A Modular Architecture for Integrating Normative Advisors in MAS

Mostafa Mohajeri Parizi[1], L. Thomas van Binsbergen[1], Giovanni Sileno[1], and Tom van Engers[1,2]

[1] Informatics Institute, University of Amsterdam, The Netherlands
[2] Leibniz Institute, University of Amsterdam/TNO, The Netherlands
{m.mohajeriparizi,g.sileno,vanengers}@uva.nl,ltvanbinsbergen@acm.org

This version of the contribution has been accepted for publication after peer review but is not the Version of Record and does not reflect post-acceptance improvements or corrections. The version of record is available online at: https://doi.org/10.1007/978-3-031-20614-6_18. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

Abstract. This paper introduces a modular architecture for integrating norms in autonomous agents and multi-agent systems. As the interactions between norms and agents can be complex, this architecture utilizes multiple programmable components to model concepts such as adoption of personal and/or collective norms (possibly conflicting), interpretation and qualification as mappings between social and normative contexts, intentionally (non-)compliant behaviors, and resolution of conflicts between norms and desires (or other norms). The architecture revolves around *normative advisors*, that act as the bridge between intentional agents and the institutional reality. As a technical contribution, a running implementation of the architecture is presented based on the ASC2 (AgentScript) BDI framework and eFLINT normative reasoner.

1 Introduction

Norms are widely used to represent ethical, legal and social aspects of multi-agent systems, and normative multi-agent systems are deemed to provide a powerful model for norm-governed complex cyber-infrastructural systems that include social agents (humans, organizations, or other bodies), infrastructural systems, norms and their interactions [11]. At least, designing computational agents that reason with norms—technical instances of *normative agents*—requires having a suitable computational model for reasoning with norms. This is a challenging task because norms are more than a set of formal rules extracted from a legislative text: they emerge from multiple sources with different degrees of priority, require interpretation before being encoded, and qualification to be applied to a social context. Furthermore, they continuously adapt, both in expression and in application [3]. This entails that there are many challenges in modelling the interactions between agents and norms. At *content* level, multiple normative sources may be concurrently relevant, and/or multiple interpretations of the same normative sources may be available (e.g. retrieved from previous cases),

and these may be possibly conflicting. Intuitively, enabling to maintain those in a modular fashion is a suitable, and, even necessary precondition for update/adaptation actions, where norms can be changed on the fly, and agents may decide at run-time e.g. to change the relative priority between normative components, requiring some explicit meta-reasoning about those norms. At *method* level, there is still an ongoing debate on what is the most adequate representation model for norms, and on methods for normative reasoning (eg. synthesizing norms [24], managing conflicts [16]). Allowing the recourse to external tools, and supporting programmability of the coordination level, greatly empowers modelers/programmers/designers to test and compare different choices. Finally, at *functional* level, most of the knowledge instilled in norms concerns a whole social system, but only part of the system is contingently relevant to the agent. Enabling the system design so that it distributes and localizes the inferential load at best (and at need) externally from the decision-making seems the most efficient option.

Contribution Based on these requirements, this work proposes an abstract architecture that encapsulates norms—encoded in terms of normative relationships as in Hohfeld's framework [20]—in a MAS. The architecture centers around *normative advisors* that can be utilized by (other) agents in the MAS as a sort of council about the institutional state of affairs and normative relations between agents, highlighting and enabling the mapping between the social and institutional views of the environment. Agents may resort to personal or to collective advisors, depending on the decentralization constraints set up by the designer. As a technical contribution, we present a practical implementation of this architecture that relies on the AgentScript BDI framework (ASC2) [23] for programming agents, and norm specification framework eFLINT [2] for encoding norms.

Related Work The B-DOING framework [16] explores logical relations between belief, desire, obligation, intention, norms and goals in agents and their interactions like conflicts and possible approaches to balance them in agent's behavior. Similarly, the BOID architecture [9, 25] proposes a belief, obligation, intention and desire architecture with a feedback loop to consider the effects of actions before committing to them. These studies (and many others, e.g., [14, 32, 12]) propose extensions to the BDI architecture to add (regulative) norms as part of the agents' mind and to solve conflicts via pre-defined rules. The main issue with these works is that putting all relevant normative sources (and logical conflict resolution rules) within the agent is typically not feasible in a real system with complex interactions between norms, actions, and their possible effects on different stakeholders. Consequently, in our approach we propose delegating the normative reasoning to external components, here named *normative advisors*.

In [10], an approach is proposed for ethical reasoning in MAS by programming *ethical governors*. In this approach, when an agent needs ethical advise about certain actions, it will ask dedicated agents named evidential reasoners, providing evidence to an arbiter agent, that in turn picks a suggestion with a predefined strategy, and send it to the requesting agent. The concept of external advisor agents is similar to our proposal. However, while their approach focuses

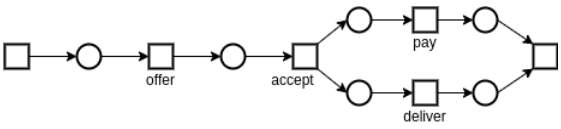

Fig. 1: A sale transaction as a *Petri net* workflow.

on agents only *querying* for suggestions when they require advise, in our approach the normative advisors keep an explicit institutional state of the environment and are able to notify the agent about different normative events (e.g. new duties or violations). The work in [21] introduces Jiminy advisors that reflect the moral stance of an agent; their approach leans towards using these advisors for coordination purposes specifically when there are multiple agents (stakeholders) that follow different norms and moral dilemmas may arise. Formal argumentation methods are then used to resolve these dilemmas. In the present work, we start from a more neutral stance towards what specific methods/approach needs to be taken to represent norms and resolve conflicts; our aim is to discuss the design of a more general system architecture, whilst presenting a specific implementation of the architecture based on certain implementation decisions.

Structure of the Document Section 2 gives background on the core components that the proposal uses by providing some detail on the AgentScript/ASC2 and eFLINT frameworks used for the implementation. Section 3 lays out the theoretical framework for the proposed architecture, whereas Section 4 describes details of its implementation. Section 5 reflects on the capabilities of our implementation, suggests future directions, and draws connections with related work.

2 Core Components

To illustrate our approach, we will consider as a running example a marketplace environment consisting of buyer and seller agents. This target domain can be seen as an abstract model of many real-world domains, e.g. data market-places and more in general data-sharing infrastructures, electronic trading infrastructures, etc. The process model of a individual sale transaction—prototypical example of bilateral contract—is represented as a workflow through a Petri-net in Figure 1. A seller offers a buyer an item for a certain price. If the buyer accepts the offer, then the seller is expected to deliver the aforementioned item to the buyer, and the buyer is expected to pay the seller the price agreed upon (in any order). The workflow is a simplified representation of the normative mechanisms in place during an actual sale transaction (cf [29]). Furthermore, it does not consider the intentional aspects on the agents during the transaction, e.g. based on which desires or goals the agents may be willing to engage in the transaction, as these concepts remain external to norms.

```
needed_item("Book1").
fair_price("Book1", 5).
have_money(10).

!init(#sale_advisor.getClass, "sale.eflint", "BuyerAdvisor").

+!init(AgentType,EFFile,Name) => #spawn_advisor(AgentType, EFFile, Name).

+offer(Item ,P) =>
    #achieve("BuyerAdvisor", perform(offer(Source, Self, Item, Price)));
    !consider_buying(Source, Item ,Price).

+!consider_buying(Seller, I, P) :
  needed_item(I) && fair_price(I, FP) && P =< FP && have_money(M) && M >= P =>
    #tell(Seller, accept(I, P));
    +pending(accept(I, P)).

+acknowledge(accept(I, P)) : pending(accept(I, P)) =>
    -pending(accept(I, P));
    #achieve("BuyerAdvisor", perform(accept(Self, Buyer, I, P))).

+duty_to_deliver(Seller,Buyer,I) : Source == "BuyerAdvisor" && Buyer == Self =>
    +expected_delivery(Seller,I).

+delivery(Sender, Item) : expected_delivery(Sender, Item) =>
    -expected_delivery(Sender, Item);
    #achieve("BuyerAdvisor", perform(deliver(Sender, Self, Item)).

+duty_to_pay(Buyer, Seller, P) : Source == "BuyerAdvisor" && Buyer == Self  =>
    !pay(Seller, P).

+!pay(Seller, P) : have_money(M) && M >= P =>
    #pay(Seller, P);
    #achieve("BuyerAdvisor", perform(pay(Self, Seller, P)).

+!pay(Seller, P) => ... ALTERNATE APPROACH TO PAYMENT ...
```

Listing 1: Buyer agent script as an ASC2 program

2.1 Intentional Agents

Intentional agents are generally approached in the computational realm via the *belief-desire-intention* (BDI) model [27], to specify agents acting in dynamic environments with rational behavior. The BDI model refers to three human mental attitudes [8]: *beliefs* are the factual and inferential knowledge of the agent about itself and its environment; *intentions* are the courses of action the agent has committed to; *desires*, in their simplest form, are objectives the agent wants to accomplish. In practice, BDI agents also include concepts of *goals* and *plans*. Goals are concrete desires, plans are abstract specifications for achieving a goal, and intentions then become commitments towards plans. Multiple programming languages and frameworks have been introduced to operationalize the BDI model, such as AgentSpeak(L)/Jason [26, 7], 3APL/2APL [13], Astra [15] and AgentScript/ASC2 [23].

AgentScript/ASC2 Agent Framework ASC2 is an agent-based programming framework and languge with a syntax very close to AgentSpeak(L), con-

sisting of initial beliefs and goals, and plans. Initial beliefs are a set of Prolog-like facts or rules that define the first beliefs the agent has, and, initial goals designate the first intentions to which the agent commits. Plans are potentially non-grounded reactive rules in the form of `E : C => A`, where `E` is the head of the plan which consists of a trigger and a predicate, the trigger can be one of `+!,-!,+,-,+?` respectively used for achievement goals, failure (of) goals, belief-updates (assertion, retraction) and test goals. The expression `C` is the context condition that can be any valid Prolog expression, and `A` is the body of the plan that consists of a series of steps that can include belief-updates (`+belief,-belief`), sub-goal adoption (`!goal`), primitive actions (`#action`) which may be any arbitrary callable entity on the class path, variable assignments, and control flow structures (loops and conditionals). It is said that a plan is *relevant* for an event `G` iff the event-type of `G` matches with the trigger and the content of `G` matches with the predicate of `E`. Furthermore, a relevant plan is *applicable*, iff `C` is a logical consequence of agent's belief-base. When an agent receives an event, as a reaction, after finding the relevant, and then applicable plans, it will use a selection function to choose a plan to execute as an intention. This process is typically called *planning* in BDI agents.

The communications interface of the agents is based on speech act preformatives and implemented with actions like `#achieve` which relays an achievement goal event, `#tell` and `#untell` which relay belief-update events, and `#ask`/`#respond` which can be used between agents as synchronous communication with test goal events. As an example of an AgentScript program and continuing with the example, Listing 1, presents the script of a buyer agent. The initial beliefs (lines 1-3), initial goals (line 5), and plan rules (line 7 and onwards) are the components of the script. The script is further explained in Section 4.2.

2.2 Norms and Normative (Multi-Agent) Systems

Following Gibbs, norms are "a collective evaluation of behavior in terms of what it ought to be; a collective expectation as to what behavior will be; and/or particular reactions to behavior, including attempts to apply sanctions or otherwise induce a particular kind of conduct" [19]. This definition is relevant to our purposes as it gives primacy to action (rather than to situations). In the context of multi-agent systems, and even more of in MAS, an action-centered approach is intuitively more suitable, as actions are the only means agents have to intervene in the environment, resulting in normative consequences.

Categories of Norms Norms are traditionally distinguished between *regulative* and *constitutive* norms [28, 5, 30]. Regulative norms regulate behavior existing independently of norms, and are generally expressed in terms of *permissions*, *obligations* and, *prohibitions* (e.g. traffic regulations). Constitutive norms determine that some entity (e.g. an object, a situation, an agent, a behaviour) "counts as" something else, creating a new institutional entity that does not exist independently of these norms (for example, money as a legal means of payment).

The concept of *institutional power* is particularly relevant in the context of constitutive norms, as it is used to ascribe institutional meaning to performances (e.g. raising a hand counts as making a bid during an auction). A conceptual framework that contains both deontic and potestative dimensions is the one proposed by Hohfeld [20], whereas deontic logics, although much more studied in normative multi-agent systems [18, 17], by definition focuses on regulative norms.

Normative Systems The term normative system can be used for a system of norms, as well as for multi-agent system guided by norms. In our work we focus on the latter. We apply the so-called *normchange* definition of normative MAS system by Boella et al. [6]: "a multi-agent system together with normative systems in which agents on the one hand can decide whether to follow the explicitly represented norms, and on the other the normative systems specify how and in which extent the agents can modify the norms". This definition does not assume any particular inner workings of the agents except that they should be able to somehow *decide* whether to follow the norms or not and they should be able to *modify* them. Furthermore, there is no assumption about the representation of the norms, except that they should be *explicit* (i.e. a 'strong' interpretation of the norms [4]) and *modifiable*.

The eFLINT Norm Language The eFLINT language is a DSL designed to support the specification of (interpretations of) norms from a variety of sources (laws, regulations, contracts, system-level policies such as access control policies, etc.) [2, 1]. The language is based on normative relations proposed by Hohfeld [20]. The type declarations introduce types of *facts*, *acts*, *duties* and *events*, that together define a transition system in which states—sets of facts—transition according to the effects of the specified actions and events. The transitions may output violations if triggered by an action with unfulfilled preconditions (e.g. only sellers can make offers) or if any *duties* are violated in the resulting state.[3]

Listing 2 shows an eFLINT specification for our running example. The `Actor` and `Recipient` clauses and `Holder` and `Claimant` clauses of act- and duty-type definitions establish constructs mapping to Hohfeldian power-liability and duty-claim relationships. The `Creates` and `Terminates` clauses describe the effects of actions when performed, enabling reasoning over dynamically unfolding scenarios. An instance of `offer` can be performed without any pre-conditions and it holds when there is a `seller` instance. The act `accept` is only available after an offer: accepting a non-existing offer is considered a violation of the power to accept offers. Acceptance of an offer creates the two act instances `pay` and `deliver` which can be performed in any order. The duties express that the `pay` and `deliver` actions are expected to be performed by their respective holder

[3] In eFLINT, actions capture a permission dimension as well as a power dimension, following from the design choice that a violation is raised when an action with unfulfilled preconditions is performed. Other computational frameworks propose a clear-cut separation between deontic and potestative categories [31].

```
// fact definitions
Fact buyer
Fact seller
Fact item
Fact price Identified by Int

// act definitions
Act offer Actor seller Recipient buyer
  Related to item, price
  Holds when seller
  Creates
    accept(buyer, seller, item, price)

Act accept Actor buyer Recipient seller
  Related to item, price
  Creates
    pay(buyer, seller, price),
    duty_to_pay(buyer, seller, price),
    deliver(seller, buyer, item),
    duty_to_deliver(seller, buyer, item)
```

```
Act pay Actor buyer Recipient seller
  Related to price
  Terminates
    duty_to_pay(buyer, seller, price)

Act deliver Actor seller Recipient buyer
  Related to item
  Terminates
    duty_to_deliver(seller, buyer, item)

// duty definitions
Duty duty_to_pay
  Holder buyer
  Claimant seller
  Related to price

Duty duty_to_deliver
  Holder seller
  Claimant buyer
  Related to item
```

Listing 2: eFLINT Specification for *Sale Transaction* norms

after they are created as part of the `accept` action. As described in Listing 2, no violation conditions are associated with the duties.

3 Normative MAS via Normative Advisors

Our approach is based on the introduction of *normative advisors* that enable intentional agents to communicate with external norm reasoners. We assume the parent agent is a BDI agent, i.e. it has the capabilities to reason with beliefs, desires and intentions. The tasks of maintaining an institutional perspective (state) and reasoning about specific sets of norms is delegated to the advisors. The advisors are initialized with a particular norm specification and maintain an institutional perspective on the environment, which is continuously updated at run-time. A normative advisor is therefore viewed as maintaining (inferential mechanisms necessary to operationalize) a *norm instance*. Both regulative and constitutive norms are taken into account. The normative (institutional) state of the world is stored in a way that can both be queried and updated at any time. An update can generate normative events that the agent is to be notified about. Through the normative advisors, a social agent acquires various capabilities to interact with norms. As a consequence, norms interactions become programmable parts of the agent, realizing our goal of using norms for behavioural coordination between agents and for specifying qualification processes between social and normative contexts. With such an infrastructure, an agent becomes:

- able to *adopt* or *drop* any number of norm sources as norm instances;
- able to *qualify* observations about their environment as normatively relevant updates, and conversely to *respond* to normative events by acting accordingly in their environment;

- able to *query*, *update*, *revert*, *reset* a normative state of any norm instance;
- able to *receive and process* or *ignore* normative events (e.g. new claims)
- able to *follow* or *violate* normative conclusions (e.g. obligations) or query responses (e.g. permissions and prohibitions)
- able to *modify* any of the above abilities at run-time.

Normative reasoning occurs based on these inputs—triggered by queries or updates— with all conclusions made available as internal events to the advisor. Note that an agent can have multiple advisors for different (instances of) sets of norms. An agent is free to qualify observations about events in the environment, other agents' actions, its own beliefs and actions—or any combinations of these—and report the resulting observations to the relevant normative advisors. In other words, this infrastructure makes possible a rich, recursive interaction between behavioral decision-making and normative reasoning. The proposed model supports a number of programmable concepts applicable to different functions:

1. *Perception*: which internal/external events are received and processed or otherwise ignored;
2. *Reaction and planning*: what are the relevant reactions to an event, which reactions are applicable in the current context and which reaction is the most preferred one to execute;
3. *Norm adoption*: when and how to adopt or drop a set of norms;
4. *Qualification of social context*: how an event or query is qualified, i.e. which is its normative counterpart for each norm instance;
5. *Querying*: when and how the normative state of an instance needs to be queried (e.g. for compliance checking);
6. *Reporting*: what events/updates are reported to which norm instances;
7. *State change*: how a normative event changes a norm instance's state;
8. *Event generation*: what normative events are created as the result of an instance's state update;
9. *Qualification of normative concepts*: which events should be raised as the result of what normative conclusions reported by a norm instance.

To concretize the proposed approach, we will discuss at higher-level why it is feasible to implement a system meeting these requirements by utilizing an AgentSpeak(L)-like BDI framework (AgentScript/ASC2, in particular) and a norm reasoner that can store an updatable and queryable normative state, generating events on updates (eFLINT, in particular). Perception, planning and execution are basic core functions of reactive BDI agents as those specified via AgentSpeak(L), i.e. when an event is received, the agent performs a sequence of actions in reaction. Qualification can be encoded as part of planning: what reaction is selected for an event (or a series of events) in any context signifies how that event is qualified. Norm adoption, querying and reporting intuitively become part of this reaction. Note however that querying can also be part of planning, as a query response may affect what reactions are applicable. State changes happen internally to the norm instance as the result of reporting, and then normative events are generated, which are in turn qualified as events by

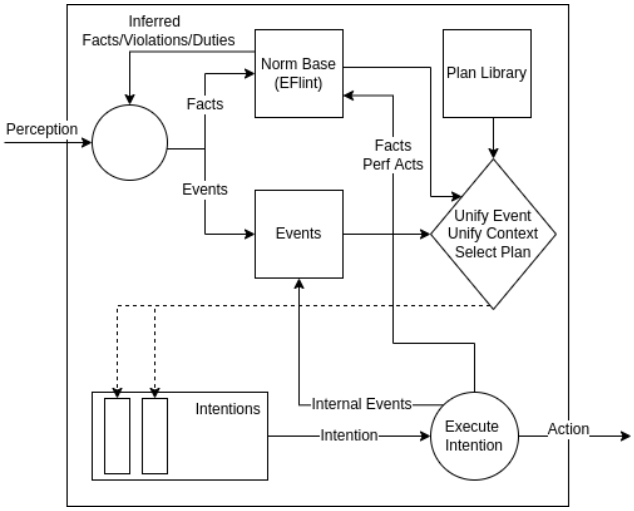

Fig. 2: The architecture of normative advisors.

the agent, creating a full circle. Finally, if both the BDI framework and norm framework allow for run-time changes, as is the case with ASC2 and eFLINT, then all aspects are changeable and dynamic.

4 Implementation

This section describes an architecture for advisors and discusses how the ASC2 BDI framework and the eFLINT normative reasoning framework are used to implement the proposed architecture. The eFLINT framework is used to implement the norm base. The advisors as well as the intentional agents that employ them are defined in ASC2. Our implementation benefits from the modularity provided by ASC2, allowing easy replacement of different parts of the agent [23] and the Java API provided by eFLINT.

4.1 Normative Advisor Architecture and Decision-Making Cycle

Figure 2 provides an overview of the architecture of a normative advisor, inspired by the BDI architecture of Jason [7]. A normative advisor can be seen as a BDI agent in which the (typically Prolog-like) belief-base is replaced by the norm reasoner, thus, the reasoning of the agent is replaced with normative reasoning. Apart from the differences between a general-purpose reasoner (e.g. Prolog) and a norm reasoner (e.g. eFLINT), the main architectural differences of an advisor with a typical BDI agents are: (1) the belief-base (in this case, the norm-base) of the agent can generate more than just belief-update (or fact-update) events, it may now also raise duty events, act (enabled/disabled) events, and violation events upon which the agent can react according to its plan library; (2) from the

execution context of a plan alongside fact-update actions (`+fact` and `-fact`), there can now be act-perform actions (`#perform(act)`). These differences arise from the fact that unlike a general-purpose reasoner like Prolog that typically uses backward-chaining to infer facts based on queries, the eFLINT framework also produces information in a forward-chaining manner, thus generating more events for the advisor to process. Despite these modifications, the core of the AgentScript DSL, and the capabilities of the framework, like goal adoption, communication, and performing arbitrary primitive actions, remain the same as with 'traditional' intentional agents.

Decision-Making Cycle When an advisor receives an external or internal event, and if it is a fact-update, then it will be sent to the norm base. If the event is an achievement or test event, it will be sent to the event queue. Events are taken from the event queue by an event-selection function, at which moment the head of the event is matched with the plan library to find all the relevant plans. The context conditions of relevant plans are checked against the normative state of the norm base in order to select only applicable plans. Then, a plan selection function selects one applicable plan and turns the (execution of that) plan into an intention, and, consequently, an intention selection function chooses intentions for execution. If the body of the plan includes any fact-update actions (`+fact` and `-fact`) or act performance (`#perform(act)`), then these are sent to the norm base. Whenever there is any update committed to the norm base, there could be multiple new events or new facts derived by the normative reasoner that are sent back to the advisor as internal events.

These new capabilities are also the result of replacing the Prolog reasoning engine with the eFLINT reasoner. Any Boolean expressions in the DSL can now refer to pre-defined predicates corresponding to eFLINT keywords for querying the norm base: `holds` is used to check if a fact (or act, duty, etc.) holds, `enabled` whether the preconditions of an act hold, and `violated` checks if a duty was violated. A comprehensive list of possible interactions with the eFLINT norm reasoner is given in the next subsection.

4.2 eFLINT Norm Base Implementation

The eFLINT language is implemented in the form of a reference interpreter in Haskell[4]. As discussed in [2], the interpreter can run in a 'server mode' in which it listens to requests on a certain port and produces responses according to some API. A layer has been developed on top of the server to maintain multiple server instances as is need for supporting multiple advisors with a norm base each. An eFLINT server instance can receive the following **requests**:

- *Fact creation/termination/obfuscation.* A created fact (instances of fact-types, act-types, duty-types and event-types are referred to as facts) is set to 'true' in the knowledge base, a terminated fact to 'false' and any existing truth-assignment is removed when a fact is obfuscated.

[4] Publicly available online https://gitlab.com/eflint/haskell-implementation.

- *Triggering an action or event.* Instances of act-types and event-types can be *triggered*, resulting in the effects of the action or event manifesting on the knowledge base (`#perform` in Listing 3). These effects create, terminate, and/or obfuscate certain facts, as listed in the corresponding (post-condition) clauses of the type declaration of the triggered action/event. Multiple actions/events can be triggered at once because of the synchronization mechanism discussion in Section 5.
- *A query in the form of a Boolean expression.* The expression is evaluated in the context of the current knowledge base and can be used to establish whether a certain fact holds true in the current knowledge base, whether an action is enabled (`holds` in Listing 3) or whether a duty is violated, etc.
- *The submission of a new type declaration* or *the extension of an existing type.* Both have the effect of modifying the norms in the sense that the underlying transition system is modified.

Every request can be associated with additional context information in the form of truth-assignment to facts that override any conflicting assignments in the current knowledge base (e.g. the current UNIX time). This mechanism can also be used to provide truth-assigments for 'open types'—types for which the closed world assumption does not hold. An eFLINT instance generally operates *synchronously*, i.e. will only send out information in **responses** to requests[5], updating the sender upon the following:

- Any created, terminated, and/or obfuscated facts. Note that this includes changes to facts that are (or were previously) derived from other facts and in this sense were indirectly modified by the incoming request
- Any changes to normative positions regarding duties, i.e. whether a duty is no longer held by an actor or whether a duty is now held by an actor (e.g. `-duty` and `+duty` in Listing 3). Violated duties are also reported as such.
- Any changes to normative positions regarding powers, i.e. which actions became (or are no longer) enabled. If the incoming request was triggering one or more actions that were not enabled, the effects of the actions still manifest, but the violations are reported.
- In response to a query, the reasoner responds with the result of the query (state is unchanged).
- If the incoming request requires the evaluation of a fact for which no truth-assignment is given and which is an instance of an open type, then an exception is raised and reported to the sender of the request. Evaluation is interrupted and the state remains unchanged.[6]

All changes to facts' truth-assignment, normative positions and violations register as internal events in the normative advisor (as shown by Listing 3), which will process and possibly report them according to its script.

[5] A clock event can be used to receive synchronous updates periodically.

[6] The exception can be used by the parent of the advisor to acquire the missing information, e.g. from another agent in the MAS.

```
+?permitted(A) : enabled(A) => #respond(true).
+?permitted(A) => #respond(false).

+!perform(A) : enabled(A) => #perform(A).
+!perform(A) => #tell(Parent, failed(A)).

+duty(D) => #tell(Parent, D).
-duty(D) => #untell(Parent, D).
```

Listing 3: AgentScript specification of a normative advisor.

4.3 Spawning and Interacting with Advisors

Scripts of normative advisors (written in AgentScript, the ASC2 DSL) run on top of the advisor architecture and give the programmer access to the norm reasoner, both providing its input in the form of queries and updates and reacting to the normative events the reasoner generates. In such sense, advisors functionally act as "bridges" or chain of transmission between institutional and social realms. Listing 3 shows a basic script for an advisor in our running example. The advisor has four plans related to acts and two related to duties. The synchronous query `+?permitted` receives an act and responds with `true` if the given act is "permitted" according to the underlying norm reasoner—in the case of eFLINT "enabled"—and `false` otherwise. The agent has similar plans to asynchronously submit (or not) the performance of acts (`+!perform`) to the norm reasoner. The last two plans are triggered when the internal norm reasoner creates (`+duty`) or terminates (`-duty`) a duty. The advisor informs their parent of these changes. The fragment demonstrates that observations about created and terminated duties are communicated to the intentional actor (`Parent`, the agent that spawned the advisor) and that an action `A` can only be performed when it is enabled according to the norm reasoner (or fails otherwise); however this script does not demonstrate all the features possibly delivered by the architecture such as internal events for violations, enabled/disabled acts, and asserted/retracted facts. Absence of power is mapped here to a prohibitions as, for example, is common in access-control systems. Other solutions may be more suitable in other contexts.

Running Example To demonstrate spawning and interacting with a normative advisor, consider again Listing 1 in which a script for a buyer agent is given. Together, Listings 1, 2, and 3 show the DSL code for buyer agent in the market-place as presented on the right side of the Figure 3. The buyer agent spawns a normative advisor, which in turn spawns an eFLINT server (norm reasoner). The buyer has its own beliefs and desires: there is a specific item

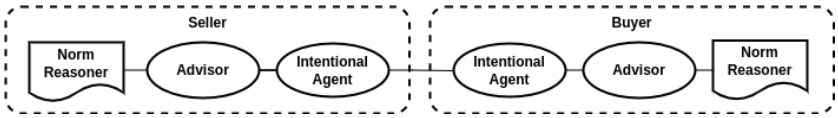

Fig. 3: Market-place model

that it needs (`needed_item`), it has a belief about the fair price (valuation) for that item (`fair_price`) and it has a belief about how much money it possesses (`have_money`). When this agent receives a `+offer` message about an item and its price, first it interprets it as an `offer` act and sends it to its advisor. Next, it adopts a goal of `consider_buying` that item for the price. This goal has one plan associated to it, which checks if the agent actually needs that item, if the price is considered a fair price and finally if the agent has enough money to buy that item. If this is all true, it sends a `accept` message to the agent that made the initial offer. Unlike before, this alone does not constitute performing the normative `accept` act. Instead, it waits until it receives a `+acknowledge` message from the seller before communicating acceptance to the advisor. This extra-institutional step for the buyer to qualify the act of `accept`, is an example of context-based qualifications in intentional agents.

When the `accept` act is submitted to the norm reasoner, the two previously mentioned duties of `duty_to_pay` and `duty_to_deliver` are generated and sent by the advisor to the intentional part of the Buyer. For the `duty_to_deliver` the agent is the *claimant* (it holds the expectation of performance); it could be that the agent asks the seller agent at this point to deliver the item, but instead, with the implicit assumption that the Seller agent is also compliant to the same set of norms, this agent simply adds this expectation to its belief-base and only when it has an observation of `delivery`, it will remove this expectation and send the `deliver` act to the advisor.

For the `duty_to_pay` the agent is the duty-holder (it has the obligation to perform) and reacts to this duty by adopting the goal `pay` (i.e. the agent desires to be compliant). There are two plans for this goal, the first one is straightforward and is applicable if the agent has the required amount of money; it will simply pay the Seller and submit this act to the advisor. However, the second plan (not implemented) is applicable if the agent does not have enough money, which means it needs to find alternative paths to relieve this duty, e.g. by borrowing from another agent or even asking another agent to pay the seller instead. Specifying these alternatives requires to further encode the models of either the agents, or the norms, or both. Although relevant in practical applications, this level of detail can be overlooked in the present context. Instead, in the next section we will elaborate on various interesting opportunities of extending this straightforward example and reflect on the design of advisors.

5 Discussion

This paper presents an approach to embed (constitutive and regulative) norms into a MAS in a modular and versatile manner, enabling autonomous agents to reason with norms.

Inline with MAS, and distributed computing in general, we consider **consistency as a consequence** of how a system is set up rather than it being ensured by the framework through which the system is built. This allows for a kind of partial consistency that enables freedom for local deviations that are not harmful

to the overall system behavior. In our approach, norm adoption and qualification is done by each individual agent, such that their view on the normative state of the world is dependent on both their script and their (bounded) perception. Particularly desirable for social simulations, we can define agents that adopt and follow the same norms but have different conclusions on the normative state of affairs because they have had different observations. Alternatively, agents do not have to follow the same norms but might still be able to behave in a coordinated fashion. An example of the latter in our sales example is a buyer that believes, on top of the existing norms of our example, that deliveries should be done before payments. The buyer can behave according to their own norms without violating the norms adopted by sellers, even though their norms are different.

As presented in the previous sections, our running example shows how **coordination** between agents is achieved by adopting norms and deciding whether to comply with norms. The example relies on the agents wanting to comply, and therefore exhibiting coordinated behavior. In more adversarial environments, additional *enforcer* agents can be added to provide (positive and negative) incentives to comply. For example, our marketplace can be extended with an agent that acts like a market authority. By responding to violations raised by their advisor(s), the market authority can apply **ex-post enforcement** of norms on the market participants. For example, a buyer refusing to pay can receive a warning or, in the case of continued non-compliance, be banned from the market altogether. This further demonstrates the versatility of our approach: it does not impose *a priori* centralized/decentralized governance or ex-ante/ex-post enforcement. Instead, this approach gives the system designer the flexibility to choose, design and test what their system requires.

Referring to the requirements in Section 3, the notion of **adopting** was illustrated in the simplest form with the buyer agent in Listing 1 with the `#spawn_advisor` to adopt a norm as an initial goal. The agents also have the `#despawn` action to and **drop** an advisor. However, by adding extra mechanisms in the agent's script, more complex archetypes can be modelled, e.g. the agent may be programmed to keep a score for a certain norm's (and advisor's) "utility" to decide if it is an effective norm to keep adopted.

The notion of **qualification**—necessary to fill the gap between computational forms of law and software [3]— can be performed at various stages, thanks to the multitude of programmable layers in our approach. An example of qualification in the sale transaction is how a seller agent perceives a *pay* act from a buyer agent. While represented as an act in the norms, in the social reality many different actions can be perceived as a payment e.g., cash payment or 3rd-party bank transaction (bank transaction) can be qualified as the act of paying. This qualification rule could have been encoded in the script of an agent. For example, a bank agent can update a seller that they have received new funds as part of a completed transaction. The seller can then determine whether these funds constitute a payment by a buyer for a particular item, and inform the corresponding advisor. The same qualification can also be performed purely within norms. In eFLINT, actions and events are synchronized such that preconditions and effects

```
Fact account
Placeholder sender    For account
Placeholder receiver  For account
Event transaction-completed Related to sender, receiver, price
  Syncs with pay(sender, receiver, price) When buyer(sender) && seller(receiver)
```

Listing 4: An eFLINT fragment connecting a bank transaction to the `pay` action.

of transitions are effectively 'inherited'. In this way, explicit 'counts as' relations between performed actions (transitions) can be specified. This is useful to a) connect actions from various normative sources which are simultaneously applicable to a system and b) connect agent-behavior to institutional counterparts with possible normative consequences. For example, through (b) it is possible to connect the concrete actions by (human or software) actors in a system to the rights and obligations laid out in a contract and through (a) to connect actions within the contract to relevant (inter)national law. Listing 4 shows for instance how a transaction event in a banking system is connected with (qualified as) a payment action in our running example. This means the intentional agent only needs to indicate to the advisor that the original event `transaction_completed` was triggered which will automatically be inferred as performance of a `pay` act.

The notions of **query**, **update**, **revert** and **reset** are already afforded by the norm reasoner where query and update are typically provided by most norm frameworks. However, eFLINT can be used to reason about the compliance of historical, hypothetical, and—most relevant here—dynamically developing scenarios: it relies upon a declarative component that lays out the norms in the form of a labelled transition system and an imperative component that describes traces in this system. Similarly to belief queries and revision, the agent is able to query and revise (assert/retract) institutional facts. But, unlike physical state, institutional state is revertible as for example, an agent may notice that its observation about performance of an act was not correct, or even, it wants to infer hypothetical effects of performance of an act before reverting them.

Another important legal/normative requirement is **adaptability** to new (interpretations of) norms. In our approach, such adaptation can be achieved in multiple ways. Apart from spawning new (and despawning old) advisors to start using the new interpretation or encoding of a set of norms, ASC2 agents are able to modify their script at run-time to change the interactions between institutional and social reality, and this is true for both intentional agents and advisors. This type of modifications are also present in other BDI frameworks such as Jason. Secondly, an existing advisor can be instructed to update the norm source of an instance by adding new type declarations or extending existing types. For example, a violation condition can be dynamically added to the payment duty by submitting the fragment `Extend Duty duty_to_pay Violated when <EXPR>` for some Boolean expression, like a parameterized timeout event. These types of modifications are particularly interesting as a future work to explore a principled approach for studying changes in the norms such as issues about consistency between variations of norms and impact of norm changes in social simulations.

The notions of **receive and process/ignore** and **follow/violate** for normative conclusions connect directly to the concept of autonomy in the agent. All of these are are already afforded by ASC2 on the language level (or AgentSpeak(L) in a broader sense) as receive and process/ignore, and, follow/violate are simply a matter of implementing the plans in the agent's script that define the reactions to such conclusions. Then, as the intentional agents' language and execution cycle are not modified in this architecture, intuitively, autonomy of the agents is also not demoted by integration of norms, particularly in comparison with any BDI agent that does not integrate norms. As a future work, these concepts—especially follow/violate—should be encoded in a more expressive and transparent manner. This can be done, for example, by utilizing declarative constructs such as preferences on the language level (see [22]) to have an explicit, yet programmable way of ordering between intentional (e.g. desires, goals) and normative (e.g. obligations) dimensions of the agent.

6 Conclusion

In this paper we present a framework for embedding norms in a MAS. It is generally acknowledged that agents in a MAS vastly benefit from utilizing norms for more effective/efficient coordination. Here it was further argued that norms, embodied as institutional views of the state of the environment, need normative advisors to facilitate the bridging between institutional and extra-institutional realms. The proposed architecture included using a BDI framework and a norm reasoning framework for creating normative advisors and was shown to address the main requirements of normative (multi-agent) systems as identified by the community. A practical running implementation of this architecture[7] using mostly off-the-shelf tools was presented via a market example to further illustrate the applicability of the approach.

As autonomous agents, norms, and their interactions deal with notions and constructs that are hard to concretize and on which it may be hard to reach an agreement, they may have different definitions and usages in different scientific communities. Alongside the proposal of the architecture and tools in themselves, this work assumed a high priority for flexibility as a requirement in frameworks utilized in designing normative (multi-agent) systems by proposing multiple programmable components varying from pure context-free and abstract norm specifications to perception/action layer of intentional agents. These components aimed at satisfying the higher level requirements of normative agents and (multi-agent) systems without putting any constraint on the language or logic used in components. In principle, the proposed infrastructure can offer a computational ground to comparing agent embeddings of alternative solutions for normative representation and reasoning.

Acknowledgements This paper has been partially funded by the Data Logistics for Logistics Data (DL4LD) project, supported by the Dutch Organisation

[7] Publicly available at: `https://github.com/mostafamohajeri/eumas2022-poc`

for Scientific Research (NWO), the Dutch Institute for Advanced Logistics TKI Dinalog and the Dutch Commit-to-Data initiative (grant no: 628.009.001) and partially funded by the AMdEX Fieldlab project supported by Kansen Voor West EFRO (KVW00309) and the province of Noord-Holland.

References

1. van Binsbergen, L.T., Kebede, M.G., Baugh, J., van Engers, T., van Vuurden, D.G.: Dynamic generation of access control policies from social policies. Procedia Computer Science **198**, 140–147 (January 2022). https://doi.org/10.1016/j.procs.2021.12.221
2. van Binsbergen, L.T., Liu, L.C., van Doesburg, R., van Engers, T.: eFLINT: A domain-specific language for executable norm specifications. GPCE 2020 - Proceedings of the 19th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, Co-located with SPLASH 2020 pp. 124–136 (2020). https://doi.org/10.1145/3425898.3426958
3. Boella, G., Humphreys, L., Muthuri, R., Rossi, P., van der Torre, L.: A critical analysis of legal requirements engineering from the perspective of legal practice. 2014 IEEE 7th International Workshop on Requirements Engineering and Law, RELAW 2014 - Proceedings pp. 14–21 (2014). https://doi.org/10.1109/RELAW.2014.6893476
4. Boella, G., Pigozzi, G., van der Torre, L.: Normative systems in computer science-ten guidelines for normative multiagent systems. Dagstuhl Seminar (March), 1–21 (2009)
5. Boella, G., van der Torre, L.: Regulative and constitutive norms in normative multiagent systems. In: Proceedings of the Ninth International Conference on Principles of Knowledge Representation and Reasoning. p. 255265. KR'04, AAAI Press (2004)
6. Boella, G., van der Torre, L., Verhagen, H.: Introduction to normative multiagent systems. Computational and Mathematical Organization Theory **12**(2-3 SPEC. ISS.), 71–79 (2006). https://doi.org/10.1007/s10588-006-9537-7
7. Bordini, R.H., Hübner, J.F., Vieira, R.: Jason and the Golden Fleece of Agent-Oriented Programming. No. January (2005). https://doi.org/10.1007/0-387-26350-0"˙1
8. Bratman, M.: Intention, plans, and practical reason (1987)
9. Broersen, J., Dastani, M., Hulstijn, J., Huang, Z., van der Torre, L.: The BOID Architecture - Conflicts Between Beliefs, Obligations, Intentions and Desires. In: In Proceedings of the Fifth International Conference on Autonomous Agents. pp. 9–16. ACM Press (2001)
10. Cardoso, R.C., Ferrando, A., Dennis, L.A., Fisher, M.: Implementing Ethical Governors in BDI. In: Alechina, N., Baldoni, M., Logan, B. (eds.) Engineering Multi-Agent Systems. pp. 22–41. Springer International Publishing, Cham (2022)
11. Chopra, A., van der Torre, L., Verhagen, H.: Handbook of Normative Multiagent Systems. College Publications (2018)
12. Criado, N., Argente, E., Noriega, P., Botti, V.: Towards a normative BDI architecture for norm compliance. CEUR Workshop Proceedings **627**, 65–81 (2010)
13. Dastani, M., Mol, C., Tinnemeier, N.A.M., Meyer, J.J.C.: 2APL: A practical agent programming language. Belgian/Netherlands Artificial Intelligence Conference (March), 427–428 (2007). https://doi.org/10.1007/s10458-008-9036-y

14. Deljoo, A., van Engers, T., van Doesburg, R., Gommans, L., de Laat, C.: A Normative Agent-based Model for Sharing Data in Secure Trustworthy Digital Market Places. Proceedings of the 10th International Conference on Agents and Artificial Intelligence (April), 290–296 (2018). https://doi.org/10.5220/0006661602900296
15. Dhaon, A., Collier, R.: Multiple inheritance in AgentSpeak(L)-style programming languages. In: AGERE! 2014 - Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Based on Actors, Agents, and Decentralized Control, Part of SPLASH 2014. pp. 109–120. No. L (2014). https://doi.org/10.1145/2687357.2687362
16. Dignum, F., Kinny, D., Sonenberg, L.: Motivational attitudes of agents: On desires, obligations, and norms. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **2296**(Section 2), 83 (2002). https://doi.org/10.1007/3-540-45941-3–"˝9
17. Gabbay, D., Horty, J., Parent, X.: The Handbook of Deontic Logic and Normative Systems, Volume 2. College Publications (2021)
18. Gabbay, D., Horty, J., Parent, X.: Handbook of Deontic Logic and Normative Systems. College Publications (2013)
19. Gibbs, J.P.: Norms: The problem of definition and classification. American Journal of Sociology **70**(5), 586–594 (1965). https://doi.org/10.1086/223933
20. Hohfeld, W.N.: Fundamental legal conceptions as applied in judicial reasoning. The Yale Law Journal **26**(8), 710–770 (1917). https://doi.org/10.2307/786270
21. Liao, B., Slavkovik, M., van der Torre, L.: The Jiminy Advisor: Moral Agreements Among Stakeholders Based on Norms and Argumentation (2018). https://doi.org/10.48550/ARXIV.1812.04741
22. Mohajeri Parizi, M., Sileno, G., van Engers, T.: Preference-based goal refinement in bdi agents. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems. p. 917925. AAMAS '22, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2022)
23. Mohajeri Parizi, M., Sileno, G., van Engers, T., Klous, S.: Run, agent, run! architecture and benchmarking of actor-based agents pp. 11–20 (2020). https://doi.org/10.1145/3427760.3428339
24. Morales, J., López-sánchez, M., Rodriguez-Aguilar, J.A., Vasconcelos, W., Wooldridge, M.: Online automated synthesis of compact normative systems. ACM Trans. Auton. Adapt. Syst. **10**(1) (mar 2015). https://doi.org/10.1145/2720024
25. Pandžić, S., Broersen, J., Aarts, H.: BOID*: Autonomous goal deliberation through abduction. In: Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems. p. 10191027. AAMAS '22, International Foundation for Autonomous Agents and Multiagent Systems, Richland, SC (2022)
26. Rao, A.S.: AgentSpeak(L): BDI agents speak out in a logical computable language. In: de Velde, W., Perram, J.W. (eds.) Agents Breaking Away. pp. 42–55. Springer Berlin Heidelberg, Berlin, Heidelberg (1996)
27. Rao, A.S., Georgeff, M.P., others: BDI agents: From theory to practice. Icmas **95**, 312–319 (1995). https://doi.org/10.1.1.51.9247
28. Searle, J.R.: Speech acts: An Essay in the Philosophy of Language. Cambridge University Press (1969)
29. Sileno, G., Boer, A., van Engers, T.: On the interactional meaning of fundamental legal concepts. In: Proceedings of JURIX 2014. pp. 39–48 (2014)
30. Sileno, G., Boer, A., van Engers, T.: Revisiting Constitutive Rules. In: Proceedings of the 6th Workshop on Artificial Intelligence and the Complexity of Legal Systems (AICOL 2015) (2015)

31. Sileno, G., van Binsbergen, L.T., Pascucci, M., van Engers, T.: DPCL: a language template for normative specifications. Workshop on Programming Languages and the Law (ProLaLa 2022), co-located with POPL 2022 (2020)
32. Tufis, M., Ganascia, J.G.: Grafting norms onto the BDI agent model. A Construction Manual for Robots' Ethical Systems. Cognitive Technologies (2015)