiate or terminate fluents at certain time points according to the predicate $Initiates(e, f, t)$ or $Terminates(e, f, t)$ (with e an event). The judgment $Happens(e, t)$ states that event e takes place at time t. A collection of axioms determines that initiated fluents hold true until they are terminated, that terminated fluents do not hold until they are initiated (again) and that the correct fluents are initiated and terminated after events happen. The facts of eFLINT correspond to the fluents of the event calculus and time points can be seen as identifiers for configurations, i.e. $HoldsAt(f, t)$ states that fact f is in configuration t. The creating and terminating post-conditions of actions and events in eFLINT correspond to judgments involving the *Initiates* and *Terminates* predicates respectively (for every possible time point). The derivation clauses of fact-type declarations introduce rules of the form $HoldsAt(f_1, t) \wedge \ldots \wedge HoldsAt(f_k, t) \implies HoldsAt(f_{k+1}, t)$. Expressions in eFLINT are thus interpreted as logical formulae with `Forall` and `Exists` implying the usage of first-order logic. Halpern and Weissman discuss the use of first-order logic to describe and reason about policies, identifying in particular a number of first-order languages with different characteristics regarding the complexity of reasoning [11]. A scenario in eFLINT corresponds to a set of concrete judgments of the form $Happens(e, t)$. However, $Happens(e, t)$ can be stated for multiple instances of e and the same t, i.e. multiple events can happen simultaneously, which is not true for the actions and events of eFLINT. Perhaps eFLINT is easily extended with declarations of composite events for this purpose. The notions of action-violation and duty-violation used by eFLINT are easily to formulate as logical predicates. A translation from eFLINT to answer set programming based on the connection with the event calculus is being considered.

Several formal languages for norms are based on extensions to the event calculus such as Symboleo [28] and InstAL [20]. Besides the event calculus, Symboleo and eFLINT are also related in their Hohfeldian foundations. In Symboleo obligations and powers are instantiated by 'triggers'. In eFLINT, derivation clauses can be used for this purpose. However, in eFLINT expressions are always evaluated in the current configuration, whereas Symboleo is true to the event calculus in that expressions concern the entire timeline.

Significant work exists about the formalization, analysis and enforcement of specific kinds of policies such as policies for access control and network policies [1] (e.g. firewall configurations), of which a survey is given by Jabal et al [14]. The eFLINT language is instead used to describe a wide variety of normative sources such as laws, regulations, policies and contracts. The Margrave Policy Analyzer tool[10] can be used to reason about access control and firewall configurations, supporting several formalisms such as the widely adopted XACML for access control [29]. The tool implements several types of reasoning, including Change-Impact Analysis [8].

Governatori et al. give a detailed overview on the interpretation and lifecycle of contracts in [9]. In terms of the elements described in [9], eFLINT is used to make interpretations explicit, including *implied terms* and norms that follow from the *integration* of the contract in a wider context. With the example of undue rectification delay and accuracy for purpose, we have shown how *open-textured terms* are treated by explicit qualification in eFLINT. The automated assessment of scenarios, explained in Section 4, can assist with *dispute resolution*. Section 4 also explains how normative actors can be used for *monitoring* and *enforcement*. The implementation also enables dynamic *modification* of contracts by removing and redefining (act- and duty-) types on-the-fly. Several avenues are being considered to support the *implementation* of contracts formalized in eFLINT, including smart contracts.

The idea of smart contracts was first introduced by Szabo [30] as software (or hardware) that facilitates the exchanges of digital items of value between two or more parties, with a security mechanism in place that ensures the exchange at a risk low enough for all parties. With the advent of blockchain technology [17], smart contracts are now typically understood as scripts that facilitate the execution of 'transactions'. The underlying blockchain technology forms a distributed ledger establishing consensus between potential witnesses about the history of transactions. A popular smart contract language is Solidity [7], running on the Ethereum platform [3, 32]. The Flint language (unrelated to eFLINT) offers a safer alternative to Solidity for writing smart contracts running on Ethereum [25]. The Marlowe DSL is used to develop smart contracts at a higher level of abstraction that run on the Cardano platform [27].

7 CONCLUSIONS

We have presented eFLINT, a novel domain-specific modeling language for formalizing norms found in laws, regulations, policies, contracts and (data-sharing) agreements. The action-oriented nature of the language makes it possible to apply the language in a variety of applications such as case analysis and monitoring the compliance of a running system. Pragmatic design decisions have been made to allow specifications to be reused for both types of reasoning. In particular, the generic concepts encountered in laws and regulations can be formalized at a high level of abstraction, whilst applications of these formalizations can effortlessly redefine

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

the concepts to match the domain of the application. Our approach involves the explicit qualification of physical reality as institutional facts, actions, events and duties. The normative positions of actors evolve over time as actions are performed and events take place. The resulting traces can be used to diagnose violations and to provide explanations about the decisions made based on the norms.

Acknowledgements. This work is supported by the NWO project (628.009.014) Secure Scalable Policy-enforced Distributed Data Processing (SSPDDP), part of the NWO research program Big Data: Real Time ICT for Logistics

REFERENCES

[1] E. S. Al-Shaer and H. H. Hamed. 2004. Modeling and Management of Firewall Policies. *IEEE Transactions on Network and Service Management* 1, 1 (2004), 2–10.

[2] A. Boer, R. Hoekstra, E. De Maat, F. Vitali, M. Palmirani, and B. Ratai. 2010. Metalex (Open XML Interchange Format for Legal and Legislative Resources). Technical Report CWA, Vol. 15710:2010. European Committee for Standardization (CEN).

[3] V. Buterin. 2018. Ethereum White Paper.

[4] Ilias Chalkidis, Ion Androutsopoulos, and Achilleas Michos. 2017. Extracting Contract Elements. In *Proceedings of the 16th Edition of the International Conference on Articial Intelligence and Law* (London, United Kingdom) *(ICAIL 2017)*. Association for Computing Machinery, 19–28. https://doi.org/10.1145/3086512.3086515

[5] Marinos Charalambides, Paris Flegkas, George Pavlou, Arosha K. Bandara, Emil C. Lupu, Alessandra Russo, Naranker Dulay, Morris Sloman, and Javier Rubio-Loyola. 2005. 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*. IEEE Computer Society, 99–108. https://doi.org/10.1109/POLICY.2005.23

[6] OASIS LegalXML eContracts TC. 2007. eContracts Version 1.0 Committee Specification.

[7] Ethereum. 2016. Solidity Documentation Online. https://solidity.readthedocs.io. [Online, accessed 11 May 2020].

[8] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. 2005. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering* (St. Louis, MO, USA) *(ICSE '05)*. Association for Computing Machinery, New York, NY, USA, 196–205. https://doi.org/10.1145/1062455.1062502

[9] Guido Governatori, Florian Idelberger, Zoran Milosevic, Régis Riveret, Giovanni Sartor, and Xiwei Xu. 2018. On legal contracts, imperative and declarative smart contracts, and blockchain systems. *Artificial Intelligence and Law* 26, 4 (2018), 377–409. https://doi.org/10.1007/s10506-018-9223-3

[10] Guido Governatori, Michael J. Maher, Grigoris Antoniou, and David Billington. 2004. Argumentation Semantics for Defeasible Logic. *Journal of Logic and Computation* 14, 5 (10 2004), 675–702. https://doi.org/10.1093/logcom/14.5.675

[11] Joseph Y. Halpern and Vicky Weissman. 2008. Using First-Order Logic to Reason about Policies. *ACM Trans. Inf. Syst. Secur.* 11, 4 (2008), 21:1–21:41. https://doi.org/10.1145/1380564.1380569

[12] H. Herrestad. 1993. Norms and Formalization. In *Proceedings of the 3th International Conference on Artificial Intelligence and Law (ICAIL 1993)*. ACM, 175–184. https://doi.org/10.1145/112646.112667

[13] W.N. Hohfeld. 1913. Some Fundamental Legal Conceptions as Applied in Judicial Reasoning. *Yale Law Journal* 23(1) (1913), 59–64.

[14] Amani Abu Jabal, Maryam Davari, Elisa Bertino, Christian Makaya, Seraphin Calo, Dinesh Verma, Alessandra Russo, and Christopher Williams. 2019. Methods and Tools for Policy Analysis. *ACM Comput. Surv.* 51, 6, Article 121 (Feb. 2019), 35 pages. https://doi.org/10.1145/3295749

[15] Andrew J. I. Jones and Marek Sergot. 1996. A Formal Characterisation of Institutionalised Power. *Logic Journal of the IGPL* 4, 3 (06 1996), 427–443. https://doi.org/10.1093/jigpal/4.3.427

[16] Robert A. Kowalski and Marek J. Sergot. 1986. A Logic-based Calculus of Events. *New Gener. Comput.* 4, 1 (1986), 67–95. https://doi.org/10.1007/BF03037383

[17] S. Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. (2008).

[18] Donald Nute. 2003. Defeasible Logic. In *Web Knowledge Management and Decision Support*, Oskar Bartenstein, Ulrich Geske, Markus Hannebauer, and Osamu Yoshie (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 151–169.

[19] Council of the EU. 2016. General Data Protection Regulation.

[20] Julian Padget, Emad Elakehal, Tingting Li, and Marina De Vos. 2016. *InstAL: An Institutional Action Language.* Law, Governance and Technology Series, Vol. 30. Springer Verlag, 101.

[21] Alessandra Russo, Rob Miller, Bashar Nuseibeh, and Jeff Kramer. 2002. An Abductive Approach for Analysing Event-Based Requirements Specifications. In *Logic Programming, 18th International Conference, ICLP 2002, Copenhagen, Denmark, July 29 - August 1, 2002, Proceedings (Lecture Notes in Computer Science, Vol. 2401)*, Peter J. Stuckey (Ed.). Springer, 22–37. https://doi.org/10.1007/3-540-45619-8_3

[22] Michele Ruta, Floriano Scioscia, Saverio Ieva, Giovanna Capurso, Agnese Pinto, and Eugenio Di Sciascio. 2018. A Blockchain Infrastructure for the Semantic Web of Things. In *Proceedings of the 26th Italian Symposium on Advanced Database Systems (CEUR Workshop Proceedings, Vol. 2161)*. CEUR-WS.org.

[23] Michele Ruta, Floriano Scioscia, Saverio Ieva, Giovanna Capurso, and Eugenio Di Sciascio. 2017. Semantic Blockchain to Improve Scalability in the Internet of Things. *Open Journal of Internet of Things* 3 (2017), 46–61.

[24] Fariba Sadri and Robert A. Kowalski. 1995. Variants of the Event Calculus. In *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, Leon Sterling (Ed.). MIT Press, 67–81.

[25] F. Schrans, D. Hails, A. Harkness, S. Drossopoulou, and S. Eisenbach. 2019. Flint for Safer Smart Contracts. https://arxiv.org/pdf/1904.06534.pdf.

[26] J.R Searle. 1996. *The construction of social reality.* Penguin Books.

[27] Pablo Lemela Seijas, Alexander Nemish, David Smith, and Simon Thompson. 2020. Marlowe: implementing and analysing financial contracts on blockchain. In *Workshop on Trusted Smart Contracts (Financial Cryptography 2020)*.

[28] Sepehr Sharifi, Alireza Parvizimosaed, Daniel Amyot, Luigi Logrippo, and John Mylopoulos. 2020. Symboleo: Towards a Specification Language for Legal Contracts. In *28th IEEE Int. Requirements Engineering Conf. (RE'20)*. IEEE.

[29] OASIS Standard. 2013. eXtensible Access Control Markup Language (XACML) Version 3.0. http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html.

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

[31] T.M. van Engers and R. van Doesburg. 2015. At Your Service, On the Definition of Services from Sources of Law. In *Proceedings of the 15th International Conference on Artificial Intelligence and Law (ICAIL 2015)*. ACM, 221–0225. https://doi.org/10.1145/2746090.2746115

[32] Daniel Davis Wood. 2014. Ethereum: a secure decentralised generalised transaction ledger.

[33] Huan Zhou, Xue Ouyang, Jinshu Su, Cees de Laat, and Zhiming Zhao. 2019. Enforcing trustworthy cloud SLA with witnesses: A game theory-based model using smart contracts. *Concurrency and Computation: Practice and Experience* (2019), e5511. https://doi.org/10.1002/cpe.5511

[34] Mirko Zichichi, Michele Contu, Stefano Ferretti, and Víctor Rodríguez-Doncel. 2020. Ensuring Personal Data Anonymity in Data Marketplaces through Sensing-as-a-Service and Distributed Ledger. In *Proceedings of the 3rd Distributed Ledger Technology Workshop Co-located with ITASEC (CEUR Workshop Proceedings, Vol. 2580)*, Franco Chiaraluce and Leonardo Mostarda (Eds.). CEUR-WS.org.