Towards a generic communication protocol for normative actor systems

Philo Decroos
philo0508@hotmail.com

July 2, 2023, 66 pages

Academic supervisor:	dr. L. Thomas van Binsbergen
	l.t.vanbinsbergen@uva.nl
Daily supervisor:	Lu-Chi Liu
	l.liu@uva.nl
Research group:	Complex Cyber Infrastructure (CCI)
	https://cci-research.nl/

Universiteit van Amsterdam
Faculteit der Natuurwetenschappen, Wiskunde en Informatica
Master Software Engineering

Abstract

Software systems are more ubiquitous in human societies than ever before. With this increasing digitisation of society, software systems are also increasingly subject to norms and regulations. Enforcement of these norms in software systems is currently largely manual. Normative domain specific languages (DSLs) have been developed in order to create a direct connection between the norms and the regulated systems, reducing cost and manual labour, and enhancing compliance. However, usage of these languages is still experimental, and tooling needs to mature to achieve the necessary robustness for usage in production systems. Because the regulated systems are often highly distributed, abstraction and standardisation in communication is desired, which is not provided by general-purpose communication protocols because these do not include normative concepts in their message semantics. In this thesis, we design a generic communication protocol for normative actor systems in which normative DSLs are used to regulate other actors. We do so by categorising existing normative systems based on their enforcement strategies, both automated and using human intervention. We introduce an approach with normative actors divided into specific roles, and define requirements based on the communication patterns we identify. We then design a formally defined messaging framework, and incrementally create a protocol that facilitates each pattern of communication. By implementing a proof of concept using the eFLINT normative DSL [1] and multiple case studies, we show that our protocol supports complex enforcement strategies, satisfying all the requirements distilled from existing systems. We conclude that our protocol's contribution to abstraction and standardisation can improve usability and stability of normative actor systems, although it is crucial to recognise the need for further rigorous evaluation in real-world, production-grade environments.

Contents

Chapter 1

Introduction

For software systems, interpretation, monitoring, control and enforcement of policies, laws, regulations and contracts are currently manual to a large extent. Formalisation of these norms as executable specifications has potential to reduce the amount of manual labour needed for enforcing norms on regulated software systems. By creating a direct connection between these systems and executable norm specifications, labour, costs, and complexity can be decreased significantly [2]. The range of application domains for this type of technology is potentially very large: from governments to businesses, computer systems are everywhere, and many are subject to norms like the European GDPR legislation, for example.

An advantage of formalisation of norms, is that it can increase trust between parties of the agreements, as the interpretation of legal texts is made explicit beforehand, instead of needing human interpretation every time the norms are applied to test for violations [2]. When having norms directly available to software systems, it is possible to evaluate beforehand (*ex-ante*) whether a potential action would violate a norm. This can lead to a compliance increase, because ex-ante evaluation could prevent violations that would currently only be evaluated ex-post, when legal professionals judge the potential offence.

The current trend towards the application of predictive machine learning models in law enforcement, while promising in terms of operational efficiency and predictive capabilities, poses significant ethical, legal, and sociotechnical challenges [3]. These systems may introduce biases embedded in training data, leading to algorithmic discrimination that disproportionately affects marginalised populations. Furthermore, the opacity of these models, often referred to as the 'black box' problem, creates issues of accountability and explainability. This lack of transparency can impede the due process and may infringe upon individuals' rights to presumed innocence, explanation, and appeal.

Some of the problems observed in the usage of predictive models seem inherent to this approach and thus hard to overcome. Executable norm specifications using declarative, normative domain specific languages (DSLs) potentially provide an alternative to predictive models for automating norm enforcement. They do not exhibit the predictive, 'black box' nature, and have the potential to instead automate enforcement by monitoring for violations. When decisions are made by such norm specifications, they have the capability of providing full traces of the steps that lead to this decision.

1.1 Problem statement

Executable norm specifications are not yet in such a state of maturity that they are widely used in practice. Many of the languages capable of implementing these specifications were developed recently, and they are often still in an experimental phase. To work towards usage of these languages in real-world systems, language specifications and tooling need to mature.

Multiple use cases for normative DSLs have been identified so far. There is proven potential for both ex-ante enforcement and ex-post enforcement [1, 2]. In many of these use cases, the governed systems are highly distributed and correctness of communication is vital. Currently, the structure of this communication is often tailored towards a specific use case, and DSL statements are passed around directly. However, if the normative actor approach is to have potential for development of large applications, abstraction and standardisation of communication are necessary to increase usability and stability.

To create this abstraction, a high-level communication protocol is desirable that includes normative semantics in its messages. Using existing protocols and messages devoid of this feature is not suitable for this goal, because to provide significant value, the protocol message exchanges and appropriate responses should be motivated by these normative concepts.

1.1.1 Research questions

In this thesis, we aim to create standardisation in communication in normative actor systems by formulating a messaging framework with a corresponding protocol specification. To do so, we will answer the following research questions:

> **RQ1**: *"Which types of systems can be identified in which compliance with norms is currently enforced, either automatically or manually?"*
>
> **RQ2**: *"How can normative actors be integrated in each type of system, and which patterns of communication between these actors emerge from the observed enforcement strategies?"*
>
> **RQ3**: *"What are the necessary components of a protocol for communication with normative actors with which the identified communication patterns can be facilitated in a generic way?"*

1.2 Research method

Multiple languages have been developed for formalising norms in an executable manner [1, 4–8]. The language that is used in this project is eFLINT [1]. Because eFLINT contains a read-eval-print loop (REPL) interpreter [2], its statements can be evaluated and executed one by one, allowing for dynamic, on-the-go interpretation of actions and events. This allows for eFLINT to be used in actor-based systems, which makes it an ideal candidate for researching communication in normative distributed systems.

We will conduct a literature survey in order to answer RQ1, creating an overview of enforcement strategies in normative systems. Using this overview, we will distil and describe communication patterns from the different enforcement strategies that we see across these systems (RQ2). From this overview of required communication capabilities, we first design a formally defined messaging framework. Using these messages, we will design multiple protocol components, which, when combined, will follow into a generic protocol specification (RQ3). Finally, we will create a proof of concept (PoC) implementation of the protocol in an actor system using eFLINT, and evaluate its effectiveness in facilitating all identified communication patterns.

1.3 Contributions

With this thesis, we make the following contributions:

1. **Analysis and categorisation of norm enforcement.** The first contribution of this project will be a description of which structural types of normative systems exist, how norms are currently enforced in these systems, and which different types of enforcement can be identified.
2. **Analysis of patterns when introducing normative actors.** The second result of this project will be an analysis of how normative actors can be deployed in each identified category of normative systems. The focus will be on which patterns of communication arise from the different types of enforcement and the positioning of normative actors in these systems.
3. **A messaging framework specially designed for normative actor systems.** The third contribution will be a formally defined framework of messages with fixed semantics, that will make up the protocol components.
4. **A protocol design for generic communication in normative actor systems.** Using the message framework, we will design a high-level generic communication protocol that facilitates all identified enforcement patterns.
5. **A proof of concept (PoC) implementation of protocol components.** We will implement a PoC implementation of the protocol to evaluate its effectiveness and usability across the identified categories of normative systems.

1.4 Outline

In Chapter 2 we will provide a theoretical framework that introduces the various concepts that this thesis builds on. Chapter 3 describes the various patterns that were observed in our literature review of normative systems and formulates communication requirements based on them. It also defines a set of actor roles based on the patterns, which will later be used as parties in the protocol. Chapter 4 describes the individual message semantics that will be the building blocks of the protocol. Chapter 5 lays out the actual design of the protocol components for each communication pattern, and makes a generalisation step from these components. In Chapter 6 we describe the implementation of our case studies, and evaluate our protocol's effectiveness based on usability for the simulated cases, relating back to requirements distilled from the observed patterns. These results are then discussed in Chapter 7. Chapter 8 contains the work related to this thesis. Finally, we present our concluding remarks in Chapter 9.

Chapter 2

Background

2.1 Norms and normative systems

Norms and policies are a fundamental element of human societies. Normative systems are everywhere, and they influence human behaviour constantly. Examples are governments introducing norms by means of laws and regulations, businesses and people concluding contracts with each other, and organisations drafting internal policies for their employees.

2.1.1 Definitions

There is quite some discussion about what the correct definition of a norm is. A large literature study conducted by Legros and Cislaghi [9] attempted to find consensus in literature about social norms. Roughly, consensus exists that norms are proscriptive or prescriptive, are related to behaviour and describe a socially acceptable way of living that is shared by members of a group or society. In this project, we will assume a broad definition of norms as posed by Sandholtz [10], where we characterise norms as equivalent to similar terms like ‘laws’, ‘policies’, or ‘rules’:

> “[Norms are] standards of appropriate behaviour for actors with a given identity”.

When referring to ‘normative systems’, we use the definition given by Carmo and Jones [11], who explicitly include the notion of agents and their behaviour in the system:

> “Sets of agents whose interactions are norm-governed; the norms prescribe how the agents ideally should and should not behave”.

Following these definitions, we see how ubiquitous normative systems are. Any system that contains multiple interacting agents and is governed by norms can be described as a normative system. These systems range from human societies, countries with laws, companies with internal policies, friend groups where social norms influence behaviour; to software systems that, for example, process data of consumers under regulation of the GDPR. In this last example, following the definition given by Carmo and Jones [11], both the software components and the people in the system (the consumer as well as the people operating the software) are agents in the same normative system, and their behaviour is regulated by the norms posed by the GDPR.

2.1.2 Hohfeld’s framework of legal relations

An influential framework for describing normative systems was developed by Hohfeld [12]. Hohfeld gave a new definition to the concepts of rights, privilege, power and duty to eliminate the ambiguity he observed in the way legal professionals used these terms. Central to this definition is the concept of ‘normative relations’: Hohfeld argues that right and duty are concepts that must always be described as a relation between individuals, and not as a position that a single person holds. Therefore, to accurately describe a right or a duty, the one must always be matched with a claim about the other. This leads to a conceptual equivalence: if Bob has a right against Alice, this relation can also be described as Alice having a duty to fulfil Bob’s right. All normative concepts that have this form of correlation are:

- **right and duty**, as described above;
- **privilege and no-right**: this relation can be seen as the opposite of the right-duty correlation. If Bob has the privilege to do something, Alice does *not* have the right to stop him from doing it;
- **power and liability**: this relation regards the legal relationships themselves rather than ordinary actions in the system. If Bob has a power against Alice, he has control over some legal relation between them: Alice is then liable (or 'bound to') to any changes Bob makes to this relation;
- **immunity and disability**: this relation is then again the opposite of the previous: if Bob is immune to the legal power of Alice, then Alice does not have the power to change relations between them (disability).

Using this form of expressing normative relations, the concept of 'liberty' can be expressed as 'the absence of both rights and duties'. Bob is free if he has no duties to perform actions or to refrain from them; and if no other individuals have a right to either Bob acting or Bob not acting.

Using Hohfeld's framework, normative systems can thus be described as a network of these normative relations between each pair of actors in the system. Clearly, these networks are not static in real systems: since rights and duties are relations, actions executed on one side of the relation can influence the other. Examples of these actions are the exercising of a power or failing to comply with a duty, which could have an effect on individuals that are the correlating party in the power-liability and right-duty relations respectively. So, when describing normative systems, this network of normative relations is not enough: we also need to be able to describe the actions and events that can occur in these systems, and how they influence these relations. Both of these elements can be formalised: to formally describe normative relations, a formal description can be given of the concepts of rights and duties, for example using a form of deontic logic [13]. Event calculus [14] is often used for formalising actions, events and their effects.

2.2 Automated norm compliance

As the proportion of human society that takes place in software systems increases constantly, software systems also need to comply with the norms that regulate them. Norms are often represented as documents written in natural language. This form of representation makes understanding and maintaining the norms accessible to human readers. However, it introduces ambiguity and a lack of precision that is inherent to the use of natural language, making human interpretation play a central role in the understanding and enforcing of the norms in the systems that they regulate [15]. Thus, currently, in many software systems the process of compliance is manual to a large extent. However, it is desirable to create direct connections between the software and the norms that govern them. Accomplishing this can decrease labour, costs and complexity needed for norm compliance [1]. By deciding on an exact interpretation beforehand, compliance automation can increase trust between parties and also increase compliance because it is no longer dependent on (potentially flawed) human intervention.

2.2.1 Regimentation

When norms are automatically complied with in currently existing systems, often they have been interpreted and taken into account at design time (the *regimentation* approach), where compliance is forced by not allowing violations by design. In this case, norm compliance is often embedded into the code, which has several drawbacks. Regimentation makes maintaining and changing the norms difficult, as they are scattered throughout the implementation of entire systems. Also, the interpretation of the norms is not made explicit, and the lack of a separate formal representation means the implementation of the norms themselves cannot be verified in isolation.

2.2.2 Enforcement

While regimentation guarantees norm compliance, a system designer might not want to avoid norm violations all together. Castelfranchi [16] argues that norm violations can be functional, especially in systems that have a society-like structure (multi-agent systems, distributed user-centred systems). If we remove the possibility for violations, the concept of a norm has no significant meaning any more: to the system agents, compliance is the only reality [17]. Also, regimentation is simply not possible in many systems. When the designer of a system does not design every interacting part of it, norm compliance of other system components cannot be guaranteed at design time. An well-known example of a type of system where this problem exists is open multi-agent systems (MAS), but it can occur in any type of system where multiple components operate together. An open API server might want to limit access to certain endpoints; since its designer cannot program all clients to not use these endpoints, it needs a different way of facilitating norm compliance. In cases where regimentation is not possible, or where we aim for our system components to interact freely, but regulated by norms, the norms need to be *enforced*.

Two different patterns of enforcement in normative systems can be identified: *ex-ante* enforcement and *ex-post* enforcement. We will briefly explain both patterns, and in the sections afterwards, we will provide some examples of how these patterns are currently applied in existing systems.

Ex-ante enforcement

Ex-ante enforcement is a pattern of enforcement that aims to avoid violations occurring [18]. To use ex-ante enforcement, an actor in a normative system might check whether a certain action is permitted before executing it, or whether it has a duty to fulfil. In an open MAS, an agent could query a dedicated service to find out whether it is allowed to carry out an action. Based on this, it can choose to comply with the norms. For this type of ex-ante enforcement to be possible, it is thus necessary that the norms are available and understandable to the actors regulated by them. Similarly, a restricted API might analyse an incoming request and make a decision about whether the client is allowed to access before letting the request pass through to the target resource.

Ex-post enforcement

When the enforcing system does not have the power or ability to change the behaviour of the regulated actors, the enforcement pattern that is used is *ex-post* enforcement [18]. This pattern is most similar to what is often the case in real-world legal systems such as a nation's justice system or a company policy: the enforcing institutes cannot directly influence the behaviour of the people they regulate. In this case, legal action is taken after a violation has occurred, rather than before. This means that a monitoring system needs to be in place: an authority needs to detect either a prohibited action being performed or a duty not being fulfilled. After this detection, a sanction might be given to the violating actor. In criminal law for example, this monitoring might be executed by a police force, where a judge imposes a sanction upon detection of a violation.

Most software systems that use automated enforcement currently apply the ex-ante pattern (mainly access control and related features). However, besides the fact that in some systems ex-ante is not possible (open MAS is an example, where agents can join a system without the system knowing anything about its internal functioning), ex-post also has advantages over ex-ante for certain situations. It allows more autonomy for actors in a system than ex-ante enforcement does. This is vital for complex social systems like MAS [16, 19]. In systems that work closely with a distributed human network, like software used by banks, hospitals or governments, we might not want to limit the freedom that employees have in their work. Rather, using ex-post enforcement, these systems could monitor actions taken in the systems and notify the employees of possible violations or duties, while leaving their autonomy untouched. Finally, ex-post enforcement can be needed because norms are variable in nature: they might be conflicting with each other, and their interpretations can change after they have been applied. These complexities cannot be captured by using ex-ante enforcement alone.

2.2.3 Examples of existing enforcement implementations

Access control

An example of ex-ante norm enforcement that is currently widely used is access control. As described before, the creator of a protected resource is often not the creator of all software systems that try to access it. Thus, enforcement is needed to make sure that not just anyone has access. Often, access control is implemented by a layer around the resource that monitors the incoming traffic and determines whether it is allowed to access. This decision can, for example, be based on authentication, certain attributes of the requesting party, or the role that it has within the system [20].

The most dominant paradigm for more advanced access control systems is attribute-based access control (ABAC) [20]. Especially for large, distributed, service-oriented architectures, the ABAC model is popular and effective. A widely used implementation of the ABAC paradigm is XACML [21]. XACML provides both an XML-based language to formalise access policies, and an architecture with different services that can be used to implement a robust access control system.

Smart contracts

Smart contracts are digital, executable implementations of legal actions according to a contract or agreement, often implemented on blockchain technology [22]. They consist of computer programs written in dedicated languages (e.g. Solidity [5]), in which contract clauses are specified that are automatically executed when predefined conditions are met. Because of this, smart contracts eliminate the need for intermediaries such as legal or financial entities, and allow for *peer-to-peer* enforcement of an agreement. Due to their autonomous, self-executing nature, smart contracts are a form of ex-ante enforcement, as they do not allow for violations. The blockchain's immutability ensures that once the terms of the smart contract are set, they cannot be altered or tampered with.

Multi-Agent Systems

Multi-agent systems (MAS) are systems where autonomous agents concurrently interact with each other. An agent's implementation is often based on the beliefs-desires-intentions (BDI) model, which intuitively maps to human behaviour. This makes agent-based systems highly suitable for modelling of social systems [23]. Next to modelling, it can be used to implement highly concurrent, 'computational social systems', like digital marketplaces or other data sharing infrastructures in production environments.

There is much research to norm enforcement in MAS. Since autonomy is an important quality of agents, and programming them in such a way that they cannot violate the norms (regimentation) hurts this autonomy, enforcement is a popular way to keep the system's behaviour as desired. Also, regimentation is only possible if the set of agents that can join a system is limited to correctly specified agents. In open multi-agent systems, where any agent can join the system, this is impossible. In this situation, enforcement is the only feasible option.

To implement enforcement for agent systems, the agent behaviour needs to be monitored. In case of norm violations, sanctions could be applied to violating agents. Examples of sanctions that can be implemented are fines, reputation damage, or temporary disabling of certain capabilities the agents have within the system [17]. Since this model of enforcement allows agents to violate norms, this is an example of how ex-post enforcement is currently implemented in software systems.

2.3 Normative specification languages

Recent research towards automated norm enforcement has resulted in the development of multiple domain specific languages (DSLs) for formal, executable specification of norms [1, 4–8]. Many of these languages include the notions of truth statements, deontic logic (permissions, obligations) and some form of event calculus [14]. Most of these DSLs are declarative in nature, although imperative ones also exist (Solidity [5] for example). We focus our attention to declarative DSLs, since they are more suited for our goals with this project.

By using a declarative DSL for formal norm specification, a form of model-driven engineering is introduced to norm compliance in system development. Instead of the algorithmic approach of embedding this compliance in the code directly, a more abstract model is created that describes the interpretation of the norms in a declarative way. This way, the norms are specified separately from the application itself. This makes it more straightforward to inspect and maintain the currently used interpretation of a set of norms, which is especially useful when norms change frequently. The fact that representation using a DSL is formal also means that the specifications are verifiable in isolation from their application. Lastly, representation using a declarative DSL often more closely resembles the way norms are represented in natural language, making it more accessible to domain professionals who do not have a programming background.

2.3.1 eFLINT

A recently developed DSL for norm specification is eFLINT, created by Binsbergen *et al.* [1]. eFLINT is suitable for specifying a wide variety of different norms, and uses the theoretical framework of normative relations developed by Hohfeld [12] that can be changed over time by actions and events. eFLINT contains a read-eval-print loop (REPL) interpreter, which means that during execution it can receive statements to dynamically update its belief base and the norms themselves; this makes it highly suitable for usage in dynamic settings like actor systems. As far as we know, eFLINT is the only normative DSL that has this explicit focus on actor systems, including a proven actor implementation by its creators. Therefore, eFLINT has been selected as the DSL we will use as a basis for our project, but the communication concepts introduced in this project could be applied to all other similar languages. An overview of eFLINT and its features is given below.

Structure

An eFLINT program consists of two parts: a *specification*, which formalises an interpretation of the applicable norms, and a *scenario* that describes the progression of the system it reasons about. Using these two elements, that are based on event calculus [14], eFLINT captures both aspects of the framework of normative relations introduced by Hohfeld. Normative relations are expressed by 'Facts' and 'Duties', where 'Acts' and 'Events' allow these relations to be changed over time. Examples of both a specification and a scenario can be seen in Figure 2.1.

```
Fact student        Identified by String
Fact assignment     Identified by String
Fact teacher        Identified by String

Fact homework       Identified by
                    student * assignment * teacher
Fact homework-due   Identified by homework

Duty finish-homework
  Holder          student
  Claimant        teacher
  Related to      assignment
  Holds when      homework()
  Violated when   homework-due()

Act submit-homework
  Actor           student
  Recipient       teacher
  Related to      assignment
  Terminates      homework()
  Holds when      homework()

// initial domain
Fact student    Identified by Alice
Fact teacher    Identified by Bob
Fact assignment Identified by Essay
```

(a) An eFLINT specification

```
// homework assigned
+homework(Alice, Essay, Bob).

// homework due, duty violated
+homework-due(homework(Alice, Essay, Bob)).

// query will return 'true'
?Violated(finish-homework).

// homework submitted, duty is terminated
submit-homework(Alice, Bob, Essay).
```

(b) An eFLINT scenario

Figure 2.1: An example of an eFLINT specification (a) that specifies norms about a student having a duty to submit homework to a teacher; and an eFLINT scenario (b) that models a series of events that can cause a violation of this duty.

The specification will often be written by domain experts who interpret the norms and formalise them, whether or not with the help of specific development tools. The scenario will then be generated by the system the specification governs. The REPL interpreter for eFLINT evaluates the statements one by one and provides immediate response about possible violations.

Facts

Facts are the basis of an eFLINT program, and they can be compared to propositions in propositional logic. The eFLINT specification defines Fact-types, which can be instantiated to a Fact in the knowledge base of the eFLINT program. Fact instances are three-valued propositions: they can take the values `True`, `False`, or they can be absent from the knowledge base (unknown value). By default, eFLINT assumes the closed-world principle: unknown Facts are treated as if they hold false.

The line `Fact student Identified by String` from Figure 2.1 creates a Fact-type `student`, which is identified by a string when instantiated. A student can then be instantiated with the `+` symbol, after which the instance holds true. For an existing instance, `-` can be used to make it hold false, and ~ can be used to remove the instance from the knowledge base (see Listing 2.1).

```
Fact student Identified by String  // Create fact-type

+student(Alice).  // Instantiate a student 'Alice'
-student(Alice).  // student(Alice) holds false
~student(Alice).  // Remove instance student(Alice) from knowledge base
```

Listing 2.1: Operations to manipulate Fact instances in eFLINT

Fact-types can have literal instances (`String` or `Int`), but they can also be *composite types*: in this case, the instance of the type is a tuple of other instances. Two examples from Figure 2.1 can be seen in Listing 2.2.

```
// Composite fact-types
Fact homework      Identified by student * assignment * teacher
Fact homework-due  Identified by homework

+homework(Alice, Essay, Bob).  // Instantiate homework for Alice
+homework-due(homework(Alice, Essay, Bob)).  // Mark homework as due
```

Listing 2.2: Composite types in eFLINT

Besides using `+`, `-` and ~ to manipulate Fact instances, their truth values can also be derived using the `Holds when` clause. So, instead of explicitly marking the homework as due in the scenario, we can derive it from whether a deadline has passed:

```
Fact deadline-passed Identified by homework

// Derived fact-type
Fact homework-due  Identified by homework
  Holds when deadline-passed(homework)

+homework(Alice, Essay, Bob).  // Instantiate homework for Alice
+deadline-passed(homework(Alice, Essay, Bob)).  // homework-due will now also hold true
```

Listing 2.3: Derivation rules for Fact-types in eFLINT

This may not seem like the most useful operation in this example, since we still need to set the `deadline-passed` instance. However, `Holds when` allows composite boolean expressions, and is especially powerful when combined with more advanced eFLINT features like `Foreach`, `Forall`, `Exists`, `Sum` or `Count` [1]. A discussion of the exact functioning of these features is outside the scope of this background section.

Acts and Events

Act-type declarations, similarly to Fact-types, describe instances, that in this case can be executed and have effects upon execution. Review the example from Figure 2.1 in Listing 2.4.

```
Act submit-homework
  Actor          student
  Recipient      teacher
  Related to     assignment
  Terminates     homework()
  Holds when     homework()
```

Listing 2.4: Specification of Act-types in eFLINT

Here, when a student executes the act `submit-homework`, the effect of the action is that the `homework` fact holds false. Acts can also terminate duties directly. Acts can be violated: note the `Holds when` clause that defines when an act is *enabled*. When there is no homework assigned, the act `submit-homework` should not be executed. If this happens, it will cause a violation.

Events are similar to Acts, but they do not have an actor or recipient. They can be used to model system events that are not specifically executed by one actor, and can have the same effects as Acts.

Duties

The final core concept of eFLINT we will discuss is the Duty. Revisit again the duty-type from Figure 2.1 in Listing 2.5.

```
Duty finish-homework
  Holder          student
  Claimant        teacher
  Related to      assignment
  Holds when      homework()
  Violated when   homework-due()
```

Listing 2.5: Specification of a Duty-type in eFLINT

Duties are the final ingredient to accurately represent the legal relations defined by Hohfeld. They describe a duty relation between a holder actor and a claimant actor in the system. In the example case, `finish-homework` is a duty from a student to a teacher, related to an assignment. For each student-teacher-assignment combination, it is created when a homework fact exists (note the `Holds when`). It is violated when the duty still exists and `homework-due` holds true. Thus, implicitly, the duty for a student is to execute `submit-homework`, which terminates the duty, before `homework-due` becomes true.

Open types

As noted before, eFLINT assumes the closed-world principle by default: if a fact that is not in the knowledge base is evaluated, it will hold false. However, a fact can be marked as *open*: when an unknown fact of an open type is evaluated, eFLINT raises an exception. This can be used to trigger an information update: when a fact is necessary to make a decision, the system can implement ways to query other knowledge bases to get its value before making a decision. An example of marking a fact as open is the following:

```
Open Fact maximum-lines Identified by assignment
```

If now for example we were to further condition the `submit-homework` act by whether the line count is below the maximum for that assignment, eFLINT would throw an exception if there is no value for `maximum-lines`. It can thus be assured that all required information is present before eFLINT decides.

2.4 The actor model

The actor model is a model of computation that is used as an abstraction for structuring concurrent systems [24]. In this model, the *actor* is the primitive for computation. An actor can send messages to other actors, create new actors and respond to messages it receives with internal actions. Message sending is the only way actors can communicate. They do not share internal state with each other. Actors are identified with an address that is used to send messages. Thus, an actor can only send messages to actors of which it knows the address: either it created the actor, or it received the address via a message.

The actor model is greatly suitable for this project since it provides the correct amount of abstraction to reason about communication without needing to know anything specific about the underlying systems. Because the actor system is highly generic, and only implements concurrency management, it can be applied to any concurrent system with multiple components. The actor model could be interpreted as a programming paradigm similar to OOP with more extreme encapsulation; but we can also model a distributed network of system components, even including humans, without changing the implementation. The actor model can be used to implement multi-agent systems (MAS), which feature the same concurrency level. The key difference is that MAS agents use a more specific form of computation, often involving some form of beliefs-desires-intentions (BDI), modelled after human behaviour. Actors do not have such requirements about their internal functioning.

We will use a well-known implementation of the actor model, *Akka* [25], that can be used in both Java and Scala. The abstraction level described before is entirely provided by Akka. It is an actor framework that has proven its usage in production systems. Because of this, we can evaluate simulations using our proof-of-concept implementation without concern about applicability to real production systems on implementation level.

2.5 Speech acts

Because monitoring and norm enforcement often happen in distributed systems, where multiple components communicate and separate actors provide normative services, communication is vital in integrating these concepts. To communicate normative concepts, components need to have understanding of normative message semantics, which will be a large part of the contribution of this thesis. The communication protocol we have developed uses a set of messages based on *speech acts* (the full language specification will be laid out in Chapter 4).

Speech acts are a concept from the domain of linguistics [26, 27]. When an individual speaks, this can be solely with the goal of giving certain information. However, when we speak we often intend more than this: by speaking we can also perform actions. For example, when we ask someone "would you pass me the salt?", we are conveying the fact that we would like to have the salt, but we are also initiating an action in the form of a request for someone to bring the salt to us.

In refining Austin's [27] initial conceptualisation of speech acts, Searle [28] established five distinct categories of speech acts:

- **Assertives**: these are statements of fact that can be true or false, such as "The sky is blue."
- **Directives**: these aim to get the listener to do something, such as "Please close the door."
- **Commissives**: these commit the speaker to a future action, such as "I will call you tomorrow."
- **Expressives**: these express the speaker's psychological state, such as "I'm sorry for your loss."
- **Declaratives**: these change the state of affairs in the world, such as "I now pronounce you man and wife."

Singh [29] adds the categories of *permissives* (e.g. "You may go now") and *prohibitives* (e.g. "Do not enter this room."), which one could argue fall under the category of directives. However, they specialise the concept of a directive speech act, and especially in the context of normative systems it can be useful to treat them as distinct categories.

Speech acts are often used to formalise message semantics, especially in MAS. By formalising our message types based on these speech acts, we can embed semantics in the message type itself, irrespective of the content of the message, and assign intuitive meaning and intention to message types.

Based on speech acts, multiple communication languages have been developed for usage in agent systems. Examples are KQML [30] and FIPA-ACL [31]. When agents use these languages to send messages to each other, speech acts are used to give exact semantics to the messages that are sent and which effect they are intended to have. Most of these languages use the *mentalistic* approach, in which the meaning of a message is semantically defined in terms of beliefs-desires-intentions (BDI) of the sending agent. An alternative to the mentalistic approach is *social semantics*, which is an approach introduced by Singh [29] that bases its semantics on the public notion of *commitments*. We will discuss these concepts in more detail in Chapter 4 when introducing our protocol message types.

Chapter 3

Communication patterns in normative systems

Communication is vital in a well-functioning normative actor system. In order to reach our goal of creating a generic, abstract protocol to facilitate this, we need knowledge of which kinds of communication are necessary to reach the desired functionality, and which information must be passed around.

In this chapter we will describe the patterns of communication that we observe in existing normative systems, both automated and non-automated. We will describe each pattern with regards to which communication is necessary to come to the goal of correct norm enforcement for a system that exhibits this pattern. Across communication patterns, certain roles with specific properties can be identified that are generally consistent. We will introduce these roles here, and later use them as parties in the specification of the communication protocol, where each role has their own properties and appropriate behaviour. By using a fixed set of roles, we aim to give structure to system implementations that use the protocol, but we also allow enough flexibility to support all (combinations of) patterns by freely assigning roles to actors. The role that an actor has will decide which sequence of actions it takes in a protocol exchange, and therefore define its normative communication behaviour in the system. We will show how we approached this for our PoC implementation in Chapter 6.

This chapter will lead to concrete requirements of communication for normative actor systems, structured by actor role. In the evaluation of our protocol components in Chapter 6, we will explicitly refer back to these requirements to conclude how well our proof of concept satisfies them. Requirements will be numbered in a structured form, such as “R1.1”. The sub-numbering is used to identify which pattern lead to which requirement. Table 3.1 at the end of this chapter shows an overview of the protocol requirements.

3.1 Patterns

We identified a total of six patterns. In this project, a pattern is defined as a specific approach to norm enforcement that leads to specific needs for communication. These patterns have been observed in existing normative systems. Some of them can already be found in existing software systems and are entirely automated; others are patterns of enforcement that are currently executed by human actors in the system, but have potential to be automated by using normative actors.

3.1.1 The access control pattern

Access control is one of the most widely observed automated enforcing patterns. In this form of ex-ante enforcement, a resource is protected by an access control layer, that regulates which entities are allowed to access this resource. The resource must therefore not be accessible directly without going through the access control system. Actors that intend to access a protected resource submit a request for access to a decision point in the system, which decides whether to allow access based on the systems policies. In our case, these policies are the eFLINT specifications.

There are many different implementations available for access control, using different paradigms. Throughout this project, we will use XACML [21] as an example of a real-world implementation of this pattern. XACML is one of the most robust and widely used access control frameworks using attribute-based access control (ABAC). The framework includes a distributed architecture that provides guidelines for implementation in production systems (see Figure 3.1). Because ABAC is a flexible and powerful form of access control, we will model our proof-of-concept actor-based implementation on this architecture. When describing the general properties of the access control pattern we will explain the approach that XACML takes to make our comparisons more concrete. However, our further contributions are focused on the requirements of communication, not on a specific form of access control. Thus, the protocol component we will develop is not limited to XACML or ABAC, but can be used to implement any paradigm.

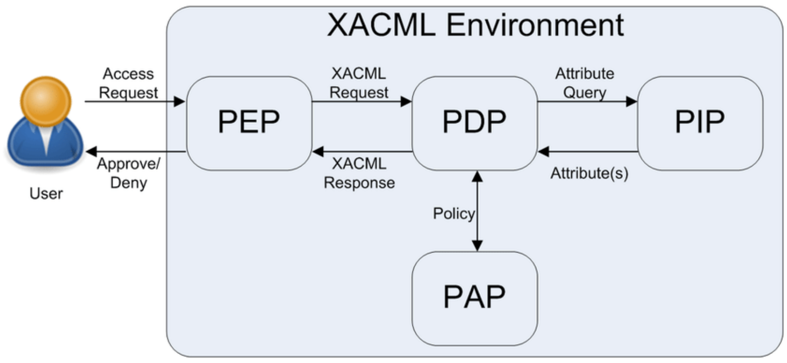

Figure 3.1: The access control architecture proposed by XACML [32]. This image shows a simplified version that omits some components: the components shown are most relevant to our discussion of the access control pattern.

Requirements

The first requirement for access control is that an entity in the system needs to be able to send an access request to some available entry point (R1.1). In the case of XACML, this entry point is called the Policy Enforcement Point (PEP). The PEP translates the source request (e.g. an HTTP request from a browser) into a XACML request containing user attributes, and the target resource and action.

When the access control layer receives a request, it needs to be able to make a decision about whether access should be granted or not (R1.2). In XACML, this reasoning is executed by the Policy Decision Point (PDP), which gets the translated request forwarded from the PEP. To make the decision whether to allow access, the PDP needs all relevant attributes of both the requesting subject and the resource. If this information is incomplete, it calls the Policy Information Point (PIP) to fetch it. The PIP component thus needs access to some application-specific data: for example, whether the user that is requesting access is in the 'admin' role. Consequently, our normative system also needs access to the data in the system: which subjects exist, which resources exist and what their relevant attributes are (R1.3).

After a decision is made, it needs to be communicated back to the requesting entity. This means that either access is permitted (R1.4) or forbidden (R1.5). XACML keeps strict separation of concerns by making the PEP the only entity that talks to the outside world, so this is also the entity that sends this response back. Finally, the access control system needs a way to actually grant access to the resource when it is permitted (R1.6). In XACML, the PEP component sits between the client and the resource, and forwards the original request to the resource if the access is permitted.

3.1.2 The monitoring pattern

Monitoring is a communication pattern that is central to more complex forms of (automated) norm enforcement. It is not in itself a form of enforcement, but it is a pattern that can play a role in multiple enforcement strategies, which is why we discuss it separately.

A first use case for monitoring is simply keeping all parts of the system up-to-date on relevant information about the system that is regulated. For example, if within our access control pattern a regulated system actor requests to execute some action on some resource, the regulating actors need to be up-to-date about the actor's properties and the resources that exist to decide whether this action is allowed. This is a requirement we already discussed for access control specifically, and it can be satisfied in different ways, but monitoring is good candidate for this. For other patterns of enforcement, mainly ex-post enforcement (see Section 3.1.3), monitoring is an essential part of the enforcement strategy.

In order to achieve effective monitoring of a software system, we need to introduce some complexity to the communication flow: simply sending and receiving messages in the philosophy of the actor model will no longer suffice. Especially if we want to monitor actors that might violate the norms, we cannot trust them to always update us about what actions they take; a monitoring implementation needs to ensure that no relevant communication can escape it.

The implementation of a monitoring system thus needs to facilitate a robust way to force all communication to be accessible to the monitoring actor(s). A system component needs to be in place that sees all communication in the system. How to achieve this is dependent on which technology is used. In our actor-based implementation using the Akka framework, we will modify the default message sending procedure to duplicate each message and forward it to all subscribed monitors. In a web-based implementation, middleware in the system web servers could distribute communication to subscribed monitors similarly.

To create full flexibility, we propose an approach where monitoring actors can subscribe to specific parts of this communication, of which they will be notified by the component that intercepts the communication. This way the designer of the system, including its regulatory actors, is free to combine multiple isolated monitoring actors that handle specific parts of the communication. In Chapter 6 we will show how we achieve this robust and flexible form of monitoring in our Akka implementation.

Requirements

As discussed, the first requirement for the monitoring pattern is that the regulating actors need free access to the interactions that happen in the system they regulate (R2.1). Secondly, if the communication that is monitored uses different message semantics than defined within our protocol, these interactions need to be understood and translated to normative concepts that have meaning within the confinement of our protocol specification (R2.2).

3.1.3 The ex-post enforcement pattern

Now that we introduced the first pattern that uses ex-ante enforcement and the general communication pattern of monitoring, we extend our discussion about monitoring by discussing one of its main use cases: ex-post enforcement. In the ex-post enforcement pattern, there are no attempts to prevent actors from violating the norms. Rather, we observe the behaviour of the system actors, and when violations occur we have the option of taking action. These actions taken after a violation has been observed are intended to have such an effect on the regulated actors that it will be in their own interest to comply with the norms. Clearly, the monitoring pattern discussed in the previous section is the basis for ex-post enforcement: in order to enforce the norms after an action was taken, we need to know about the action and understand it so that it can be evaluated against the norms.

Even though most software systems currently use regimentation or ex-ante enforcement, ex-post enforcement using monitoring is a wide-spread form of enforcement outside of software systems. It is the default pattern we see in human societies (governments, companies, social structures) [33]. Because humans cannot be influenced in an “ex-ante” way, at least not as rigidly as programmers can influence the programs they write, we need monitoring to enforce societal norms. This monitoring is often executed by people, for example a police force or managers. This pattern is also common in ensuring compliance when using software: to make sure that the usage of software follows the norms, the behaviour of the software and the users themselves are monitored by people.

Existing examples

Besides the observation that ex-post enforcement is ubiquitous in human social environments, some software systems already use automated ex-post enforcement. An example is cloud monitoring [34, 35], where metrics are collected from cloud services, and monitoring services can trigger automated responses to violations or security breaches, such as alerting administrators, isolating affected resources, or even shutting down compromised instances.

Another example is spectrum sharing [36, 37]: this is a technique used in wireless communication systems to allow multiple users or devices to access the radio frequency spectrum efficiently. In this context, ex-post enforcement is achieved by monitoring the usage of the shared spectrum and identifying violations, such as unauthorised access or interference. Upon detecting a violation, the monitoring system can take corrective measures, such as reassigning frequencies or alerting the responsible authorities.

Finally, ex-post enforcement is an active research subject in open multi-agent systems (MAS) [19, 38]. Open multi-agent systems consist of agents that are designed to operate autonomously and can come from different sources, possessing varied goals and behaviours. They are often used to model social systems, which, as discussed earlier, are inherently bound to ex-post enforcement because the social entities cannot be programmed to follow the norms.

Active and passive enforcement

Broadly, we identify two variants for actions the normative system might take when observing violations: *passive* and *active* enforcement. While both approaches utilise the same monitoring mechanism, they differ in the degree of intervention employed in response to violations. In passive enforcement, the regulatory system solely focuses on notifying other relevant actors about observed violations. It relies on other system actors to take appropriate actions in response to the information they receive. For example, in a file-sharing system, passive enforcement might involve monitoring for unauthorised access attempts and alerting the system administrators or resource owners about these incidents. The administrators or resource owners would then be responsible for taking further action, such as revoking access permissions or conducting an investigation. On the other hand, active enforcement includes actual interventions in the system, such as applying sanctions. Using the file-sharing system example, a sanction upon detecting unauthorised access could be to automatically suspend the offending user's account.

Requirements

For ex-post enforcement, the basis is the set of requirements posed for the monitoring pattern. To use monitoring for ex-post enforcement, the extracted protocol messages need to be evaluated by the norm specification for their compliance with the norms (R3.1). When violations occur, for passive enforcement, the normative actors need a way to notify the relevant actors about a violation (R3.2); for active enforcement, the normative system should be able to intervene in the system, for example by changing some data, blocking some action or handing out a sanction (R3.3).

3.1.4 The duty monitoring pattern

The first form of ex-ante enforcement we discussed, access control, is a form of ex-ante enforcement that involves a regulatory service preventing actors from executing actions they are not allowed to. A different form of ex-ante enforcement is what we will call duty monitoring. This is a form of ex-ante enforcement that can be used in systems where a monitoring system is in place, to help the system actors comply with the norms by actively informing them about their duties. Thus, the system uses the same pattern of monitoring as discussed before, but besides acting upon violations, it will also distribute information about duties that arise or terminate because some action was taken or some information was updated. This is directly related to how normative DSLs like eFLINT and Symboleo [4] work: when some action is taken in the system (e.g. homework being assigned to a student), it can directly enable a duty (the student must submit the homework).

Requirements

Besides using monitoring for detecting violations as we described in the ex-post pattern, for duty monitoring we need our system to be able to monitor duties coming into existence. The requirements for monitoring itself do not change, but the handling of it is extended by also distributing information about duties. Thus, when a system event leads to a duty becoming active, the normative actors need a way to notify the relevant system actors (R4.1). Similarly, when an existing duty is terminated, we also want to be able to notify the relevant actors (R4.2).

3.1.5 The query pattern

Next to duty monitoring, in which the regulatory services actively update the system actors of their normative positions, we define the query pattern. In this pattern, the system actors proactively take measures themselves to make sure that they comply with the norms. In a system where ex-post enforcement and monitoring are used, it is in the system actors' best interest to comply, because they intend to avoid sanctions. In order to facilitate this, especially when duty monitoring is absent, system actors might intend to query for their normative positions. This pattern thus includes information requests such as "do I have a duty I need to fulfil?", or "am I allowed to execute this action?". The latter question is of course also central in access control, so we will not elaborate on it extensively again, but it can also be relevant outside of access control when violations are allowed by design.

Similarly to the patterns that use monitoring, this pattern is especially commonplace in real-world social systems: individuals are aware of the existence of rules and the consequences of violating them. As a result, it is in their best interest to comply with these rules. Similarly, when someone has a duty towards us, we might want to know so that we can take appropriate action if the duty is not fulfilled, such as protesting to authorities or reminding the other party. To make informed decisions in this regard, we need to be aware of our normative positions. Individuals might read law documents or learn from other individuals in order to achieve this; in our actor systems, actors will be able to query the governing normative actors for this information.

Requirements

System actors should be able to query their regulating actor system about which duties exist in which they are a party (either they have a duty, or someone else has a duty towards them) (R5.1). Because it might not always be acceptable for actors to query any information they want, we also specify the need for implementing some 'meta-policy'. With this policy, the receiving party of a request can determine whether they should answer it or not. The regulating system should then be able to reject a request if this is not the case (R5.2). If this is the case, the regulating system needs a way to communicate to other actors whether any of the requested duties exist, including the specifics of the duties (R5.3). Note that queries about actions are covered by the requirements posed in the access control pattern.

3.1.6 The information fetch pattern

When a normative system is expected to make a decision about the legality of some action, it needs access to all up-to-date information that might be relevant for the outcome of the decision. Monitoring can be an important strategy for achieving this. However, a regulating actor might still find itself missing relevant information upon receiving a request to assess a normative situation. The information fetch pattern mitigates this, by providing the reasoning actors with infrastructure to fetch missing data, possibly from sources outside the system itself. This pattern is also observed in XACML, which uses its designated PIP component for this purpose, but it can also be applicable outside of access control.

Requirements

In order to achieve this pattern, the norm specification that is used for evaluation needs functionality to mark information as missing and halt the decision making process until this information is retrieved (R6.1). Secondly, the reasoning actor needs to be able to send requests for specific information to the applicable data source (R6.2). This data source needs to be able to understand these requests and correctly translate them to its underlying data structure to fetch the information (R6.3). Finally, the data source should be able to send back this information (R6.4) after which the reasoning actor should be able to resume or retry the original evaluation (R6.5).

3.2 Actor roles

From the patterns that we described, we introduce structure in the requirements by defining roles for the actors that will take part in the protocol, and define their behaviour in it. We observe similarities and common features in the communication patterns that we can generalise, and we do so with these roles. We also introduce a separation of concerns, similar to how XACML achieves this by using its dedicated components. Specific features of actor roles can also yield specific requirements for their implementation, which we will briefly introduce here, and elaborate upon later.

3.2.1 Application role

The application role is the default role we assign to actors that are part of the system regulated by the norms. In access control for example, both the actors that ask for access and the actors that manage the protected resources are application actors. We cannot say much in general about the characteristics of actors of this role, since their functioning is (almost) entirely application specific. However, if application actors are to participate in normative communication, they will need to implement some of our messages. An application actor that wants to access a protected resource, for example, will need to be able to send a request for access and understand the response. In general, all actors that do not have any of the other roles are application actors.

3.2.2 Enforcer role

The enforcer role can be seen as the central point of authority in the system that is regulated, and the direct point of contact between the application actors and the other regulating actors. For example, application actors can query the enforcer to learn about their normative positions, and the enforcer can also notify application actors about their duties.

An enforcer actor has knowledge about the system it regulates that is necessary to bridge the gap between the generality of our protocol and the specificity of the system it is deployed in. The enforcer is therefore also the actor that takes action upon violation: it notifies the relevant actors, and can execute sanctions appropriately. This is also why we separate the enforcer from the actors that actually query eFLINT: by having this separation of concerns, the reasoner's only responsibility (see Section 3.2.3) is to receive protocol messages, evaluate with eFLINT and return the result to the enforcer. A system designer may not always want to notify every application actor of a violation they commit, and sanctions are inherently application specific; the enforcer actor can be used to program appropriate actions in reaction to updates received from the reasoner actor.

3.2.3 Reasoner role

An actor of the reasoner role is the actor that reasons about the norms. This is the actor that the eFLINT knowledge base and REPL run in. The reasoner receives queries from enforcer actors and updates the enforcers about normative positions, violations and answers to queries. It also receives information from the monitors (see Section 3.2.4) about the system, and uses its knowledge base and a specification of the norms (in our case in eFLINT) to make decisions about the legality of the behaviour of the system.

The reasoner can thus be seen as the central point within the system of regulatory actors: it makes all the decisions about the norms, and only communicates with other regulatory actors, not directly with application actors. The enforcers and the monitors then form the bridge between the reasoner and the application, and handle all actions that need application knowledge: translating communication, controlling who has access to query, and firing the mechanisms for intervening upon violations.

Another reason for this separation is that the reasoner needs to know who to trust. For the reasoner to be able to make decisions about norm compliance in the system it needs to be continuously updated about the state of the system. However, if the application actors can talk directly to the reasoner, it would need to implement a way to determine 'who to believe'. Application actors could namely try to inform the reasoner incorrectly about the system if it benefits their normative positions, and different actors could try to persuade the reasoner of conflicting beliefs. Thus, this responsibility is delegated to actors of other roles, which we assign absolute authority. The reasoner blindly trusts the enforcer and the monitor, and these actors implement application specific ways to update the reasoner about the system state and to make sure that this state is accurate.

In this project we will limit the amount of reasoner actors to one in every case study that we do. When multiple reasoners are present in a single system, we have multiple eFLINT knowledge bases and norm specifications, which could conflict with each other and need a mechanism for synchronisation. Resolution for this problem is out of scope for this project.

eFLINT evaluation

The reasoner actors will be the actors that internally run a normative DSL (eFLINT in our case) to evaluate norm compliance. One of the main goals of our generalisation into a protocol, is that applications that need norm enforcement do not require writing eFLINT statements directly embedded into business logic. Instead, communication will be confined to a strictly defined set of messages with fixed semantics. Thus, in order to evaluate these, the reasoner actor needs a way to translate protocol messages to statements that can be interpreted by the language that does the normative reasoning. Internal to a reasoner actor, there needs to be some adaptor that does this translation. In Chapter 4 we will introduce the set of messages and how they relate to eFLINT, from which the approach for this translation will follow quite easily. In our PoC implementation (Chapter 6) of the protocol in Scala, we will create such an adaptor for eFLINT and discuss how this can be done.

When the reasoner is able to translate messages to eFLINT and parse its responses correctly, the rest of the evaluation process is trivial for this actor: the actual evaluation happens entirely within the normative language itself. It will respond with answers to queries and notify the reasoner when violations occur. The most important aspect of the evaluation process is therefore whether the eFLINT specification has been written correctly.

3.2.4 Monitor role

Actors of the monitor role play a central part in multiple of the enforcement patterns. They monitor the application actors and send updates to the reasoner. The events and communication that the monitor observes are obviously application specific, so the monitor needs application specific implementation as well. It needs to be able to understand the events in the system, and implement a way to translate these events to messages that can be understood by the reasoner actor, which will always be fully generic.

Also, monitors need to be facilitated by their enclosing implementation in order to function: if they do not receive all relevant communication in the system, they cannot function correctly. At system design time this requirement needs to be considered, with the possibility of introducing subscriptions for different parts of the system communication, each of which could be connected to a dedicated monitor actor.

3.2.5 Information role

The information role is reserved for actors that can be used by the reasoner to serve as an information point. When the reasoner decides that it is missing required information to make a decision, it can query these actors to fetch the required information. These actors have an internal data representation, and thus need to implement a way to translate incoming requests to this representation, fetch the required data, and then embed this data into response protocol messages.

3.3 Requirements overview

Table 3.1 shows a full overview of the requirements that were distilled from the evaluation of the patterns we have found in real normative systems. Now that we have also defined the roles for the actors that will take part in the protocol, we slightly rephrase our requirements to take into account which roles should fulfil which requirements.

Table 3.1: The full list of requirements that will be used to evaluate our protocol.

Ref.	Pattern	Requirement
R1.1	Access control	An application actor needs to be able to send a request to an enforcer actor for access to a protected resource.
R1.2	Access control	The reasoner actor running the norm specification needs to be able to make decisions about any incoming request and send the decision to the enforcer.
R1.3	Access control	The reasoner actor needs to be continuously updated about the state of the system and all relevant attributes of the requesting subject and the resources.

R1.4	Access control	Based on the decision, the enforcer needs to be able to send back to the application actor that its access request was permitted.
R1.5	Access control	Based on the decision, the enforcer needs to be able to send back to the application actor that its access request was forbidden.
R1.6	Access control	When access is permitted, the enforcer actor needs a way to actually facilitate the requesting actor to access the resource.
R2.1	Monitoring	The system needs to facilitate a way for the monitoring actor to receive all relevant events that happen in the system.
R2.2	Monitoring	The monitoring actor needs to be able to interpret the events that happen in the system, and translate them to protocol messages.
R3.1	Ex-post enforcement	The monitoring actor should send updates to the reasoner actor in order to evaluate their compliance status.
R3.2	Ex-post enforcement	When the reasoner decides that a violation has occurred, it needs a way to communicate this violation to the relevant actor.
R3.3	Ex-post enforcement	When the reasoner decides that a violation has occurred, the system needs a way to intervene in the system by blocking actions or handing out sanctions.
R4.1	Duty monitoring	When a system event observed by a monitor actor leads to a duty becoming active, the system needs a way to notify the relevant application actors.
R4.2	Duty monitoring	When a system event observed by a monitor actor leads to a duty being terminated, the system needs a way to notify the relevant application actors.
R5.1	Query	Application actors should be able to query the enforcer actor about which duties either they have, or someone has towards them.
R5.2	Query	The enforcer actor should be able to determine whether an application actor is allowed to request information about a duty, and reject a request if this is not the case.
R5.3	Query	The enforcer actor should be able to query the reasoner actor for existing duties, and inform the requesting application actor of these duties.
R6.1	Information fetch	The norm specification that runs within the reasoner actor should be able to mark information as missing and halt the evaluation.
R6.2	Information fetch	When the norm specification marks information as missing, the reasoner actor should be able to send a request to an information actor to fetch this information.
R6.3	Information fetch	The information actor should be able to translate the information requests to its internal data layer and fetch the requested data.
R6.4	Information fetch	The information actor should send the result of the data query back to the reasoner.
R6.5	Information fetch	The reasoner actor should be able to resume or retry the original evaluation after an information request was fulfilled.

Chapter 4

The normative communication language

In the previous chapter we introduced the communication patterns that can be found in normative systems, and which requirements they lead to for our protocol. In this chapter, we introduce the message types that will make up the protocol, and what their semantics are. By embedding normative concepts directly in the semantics of the message types, we eliminate imprecision and ambiguity in the use of these messages, simplifying the implementation process. We will describe our messages using formal notation, to further enhance exactness, allow reuse of the introduced syntax in our protocol specifications, and to facilitate potential further use cases with regards to formal verification and code generation.

4.1 Message semantics: speech acts and commitments

We design our communication language similar to existing message semantics used in agent communication languages (ACLs) like the ones introduced in [29–31, 39, 40]. As most of these existing ACLs, we use the concept of speech acts as a basis for our messages. The theory of speech acts treats language and communication as actions. The core concept is the *illocution* of communication, which is the intention of a message (what the speaker intends to accomplish with it) [29]. Its use in software systems is also motivated by the fact that speech acts provide an adequate approach to human communication, allowing the use of a single specification to describe interactions between humans and software agents [39].

As introduced in Section 2.5, some established ACLs like KQML [30] and FIPA-ACL [31] use the beliefs-desires-intentions (BDI) framework to define the message semantics. In this approach, the semantics of a message are “mentalistic": their semantics are defined by what the sender believes or intends to be the case [29]. To illustrate this, take the speech act *inform*. In the BDI approach, the semantics of the *inform(a)* message are that the sender ‘believes a’, or ‘has a in its knowledge base’. However, this is a mental concept private to the sender, which may not be verifiable by the receiver: the sender could be misleading the receiver to reach some other goal. Whether the sender actually believes a cannot be verified without access to its internal construction, which cannot be assumed in many normative systems.

4.1.1 Commitment

While specifying message semantics using the mentalistic approach has its benefits in many BDI based agent systems, the fact that internal behaviour is not accessible or verifiable in more open systems is problematic. Aligning our views on normative actor systems with real-world views on norm enforcement, we disregard the internal process of actors in these systems: the only relevant part of their behaviour is that which is visible from the outside. In other words, we are not interested in *why* an actor makes a decision (its internal process), but only in *what* it does and whether it follows the rules (its external behaviour). Therefore, we base our communication language on the commitment-based *social semantics* approach introduced by Singh [29] and further developed in [39, 40], as an alternative to the mentalistic approach. In this approach, message semantics are defined in *commitments*: *inform(a)* no longer means ‘I believe a’, but is now defined as ‘I commit to you that a is true’. Thus, the semantics do not depend on the internals of the sender: the fact that it sends the message creates a commitment that can be verified by some authority in the system.

To use commitments in our formal specifications, we use a notation that is mostly derived from the work of Fornara and Colombetti [39]. We define a commitment object C with the following fields:

- **State**: a commitment can have multiple states. An active commitment is indicated with the letter a. This is how most commitments begin. An active commitment can either become fulfilled (f) or violated (v). Besides active commitments, we can also define precommitments (p). These are commitments that will come into play when we define requests. Since system actors cannot simply create active commitments for other actors at will, when they request an actor to execute some action, they create a precommitment for them. The actor that receives the request is then allowed to execute the requested action, or to decline the request, which leads to the final commitment status, namely cancelled (c).
- **Debtor**: the actor that makes the commitment and can thus be held responsible for it.
- **Creditor**: the actor relative to which the commitment is made.
- **Content**: the content of the commitment. This is a propositional formula, in which the propositions are specifically defined *normative objects* (see the next section, 4.2), and messages that should be sent.

A commitment object is then used in our writings as $C(\mathit{state}, \mathit{debtor}, \mathit{creditor}, \mathit{content})$. In this project, we will use commitments only as a way to accurately and formally define the semantics of our messages. They have no direct function in the protocols or in the implementation of our PoC. However, since commitments are inherently public, they can potentially be used for verification purposes. Since the state changes of a commitment (e.g. from active to fulfilled) depend on whether the content of the commitment is ‘actually true’, in verification settings we could introduce a component in the system that contains an interpretation of these truth values and verifies the claims made by system actors. Explicitly keeping a list of commitments and updating their status can then provide insights into which actors behave correctly within the system. This can be used for testing a system, or even in production settings to monitor the correct behaviour of the actors.

4.2 The normative objects

To accurately capture the content of a message, we define four different objects we will use in our definitions. These objects relate closely to concepts from Hohfeld’s legal framework [12] that eFLINT and other normative DSLs are based on. This way, we keep translation to statements that can be used for evaluation by eFLINT straight-forward.

The proposition object

The proposition object P has the following fields:

- A type name, denoted $P.\mathit{type}$. If this type name corresponds to a Fact-type, Predicate-type or Invariant-type in the eFLINT specification(s), it can be used in a query;
- an instance, denoted $P.\mathit{instance}$, that uniquely identifies this proposition;
- a value, $P.\mathit{value} \in \{0, 1, \bot\}$, that represents the truth value of this proposition object. Here, the value $\bot$ means the truth value of the predicate is unknown.

The event object

The event object E has the following fields:

- A type name, denoted $E.\mathit{type}$, that can be used to translate events in the system to Event-types in eFLINT;
- a timestamp, denoted $E.t$ at which the event took place.

The act object

The act object A has the following fields:

- A type name, denoted $A.\mathit{type}$, that can be used to translate actions in the system to Act-types in eFLINT;
- a timestamp indicating the execution time of the action, denoted $A.t$;
- an actor, $A.\mathit{actor} \in \alpha$ that represents the actor that executed (or will execute) the action at $A.t$;

When an act object is used as a proposition, we consider it true if the action has been executed (or will be executed) at $A.t$, and false otherwise.

The duty object

The duty object D has the following fields:

- A type name, denoted $D.type$, that can be used to translate duties in the system to Duty-types in eFLINT;
- a holder actor $D.holder \in \alpha$;
- a claimant actor $D.claimant \in \alpha$;
- a terminating action $D.A_T$ that, when executed, terminates the duty;
- a violation proposition $D.P_V$ that, when it becomes true, makes the duty violated if it has not been terminated.
- $D.violated \in \{0, 1\}$ that signifies whether a duty is currently in a violated state.

4.3 Formal model

Normative communication inherently includes a notion of time (i.e. 'you must execute this action before this happens'). We use temporal logic to express this part of the semantics of a message. For this, we use a subset of Computation Tree Logic (CTL), largely based on the model used by Singh [29] to formalise his commitment based message semantics. CTL is a model of a system in a tree form, in which nodes are the states the system can be in, and the edges represent the ways that the system can evolve.

We start from Φ, which is a collection of atomic propositions, namely normative objects or messages. For messages, the truth value corresponds to whether the message was (or will be) sent or not. We define $M = \langle \alpha, \mathbf{S}, \prec, \mathbf{N}, \mathbf{R} \rangle$ as our formal model, with:

- α as the set of actors in the system;
- $\mathbf{S}$ as the set of states;
- $\prec \subseteq \mathbf{S} \times \mathbf{S}$ as a strict partial order, that signifies the temporal branching between states;
- $\mathbf{N} : \mathbf{S} \mapsto 2^{\Phi}$ as an interpretation over the propositions, that determines whether they are true or not in a given state;
- $\mathbf{R} : \mathbf{S} \mapsto \mathbf{P}$ as the real path that will be taken from a state.

Here, $\mathbf{P}$ is the set of all possible paths, derived from $\prec$. For a $t \in \mathbf{S}$, $\mathbf{P}_t$ is the set of paths starting from t. Now, for a formula p, $M \models_t p$ expresses 'M satisfies p at t' and $M \models_{P,t} p$ expresses 'M satisfies p at t along path P'.

$M1.$ $M \models_t p$ iff $p \in N(t)$, where $p \in \Phi$

$M2.$ $M \models_t \neg p$ iff $M \not\models_t p$

$M3.$ $M \models_t p \wedge q$ iff $M \models_t p$ and $M \models_t q$

$M4.$ $M \models_t p \vee q$ iff $M \models_t \neg(\neg p \wedge \neg q)$

$M5.$ $M \models_t \mathrm{E}p$ iff $(\exists P : P \in \mathbf{P}_t$ and $M \models_{P,t} p)$

$M6.$ $M \models_t \mathrm{R}p$ iff $M \models_{\mathbf{R}_t,t} p$

$M7.$ $M \models_{P,t} p\mathbf{U}q$ iff $(\exists t' : t \leq t'$ and $M \models_{P,t'} q$ and $(\forall t'' : t \leq t'' \leq t' \implies M \models_{P,t''} p)$

Thus, informally, $p\mathbf{U}q$ means 'q will eventually hold, and p holds until q holds' for a given path. $\mathrm{F}p$ is an abbreviation for $true\mathbf{U}p$, and thus means 'eventually p will hold' for a given path.

Every actor in the model has an internal knowledge base denoted K. The syntax $K_i(P)$ will be used to denote a lookup in the knowledge base of actor i for the value of proposition P. Finally, we will use $\rightsquigarrow$ to identify a causal relation internal to the norm specification. Actions or truth values of propositions can have consequences that are defined in eFLINT in our case. For example, the fact that an action was executed might lead to a violation: this is denoted as $A \rightsquigarrow V_A$, and thus implies that A is not allowed at $A.t$.

4.4 Message types

Using our formal model, we will now describe all messages we will use in our protocol specification, and formally define their semantics in terms of commitments. The messages we define are categorised as *assertive*, *directive*, *permissive* or *prohibitive* messages. Other categories of speech acts, i.e. *commissives* (e.g. promising) or *expressives* (e.g. wishing something) currently have no use within the patterns of communication we identified, so we do not define messages in these categories. The notation we will use for a message is $messageType(sender, receiver, content)$.

4.4.1 Assertives

The first class of messages we introduce are *assertives.* These are messages that convey information, and are used to describe the state and evolution of the system and its normative positions.

inform The first assertive message type is the *inform* message. It is used to inform another actor about a proposition value P. Its semantics are defined as follows:

$$\mathit{inform}(i, j, P) =_{def} C(a, i, j, P) \quad \text{with}\, i, j \in \alpha$$

Thus, when an actor i informs another actor j about P, it creates an active commitment towards j that the truth value of $P.instance$ is $P.value$ at the time of sending the message. The receiving actor can then choose to adopt the same truth value for P. Within our regulatory actor framework, whether an actor indeed does this depends on who the message comes from: reasoner actors, for example, will only accept these messages from trusted actors (an enforcer, monitor or information actor).

informAct When an action is executed, the system uses the *informAct* message. Here again, the sender is committing to the fact that the action was executed at time $A.t$. Note that the actor executing the action does not need to be i: actors can communicate about actions taken by other actors.

$$\mathit{informAct}(i, j, A) =_{def} C(a, i, j, A) \quad \text{with}\, i, j \in \alpha$$

informEvent The *informEvent* message is the same as *informAct*, but for system events, which are not carried out by a specific actor. Thus, we define *informEvent* as a commitment from the sending actor that event E has taken place at time $E.t$.

$$\mathit{informEvent}(i, j, E) =_{def} C(a, i, j, E) \quad \text{with}\, i, j \in \alpha$$

informDuty *informDuty* is the first message that contains specific normative semantics. Unlike the previous messages, where we are only informing about truth values or system events, with this message we can inform an actor about a normative relation to another actor. We define duties themselves as commitments as well, in this case commitments from the holder of the duty towards the claimant of the duty. The semantics of *informDuty* are as follows:

$$\mathit{informDuty}(i, j, D) =_{def} C(a, i, j, C(a, D.\mathit{holder}, D.\mathit{claimant}, \mathrm{R}(\neg D.P_V\ \mathbf{U}\ D.A_T))) \quad \text{with}\, i, j \in \alpha$$

So, the claim that a duty exists creates a commitment for the holder of that duty that $D.P_V$ will remain false until $D.A_T$ is executed. In other words, that the holder will execute the terminating action before the violation condition of the duty becomes true. Note that this is not a precommitment, because the holder of the duty is not free in its choice to accept the commitment: the duty is imposed by the norm specification of the system, and cannot be declined. This also means that the position of authority we assign to certain actor roles becomes important here. In actor systems where there is no explicit authority, we would not expect actors to just be able to create commitments for other actors by sending them that they have a duty. Thus, within the protocol we will limit the ability for sending these messages to the regulating actors, and system designers should make sure that this limit is enforced.

informViolatedDuty When the violation condition of a duty becomes true before it is terminated, the duty is violated. First, we observe that from the definition of the duty in the semantics of *informDuty*, we can conclude that if the violating proposition holds, but the terminating action was not executed, it implies that the original commitment has been violated:

$$(D.P_V \wedge \neg D.A_T) \implies C(v, D.\mathit{holder}, D.\mathit{claimant}, \mathrm{R}(\neg D.P_V\ \mathbf{U}\ D.A_T))$$

We can then communicate a duty violation using the *informViolatedDuty* message type. Its semantics are defined as follows:

$$\mathit{informViolatedDuty}(i,j,D) =_{def} C(a,i,j,D.P_V \wedge \neg D.A_T \wedge C(a,D.\mathit{holder},D.\mathit{claimant},\mathrm{R}D.A_T)) \quad \text{with } i,j \in \alpha$$

Thus, the sender commits to the receiver that $D.P_V$ has become true, but $D.A_T$ was not executed, which, as said, implies that the original commitment defining the duty was violated. Also, since violating a duty does not relieve the holder of the duty, this message creates a new commitment to execute the terminating action, but without the timing constraint of the violation proposition.

informDutyTerminated Eventually, a duty can be terminated. This can happen because A_T was executed, or because something else changed in the system that had the duty termination as a side effect. When a duty is terminated, it can already be in a violated state. The duty termination changes the state of the currently active commitment relating to the duty. The semantics of *informDutyTerminated* are as follows:

$$\mathit{informDutyTerminated}(i,j,D) =_{def} \begin{cases} C(a,i,j,C(f,D.\mathit{holder},D.\mathit{claimant},\mathrm{R}D.A_T)), & \mathit{if}\ D.\mathit{violated} \\ C(a,i,j,C(f,D.\mathit{holder},D.\mathit{claimant},\mathrm{R}(\neg D.P_V\ \mathbf{U}\ D.A_T))), & \mathit{otherwise} \end{cases} \quad \text{with } i,j \in \alpha$$

Thus, when the duty was not violated yet, the original commitment becomes fulfilled; if it was already violated, the commitment created upon violation becomes fulfilled.

informViolatedAct The message *informViolatedAct* conveys the fact that action A was executed, but this execution was not allowed, causing a violation V_A when evaluated against the norm base.

$$\mathit{informViolatedAct}(i,j,A) =_{def} C(a,i,j,A \wedge A \rightsquigarrow V_A) \quad \text{with } i,j \in \alpha$$

informViolatedInvariant *informViolatedInvariant* is a message that is used to communicate invalid state in the system: the fact that invariant proposition P is a violation by the definition of an invariant. An invariant is a proposition that is automatically in a violated state when its value is false.

$$\mathit{informViolatedInvariant}(i,j,P) =_{def} C(a,i,j,P \wedge P \rightsquigarrow V_I) \quad \text{with } i,j \in \alpha, P.\mathit{value} = 0$$

permitted When using the *permitted* message, the sender commits to the receiver that an action is permitted by the norms in the system at $A.t$. It is defined as follows:

$$\mathit{permitted}(i,j,A) =_{def} C(a,i,j,A \not\rightsquigarrow V_A) \quad \text{with } i,j \in \alpha$$

Note that this message is not a permissive message: it solely conveys the fact that A will not cause a violation. It is not a commitment to not prevent A from happening. For this actually permissive use case we use the *permit* message (see Section 4.4.3).

forbidden For prohibiting actions, we use the same distinction as between *permitted* and *permit*. Thus, the *forbidden* message creates a commitment to the fact that A is not allowed by the norms and would cause a violation if executed:

$$\mathit{forbidden}(i,j,A) =_{def} C(a,i,j,A \rightsquigarrow V_A) \quad \text{with } i,j \in \alpha$$

4.4.2 Directives

Next to assertive messages, we also define a class of *directives*. These messages do not only convey information about the state of the system, but intend to initiate an action by another actor. Here we will see the usage of precommitments: sending a request does not create a commitment for the sender, but for the receiver. Because the subject of the commitment is not the actor sending the message, we first create a precommitment, which can then either turn into an active commitment if the receiver accepts the request, or a cancelled commitment if the receiver does not accept the request.

request The message type *request* is a way to ask for another actor to *inform* about a proposition. Thus, it creates a precommitment for the receiving actor to reply with an *inform* message containing the truth value of P in its knowledge base, at some point in the future.

$$\mathit{request}(i,j,P) =_{\mathit{def}} C(p,j,i,\mathrm{RF}\ \mathit{inform}(j,i,K_j(P))) \quad \text{with } i,j \in \alpha$$

requestAct The message type *requestAct* is a a directive message that can be used to ask for permission. It is defined as a precommitment for the receiver of the message to respond with either a permissive or a prohibitive message that conveys whether the action A is allowed by the norms at time $A.t$.

$$\mathit{requestAct}(i,j,A) =_{\mathit{def}} C(p,j,i,\mathrm{RF}\ \mathit{message}(j,i,A))$$
$$\text{with } i,j \in \alpha,\quad \mathit{message} \in \{\mathit{permitted}, \mathit{forbidden}, \mathit{permit}, \mathit{forbid}\}$$

requestDuty Using the *requestDuty* message, an actor can query other actors for duty objects they might know about. It is defined as follows:

$$\mathit{requestDuty}(i,j,D) =_{\mathit{def}} C(p,j,i,\mathrm{RF}(\forall D' \in K_j(D)\ \ \mathit{informDuty}(j,i,D'))) \quad \text{with } i,j \in \alpha$$

So, this message is a request for the receiver to check their knowledge base for whether they know of any duties that currently exist. Since a duty is not a simple single-value object like a proposition is, a knowledge base lookup for a duty object for some actor $i \in \alpha$ is defined specifically as follows:

$$K_i(D) = \{D' \in K_i \mid D.\mathit{holder} = D'.\mathit{holder} \lor D.\mathit{claimant} = D'.\mathit{claimant}\}$$

Thus, informally, the *requestDuty* message is a request to send back all duties of which either the holder or the claimant equals the corresponding field in the duty object from the request.

4.4.3 Permissives and Prohibitives

The permissive and prohibitive messages are specific forms of directive messages, that convey the intention of the sender to influence the behaviour of the receiver in regards to compliance. This can be in response to a *requestAct* message, as an alternative to the previously discussed *permitted* and *forbidden.*

permit When the *permit* message is sent, the sender commits that they will not prevent A from happening in the future, and that it is therefore possible to execute it:

$$\mathit{permit}(i,j,A) =_{\mathit{def}} C(a,i,j,\mathrm{EF}A) \quad \text{with } i,j \in \alpha$$

forbid The opposite of *permit* is *forbid*, and this message creates an active commitment for the receiver to make sure A is not executed, as opposed to *forbidden* simply conveying the fact that an action is not allowed.

$$\mathit{forbid}(i,j,A) =_{\mathit{def}} C(a,j,i,\neg\mathrm{RF}A) \quad \text{with } i,j \in \alpha$$

reject The message type *reject* can be used to cancel precommitments: it conveys the fact that the receiver of a *request*, *requestAct* or *requestDuty* message does not intend to fulfil the request. So, for some precommitment about formula x, the semantics of the *reject* message are as follows:

$$\mathit{reject}(i,j,C(p,i,j,x)) =_{\mathit{def}} C(c,i,j,x) \quad \text{with } i,j \in \alpha$$

The reject message cannot really be seen as prohibitive, but more as a meta message type. It only has meaning towards existing commitments, and in a real system they need to be related to the request that created the precommitment in order to be functional. Thus, in real systems, and also in our implementation, instead of using a commitment object directly, we will send *reject* messages that contain the normative object (proposition, act or duty) that was also contained in the request. This is more useful to the original requester.

4.4.4 Appropriate responses

Logically, not every message type is a valid response to every other message type, regardless of which protocol they are used in. It does not make sense to reply *permit* to a request to *inform*. Thus, in Table 4.1 we provide an overview of appropriate responses to each message type.

Message type	**Appropriate responses**
request	*inform* *reject*
requestAct	*forbidden* *permitted* *forbid* *permit* *reject*
requestDuty	*informDuty* *reject*

Table 4.1: The full list of generally appropriate responses to each message type. Messages that are omitted here do not have any appropriate responses.

Chapter 5

Protocol design

Now that we have defined the message semantics used in our framework for normative actor systems, in this chapter we will define protocol components composed of these messages. These components will each facilitate a standard implementation of one of the patterns discussed in Chapter 3, aiming to satisfy all requirements. We will make a clear distinction between our protocol components that implement generic solutions, and parts of the functionality that will remain application specific upon implementation of the protocol in a real system.

5.1 Running example: real estate management application

We will develop our protocol incrementally on the basis of a hypothetical piece of software with components that together exhibit all of the patterns we based our protocol requirements on. This software is an application for the management of real estate. This example is suitable for usage as a running example because it is inherently a normative system: the data in this system is a representation of legal relations in the real world, and thereby regulated by norms. In this chapter, we will only use small examples from this application to make our discussions more concrete. We will not try to be complete or exhaustive yet. In Chapter 6, we will make an effort to demonstrate the effectiveness of the protocol with more elaborate use cases including implementation details.

The features of the example system are as follows. At the core is the management portal for the owner of the real estate. In this portal, the owner can upload data about buildings, tenants, rental agreements, documents, financial administration, and so forth. Through the software, communication with tenants of the real estate is facilitated. Tenants might need access to certain documents, or submit requests for repairs that are needed in their building. The application can be modelled as an actor system, where we view each tenant, the owner, and the database as actors that communicate with each other. These actors take the role of application actor, and we introduce our new normative actors to the system for regulation: an enforcer, a reasoner, and a monitor. We also assign the role of information actor to the database, so that the reasoner can query it directly, and we introduce an outside data source as a second information actor. See Figure 5.1 for a schematic overview of this actor representation of our running example application.

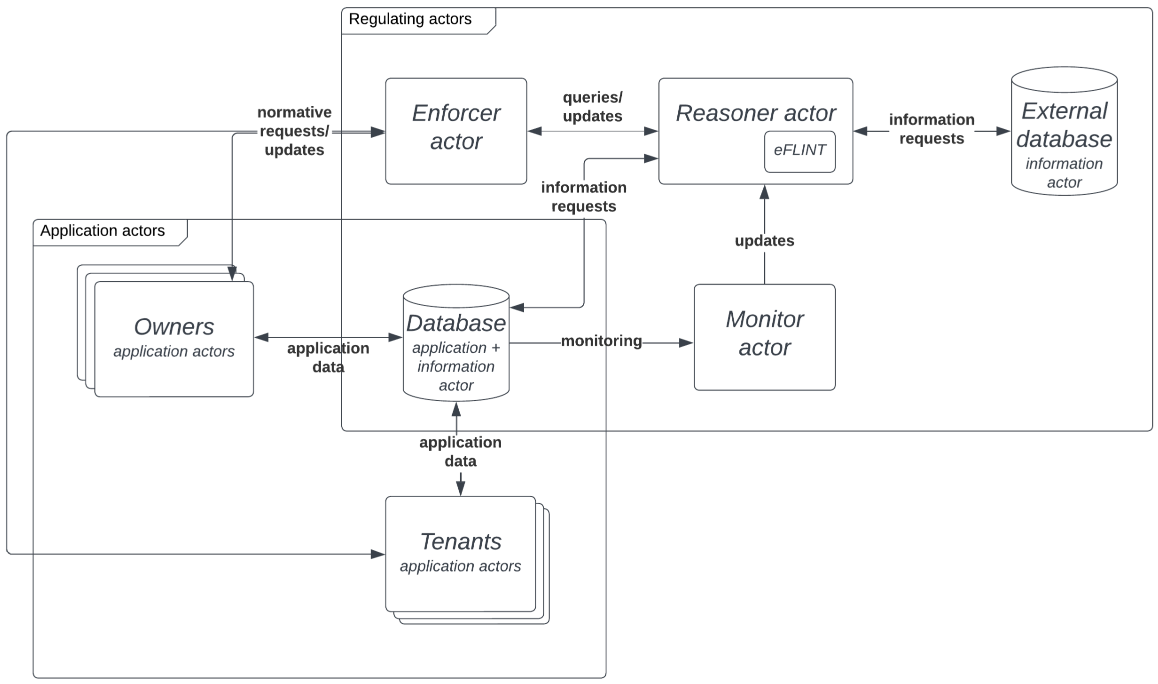

Figure 5.1: A schematic overview of the actor representation of our running example: a real estate management application that includes an owner's portal, a tenant's portal and a central database. The system is regulated by a set of normative actors: an enforcer as the central authority, a reasoner that contains the norm specification, a monitor that intercepts and translates system events, and an external database that can be used for fetching information needed for norm evaluation.

5.2 Pattern protocol components

For specifying which roles partake in a message exchange, we will use letters to denote an actor that has a specific role: a is an application actor, e is an enforcer actor, r is a reasoner actor, m is a monitor actor, and i is an information actor. When describing a protocol interface of a certain actor, the keyword *self* will be used to denote the address of the actor itself. We will use the syntax $N(...)$ as an evaluation of a normative object against the norms (and thus an eFLINT statement in our case). For example, $N(A)$ is an evaluation of an executed action A, that can cause an action violation, denoted $N(A) \rightsquigarrow V_A$. Other violations are duty violations V_D and invariant violations V_I. Next to evaluations, we can also query the norm specification, denoted $N(?...)$. So, $N(A)$ is the evaluation of an executed action against the norms, and $N(?A)$ is a query that returns whether A is currently allowed by the norms. Finally, we reuse the knowledge base lookup syntax from Chapter 4, where $K_i(P)$ is a lookup in the knowledge base of actor i for the value of proposition P.

For each component, we will define a listening loop for each actor role, and combine these into a state chart that represents the entire component. In these state charts, the nodes are states of the system, and branches indicate messages that can be sent in that state, through which the system moves to a different state. Listening loops will be defined in a structured pseudo-code-like form.

A listening loop looks as follows:

```
while listening
    if msg <- receive:
        switch msg
            case messageType1(...): ...
            case messageType2(...): ...
        end
end
```

Protocol 5.1: An example of the exact semantics of a listening loop.

Because this loop will look the same every time, we will abbreviate this syntax to the form seen in Protocol 5.2. If next to having a listening loop an actor is by itself able to start the protocol component sequence by sending the first message, this message will be shown at the top level before the listening loop starts. Of course, depending on the implementation, actors can implement asynchronousness and handle multiple protocol sequences and listening loops simultaneously. For simplicity, this is not reflected in these specifications.

To clearly make the distinction between operations that are generic and operations that are placeholders for application specific procedures, we will mark the latter in red in every specification.

```
send sequenceStartMessage(...)
listen
    case messageType1(...): app-specific procedure
    case messageType2(...): ...
end
```

Protocol 5.2: The abbreviated listening loop we will use from this point on. Application specific procedures will be marked in red.

5.2.1 The access control pattern

The first pattern we will devise a protocol component for is access control. In our hypothetical example application, a tenant might want to access a certain document that was uploaded to the system by their landlord. The norms that regulate the system include which tenant is allowed to access which document under which circumstances, and these norms are registered with the reasoner actor as an eFLINT specification. When the tenant initiates this action (through a UI), the application actor that runs the tenant portal will send a request for executing an action to the enforcer: $requestAct(a, e, A)$. Of course, A contains all information about the request. The actor then waits for response before proceeding. When it receives permission, it accesses the resource. This action is application specific and therefore not worked out within the protocol (see Protocol 5.3).

```
send requestAct(self,e,A)
listen
    case permit(e,self,A): access resource
    case forbid(e,self,A): sequence ended
    case reject(e,self,A): sequence ended
end
```

Protocol 5.3: The protocol component for an application actor implementing access control

In the access control pattern, the application actor is the only actor that can initiate a sequence. Thus, the other actors are in a listening state until they receive a message. The enforcer role receives the request from the application actor, and forwards it to the reasoner to ask for a decision. When the decision is made, the enforcer expects the reasoner to reply with a message indicating whether the action is allowed or not, after which it informs the application actor of the decision. Because of the requirements for access control, the enforcer uses *permit* and *forbid* towards the application actor, to commit to the fact that the action is not only forbidden, but will also be prevented. Since there is ordering in this exchange, the enforcer needs to keep some state: it listens for an answer to a specific request, and therefore only starts listening for reasoner responses once a request has been sent. After sending the decision back to the requesting actor, it is the responsibility of the enforcer to provide or block the actual access. This security procedure is handled in an application specific way. See Protocol 5.4 for the specification.

Since the enforcer is the point of authority over the application actors, this sequence can be further extended in an application specific way. For example, the enforcer might determine first whether it knows the requesting tenant and whether it is allowed to request anything at all. If not, it can send $reject(self, a, A)$ without including the reasoner actor to inform the application actor that its request will not be considered. This is the first example we see of how the enforcer is the real point of authority over the system: it decides who to allow access to the regulatory services and how to respond to input from eFLINT.

```
listen
    case requestAct(a,self,A):
        if request accepted
            send requestAct(self,r,A)
            listen
                case permit(r,self,A):
                    send permit(self,a,A)
                    allow access
                case forbid(r,self,A):
                    send forbid(self,a,A)
                    prevent access
            end
        else
            send reject(self,a,A)
end
```

Protocol 5.4: The protocol component for an enforcer actor implementing access control

The third and last actor role that is involved in the access control component is the reasoner actor that actually makes the decision about whether the tenant is allowed to access the document. It receives the request from the enforcer, translates the request and its content to an eFLINT query and sends it to its internal eFLINT actor to get a decision. For *requestAct*, this decision is either *permitted* or *forbidden*. See Protocol 5.5 for the specification.

```
listen
    case requestAct(e,self,A):
        if N(?A)
            send permitted(self,e,A)
        else
            send forbidden(self,e,A)
end
```

Protocol 5.5: The protocol component for a reasoner actor implementing access control

The individual listening loops of the three involved actor roles lead to the communication shown in the protocol state chart in Figure 5.2.

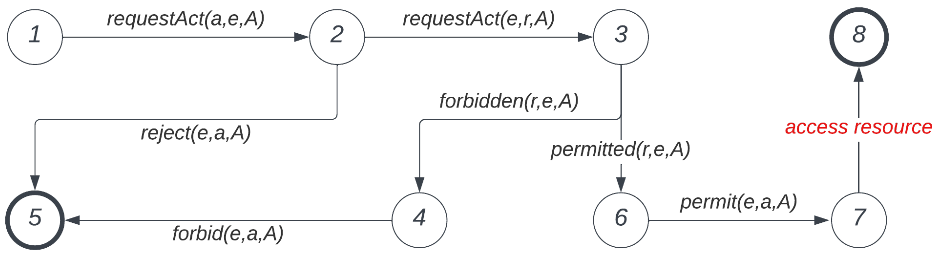

Figure 5.2: The state chart for the access control protocol component, that follows from the individual listening loops of the involved actor roles.

5.2.2 The ex-post enforcement pattern

For our ex-post pattern, we start by introducing the monitor actor's role in the protocol. As discussed before, the functionality of this actor role is for a large part application specific. Its task is to observe the events that it receives from the system, translate them to one of three possible message types, and send these messages to the reasoner actor in the system. We assume here that the streaming of information to the monitor actor is correctly set up, as this implementation falls outside of the protocol entirely. See Protocol 5.6 for the monitor's specification.

```
listen
    case msg:
        e <- translate(msg)
        switch e:
            case e is status update:
                send inform(self,r,P)
            case e is action:
                send informAct(self,r,A)
            case e is system event:
                send informEvent(self,r,E)
end
```

Protocol 5.6: The protocol component for a monitor actor in the ex-post enforcement pattern.

To illustrate using our running example application: events that could be received and translated by the monitor are communications between the other system components. For example: the owner of the real estate administers a rent raise or a contract termination in the system, which gets saved to the database. This data flow gets streamed to the monitor, and the monitor translates it to action objects A and sends an *informAct* message to the reasoner. An example of a status update could be an update to the registered income of the tenant, which might be relevant to the norms regarding their rental contract; this will be translated to an *inform* message containing a proposition P. Finally, we discern system events that are not executed by some actor, e.g. a monthly event that is triggered when the deadline has passed for all rent payments. Events are translated to an *informEvent* message with an event object E.

The reasoner receives all messages from the monitor, and evaluates them using its eFLINT specification. When eFLINT emits violations, the reasoner actor translates these back to messages, and sends them to the enforcer actor (see Protocol 5.7).

```
listen
    case inform(m,self,P):
        if N(P) $\rightsquigarrow V_D$:
            send informViolatedDuty(self,e,D)
        if N(P) $\rightsquigarrow V_I$:
            send informViolatedInvariant(self,e,P)
    case informEvent(m,self,E):
        if N(E) $\rightsquigarrow V_D$:
            send informViolatedDuty(self,e,D)
        if N(E) $\rightsquigarrow V_I$:
            send informViolatedInvariant(self,e,P)
    case informAct(m,self,A):
        if N(A) $\rightsquigarrow V_D$:
            send informViolatedDuty(self,e,D)
        if N(A) $\rightsquigarrow V_I$:
            send informViolatedInvariant(self,e,P)
        if N(A) $\rightsquigarrow V_A$:
            send informViolatedAct(self,e,A)
end
```

Protocol 5.7: The protocol component for a reasoner actor in the ex-post enforcement pattern.

Protocol 5.8 shows the specification of the enforcer actor in the ex-post patterns. It receives violations that occurred from the reasoner, and then has the authority to intervene in the system. This implementation will be application specific, and the system designers make the choice here between passive or active enforcement. For example, when the owner of the real estate raises the rent more than is allowed by law, the enforcer actor receives an *informViolatedAct* message, and can then choose to display this violation as a message to the owner (passive) or to escalate within the system, e.g. by notifying the tenant that they can file a complaint (active).

```
listen
    case informViolatedDuty(self,e,D):
        intervene
    case informViolatedInvariant(self,e,P):
        intervene
    case informViolatedAct(self,e,A):
        intervene
end
```

Protocol 5.8: The protocol component for an enforcer actor in the ex-post enforcement pattern.

The protocol specifications of the ex-post enforcement pattern lead to the protocol state chart shown in Figure 5.3.

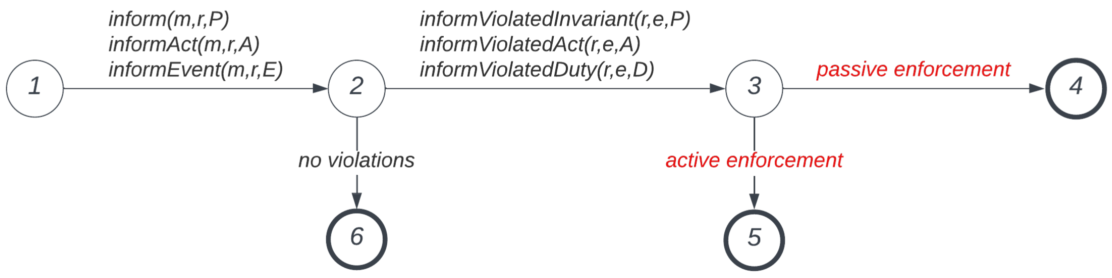

Figure 5.3: The state chart for the ex-post enforcement protocol component, that follows from the individual listening loops of the involved actor roles.

5.2.3 The duty monitoring pattern

To add duty monitoring functionality to our protocol, we will reuse the specification of the monitor actor introduced in the previous section. We then extend the specifications of the reasoner and enforcer actors, to not only monitor for violations, but also update relevant actors in the system about other normative concepts.

We discussed how ex-post enforcement can play a role in our example application. Now, we look at examples where duties come into existence within the system. For example, when a tenant submits a repair request in their portal, a duty arises for the owner to respond to it within a certain time span. When the monitor observes this action and forwards it to the reasoner, the effect of the evaluation of the message will not be a violation, but a created duty. Similarly, duties can be terminated (e.g. when the request is marked as handled). In this ex-ante pattern, we might want to notify the owner and tenant of these duty updates. Therefore we need to extend the protocol specification of the reasoner actor to facilitate this. This extension can be seen in Protocol 5.9.

```
listen
    case inform(m,self,P):
        if N(P)⇝ D:
            send informDuty(self,e,D)
        if N(P)⇝ ¬D:
            send informDutyTerminated(self,e,D)
    case informEvent(m,self,E):
        if N(E)⇝ D:
            send informDuty(self,e,D)
        if N(E)⇝ ¬D:
            send informDutyTerminated(self,e,D)
    case informAct(m,self,A):
        if N(A)⇝ D:
            send informDuty(self,e,D)
        if N(A)⇝ ¬D:
            send informDutyTerminated(self,e,D)
end
```

Protocol 5.9: The protocol component for a reasoner actor in the duty monitoring pattern.

Now, we must also extend the specification of the enforcer actor to receive these messages and act upon them. Again, similarly to ex-post enforcement, upon receiving such updates the enforcer actor uses an application specific implementation to decide how to act. A straight-forward example of how an enforcer might do this is to update the holder and claimant of a duty upon its creation, which will be sufficient for many systems. Assuming this behaviour, we can replace the handling of *informDuty* with the following:

```
send informDuty(self,D.holder,D)
send informDuty(self,D.claimant,D)
```

However, since we intend to keep our protocol generic, and systems might require different handling of duty updates (e.g. only updating the claimant and letting that actor take control of its rights), we mark the selection of recipients as application specific (see Protocol 5.10).

```
listen
    case informDuty(r,self,D):
        foreach a in relevant actors:
            send informDuty(self,a,D)
        end
    case informDutyTerminated(r,self,D):
        foreach a in relevant actors:
            send informDutyTerminated(self,a,D)
        end
end
```

Protocol 5.10: The protocol component for an enforcer actor in the duty monitoring pattern.

For completeness, we also add the specification for the application actors, which, if they intend to use the information fed to them through duty monitoring, need to listen for the relevant messages and process them. They can then make a risk assessment based on this information. See Protocol 5.11.

```
listen
    case informDuty(e,self,D):
        proceed with updated knowledge
    case informDutyTerminated(e,self,D):
        proceed with updated knowledge
end
```

Protocol 5.11: The protocol component for an application actor in the duty monitoring pattern.

The duty monitoring pattern is an extension of the ex-post enforcement pattern, so we combine the two into a single state chart starting from a monitored event (see Figure 5.4).

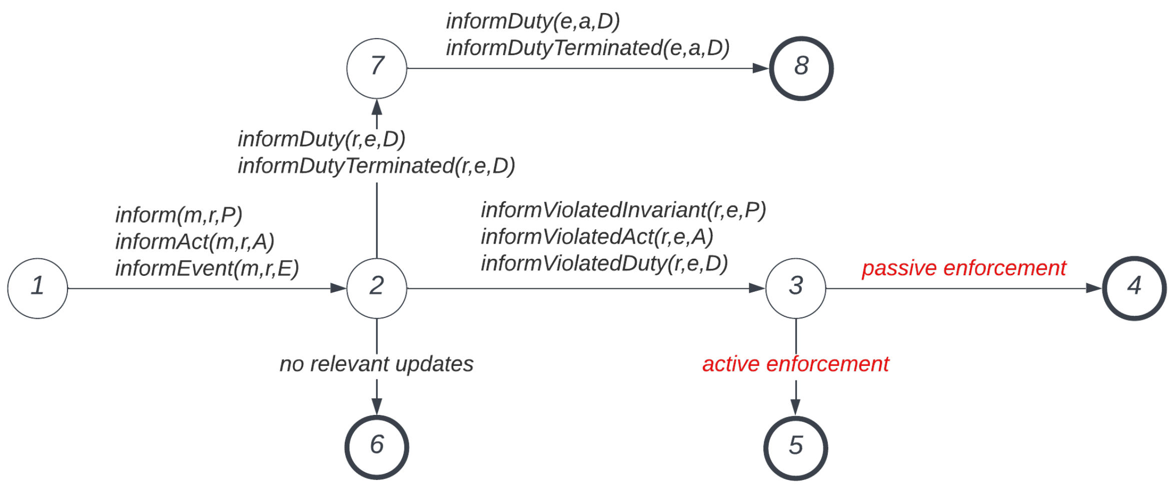

Figure 5.4: The combined state chart for both components using monitoring, that follows from the individual listening loops of the involved actor roles in both patterns.

5.2.4 The query pattern

Through duty monitoring, we have now defined a way to update all system actors of their normative positions. However, using monitoring for this purpose might not always be the optimal form. The query pattern reaches the same goal by assigning the initiative to the application actors themselves. Instead of setting up a full monitoring implementation, we could also design our hypothetical application to make the owner actors query for the duties that the real estate owner has. This is a more straight forward implementation if monitoring is not needed for other purposes in the system.

This pattern has a second component, which is very similar to access control: application actors can query whether an action is allowed. This can be used in more open normative systems, where we don't necessarily want to use full ex-ante enforcement to block the possibility of the action in question, but still allow actors to make sure the actions they intend to take are permitted.

The application actor specification (see Protocol 5.12) starts the sequence by sending a general request for a duty or an action. By shaping the message content (the duty object D) in such a way that either the holder or the claimant field is populated with the sending actor's address, it can make the query more specific.

```
send requestDuty(self,e,D) or send requestAct(self,e,A)
listen
    case informDuty(e,self,D):
        proceed with updated knowledge
    case permitted(e,self,A):
        proceed with updated knowledge
    case forbidden(e,self,A):
        proceed with updated knowledge
    case reject(e,self,D) or reject(e,self,A):
        sequence ended
end
```

Protocol 5.12: The protocol component for an application actor querying for duties or actions.

The enforcer actor, shown in Protocol 5.13, receives the request and first determines whether it is appropriate. This process is again application specific: the system designer might want to only allow actors to query duties they have (`D.holder = a`), duties some actor holds towards them (`D.claimant = a`) or not apply any restrictions at all. Similar restrictions can be put upon action requests. If the request is accepted, we see the familiar process again where the enforcer forwards the requests to the reasoner, listens for the result and returns it to the requesting actor.

```
listen
    case requestDuty(a,self,D):
        if request accepted:
            send requestDuty(self,r,D)
            listen
                case informDuty(r,self,D):
                    send informDuty(self,a,D)
            end
        else
            send reject(self,a,D)
    case requestAct(a,self,A):
        if request accepted:
            send requestAct(self,r,A)
            listen
                case permitted(r,self,A):
                    send permitted(self,a,A)
                case forbidden(r,self,A):
                    send forbidden(self,a,A)
            end
        else
            send reject(self,a,A)
end
```

Protocol 5.13: The protocol component for an enforcer actor receiving normative queries.

Note that for the *requestAct* message, here we see the difference between access control and the more generic query pattern. Instead of *forbid*, the prohibitive message that explicitly commits the application actor to refrain from executing the action, the enforcer sends *forbidden*, which does not include this directive. It merely conveys the fact that the action is currently not allowed, and execution could lead to consequences.

The reasoner specification can be seen in Protocol 5.14. It receives the requests from the enforcer, evaluates against the norms and returns the result. Note here that the knowledge base lookup using the duty instance is again defined as it was in Section 4.4.2.

```
listen
    case requestDuty(e,self,D):
        send informDuty(self,e,K_self(D))
    case requestAct(e,self,A):
        if N(?A)
            send permitted(self,e,A)
        else
            send forbidden(self,e,A)
end
```

Protocol 5.14: The protocol component for a reasoner actor receiving duty queries.

This protocol leads to the state chart in Figure 5.5. Note that, because we are solely talking about information passing here, the actions the requesting actor takes are not specified: even if the response to an action query is *forbidden*, the requesting actor can choose to still execute the action.

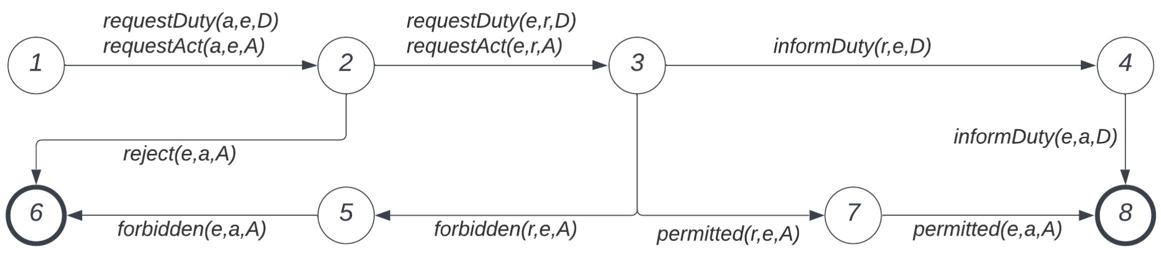

Figure 5.5: The state chart for the query pattern, that follows from the individual listening loops of the involved actor roles in this pattern.

5.2.5 The information fetch pattern

The protocol specification for the information fetch pattern is quite straight-forward, provided that the norm evaluation $N(msg.content)$ implements a way to mark information as missing. eFLINT achieves this by marking fact types as 'open', which makes evaluation that uses a fact of this type throw an exception if the fact is not in the knowledge base (see Section 2.3.1). The reasoner can then catch this exception, and send a request for the missing information to the relevant information actor (see Protocol 5.15). If the information point does not provide the information, the reasoner cannot make a decision. In most cases this will mean that the legality of the statement cannot be evaluated. Mitigation of this problem is not included in our protocol specification. Also, depending on the information actor and its data structure, fetching the information could take a long time; the reasoner actor implementation should wait for a response asynchronously in order to remain responsive to other requests.

An example exchange for this pattern using our real estate application could again be that the rent is raised by the owner. Rent increases are often limited, for example by the tenant's income. If the tenant's income is not in the eFLINT knowledge base yet and the owner executes a rent increase, we can mark the income fact as open, and make the reasoner fetch this data, for example from the system database or from the relevant country's tax authority.

```
listen
    case msg:
        if N(msg.content) raises Exception for missing P:
            send request(self,i,P)
            listen
                case inform(i,self,P):
                    if P.value ≠ ⊥:
                        retry N(msg.content)
            end
end
```

Protocol 5.15: The protocol component for a reasoner actor implementing its ability to fetch missing information.

Protocol 5.16 shows the specification for the information role: when it receives a request to inform the reasoner about some proposition, it looks it up in its knowledge base and returns the result with an *inform* message.

```
listen
    case request(r,self,P):
        send inform(self,r,$K_{self}$(P))
end
```

Protocol 5.16: The protocol component for an information actor implementing its ability to reply to information requests.

5.3 Generalisation

For reference during implementation, and for provision of an adequate overview of the functionality that actor roles should implement to comply with the protocol, we combined all of these pattern specifications into single specifications per actor role. Because these are mostly repetitions of the specifications in this chapter, these generic specifications can be found in Appendix A.

Chapter 6

Evaluation

In this chapter we will discuss our PoC implementation of the developed protocol. We implemented multiple case studies in Akka using the Scala language and eFLINT. We will briefly discuss some general implementation details, and then further describe the case studies we did, which communication sequences they lead to and how many of the requirements we were able to satisfy. We end the chapter with some comments about the genericness of the implementation process.

6.1 Implementation

We aimed to make our implementation as modular and generic as possible. All of the generic parts of the defined actor roles were implemented as partially abstract actor classes, making the implementation process of a system using the protocol as trivial as possible. It comes down to creating Akka actors that inherit from the generic actors and implementing the methods that are available for handling application specific parts of the protocol. For using the protocol in other technologies than Akka and Scala, a similar approach would be advised. Our implementation is publicly available on GitHub[1].

6.1.1 Akka environment

The implementation of our PoC was executed in Akka [25] using the Scala programming language. Akka is an open-source toolkit and runtime simplifying the construction of concurrent and distributed actor-based applications on the JVM. It provides us with an out-of-the-box implementation of the actor model. An Akka actor implementation that can run an internal eFLINT REPL instance is already available, and hence, Akka is a suitable choice for implementing our PoC.

6.1.2 The eFLINT adaptor

Internally, the reasoner actor is structured like the schematic representation shown in Figure 6.1. The reasoner receives messages specified in the protocol and translates them to eFLINT statements. This can be done in a structured way, since message semantics and content are closely related to how eFLINT statements are structured. For example, a Proposition object (as used in Chapter 4) is represented using a Scala class as shown in Listing 6.1.

[1]`https://github.com/philok55/master-thesis`

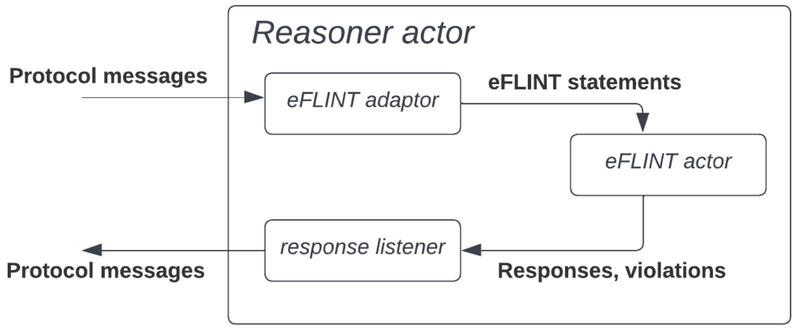

Figure 6.1: The internal structure of the reasoner actor.

```scala
trait PVal {}
final case class PString(value: String) extends PVal
final case class PInt(value: Int) extends PVal
case class Proposition(
    identifier: String,
    instance: List[PVal],
    state: TruthValue = True
) extends PVal
```

Listing 6.1: The Scala implementation of the Proposition object introduced in Chapter 4. It is used as message content and can be translated to eFLINT.

If a Proposition object is instantiated, `Proposition("tenant", List(PString("John␣Doe")), True)`, the eFLINT adaptor component maps the fields of the object into the eFLINT statement `+tenant("John Doe").`, which instantiates a `tenant` fact in the eFLINT knowledge base when evaluated. Because these objects are represented as Scala case classes, it is possible to abstract the concept of a proposition away by creating a Tenant class that inherits from Proposition. This way, concepts related to eFLINT are entirely abstracted away when using these objects in the system and the protocol messages.

When the eFLINT specification sends updates back to its reasoner 'parent' actor, such as violations or created duties, a response listener component translates these responses back to protocol messages that can then be used for further enforcement by the enforcer actor. The content of these responses are Scala objects generated by the internal eFLINT actor, which are already highly similar to the objects we use in our messages, and the translation is therefore trivial.

6.1.3 Actor roles base implementation

As said, for each actor role we implemented an inheritable Akka actor that contains all functionality that is generic across implementations. We will briefly discuss each role, what parts are generically implemented and how we used this to integrate application specific handling of protocol exchanges.

Application actor The functioning of application actors will be mainly application specific. The base implementation's primary function is to define which protocol messages can be received by application actors, and implement a listener that receives these messages. Upon message reception, the listener calls abstract handler functions that can be implemented by a concrete application actor (like `Tenant` in our implementation). Examples of these are `actPermitted` or `dutyReceived`. Furthermore, the abstract implementation provides an interface for initiating communication with the enforcer actor through the `sendQuery` method, which allows the application actor to send *requestAct* or *requestDuty* messages.

By defining as much functionality as possible in this abstract actor class, we remove complexity from the implementation process. A designer now only needs to implement reactions to messages and query the enforcer at appropriate times from the business logic. There is no longer a need to worry about using the correct messages or following the protocol; this is already included from the start.

Enforcer actor The abstract enforcer actor implements all generic behaviour that is defined in the protocol components. It thus listens for requests from application actors and updates from the reasoner actor. The necessary communication with the reasoner actor is also generically implemented for queries: a *requestAct* message is always forwarded to the reasoner actor. The application specific implementation comes in when the query result is available, in order to act accordingly.

To allow full flexibility and implement all of the enforcement strategies we have discussed, multiple abstract callback methods are again defined that can be implemented in a concrete enforcer actor to enforce decisions made by the reasoner. Examples of these callbacks methods are `actPermitted` or `violatedDuty`. To determine whether a request should be accepted or rejected, an enforcer can also implement an abstract method. Finally, we allow for specifying the difference between responding to *requestAct* with *permit* or *permitted* by overriding a list of action names that should be actively blocked to achieve access control; when the enforcer receives a request for an action that is in this list, it will reply with *permit* or *forbid*. Actually facilitating or blocking the action should then be implemented in the other, previously mentioned callback methods.

Reasoner actor The reasoner actor is implemented almost entirely generically. It handles all messages from the enforcer and monitors automatically by querying the eFLINT reasoner, listening for a response, and forwarding decisions to the enforcer through protocol messages. The only part of the reasoner that currently needs some application specific implementation is sending the requests to information actors upon required input needed by eFLINT. This includes registering a information actor and translating the exception thrown by eFLINT into a protocol message that can be sent to the information actor. This can again be done by implementing an abstract method that receives the exception and returns a `Propostion` object that the reasoner can use in a *request* message.

Monitor actor The monitor implementation focuses on translating system events to protocol messages that are sent to the reasoner. As noted before, this process is mostly application specific. If the system event implementations are closely related to objects that can be used in the messages, as is the case in our PoC, this translation is quite straight-forward, but still needs to be implemented upon integrating a monitor in a new system.

Information actor The information actor will often be positioned outside of the system that is regulated. This means that in real systems, its implementation will be using different technology. The implementation we created will therefore often not be directly translatable to a new system. However, we did create a generic information actor for usage within Akka. It includes a knowledge base with key-value entries, that can be queried with a *request* message. To make this implementation work, a concrete information actor needs to implement translation methods from a *request* message to a knowledge base key and from a knowledge base value to an *inform* message.

6.2 Case studies

In this set of case studies we will evaluate our protocol based on the requirements posed in Chapter 3 (Table 3.1). We will implement a more complete version of the hypothetical real estate management application that we used as a running example in Chapter 5. In this application, each of the described patterns will be present in some way, and we will explore the communication sequences that our protocol can facilitate within and across these patterns. Note that some parts of the PoC that are irrelevant to the protocol itself are hypothetical, like the application's user interface or email sending functionality; all communication that would be necessary to implement these side effects have actually been implemented.

To accurately represent the observed communication flows, we will use multiple sequence diagrams. In these sequence diagrams, we will clearly mark application specific components again: communication that does not use our protocol messages will be marked with red arrows, and execution of application specific logic will be shown in red as well. The eFLINT specification that was used for the case studies can be found in Appendix B.

6.2.1 Monitoring implementation

When a monitor actors are created, they subscribe themselves to a global registry that tracks all active monitor actors in the system. It is supported to create multiple of these registries to split the monitoring setup into different 'subscriptions' that can be consumed by different monitor actors.

During their functioning, application actors generate system events, for example by executing some action, sending a message, or saving something to the database. We created a custom message sending procedure that automatically generates corresponding system events when application actors send a message. System events are set up to automatically notify the relevant registry of their occurrence. The registry, which holds references to all subscribed monitor actors, responds to incoming system events by propagating these events to all monitor actors.

Monitors then translate these events to messages that have a place in our protocol again, before they can update the reasoner actor. To achieve this, we use a strategy similar to that of the translation to eFLINT. We create Scala classes that extend from Predicate or Act and represent the application specific objects. These can then be used in the system events without any system actors needing to know about the concept of propositions or acts, but can also directly be used as the content of protocol messages.

6.2.2 Generically satisfied requirements

Some requirements have already been satisfied by our generic base implementations of the actor roles and their surrounding functionality. Requirement **R1.2** is already satisfied generically by the reasoner actor and its eFLINT adaptor: any protocol message that is sent to it can be translated to an eFLINT statement and evaluated. The generic reasoner implementation always sends results to its registered enforcer actor.

The generic monitoring requirements have also been satisfied by our implementation. Requirement **R2.1** we satisfied using our approach with system events, that are directly generated when an application actor executes some action that we want to monitor. Requirement **R2.2** was satisfied by translating these events to messages as discussed in the previous section. Our generic monitor actor also automatically forwards all correctly translated messages to its registered reasoner actor, satisfying **R3.1**.

6.2.3 Access control

We start our set of case studies by implementing an access control feature into our real estate management application. The actors involved here are application actors (the owner, tenants and a database), a monitor actor, a reasoner actor and an enforcer actor. The owner and tenant actors are of course application actors, and inherit from the generic abstract application actor that implements the handling of normative messages. Through the exposed `sendQuery` method, the tenant actor can send action requests to the enforcer to ask for document access, fulfilling requirement **R1.1** in a satisfactory way: a query can be sent from application specific procedures using only two lines of Scala code.

We use our monitor actor to achieve requirement **R1.3**, namely to keep the reasoner actor and its eFLINT knowledge base up-to-date about the system state. When the owner actor creates a tenant or an agreement for a tenant and saves these objects to the database, system events are generated and broadcast to the monitor actor. The monitor translates the events to *inform* messages and sends these to the reasoner actor. This way, the reasoner can make decisions based on the correct state of the system.

When the enforcer forwards the request from the tenant to the reasoner, the decision making within the reasoner is fully generic. When the enforcer receives its response (either *forbidden* or *permitted*), it sends the permissive and prohibitive variants of these messages to the requesting tenant. This is also implemented generically, and satisfies requirements **R1.4** and **R1.5**. In order to actually enforce the decision, the enforcer implements the callback methods to add the application specific handling of the actual document access. In the case of our PoC, the tenant is unable to access the database directly, which achieves the prohibitive sequence; when access is permitted, the enforcer fetches the document from the database and sends it to the tenant. Thereby we also satisfied requirement **R1.6**, completing our access control case study.

See Figure 6.2 for a sequence diagram of a successful access request in our implementation. Of course, in the case of a denied request, the reasoner sends *forbidden*, the enforcer sends *forbid* to the tenant, and that finalises the sequence.

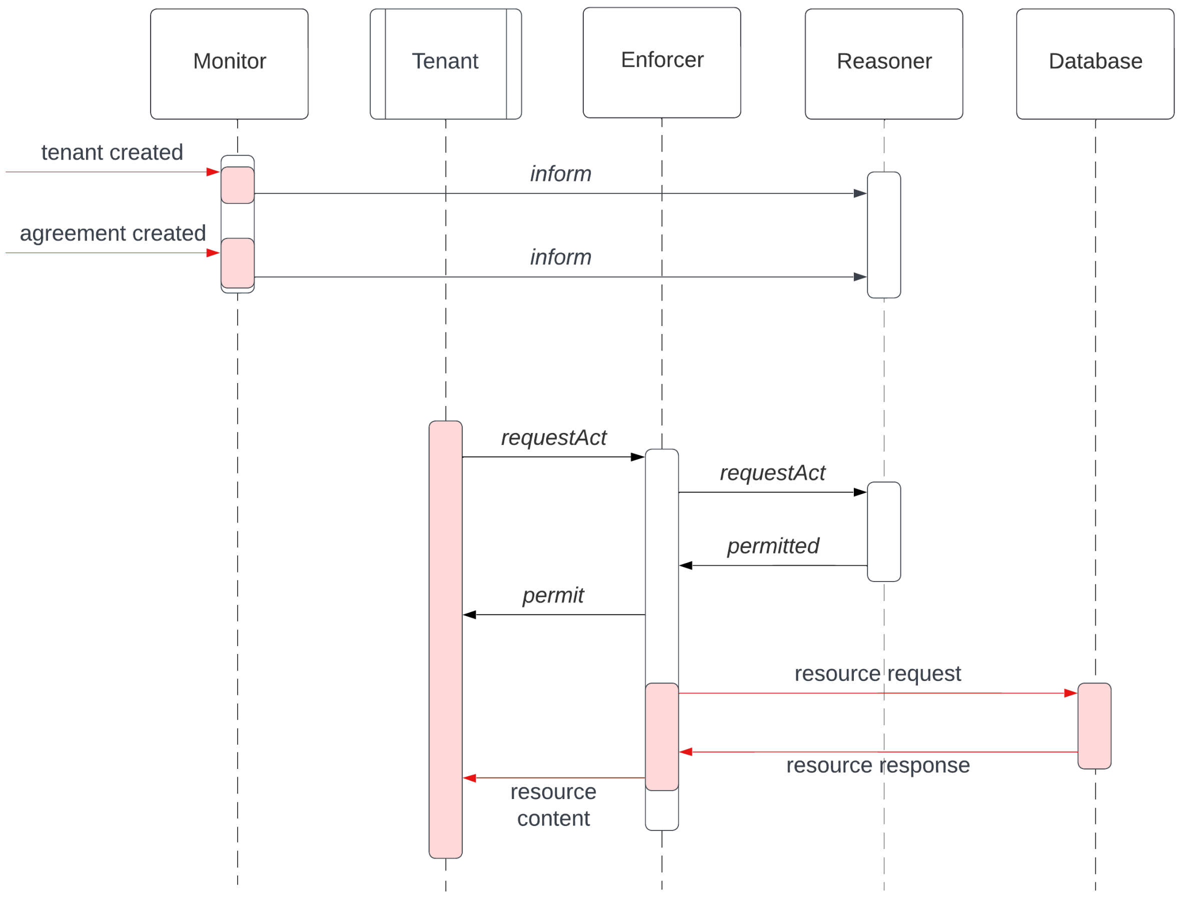

Figure 6.2: The sequence diagram for a successful request to access a document by a tenant.

6.2.4 Ex-post enforcement

To evaluate the fulfilment of our requirements for ex-post enforcement, we discuss two scenario's: one in which an action causes a violation, and one in which a duty is violated. For each, we will employ a different enforcement strategy: passive enforcement for the action violation, and active enforcement for the duty violation.

Action violation

When renting out residential real estate to private tenants, most countries have laws that prohibit excessive rent increases. For this case study we assume that rent indexations are limited to 4%. Thus, the action of raising the rent of a contract with more than 4% is a violation. We extend our monitoring implementation with a system event for an indexation, raised when a rent increase is registered in the database. The monitor translates this event to an *informAct* message and forwards it to the reasoner. If it causes a violation, the reasoner notifies the enforcer through its generic behaviour. Then, the enforcer uses its position of authority and application specific implementation to execute the actual enforcement. In this case, it sends a message to the owner portal that is displayed to the user through its user interface. This is how we achieve passive enforcement in this use case, satisfying requirement **R3.2**. It is up to the user of the owner portal to decide whether they intend to risk breaking the rules. The sequence diagram for this case can be seen in Figure 6.3.

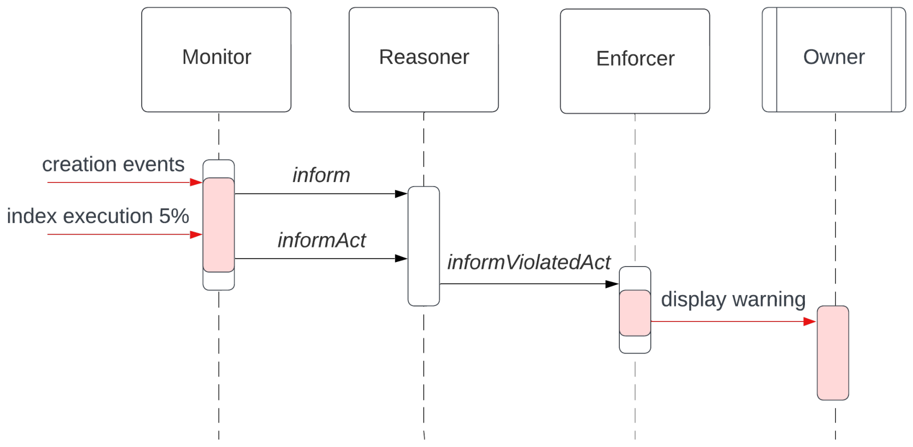

Figure 6.3: The sequence diagram for ex-post enforcement of action violations, with passive enforcement.

Duty violation

In the second ex-post enforcement case, we implemented duty violations. In this case, we assume that a tenant has a duty to pay a month's rent on the first day of that month at the latest. We use two system events that we add to the monitoring: one for setting the payment deadline, and one for marking the payment as due. Because eFLINT does not have functionality to work with dates yet, we need this second event as well: marking the payment as due needs to be done by the owner actor by raising this event. When this is done before the tenant executes the terminating `make-rent-payment` action, the duty becomes violated and the reasoner informs the enforcer. The enforcer is again the actor that chooses the enforcement strategy: a possibility for active enforcement in this scenario would be sending an email to the tenant about the missing rent payment and warning them about a possible debt collection procedure.

While requirement **R3.3**, active enforcement, cannot be satisfied generically, this is an example of how a system can satisfy it in an application specific way. The enforcer implements a procedure in the callback that is called when a duty is violated, and therefore when designing a system we only need to implement the actual sanction. In our case, sending an email to the tenant about the rent payment.

The sequence diagram for a duty violation can be seen in Figure 6.4.

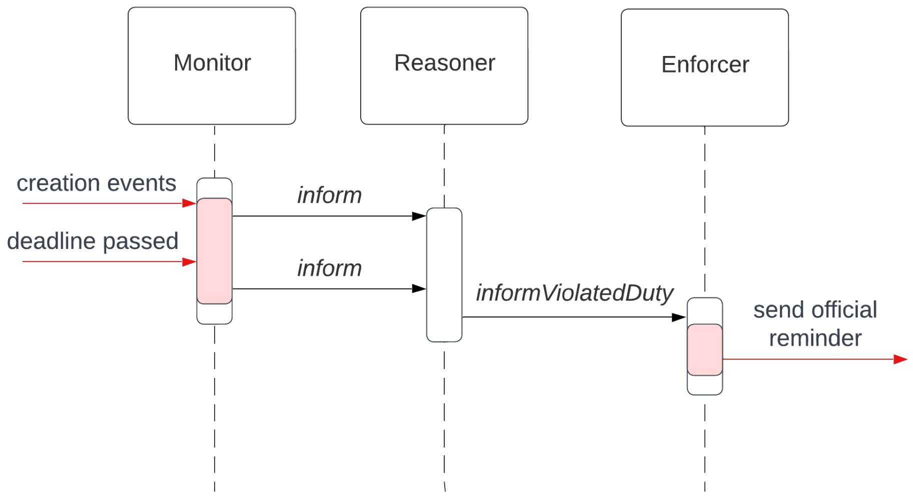

Figure 6.4: The sequence diagram for ex-post enforcement of duty violations, with active enforcement.

6.2.5 Duty monitoring

For the pattern of duty monitoring, we simulate a case in which a duty becomes active, and afterwards an action is executed that terminates the duty again. We monitor for actions that cause these duty mutations. Say that a tenant terminates their rental contract. When starting to rent their living space, they paid a deposit, which the owner should now give back to the tenant. Thus, we create system events for terminating a contract, which is an action the tenant takes that leads to a duty becoming active for the owner to refund the deposit. The terminating action of refunding the deposit raises a second event that the monitor picks up. Using our protocol component of duty monitoring, the reasoner receives the events from the monitor, notifies the enforcer of the active duty, and then the enforcer decides who to notify, satisfying requirement **R4.1**. In this case, we notify both the owner and the tenant: the owner should pay back the deposit, and the tenant might want to remind them if this doesn't happen timely.

The second part of the duty monitoring pattern kicks in when the duty is terminated. The flow and approach of this variant are exactly the same, but this time the reasoner sends *informDutyTerminated* to the enforcer. The enforcer might act differently in reaction to this in other applications; for our implementation, it is enough to just notify both involved actors again. Through this, we have also satisfied requirement **R4.2**. See Figure 6.5 for the sequence diagram that includes both variants of the communication flow for the duty monitoring protocol component.

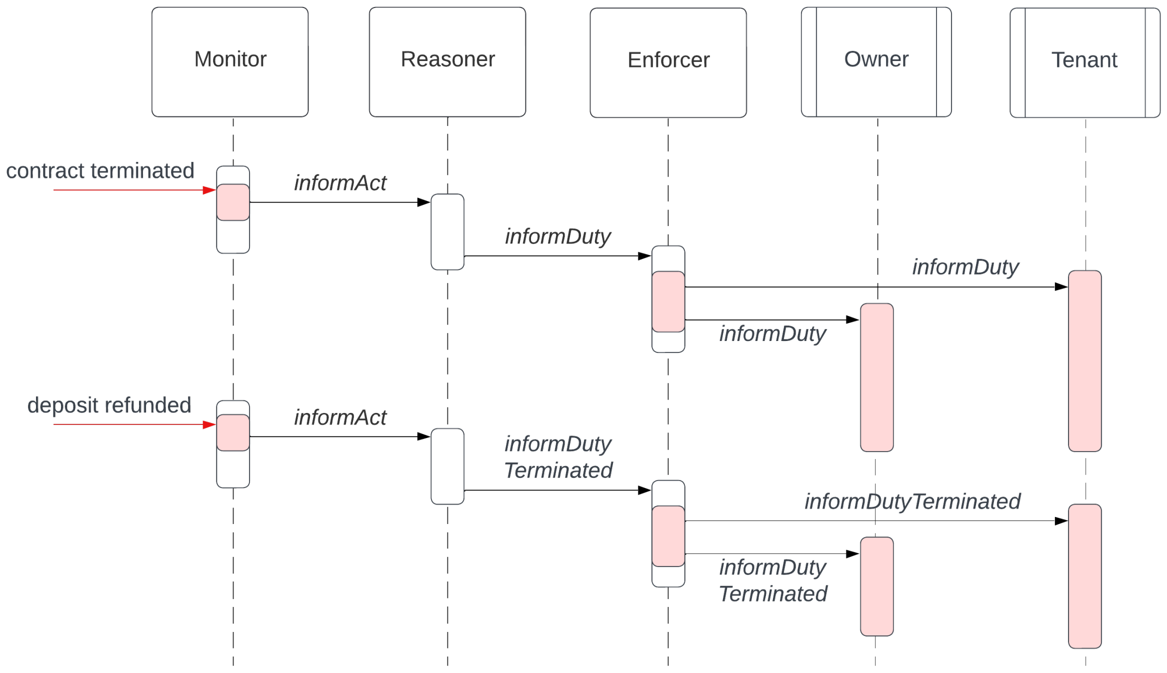

Figure 6.5: The sequence diagram for our case study towards duty monitoring.

Note here, that this pattern integrates naturally with the ex-post enforcement approach discussed previously. If, for example, we would specify a deadline for the refund of the deposit, we could combine ex-post enforcement with the ex-ante approach of duty monitoring to achieve fully flexible enforcement. We would notify all involved actors about existing duties to give them the opportunity to comply; if they don't, we can monitor for violations and further enforce the norms using sanctions, for example.

6.2.6 Queries

To demonstrate the query pattern, we have implemented duty queries. We use the same scenario as we did in the previous case study, where the tenant terminates a contract and a duty comes into existence for the owner to refund the deposit. When this duty is active, the owner actor can query its enforcer actor to retrieve a list of active duties. In this case, we make the owner actor query for any duties where this actor is the holder. In a real system, this query would be useful when displaying a user interface which can then point out the active duties to the user.

In order to query for duties, the Akka actor that represents the owner can again use the exposed `sendQuery` method with a partially populated Duty object. In this case, we set the `holder` field to the owner actor, and leave the `claimant` field empty. The `sendQuery` method then sends a *requestDuty* message to the enforcer, satisfying requirement **R5.1**. To fulfil requirement **R5.2**, the enforcer can again implement an abstract method that examines the request and possibly sends *reject* without including the reasoner. In our implementation, we use this functionality to limit duty requests to duties where the requesting actor is the holder. The final requirement, **R5.3**, is fulfilled generically by the enforcer implementation: duty requests are forwarded to the reasoner, and the duties that the enforcer receives in response are sent back to the requesting actor.

Querying all duties for whether the claimant or holder matches the request is not a feature that eFLINT can provide us with currently. Therefore, we implemented this in the Akka actor that surrounds the eFLINT REPL, by iterating through all duties and comparing these fields.

The sequence diagram for the implementation of the query pattern example can be seen in Figure 6.6. We did not implement an extra case for querying for actions; this sequence would namely be exactly the same as the basis of the access control sequence, and all of its requirements have already been satisfied. To illustrate, an action query has still been included in the sequence diagram.

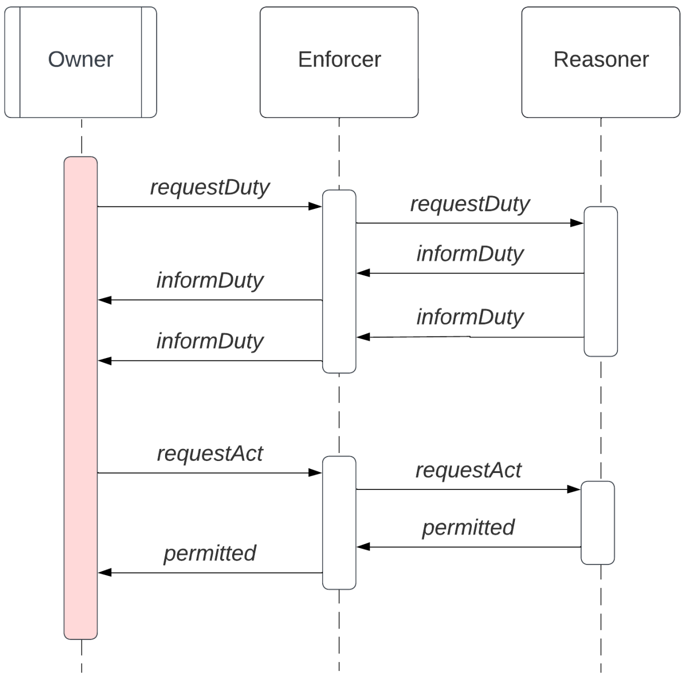

Figure 6.6: The sequence diagram for the usage of queries in our case study.

6.2.7 Information fetches

For our final case study, we implement information fetching upon missing input required by eFLINT to make a decision. We take another look at the action of increasing the rent. The legality of this action can also be dependent on the income of the tenant. In our case, we assume that a rent increase cannot exceed 2% if the tenant's yearly income is below €45,000. Now, the eFLINT reasoner needs to have a value for the tenant's income in its knowledge base in order to make a decision. To enforce this, we mark the Fact `yearly-income` as Open:

```
Open Fact yearly-income Identified by tenant * price
```

Now, when eFLINT tries to evaluate an Act that is conditioned by a value for `yearly-income`, it will throw an exception specifying that input is missing. This exception is caught by the reasoner's internal Akka actor running the eFLINT REPL, and this actor notifies the reasoner, satisfying requirement **R6.1**. The reasoner actor then parses the exception and forms a `Proposition` object that is used in a *request* message that is sent to a registered information actor. Currently, this parsing still needs to be implemented by the application designer, but this process could be implemented generically in the future. With this, requirement **R6.2** is satisfied, albeit not in a generic manner yet. Requirement **R6.5** is implemented generically, by making the reasoner temporarily store the original message and retrying the evaluation after receiving an *inform* message from the information actor.

For our example, we implemented an information actor that represents the tenant's country's tax authority, where the tenant's income can be fetched if eFLINT doesn't have it yet. This actor satisfies requirements **R6.3** and **R6.4** by translating between messages and its internal data representation. Our strategy for implementing this was discussed in Section 6.1.3. See Figure 6.7 for the sequence diagram of this case study.

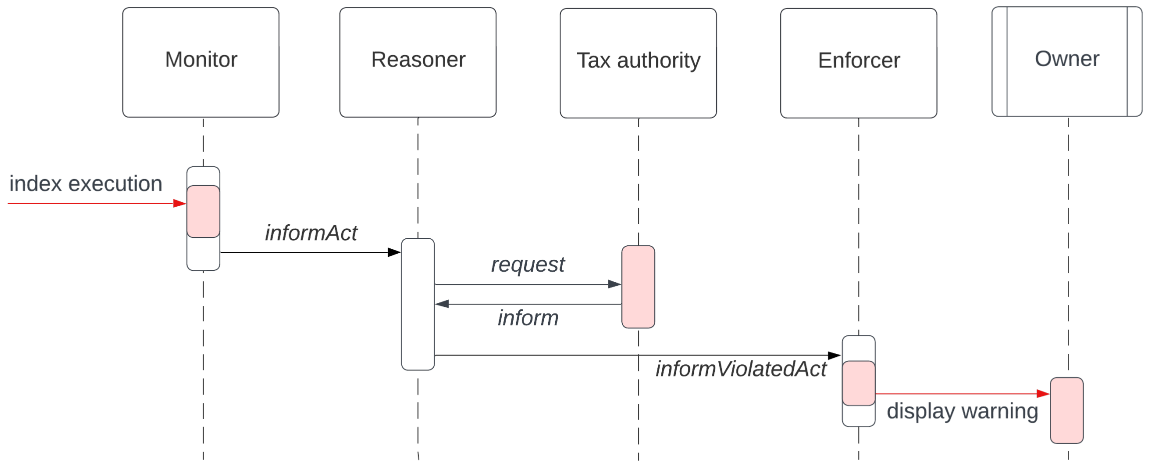

Figure 6.7: The sequence diagram for how we use the information fetch pattern in our case study.

6.3 Implementation considerations

Our approach for implementing the protocol as a PoC has lead to several considerations for implementation into a new system. We briefly discuss how generic the implementation is, and how much boilerplate code is needed to implement the protocol into a new system. Furthermore, we discuss how a higher level of distribution could influence the design considerations.

6.3.1 Generality

We attempted to make our implementation as generic and modular as possible. Some parts of the implementation that currently are not generic, like the parsing of "input required" exceptions by eFLINT, could be made generic with extra effort, which we reserve for future work. However, as can be seen in the protocol specifications themselves, many procedures in a normative actor system are inherently application specific. Our efforts have lead to a module of message types and abstract actor implementations. This module can be used directly in applications using the Akka framework, where business logic inside application actors interacts with the methods that are exposed in this module. By using inheritance and implementing abstract methods, we limit boilerplate code to functionality that is actually needed in the system; protocol functionality that is not used can be left out, and does not increase the amount of boilerplate code. When functionality is needed, integration efforts are limited to implementing abstract methods, which we found does not lead to extreme or unmanageable amounts of boilerplate code.

We took care to limit unintended interference, especially with application-specific messages, by clearly distinguishing normative messages from application messages. The functioning of the protocol never relies on application specific messages, and we believe, from our implementation experience so far, that application-specific messages do not interfere with the protocol implementation whatsoever when sent in between an exchange, if the actors are correctly implemented.

6.3.2 Distribution

In our implementations, we have often limited the amount of normative actors to one of each role. Larger, more distributed systems might require multiple reasoner, monitor or enforcer actors, which would yield more challenges than discussed in this project. For example, multiple reasoner actors would run different eFLINT specifications that might conflict with each other. When we introduce multiple enforcers, the application actors would need to know who to contact for queries. Also, when multiple monitor actors inform the same reasoner actor, they might do this with conflicting statements.

Addressing these issues and ensuring consistency in distributed normative actor systems is a nontrivial problem. While it falls outside the scope of the current project, it presents a challenging direction for future research, aiming for resilience and adaptability in large-scale, distributed normative actor systems.

Chapter 7

Discussion

In this chapter, we answer the research questions that we posed in our introduction. We will reflect on the design of our message framework and protocol, and their implementation, evaluation and applicability in real systems. Next to this, we identify some threads to validity and discuss the efforts we made to mitigate them. Finally, we give suggestions for future work.

7.1 Research questions

In our introduction section, we posed three research questions. The first two questions lay the ground work of understanding and analysis that were necessary for our most important contributions: the design approach in answering RQ3. We will discuss briefly how well we succeeded in laying this ground work, how comprehensive our requirements process was, and then focus our discussion towards our design efforts and their evaluation.

7.1.1 Research question 1

Our first research question was the following: *"Which types of systems can be identified in which compliance with norms is currently enforced, either automatically or manually?"*. The primary goal here was to establish a solid foundation of understanding regarding the landscape of norm compliance in contemporary systems. We used a literature survey in order to create a broad exploration of insights into current practices and the state of automation in enforcing norms. Although there are always concerns about comprehensiveness when basing such an overview of the state-of-the-art on a literature survey (also see Section 7.3, threats to validity), we believe we have succeeded in creating an extensive high-level overview of the currently existing types of normative systems and which approaches for enforcement are currently used, both automated and with human intervention.

7.1.2 Research question 2

We posed the second research question as *"How can normative actors be integrated in each type of system, and which patterns of communication between these actors emerge from the observed enforcement strategies?"*. In Chapter 3, we combined the results of RQ1 with an analysis aimed towards answering RQ2. The focus here was on uncovering effective strategies for integrating normative actors within the various normative systems we identified. From the resulting enforcement strategies, we identified six distinct patterns of communication. By analysing these patterns, we were able to provide structure to the foundations for our protocol by categorising actors into specific roles, and finalising Chapter 3 with a list of 21 requirements for a protocol facilitating normative actor systems generically.

7.1.3 Research question 3

The final research question is: *"What are the necessary components of a protocol for communication with normative actors with which the identified communication patterns can be facilitated in a generic way?"*. In order to answer this question, we first designed our message semantics in Chapter 4. Our approach using speech acts and commitment semantics allowed us to accurately capture the normative semantics of the messages that were needed to base our protocol on. This significantly lowered the complexity of the protocol specifications themselves, since it removed the burden off of the protocol to define what it means to, for example, 'have a duty'. If we were limited to existing message frameworks without normative semantics, we would need to include these semantics in the contents of a message (e.g. *inform* as used in FIPA-ACL), making protocol specifications dependent on (possibly unrestricted) message contents. The approach of using commitment-based semantics allowed us to disregard the internal architecture of the actors, making our message semantics widely applicable. Also, it allows for further research into verification environments for systems that use the protocol (see Section 7.4.4, future work).

Using the newly formalised messages, we systematically designed our protocol specifications based on the patterns and requirements resulting from RQ2. We took an incremental approach, in which we defined parts of the specification to facilitate single patterns. This approach resulted in a full specification of the necessary components, which were then combined into single specifications per actor role.

7.2 Evaluation

We have used Chapter 6 to discuss our PoC implementation, and thereby show how the protocol can be used in an application that exhibits each of the patterns we discussed. By using all of the patterns in a single application, we explored how we can combine patterns to create complex enforcement strategies. With our PoC, and some extra implementation efforts towards the existing tooling around eFLINT, we were able to satisfy all 21 requirements. Eight of these were satisfied in a fully generic way, without any need for application specific procedures. The remaining 13 need some application specific implementation, but as we discussed in Section 6.3.1, adding this logic can be done through implementing abstract methods, which does not require much boilerplate code. Also, most communication, especially between normative actors, is generic; we mainly observe a need for application specific procedures, which are internal to a single actor and thus much more manageable than application specific communication.

Our approach comes with limitations that are inherent to our intention to capture the full complexities of normative systems, as opposed to limiting our view to specific system requirements. As discussed in section 6.3.2, distributed norms and system state can be conflicting with each other. Another unresolved problem is that of 'full qualification': in order for an application actor to understand the full implications of a message, the current amount of information that is available for message passing can be insufficient. Take the example of the *informDuty* message. When we tell an application actor "you have a duty to submit homework", this application actor needs to already be familiar with the concept of 'homework', and know how to act in order to fulfil its duty. If none of this information is available to the receiver, sending *informDuty* alone is currently insufficient for them to comply. This remains an open problem that is suitable for further research efforts.

7.2.1 Contributions to usability

Our efforts towards abstraction and generalisation are motivated by usability and stability requirements, in order to allow normative DSLs to mature and to be adopted more widely. While the current approach of using DSL statements directly in application code offers full flexibility, it will not be suitable for effective usage in distributed production systems on which multiple persons or teams work together. When multiple components communicate, especially when not designed by the same person, standardisation is essential for correct functioning. Also, by abstraction of lower-level concepts like propositions and actions in the implementation of the protocol, programmers can simply use object-oriented approaches in programming languages they are familiar with, without needing to work directly with the DSL that is used to evaluate against the norms.

The same goes for our approach of separation of concerns, which we achieve by assigning different roles to actors. When adopted in large systems, there will often be people with very different skill-sets working on the system design. We anticipate that the tasks of writing eFLINT specifications and writing the enforcement strategies inside the enforcer actors will often be carried out by different people, which is facilitated by our approach of separating these functionalities.

Through this standardisation, abstraction and separation of concerns, we believe our message framework and protocol are a step in the right direction towards the goal of usage of normative DSLs in production systems.

7.2.2 Implementation dependence

A key choice when creating our PoC was the reliance on Akka. This decision was motivated by the robustness and advanced capabilities Akka provides. The framework simplifies actor management, communication, and coordination tasks, thereby enhancing productivity and system stability. However, the native messaging mechanisms in Akka do not translate readily to conventional web protocols such as HTTP, impeding a seamless transition to an implementation using such technologies. This issue also draws attention to a broader point: not all systems are explicitly structured as actor systems. As such, some more effort might be needed to move the PoC to an implementation that uses different technology as its basis.

The implementation we created is, however, only a representation of the specifications of the protocol we created. None of the behaviour specified in Chapter 5 is dictated by the implementation that was chosen; all of the generic behaviour is in these specifications. Thus, by no means is Akka a prerequisite for implementing a system using the protocol; usage of other technologies would only create the extra burden of not having an out-of-the-box implementation of the actor model, but conceptually the underlying architecture is of no influence.

7.3 Threats to validity

This study, as any research effort, comes with potential threats to its validity, which we recognise and analyse here for the sake of transparency and future work.

A first concern stems from the empirical nature of our requirement discovery process. This approach, while invaluable in obtaining pragmatic and real-world pertinent insights into norm enforcement patterns, might have overlooked certain cases due to its inherently heuristic basis. The richness and complexity of the target domain could imply that not all possible scenarios were captured, thereby potentially compromising the comprehensiveness of the requirements. Although we have attempted to cover a broad spectrum of systems and enforcement approaches, and we were methodical in our efforts towards finding relevant literature, we cannot rule out the possibility of enforcement patterns in existence that were not covered. However, if in the future we were to find different patterns that should be included, the structured, formal approach and modular implementation of our protocol should make this addition quite straight-forward.

Secondly, the evaluation of our normative actor protocol was conducted with a system similar to that which it was designed for. This overlap between design and evaluation contexts poses a potential threat to the validity of our findings as it might have led to missed cases, particularly those that might emerge in distinct or unanticipated system environments or architectures. In order to mitigate this threat in the future, it would be desirable to execute more PoC implementations in different settings.

Additionally, our protocol has yet to be evaluated in actual production systems. The true robustness and adaptability of the protocol can only be conclusively demonstrated in real-world, production-grade environments where unpredictable dynamics and constraints often come into play, combined with the previously discussed higher degree of distribution. Although Akka is a framework that is actually used in production settings, and we believe we can approximate them quite closely with our case studies, without concrete evidence of the protocol's effectiveness and usefulness in actual production settings it remains difficult to determine the extent to which our normative actor protocol can capture all possible communication patterns in a satisfactory way.

7.4 Future work

The following section outlines potential future research directions that could expand upon our understanding, address limitations, and further enhance the applicability of the normative actor protocol in real-world settings.

7.4.1 Possibilities for extension

Some possibilities for extension of the messaging framework and the protocol were considered, but not added yet due to a lack of direct use cases in the enforcement patterns we found in literature. However, we do not rule out the possibility for these options to have use cases that we did not identify, so we list them here for possible future implementation. These possibilities include the following:

- ***request* messages outside of the information fetch pattern:** in our current specification, the *request* message is only used for the information fetch pattern. However, since this message could be used to request any information from any actor, there might be other use cases, for example where application actors request information updates from their enforcer actor.
- **Reasoner fact updates:** we have shown how evaluating eFLINT statements can lead to violations or newly active duties, of which the enforcer gets notified. Other side effects could be truth value changes for facts. There might be use cases for updating the enforcer about these fact changes, for example when sanctions for specific actors are programmed into the eFLINT specification.
- **Enabled or disabled actions:** another option for extending the information provided by the reasoner would be to include enabled or disabled actions, for which there could be a similar distribution setup as we have created for duties in the duty monitoring pattern. Through this mechanism, application actors could be notified of actions that are allowed from now on, or no longer allowed.

7.4.2 Issues with distribution

As we discussed in Section 6.3.2, there are still some potential issues when production systems using the protocol will require a higher level of distribution. To further increase suitability in large production systems, and increase the stability and usability of the protocol, further research would be needed towards resolving these distribution issues. This would require advanced synchronisation mechanisms in order to mitigate conflicting norm interpretations, decisions by reasoner actors, information updates by monitor actors, and enforcement strategies by enforcer actors.

7.4.3 Meta policies

Another promising direction for further generalisation could be the specification of 'meta-policies'. Especially in the application specific behaviour of the enforcer actor, we make use of such meta-policies: who to send duty updates to, whether to send *forbid* or *forbidden*, or blocking duty queries when requested by an actor that is not a party of the duty. An option for future work could be the exploration of specifying these meta policies in a language with fixed semantics. This language could then be used in the enforcer actors as an internal actor, similar to how eFLINT is now used within the reasoner. With such a language, we could make use of similar model-driven design approaches for the meta-policies as is currently used for the norm specifications themselves, with all the benefits that come with it.

7.4.4 Verification environments

As discussed in Chapter 4 when introducing our formal model used for the message semantics, we see our formalisation using commitments as an opportunity for creating verification procedures for systems that use our protocol. Based on the semantics of the messages that are observed in a running system, some newly introduced component could keep a list of these commitments, and generate reports about which actors behave correctly and fulfil the commitments that they make through the messages that they send. Such a testing environment could be of use when integrating the protocol into large production systems where manual testing becomes too labour-intensive.

7.4.5 Code generation

To further ease the process of creating systems with our protocol, an interesting research direction would be the generation of code from specifications. Because our messages are formally defined, and we specify our protocol components in a structured, exact manner, some prerequisites for code generation are already available. It might be possible to specify the application specific parts of the desired program in a similar form as the protocol specifications, and generate a fully functional integration of generic and application specific logic in different target architectures.

7.4.6 Explainable AI and conversational protocols

eFLINT itself has some interesting possibilities for further feature development. Because normative DSLs like eFLINT do not exhibit the 'black-box' problem that is so prevalent in statistics-based AI models, there are opportunities for providing explanations for decisions that were taken. eFLINT is capable of providing full traces of its knowledge base evolution, providing insights into the reasoning behind the decisions it makes. Further clarifying information could also be added to existing features. Currently, eFLINT tells us when an action caused a violation, but not *why*. Similarly, the notifications about duties could be extended with more information about which actions could be taken to terminate them, which could also bring us closer to a resolution for the aforementioned qualification problem.

Using these features, the protocol could be extended towards a more conversational version, in which application actors are able to appeal decisions or ask for an explanation or motivation for a decision. We see options for adding this functionality in the form of new messages that could be added to our framework, like *appeal*, *challenge* or *clarify*. The reasoner actor could then use eFLINT traces in response to these messages, possibly with further analysis over it, to facilitate this conversational approach.

Chapter 8

Related work

In this chapter, we discuss work related to that presented in this thesis. We briefly revisit the existing forms of automated norm enforcement that were discussed previously, and compare these approaches to ours. Secondly, we take another look at existing agent communication languages that employ similar forms of messages for communication using speech acts. We then discuss some communication protocols seen outside of normative systems, and finally focus our attention to normative multi-agent systems (nMAS), where much research exists towards ex-post enforcement and normative communication.

8.1 Automated enforcement

As discussed in Chapter 2, there are multiple technologies which are used to automate enforcement of norms in specific contexts. Smart contracts [5, 22] can achieve ex-ante enforcement in a peer-to-peer setting for reliably enforcing different types of policies. Another example of automated ex-ante enforcement is access control [20]. We observe similarities between our approach and a popular framework for access control, XACML [21]: they take a similar stance on separation of concerns with their architecture components, separating enforcement from reasoning and attribute storage. However, XACML is only focused on access control, and thus misses many of the features we introduced.

Next to ex-ante enforcement, implementations of ex-post enforcement exist. Examples, as previously discussed, are spectrum sharing [36, 37] and cloud monitoring [34, 35]. While these implementations work sufficiently in their respective fields, they are specifically designed for them, and don't translate to other types of systems. Ex-post enforcement is also an active research subject in normative multi-agent systems (nMAS), which is discussed in Section 8.4.

A difference we can identify between all of these technologies and our contribution, is that the existing approaches focus on a specific use case in specific system architectures. We have captured the communication patterns that they exhibit in a more abstract way by starting from norm theory based on Hohfeld's framework. Therefore, many of the observed enforcement strategies come back in our approach, but in a more generically applicable way that is independent of the specific types of enforcement and the regulated system itself.

8.2 Agent communication languages

Many existing agent communication languages (ACLs) [29–31, 39–42] define messages similarly to our approach. Speech acts are the most popular formalisation of these messages. Approaches for further semantics differ, with two notable categories: *mentalistic* and *social* semantics (see Section 4.1.1). Often, ACLs are presented as separate messaging frameworks for generic usage with MAS, and don't include protocols; specifying these is then left to system designers. For many use cases this is sufficient, as protocols are often use-case-specific. A notable exception is the work of Pitt and Mamdani [41]: they include protocols in their message semantics, so that the meaning of a message is defined by the expected response: this is similar to how we define some of our directive message types. Kibble [40] uses commitment-based semantics as a basis to add *conversational* messages to, which we also mentioned as possible future work in our project. Extending our framework with more options for appeal and challenging of decisions would be useful, and could be based on the work by Kibble, who also provides example protocols for usage of these conversational messages.

Another difference between most ACLs and our messages, is the inclusion of normative semantics. One example of effort towards creating this feature is the work of Agerri and Alonso [42], where ACL semantics are extended with deontic logic. This allows agents to create and remove rights for each other by executing actions. Despite these advances, an exclusive focus on MAS, the absence of protocols and a less complete capture of normative reasoning compared to eFLINT limit the applicability of this approach.

8.3 Communication protocols

Communication protocols are often created for very specific use cases. This makes it difficult to transfer an approach taken by a protocol specification and transfer it to another domain. Thus, while there are many protocols in existence (well-known low-level protocols like TCP, HTTP, or more application specific protocols, e.g. for self-driving cars [43, 44] or internet of things [45]), direct comparison to ours is hardly useful. Protocols specifically created for usage in normative systems are only really seen in the context of nMAS, some of which we will discuss in the next section.

8.4 Multi-agent systems

A significant body of work in the area of normative multi-agent systems (nMAS) focuses on various aspects of norm enforcement and agent communication, often employing unique strategies to address specific challenges in this domain. Criado *et al.* [46] propose a distributed architecture for enforcing norms in open multi-agent systems within the Magentix [47] platform, a framework for MAS that uses FIPA-ACL for its communication. The proposed architecture is similar to ours in that it separates the concerns of norm management, which registers norms and instances, and norm enforcement, which controls agent behaviours. However, it does not present a generic protocol for communication and instead appears only defined within the features of Magentix itself. The norm specification also seems limited, as it only operates with agent roles.

Parizi *et al.* [48] introduce a modular architecture for integrating normative advisors in MAS, using eFLINT actors that are internal to agents, showing similarity to our approach of embedding eFLINT in our reasoners. Through this feature, agents maintain their own 'conscience' and are responsible for keeping their norm base up-to-date, depending on their functioning and perception. They communicate with eFLINT through a translation mechanism similar to our eFLINT adaptor, translating between AgentScript [23] and eFLINT. While the usage of eFLINT provides them with the same rich capturing of complex norm structures, a key difference compared to our approach is that they focus primarily on the agent's subjective standpoint, with each agent deciding for itself whether to adopt a set of norms. There is no explicit focus on normative communication between agents, and the system is tailored for BDI agents using the AgentScript language, limiting its extensibility to other software systems.

Artikis and Sergot [8] introduce an executable specification norms in open multi-agent systems, using an event calculus as a direct formalisation. Their messages are not generic, being defined only for merchant-type applications, and there's no comprehensive representation of Hohfeld's concepts, because permissions are solely defined based on roles.

Another approach that is promising in nMAS is *norm negotiation* [49, 50]. In this approach, the agents, through specific protocols, make 'deals' with each other about which norms are accepted across the system. The norms are not formally specified but instead are a result of consensus between agents, leading to emergent social norms. This is achieved by essentially merging mentalistic and commitment approaches by making beliefs and desires 'common' or 'public' and communicating them through speech acts. A norm, once accepted by an agent, influences the agent's goals and behaviours, which is similar to the 'internal advisor' approach introduced in [48]. This approach is different from ours as it facilitates social norms, but no enforcement from an authority perspective.

Finally, Alechina *et al.* [51] introduce a fully decentralised approach to norm monitoring for open multi-agent systems. In this model, agents monitor each other and are incentivised with tokens when they detect violations, which can then be spent on taking actions. The main limitation is that this approach is only applicable to permanently forbidden actions and doesn't consider more complex norms that arise dynamically.

In general, while advancements in norm enforcement in MAS are promising, they often take an approach that is tailored to a specific use case and do not attempt to capture all the complexities of normative systems in a generic way. Focus is primarily on ex-post enforcement and violation detection, as opposed to our approach in which agents can actively query for and get notified of normative relations.

Chapter 9

Conclusion

In this thesis, we made efforts towards designing a generic communication protocol for normative actor systems. In order to capture a wide range of normative systems and enforcement strategies for the protocol to support, we laid a foundation in the form of a categorisation of currently existing normative systems, both automated and with human intervention. To the identified categories, we then applied a new approach with the introduction of normative actors, which we divided into distinct roles in order to create structure in the initially unstructured categorisation of systems. By analysing the communication this introduction lead to, we identified six distinct communication patterns that should be supported, and created a list of 21 requirements for the protocol, divided between the actor roles. From this foundation, we started our design efforts by formulating a messaging framework, with formally defined messages based on speech acts. Our approach leveraged a commitment-based formalisation to capture the necessary normative semantics in the messages, effectively simplifying the protocol specifications and providing a flexible base for application to diverse system architectures. Using this framework, we incrementally built up our protocol specifications per pattern and actor role, eventually providing single, role-specific specifications that satisfy all of the requirements. We evaluated the protocol based on a proof-of-concept implementation with multiple case studies, in which we simulated a system that exhibits all of the patterns of enforcement we identified, and related our observations back to the requirements we posed. Our evaluation demonstrated the protocol's potential effectiveness in applications that exhibit all identified patterns, highlighting the ease of usage in normative actor systems to facilitate complex, distributed enforcement strategies. We expect that the abstraction and generalisation provided by the protocol will promote the usability and stability of normative DSLs in production systems. However, it is worth noting that our protocol has yet to be rigorously evaluated in real-world, production-grade environments. Our results also highlight areas that require further research: reaching full qualification in message passing, mitigating synchronisation complexity upon high distribution of normative actors, and extensions of the protocol with further features.

Bibliography

[1] L. T. van Binsbergen, L.-C. Liu, R. van Doesburg, and T. van Engers, "Eflint: A domain-specific language for executable norm specifications," in *Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*, 2020, pp. 124–136. DOI: `10.1145/3425898.3426958`.

[2] L. T. van Binsbergen, M. G. Kebede, J. Baugh, T. van Engers, and D. G. van Vuurden, "Dynamic generation of access control policies from social policies," *Procedia Computer Science*, vol. 198, pp. 140–147, 2022. DOI: `10.1016/j.procs.2021.12.221`.

[3] E. Berman, "A government of laws and not of machines," *Bul rev.*, vol. 98, p. 1277, 2018. DOI: `10.2139/ssrn.3098995`.

[4] S. Sharifi, A. Parvizimosaed, D. Amyot, L. Logrippo, and J. Mylopoulos, "Symboleo: Towards a specification language for legal contracts," in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, IEEE, 2020, pp. 364–369. DOI: `10.1109/re48521.2020.00049`.

[5] Solidity. "Solidity programming language." (2014), [Online]. Available: `https://soliditylang.org` (visited on 02/01/2022).

[6] J. Padget, E. E. Elakehal, T. Li, and M. De Vos, "Instal: An institutional action language," in *Social coordination frameworks for social technical systems*, Springer, 2016, pp. 101–124. DOI: `10.1007/978-3-319-33570-4_6`.

[7] T. Athan, G. Governatori, M. Palmirani, A. Paschke, and A. Wyner, "Legalruleml: Design principles and foundations," in *Reasoning Web International Summer School*, Springer, 2015, pp. 151–188. DOI: `10.1007/978-3-319-21768-0_6`.

[8] A. Artikis and M. Sergot, "Executable specification of open multi-agent systems," *Logic Journal of the IGPL*, vol. 18, no. 1, pp. 31–65, 2010. DOI: `10.1093/jigpal/jzp071`.

[9] S. Legros and B. Cislaghi, "Mapping the social-norms literature: An overview of reviews," *Perspectives on Psychological Science*, vol. 15, no. 1, pp. 62–80, 2020. DOI: `10.1177/1745691619866455`.

[10] W. Sandholtz, "International norm change," in *Oxford Research Encyclopedia of Politics*, 2017. DOI: `10.1093/acrefore/9780190228637.013.588`.

[11] J. Carmo and A. J. Jones, "Deontic logic and contrary-to-duties," in *Handbook of philosophical logic*, Springer, 2002, pp. 265–343. DOI: `10.1007/978-94-010-0387-2_4`.

[12] W. N. Hohfeld, "Some fundamental legal conceptions as applied in judicial reasoning," *Yale Law Review*, vol. 23, no. 1, p. 37, 1913. DOI: `10.1515/9780691186429-004`.

[13] A. J. Jones and M. Sergot, "Deontic logic in the representation of law: Towards a methodology," *Artificial Intelligence and Law*, vol. 1, no. 1, pp. 45–64, 1992. DOI: `10.1007/bf00118478`.

[14] R. Kowalski and M. Sergot, "A logic-based calculus of events," in *Foundations of knowledge base management*, Springer, 1989, pp. 23–55. DOI: `10.1007/978-3-642-83397-7_2`.

[15] G. Ferraro *et al.*, "Automatic extraction of legal norms: Evaluation of natural language processing tools," in *JSAI International Symposium on Artificial Intelligence*, Springer, 2019, pp. 64–81. DOI: `10.1007/978-3-030-58790-1_5`.

[16] C. Castelfranchi, "Formalising the informal?: Dynamic social order, bottom-up social control, and spontaneous normative relations," *Journal of Applied Logic*, vol. 1, no. 1-2, pp. 47–92, 2003. DOI: `10.1016/s1570-8683(03)00004-1`.

[17] D. Grossi, H. Aldewereld, and F. Dignum, "Ubi lex, ibi poena: Designing norm enforcement in e-institutions," in *International Workshop on Coordination, Organizations, Institutions, and Norms in Agent Systems*, Springer, 2006, pp. 101–114. DOI: `10.1007/978-3-540-74459-7_7`.

[18] L.-C. Liu, G. Sileno, and T. van Engers, “Digital enforceable contracts (dec): Making smart contracts smarter,” in *Legal Knowledge and Information Systems: JURIX 2020: The Thirty-third Annual Conference, Brno, Czech Republic, December 9-11, 2020*, IOS Press, vol. 334, 2020, p. 235. DOI: `10.3233/faia200872`.

[19] S. Modgil, N. Faci, F. Meneguzzi, N. Oren, S. Miles, and M. Luck, “A framework for monitoring agent-based normative systems,” in *Eighth International Joint Conference on Autonomous Agents and Multiagent Systems*, International Foundation for Autonomous Agents and Multiagent Systems, 2009, pp. 153–160.

[20] E. Yuan and J. Tong, “Attributed based access control (abac) for web services,” in *IEEE International Conference on Web Services (ICWS’05)*, IEEE, 2005. DOI: `10.1109/icws.2005.25`.

[21] OASIS. “Extensible access control markup language (xacml) version 3.0.” (2003), [Online]. Available: `http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html` (visited on 10/10/2022).

[22] Z. Zheng *et al.*, “An overview on smart contracts: Challenges, advances and platforms,” *Future Generation Computer Systems*, vol. 105, pp. 475–491, 2020. DOI: `10.1016/j.future.2019.12.019`.

[23] M. Mohajeri Parizi, G. Sileno, T. van Engers, and S. Klous, “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*, 2020, pp. 11–20. DOI: `10.1145/3427760.3428339`.

[24] P. Haller, “On the integration of the actor model in mainstream technologies: The scala perspective,” in *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, 2012, pp. 1–6. DOI: `10.1145/2414639.2414641`.

[25] L. Inc. “Akka: Build powerful reactive, concurrent, and distributed applications more easily.” (2021), [Online]. Available: `https://akka.io` (visited on 12/23/2021).

[26] J. Sadock, “Speech acts,” *The handbook of pragmatics*, pp. 53–73, 2004. DOI: `10.1002/9780470756959.ch3`.

[27] J. L. Austin, *How to do things with words*. Oxford university press, 1975. DOI: `10.1093/acprof:oso/9780198245537.001.0001`.

[28] J. R. Searle, *Speech acts: An essay in the philosophy of language*. Cambridge university press, 1969, vol. 626. DOI: `10.1017/cbo9781139173438`.

[29] M. P. Singh, “A social semantics for agent communication languages,” in *Issues in agent communication*, Springer, 2000, pp. 31–45. DOI: `10.1007/10722777_3`.

[30] T. Finin, R. Fritzson, D. McKay, and R. McEntire, “Kqml as an agent communication language,” in *Proceedings of the third international conference on Information and knowledge management*, 1994, pp. 456–463. DOI: `10.1145/191246.191322`.

[31] Fipa, “Fipa acl message structure specification,” *Foundation for Intelligent Physical Agents*, 2002. [Online]. Available: `http://www.fipa.org/specs/fipa00061/SC00061G.html` (visited on 06/07/2022).

[32] M. Ferdous *et al.*, “User-controlled identity management systems using mobile devices,” Ph.D. dissertation, University of Glasgow, 2015.

[33] S. Kube and C. Traxler, “The interaction of legal and social norm enforcement,” *Journal of Public Economic Theory*, vol. 13, no. 5, pp. 639–660, 2011. DOI: `10.2139/ssrn.1626131`.

[34] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, “Cloud monitoring: A survey,” *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013. DOI: `10.1016/j.comnet.2013.04.001`.

[35] J. S. Ward and A. Barker, “Observing the clouds: A survey and taxonomy of cloud monitoring,” *Journal of Cloud Computing*, vol. 3, no. 1, pp. 1–30, 2014. DOI: `10.1186/s13677-014-0024-2`.

[36] A. Malki and M. B. Weiss, “Automating ex-post enforcement for spectrum sharing: A new application for block-chain technology,” *Available at SSRN 2754111*, 2016. DOI: `10.2139/ssrn.2754111`.

[37] ——, “Ex-post enforcement in spectrum sharing,” in *2014 TPRC Conference Paper*, 2014. DOI: `10.2139/ssrn.2417006`.

[38] D. Grossi, H. Aldewereld, and F. Dignum, "Ubi lex, ibi poena: Designing norm enforcement in e-institutions," in *Coordination, Organizations, Institutions, and Norms in Agent Systems II: AAMAS 2006 and ECAI 2006 International Workshops, COIN 2006 Hakodate, Japan, May 9, 2006 Riva del Garda, Italy, August 28, 2006. Revised Selected Papers*, Springer, 2007, pp. 101–114. DOI: 10.1007/978-3-540-74459-7_7.

[39] N. Fornara and M. Colombetti, "Operational specification of a commitment-based agent communication language," in *Proceedings of the first international joint conference on Autonomous agents and multiagent systems: part 2*, 2002, pp. 536–542. DOI: 10.1145/544862.544868.

[40] R. Kibble, "Speech acts, commitment and multi-agent communication," *Computational & mathematical organization theory*, vol. 12, no. 2, pp. 127–145, 2006. DOI: 10.1007/s10588-006-9540-z.

[41] J. Pitt and A. Mamdani, "A protocol-based semantics for an agent communication language," in *IJCAI*, vol. 99, 1999, pp. 486–491.

[42] R. Agerri and E. Alonso, "Normative pragmatics for agent communication languages," in *International Conference on Conceptual Modeling*, Springer, 2005, pp. 172–181. DOI: 10.1007/11568346_19.

[43] X. Yang, L. Liu, N. H. Vaidya, and F. Zhao, "A vehicle-to-vehicle communication protocol for cooperative collision warning," in *The First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004.*, IEEE, 2004, pp. 114–123. DOI: 10.1109/mobiq.2004.1331717.

[44] E.-R. Olderog and M. Schwammberger, "Formalising a hazard warning communication protocol with timed automata," *Models, Algorithms, Logics and Tools: Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, pp. 640–660, 2017. DOI: 10.1007/978-3-319-63121-9_32.

[45] S. Al-Sarawi, M. Anbar, K. Alieyan, and M. Alzubaidi, "Internet of things (iot) communication protocols," in *2017 8th International conference on information technology (ICIT)*, IEEE, 2017, pp. 685–690. DOI: 10.1109/icitech.2017.8079928.

[46] N. Criado, E. Argente, P. Noriega, and V. Botti, "A distributed architecture for enforcing norms in open mas," in *Advanced Agent Technology: AAMAS 2011 Workshops, AMPLE, AOSE, ARMS, DOCM 3 AS, ITMAS, Taipei, Taiwan, May 2-6, 2011. Revised Selected Papers 10*, Springer, 2012, pp. 457–471.

[47] J. M. Alberola, J. M. Such, A. Espinosa, V. Botti, and A. Garcıa-Fornes, "Magentix: A multiagent platform integrated in linux," in *Proceedings of the sixth European workshop on multi-agent systems (EUMAS-2008)*, 2008.

[48] M. M. Parizi, L. T. van Binsbergen, G. Sileno, and T. van Engers, "A modular architecture for integrating normative advisors in mas," in *European Conference on Multi-Agent Systems*, Springer, 2022, pp. 312–329. DOI: 10.1007/978-3-031-20614-6_18.

[49] G. Boella, P. Caire, and L. van der Torre, "Norm negotiation in online multi-player games," *Knowledge and information systems*, vol. 18, pp. 137–156, 2009. DOI: 10.1007/s10115-008-0162-2.

[50] A. Artikis, "Formalising dynamic protocols for open agent systems," in *Proceedings of the 12th International Conference on Artificial Intelligence and Law*, 2009, pp. 68–77. DOI: 10.1145/1568234.1568243.

[51] N. Alechina, J. Y. Halpern, I. A. Kash, and B. Logan, "Decentralised norm monitoring in open multi-agent systems," *arXiv preprint arXiv:1602.06731*, 2016.

Acronyms

ABAC attribute-based access control. 9, 15, 16
ACL agent communication language. 23, 55, 56
BDI beliefs-desires-intentions. 10, 13, 14, 23, 56
DSL domain specific language. 1, 4, 10, 11, 18, 24, 51, 52, 54, 57
MAS multi-agent systems. 8–10, 13, 14, 18, 55, 56
nMAS normative multi-agent systems. 55, 56
OOP object-oriented programming. 13
PoC proof of concept. 5, 15, 21, 24, 40, 42, 43, 49, 51, 52
REPL read-eval-print loop. 5, 10, 11, 20, 47, 48

Appendix A

Full actor role protocol specifications

```
listen
    case requestAct(a,self,A):
        if request accepted
            send requestAct(self,r,A)
            listen
                case permitted(r,self,A):
                    if blockable action
                        send permitted(self,a,A)
                        allow access
                    else
                        send permitted(self,a,A)
                case forbidden(r,self,A):
                    if blockable action
                        send forbid(self,a,A)
                        prevent access
                    else
                        send forbidden(self,a,A)
            end
        else
            send reject(self,a,A)
    case requestDuty(a,self,D):
        if request accepted:
            send requestDuty(self,r,D)
            listen
                case informDuty(r,self,D):
                    send informDuty(self,a,D)
            end
        else
            send reject(self,a,D)
    case informDuty(r,self,D):
        foreach a in relevant actors:
            send informDuty(self,a,D)
        end
    case informDutyTerminated(r,self,D):
        foreach a in relevant actors:
            send informDutyTerminated(self,a,D)
        end
    case informViolatedDuty(self,e,D):
        intervene
    case informViolatedInvariant(self,e,P):
        intervene
    case informViolatedAct(self,e,A):
        intervene
end
```

Protocol A.1: Generic enforcer actor specification.

```
send requestAct(self,e,A) or send requestDuty(self,e,D)
listen
    case informDuty(e,self,D): proceed with updated knowledge
    case informDutyTerminated(e,self,D): proceed with updated knowledge
    case permitted(e,self,A): proceed with updated knowledge
    case forbidden(e,self,A): proceed with updated knowledge
    case permit(e,self,A): execute action
    case forbid(e,self,A): sequence ended
    case rejected(e,self,A): sequence ended
end
```

Protocol A.2: Generic application actor specification.

```
listen
    case requestAct(a,self,A):
        if N(?A)
            send permitted(self,e,A)
        else
            send forbidden(self,e,A)
    case requestDuty(e,self,D):
        send informDuty(self,e,K_self(D))
    case inform(m,self,P):
        if N(P) ⇝ V_D:
            send informViolatedDuty(self,e,D)
        if N(P) ⇝ V_I:
            send informViolatedInvariant(self,e,P)
        if N(P) ⇝ D:
            send informDuty(self,e,D)
        if N(P) ⇝ ¬D:
            send informDutyTerminated(self,e,D)
    case informEvent(m,self,E):
        if N(E) ⇝ V_D:
            send informViolatedDuty(self,e,D)
        if N(E) ⇝ V_I:
            send informViolatedInvariant(self,e,P)
        if N(E) ⇝ D:
            send informDuty(self,e,D)
        if N(E) ⇝ ¬D:
            send informDutyTerminated(self,e,D)
    case informAct(m,self,A):
        if N(A) ⇝ V_D:
            send informViolatedDuty(self,e,D)
        if N(A) ⇝ V_I:
            send informViolatedInvariant(self,e,P)
        if N(A) ⇝ V_A:
            send informViolatedAct(self,e,A)
        if N(A) ⇝ D:
            send informDuty(self,e,D)
        if N(A) ⇝ ¬D:
            send informDutyTerminated(self,e,D)
end
```

Protocol A.3: Generic reasoner actor specification. We exclude the specification of information queries upon missing input, because this is a reactive action that can happen during the evaluation of each of the messages defined here. For reference, the separate specification of the information fetch pattern for the reasoner is specified again in Listing A.4.

```
listen
    case msg:
        if N(msg.content) raises Exception for missing P:
            send request(self,i,P)
            listen
                case inform(i,self,P):
                    if P.value ≠ ⊥:
                        retry N(msg.content)
            end
end
```

Protocol A.4: Specification for the behaviour of the reasoner upon missing input: this specification can be executed during the evaluation of each of the messages in the generic specification above.

```
listen
    case msg:
        e <- translate(msg)
        switch e:
            case e is status update:
                send inform(self,r,P)
            case e is action:
                send informAct(self,r,A)
            case e is system event:
                send informEvent(self,r,E)
end
```

Protocol A.5: The generic specification for the monitor actor role.

```
listen
    case request(r,self,P):
        send inform(self,r,$K_{self}$(P))
end
```

Protocol A.6: The generic specification for the information actor role.

Appendix B

Case studies: eFLINT specification

In our case studies we used a single eFLINT specification which contains all the norms that regulated our system. This eFLINT specification can be seen in Listing B.1.

```
Fact user
Fact owner Identified by user
Fact tenant Identified by user
Fact price Identified by Int
Fact percentage Identified by Int

Fact document
Fact rental-agreement Identified by tenant * document
Fact rent-price Identified by rental-agreement * price
Fact is-social-housing Identified by rental-agreement

Fact deposit Identified by rental-agreement * price

Fact deadline
Fact rent-payment Identified by tenant * price * deadline
Fact rent-due Identified by rent-payment

Open Fact yearly-income Identified by tenant * price

// Access control policy for tenant document access
Act access-document
  Actor tenant
  Related to document
  Holds when
    rental-agreement(tenant, document)
  Conditioned by
    rental-agreement(tenant, document)

// Indexation action
Act index-agreement
  Actor owner
  Related to rent-price1, percentage
  Holds when rent-price1
  Conditioned by
        percentage <= 2
    || (percentage <= 3
        && (Exists yearly-income : yearly-income.tenant == rent-price1.rental-agreement.tenant
                                && yearly-income.price > 45000))
    || (percentage <= 4 && !is-social-housing(rent-price1.rental-agreement))
  Terminates rent-price1
  Creates rent-price(rent-price1.rental-agreement, rent-price1.price * (1 + percentage / 100))

// Duty to pay rent
Duty pay-rent
  Holder tenant
  Claimant owner
  Related to rent-payment
  Holds when rent-payment
  Violated when rent-due(rent-payment)


```

```
// Act of paying rent, terminates the pay-rent duty
Act make-rent-payment
  Actor tenant
  Recipient owner
  Related to rent-payment
  Holds when rent-payment
  Terminates pay-rent()

// Act of terminating an agreement, leads to refund duty
Act terminate-agreement
  Actor tenant
  Recipient owner
  Related to rental-agreement
  Holds when rental-agreement
  Creates refund-deposit()

// Duty to refund the deposit
Duty refund-deposit
  Holder owner
  Claimant tenant
  Related to deposit

// The act of refunding a deposit, terminates the duty
Act refund
  Actor owner
  Recipient tenant
  Related to deposit
  Holds when refund-deposit()
  Terminates refund-deposit()
```

Listing B.1: The full eFLINT specification used for our case studies.