Custard: Computing Norm States over Information Stores

Submission #7

ABSTRACT

Custard is a novel language for specifying norms. A crucial innovation of Custard is that it models the information-centric aspects of norms. Specifically, Custard enables treating a norm specification on par with an information schema and gives a mapping of the norm to queries that give the instances of the norm in the various states of the norm's canonical lifecycle. In essence, the mapping creates a norm-based abstraction layer over underlying information stores such as databases and logs.

Custard supports expressing the occurrence of complex events, including based on aggregation, sophisticated deadlines, and nested norms. We prove important correctness properties of our language, including stability (the idea that once an event has occurred, it has occurred forever) and safety (the idea that a query returns a finite set of tuples). We have implemented a compiler that generates SQL queries from Custard specifications. Writing out such SQL queries by hand would be extremely difficult even for simple norms, thus demonstrating Custard's practical benefits.

1. INTRODUCTION

Norms provide a natural basis for modeling and enacting interactions in sociotechnical systems involving multiple autonomous social principals, who may be either humans or organizations. The importance of norms derives from the fact that they serve as the rules of encounter among the principals, and thereby as a standard of correctness, in settings where it is either not possible or not desirable to regiment interactions [11]. Important kinds of norms studied in the literature include commitment, authorization, prohibition, and power [3, 20, 13, 17].

Following convention, we understand an agent as a software entity that acts on behalf of a principal in a sociotechnical system. Several researchers have advocated intelligent agents that reason about the applicable norms in deciding a course of action [6, 14, 2, 24]. In fact, norms are widely considered as social component of an agent's deliberation. For example, in an extended healthcare setting, an agent representing a healthcare authority may sanction an Electronic Health Record (EHR) service provider in case the provider has violated norms related to the disclosure of information. Or, an agent representing a fitness center may provide full exercise and diet plans to only those agents to whom it is committed to providing such plans as a consequence of their signing up with the center; to others, the center may just provide basic fitness tips.

Supporting such intelligent agent behavior in practical settings presumes being able to reason about the *states* of the norms relevant to the setting from low-level information recorded in databases and logs. This in turn requires distinguishing between norm specifications and norm instances. Returning to the above example of inappropriate disclosures, it may be one specific disclosure norm, say a prohibition, several instances of which have been violated. This distinction between norm specifications, or simply norms, and their instances has not been adequately formalized. In general, what we would like to do is treat a norm as an information schema and treat information stores such as relational databases and logs as a store of the norm's instances in various lifecycle states. For example, a certain state of the information store may imply an instance of a prohibition that has been created but not violated and many instances of the same prohibition that have been violated. Figure 1 captures our target information architecture.

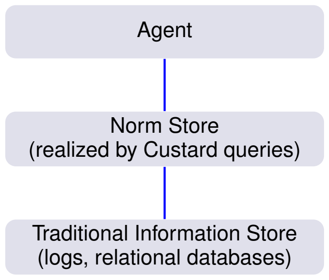

Agent
Norm Store
(realized by Custard queries)
Traditional Information Store
(logs, relational databases)

Figure 1: Custard enables realizing a norm-based view of information on top of commonly used information stores by way of queries.

With a view to realizing our target architecture, we propose *Custard*, a language for specifying information-based norms. In Custard, one can specify commitments, authorizations, prohibitions, and powers. Custard is event-based: important stages in the lifecycle of a norm instance, specifically, creation, detachment, expiration, discharge, and violation, are event instances and inferred from event instances recorded in the underlying information store. Custard is highly expressive: one can specify complex event expressions involving logic-style operators, aggregation operators, relative time intervals within which events should occur, and nested norms that involve multiple kinds of norms. For example, the commitment to perform inform patients in case of a security breach, which would itself be the violation of a prohibition.

We give the semantics of Custard specifications in terms of queries in the tuple relational calculus (TRC) [8]. Effectively, for every norm specified in Custard, we define a query for each event in the

Appears in: *Proceedings of the 15th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2016), John Thangarajah, Karl Tuyls, Stacy Marsella, Catholijn Jonker (eds.), May 9–13, 2016, Singapore.*

norm's canonical lifecycle. The query definition gives the instances of the event. The benefit of using the TRC is that a query is defined directly as a set of instances and paves the way for easy implementation in widely used query languages such as SQL. In fact, we implemented a compiler that generates SQL queries from Custard specifications.

Because of the richness of our language, especially the support for specifying the nonoccurrence of an event, aggregation, and expressive time intervals, the queries turn out to be nontrivial. We formulate and prove two desirable properties for our queries. One, *stability*, a kind of monotonicity, which captures the idea that once an event instance has occurred, it stays occurred forever. Thus, for example, a prohibition instance determined violated at a certain moment in time must be determined violated at all future moments. Two, safety, which captures the idea that queries map to finite sets.

We demonstrate the effectiveness of Custard by modeling a real-world privacy consent scenario being considered by Health Level Seven (HL7) [10], which is a leading standardization body for health information systems.

Organization. The rest of the paper is organized as follows. Section 2 describes our metamodel for norms with the help of examples from the HL7 use case. Here, we express the norms in Custard with the aim of clarifying the notions involved. Section 3 describes Custard formally, including its syntax, semantics, and important properties. Section 4 describes an implementation of Custard that generates SQL queries. Section 5 discusses related work and lays out future directions.

2. SAMPLING CUSTARD

We take as our point of departure recent work that understands norm types such as commitment, authorization, prohibition, and power as *social expectations* between agents [20, 21]. The import of this formulation is that it yields a basis for accountability in sociotechnical systems: the *expectee* is accountable to the *expector* for the satisfaction of the expectation. In other words, the expector may legitimately demand that the expectee give an account about the status of the expectation. Let's consider an example to illustrate the notions involved.

EXAMPLE 1. *A commitment from a hospital to encrypt sensitive private health information (PHI) represents the patient's expectation that the hospital will do that. The commitment implies that the patient has a basis for demanding an account from the hospital about whether his or her PHI has been encrypted, and if not, why not. A failure to encrypt PHI would in fact be a violation, for which the hospital may be sanctioned.*

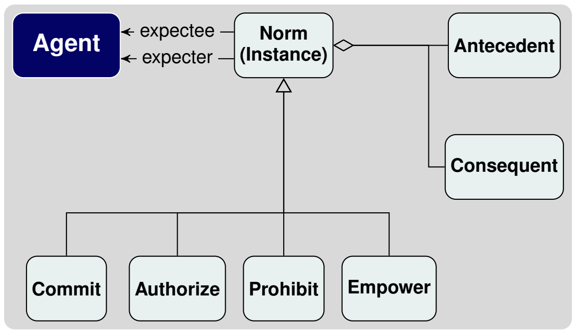


Figure 2: Norms metamodel (based on [20]).

Figure 2 (based on [20]) shows the important elements of our conceptual model. Each (created) norm instance is an expectation and has expector and expectee agents. The expector and expectee represent the *privileged* and *liable* parties, respectively. A norm instance is, in general, conditional, with the *antecedent* stating the condition under which the *consequent*, which represents the force of the norm, applies. We consider four types of norms: commitment, authorization, power, and prohibition.

Viewing norms as expectations between two agents leads to interesting questions that have not received adequate attention in the literature. For example, when is an authorization violated and who is accountable to whom for the violation? In this paper, we adopt conventions that help address these conceptual issues.

To illustrate our concepts and techniques, we use examples from HL7's collaborative health care scenarios. A patient signs up with a cloud-based health vault to store and manage access to its private health information (PHI). This information may include regular records of the patient's vital health signs such as pulse, pressure, blood sugar, and so on, that are monitored by wearable devices and uploaded to the vault. The patient may authorize third parties to receive the PHI from the vault by indicating consent. Alternatively, the patient may revoke previously granted consent. For example, the third party could be a fitness center that gives the patient exercise and diet plans based on the information in the vault. The power to grant or revoke authorizations is given to the patient by the jurisdiction in which the patient is resident. In general, third parties granted authorization by the patient are prohibited from forwarding the information they receive to yet other parties. Parties may be sanctioned for violating this prohibition.

Listing 1 shows an information schema for such a healthcare settings. It describes a number of event specifications as relations, each annotated with its key and timestamp attributes. No two instances of an event (specification) will have identical bindings for the key, and for every instance, the timestamp attribute records the time of occurrence of the instance. The key of one event may occur in another. For example, accID occurs in Allowed. Such patterns are crucial to correlation. Thus every Allowed instance can be correlated by a Signedup instance via the binding for accID in the former. In general, correlations may be effected via chains. For example, a Revoked instance is correlated to an Allowed instance by discID, and, therefore, to Signedup via accID.

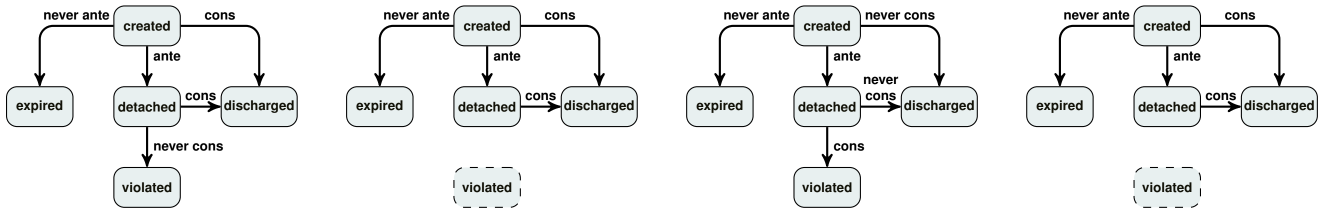
Further, notice that there can be at most one SentCred instance for an Allowed instance as their keys are identical. For every disclosure to a third party, there can be many requests for data from that party to the vault (ReqData). For every request, there can be at most one access (Accessed). A third party may forward data accessed via a request to yet other parties many times (Forwarded). Every Forwarded instance is correlated with a Signedup instance via a chain of correlations (forID to reqID to discID to accID). Methodologies for designing the appropriate information schemas are outside the scope of the current paper.

Listing 1: Example schema for the healthcare scenario.

```
schema
 //Patient pID becomes resident in jurisd. jID
  Registered(pID, jID, resID, council)
  key resID time t

 //pID signs up with health vault hID
 Signedup(pID, hID, accID)
  key accID time t

 //pID allows disclosure to third party tpID
 Allowed(pID, hID, discID, accID, tpID, info)
  key discID time t
```



(a) Commitment (b) Authorization (c) Prohibition (d) Power

Figure 3: Lifecycles for the various norms. For each norm type, each box refers to a lifecycle event (not a state). The transitions refer to events as well—the occurrence or the impossibility of occurrence of the antecedent and consequent events.

```
//pID revokes disclosure for tpID
Revoked(pID, hID, discID)
 key discID time t

//hID sends creds. to tpID if allowed disclosure
SentCred(hID, tpID, discID, credentials)
 key discID time t

//tpID requests patient data from hID
ReqData(tpID, hID, reqID, discID, request)
 key reqID time t

//tpID gets access to the requested data
Accessed(tpID, hID, reqID, response)
 key reqID time t

//tpID forwards data to party otherID
Forwarded(tpID, otherID, forID, reqID, response)
 key forID time t

//hID sanctions tpID for mishandling accID
Sanction(hID, tpID, accID, details)
 key accID time t
```

2.1 Commitment

Our notion of commitment corresponds to the standard one in literature: the debtor commits to the creditor to bring about a condition (the consequent) if some condition (the antecedent) holds. We give an example from the foregoing scenario.

EXAMPLE 2. *When a patient signs up with a health vault, the vault commits to the patient that if disclosure is allowed to a third party, then the vault will authorize the party within a day.*

Listing 2 shows a commitment listing in Custard. DisclosureCom is a name for the listing. The commitment is from hID to pID; it is created upon the occurrence of a Signedup instance; detached upon the occurrence of a correlated Allowed disclosure instance, provided that disclosure has not been Revoked (this allows modeling situations where patients may change their minds); it is discharged when a correlated instance of DisclosureAuth (an authorization, see Listing 3) is created. The detach and discharge clauses represent the antecedent and the consequent, respectively. In an expression of the form $E[l, r]$, $[l, r]$ is a time interval; the expression says that E should occur within this interval. If $l = 0$, we may omit specifying l; if $r = \infty$, we may omit specifying r; if $l = 0$ and $r = \infty$, we may omit specifying the time interval.

Listing 2: Captures Example 2.

```
commitment DisclosureCom hID → pID
 create Signedup
 detach Allowed except Revoked
 discharge created DisclosureAuth[0, Allowed+1]
```

The lifecycle of a created commitment instance is shown in Fig. 3a. The instance is violated if the antecedent occurs but the consequent cannot; it expires if the antecedent cannot occur; and it is discharged if the consequent occurs. We adopt the convention that a commitment may be both expired and discharged: this happens if the antecedent cannot occur but the consequent occurs.

2.2 Authorization

An agent authorizes another to bring about a condition (as captured by the consequent) in a certain condition (as captured by the antecedent).

EXAMPLE 3. *A party is authorized by the health vault, via the sending of credentials, that data requested will be available within a day of the request for ten days after the request.*

Listing 3: An authorization in Custard.

```
authorization DisclosureAuth tpID by hID
 create SentCred
 detach ReqData
 discharge Accessed[ReqData+1, ReqData+10]
```

In authorizations, we treat the authorizing party as the expectee and the authorized party as the expector. Thus, in Listing 3, the expector is tpID and the expectee is the vault service. Specifically, tpID expects access to patient information if it has been allowed disclosure to. This aligns with the intuition that an authorization is the authorized party's privilege, not a liability. The authorization would in fact be violated if the tpID had been allowed disclosure to by the patient but were blocked by the vault service from accessing the patient's information.

Although in principle, an authorization may be violated as discussed above, we make a design assumption that makes the violation of authorization impossible. We assume that authorizations are regimented by the agent, for example, via authentication and access control mechanisms. Thus, access to a patient's record is regimented by the health vault via authentication. This also implies that if a party is unable to gain access despite its being on the disclosure list, then that is not a violation, but simply an implementation error. Figure 3b captures the authorization lifecycle. The *violated* event is unreachable to capture the fact that authorizations cannot be violated.

2.3 Prohibition

An agent prohibits another from bringing about the consequent if the antecedent has occurred. Unlike authorizations, we model as prohibitions those cases where either events are not easily regimented or not desired to be regimented. For example, it is not easy to regiment a system so that smoking in a space (or littering) is impossible. We capture the undesirability of such behaviors by placing prohibitions on them. We give an example from the health scenario.

Example 4. *Parties that have been allowed disclosure by the patient are prohibited by the jurisdictional authority from forwarding the information they receive to other parties.*

Listing 4 captures Example 4.

Listing 4: A prohibition in Custard.

```
prohibition DisclosureProh tpID by jID
 create Accessed
 violate Forwarded
```

Figure 3c captures the prohibition lifecycle. A prohibition is violated if the antecedent and the consequent have both occurred. In a prohibition, the expectee and expector are the prohibited and prohibiting parties, respectively.

2.4 Power

An agent empowers another to bring about certain states of affairs by simply "saying so" provided some conditions hold. We adopt Jones and Sergot's distinction between power and authorization[11]: power is the ability of an agent to modify norms among other agents. Authorization, by contrast, is the ability to access resources and, as discussed above, in our model, authorizations are regimented in software.

Example 5. *The jurisdictional authority empowers the patient to authorize any party to receive its information from the storage service by simply filling out the appropriate form.*

Listing 5 captures Example 5. It says that a patient who is registered in a jurisdiction and who has signed up with a vault after becoming a resident has to power to authorize (and revoke) disclosure to other parties.

Listing 5: A power in Custard.

```
power ConsentPower pID by jID
 create Registered
 detach Signedup[Registered,]
 discharge Allowed[Signedup,] and Revoke[Signedup,]
```

In a power instance, the empowered agent is the privileged party, that is, the expector. The empowering agent is the liable party, that is, the expectee. However, a power cannot be violated for the simple reason that "saying so" under the right conditions is enough to fully exercise the power. The "effects" of the power will be realized through other norms. In Example 5, the effect of the power is realized through the vault becoming committed to authorize those parties who are allowed by the patient to receive information. Of course, the vault may refuse to comply, thus violating its commitment, but the power itself is inviolable.

If the antecedent condition does not hold, then the exercise of the power may be considered invalid but that is not the same as violation (recall that the privileged party is the empowered agent and therefore an invalid exercise of the power cannot be a violation). Figure 3d captures the lifecycle of power.

2.5 Aggregation

Many norms involve aggregation, as Example 6 illustrates.

Example 6. *The jurisdiction has the been empowered by the patient to declare a party as "out-of-compliance" if in the year following it being allowed to receive information, it violates the prohibition to forward information to others more than twice.*

Listing 6 captures Example 6. The aggregation syntax is based on the standard one in database theory: specify the attribute to aggregate over (forID), the query in which the attribute appears (Signedup and Allow and violated DisclosureProh), how to group the tuples of the query (by accID), and the attribute that holds the aggregated value (numViol). The aggregate event occurs when the count of forID over the specified interval is greater than two.

Listing 6: A norm involving aggregation in Custard.

```
power SanctionPower jID by pID
 create Signedup and Registered
 detach count[Signedup,Signedup+365] forID of
      (Signedup and Allow and violated
      DisclosureProh) as numViol group by accID > 2
 discharge Sanction[,detached SanctionPower+10]
      where details=‘‘Out of compliance’’
```

3. TECHNICAL FRAMEWORK

Let $\mathcal{D} = \{\mathcal{D}_1 \ldots \mathcal{D}_n\}$ be a set of domains where $\mathcal{T} \in \mathcal{D}$ is the domain of time instants; in particular, $\mathcal{T} = \mathbb{N} \cup \{\infty\}$, where $\mathbb{N}$ is the set of natural numbers and ∞ is an infinitely distant time instant. Below, $\mathcal{A}$ and $\mathbb{R}$ are the sets of agent names and the real numbers, respectively. Table 1 defines the syntax of Custard.

Table 1: Syntax of Custard.

Event	$\longrightarrow$	Base | LifeEvent
LifeEv	$\longrightarrow$	created Spec | detached Spec | discharged Spec expired Spec | violated Spec
Expr	$\longrightarrow$	Event[Time, Time] | Aggr | Expr where φ | Expr EvOp Expr
EvOp	$\longrightarrow$	$\sqcap$ | $\sqcup$ | $\ominus$
Aggr	$\longrightarrow$	Func[Time, Time] $\mathcal{D}$ of Expr as $\mathcal{D}$ group by GSpec Comp $\mathbb{R}$
GSpec	$\longrightarrow$	$\mathcal{D}$ | GSpec,$\mathcal{D}$
Func	$\longrightarrow$	sum | count | min | max | avg
Comp	$\longrightarrow$	> | >= | < | <= | = | !=
Time	$\longrightarrow$	Event – $\mathcal{T}$ | Event + $\mathcal{T}$ | $\mathcal{T}$
Norm	$\longrightarrow$	commitment | prohibition | authorization | power
Spec	$\longrightarrow$	Norm($\mathcal{A}$, $\mathcal{A}$, Expr, Expr, Expr)

All expressions of type Expr are essentially (complex) events. The Custard listings shown earlier use a surface syntax for Spec expressions in the formal grammar. In the surface syntax, we write and, or, and except for $\sqcap$, $\sqcup$, and $\ominus$ respectively. In time intervals, we omit lower and upper instants when they are 0 and ∞, respectively. An omitted detach clause means the norm is unconditional. Further, in the surface syntax, we label norms to simplify writing nested commitments. For instance, the commitment in Listing 2 in the formal grammar is: commitment(hID, pID, SignedUp, Allowed $\ominus$ Revoked, created authorization(tpID, hID, SentCred, ReqData, Accessed[ReqData+1, ReqData+10])).

3.1 Semantics

An *information schema* is a nonempty set of events, each modeled as a relation with a superkey and a distinguished timestamp column. Definition 1 defines an information schema formally.

Definition 1. *For convenience, we identify a domain with its set of possible values. An* event schema *over $\mathcal{D}$ pairs a nonempty set of* attributes *and a* key. *That is, $E = \langle A, K \rangle$, where $A \subseteq \mathcal{D}$, $\mathcal{T} \in A$, and $K \subseteq A$. (Treating each attribute as unique with its own domain simplifies the notation without loss of generality.) $\mathcal{E}_{\mathcal{D}}$*

is the set of all possible event schemas over $\mathcal{D}$. An information schema *I over $\mathcal{D}$ is a nonempty set of event schemas over $\mathcal{D}$. That is, $I \subseteq \mathcal{E}_{\mathcal{D}}$.*

Definition 2 defines the intension of an event schema E.

DEFINITION 2. *The* universe over *E, is the set of all possible instances of E. That is, if $A = \{A_1 \ldots A_m\}, \mathcal{U}_E = A_1 \times \ldots \times A_m$. The* intension of *$E$ is the powerset of its universe restricted to sets that satisfy E's key, that is, any two E instances that agree on the key attributes must agree on every attribute (i.e., they are the same instance). That is, $\langle\!\lbrack E \rbrack\!\rangle = \{Y | Y \in \mathcal{U}_E$ and $(\forall u_i, u_j \in Y$ if $u_i||K = u_j||K$ then $u_i = u_j)\}$, where $||$ indicates projection to the specified set of attributes.*

A model M of an information schema specifies, for each event schema E, an extension of that schema $\llbracket E \rrbracket$ as the set of instances of that event schema, respecting the event schema's key. Definition 3 defines models formally.

DEFINITION 3. *A* model *of an information schema is a function that maps each of its event schemas to its extension, i.e., a member of its intension. Specifically, $M : \mathcal{E}_{\mathcal{D}} \mapsto \langle\!\lbrack E \rbrack\!\rangle$. We term $M(E)$ the* extension of *E and write it as $\llbracket E \rrbracket^M$, omitting the superscript when M is understood.*

The model defines $\llbracket E \rrbracket$ for Base E. The semantic postulates below lift the $\llbracket\ \rrbracket$ to all expressions in Custard via the TRC. In the TRC, quantification is over tuples; for a tuple τ, $\tau.a$ gives the value of attribute a of τ. Below, t is the distinguished timestamp attribute of all event schemas; $\{c, d\} \subseteq \mathcal{T}$; $E, F, \ldots$ are expressions of type Event; $X, Y, \ldots$ are expressions of type Expr; l and r are Time expressions; $\oplus$ is either '+' or '–'; $\otimes$ is a Comp expression; N is a Norm expression; g is a GSpec expression.

We use the following auxiliary functions: *att* gives the nontimestamp attributes of an event schema; *cmn* gives the common nontimestamp attributes of two event schemas; *eq* takes two tuples and a set of attributes and returns true if and only if the tuples are equal for each of those attributes; *nul* takes a tuple and a set of attributes and returns true if and only if each attribute's value is null in that tuple; *max* and *min* have their usual meanings; *sumf* takes four values, a set of tuples S, an attribute col (which needs to be summed), a set of columns g (to group S by), and an attribute colsum (whose value will be the sum), and gives a set of tuples with the attributes g and colsum; *holds* takes a constraint and a tuple and returns true iff the tuple satisfies the constraint.

D_1. $\llbracket E[c, d] \rrbracket = \{\tau | \tau \in \llbracket E \rrbracket \wedge c \leqslant \tau.t < d\}$. Select all events in E that occur after (including at) c but before d.

D_2. $\llbracket E[F \oplus c, d] \rrbracket = \{\tau | \exists \tau'\ \tau \in \llbracket E \rrbracket \wedge \tau' \in \llbracket F \rrbracket \wedge eq(\tau, \tau', cmn(E, F)) \wedge \tau'.t \oplus c \leqslant \tau.t < d\}$. Select E if F occurs and E occurs after (or before, depending on what $\oplus$ is) c moments of F's occurrence but before d.

D_3. $\llbracket E[c, F \oplus d] \rrbracket = \{\tau | \exists \tau' \tau \in \llbracket E \rrbracket \wedge \tau' \in \llbracket F \rrbracket \wedge eq(\tau, \tau', cmn(E, F)) \wedge c \leqslant \tau.t < \tau'.t \oplus d\}$. Select E if F occurs and E occurs after c but before d moments have passed since F's occurrence (or at least d moments before F's occurrence, depending on what $\oplus$ is).

D_4. $\llbracket E[F \oplus c, G \oplus d] \rrbracket = \{\tau | \tau \in \llbracket E[F \oplus c, \infty] \rrbracket \wedge \tau \in \llbracket E[0, G \oplus d] \rrbracket\}$. Combines D_2 and D_3.

We give definitions for aggregate events involving sum. The definitions for the events involving the other Func expressions (min, max, count, avg) are analogous. We skip them for brevity.

D_5. $\llbracket \mathsf{sum}[l, d]\ col$ of X as $colsum$ group by $g \otimes n \rrbracket = \{\tau | \tau.t = d \wedge S = \{\tau' | \tau' \in \llbracket X \rrbracket \wedge l \leqslant \tau''.t < d\} \wedge T = sumf(S, col, colsum, g) \wedge \exists \tau''\ \tau'' \in T \wedge eq(\tau, \tau'', att(T)) \wedge \tau''.colsum \otimes n\}$.

D_6. $\llbracket \mathsf{sum}[l, F \oplus d]\ col$ of X as $colsum$ group by $g \otimes n \rrbracket = \{\tau | \exists \tau'\ \tau' \in \llbracket F \rrbracket \wedge eq(\tau, \tau', g) \wedge \tau.t = \tau'.t \oplus d \wedge S = \{\tau'' | \tau'' \in \llbracket X \rrbracket \wedge l \leqslant \tau''.t < \tau'.t \oplus d\} \wedge T = sumf(S, col, colsum, g) \wedge \exists \tau'''\ \tau''' \in T \wedge eq(\tau, \tau''', att(T)) \wedge \tau'''.colsum \otimes n\}$.

D_7. $\llbracket X \sqcap Y \rrbracket = \{\tau | \exists \tau' \exists \tau''\ \tau' \in \llbracket X \rrbracket \wedge \tau'' \in \llbracket Y \rrbracket \wedge eq(\tau, \tau', att(X)) \wedge eq(\tau, \tau'', att(Y)) \wedge \tau.t = max(\tau'.t, \tau''.t)\}$. Select (X, Y) pairs where both have occurred; the timestamp of this composite event is the greater of the two.

D_8. $\llbracket X \sqcup Y \rrbracket = \{\tau | (\exists \tau' \exists \tau''\ \tau' \in \llbracket X \rrbracket \wedge \tau'' \in \llbracket Y \rrbracket \wedge eq(\tau, \tau', att(X)) \wedge eq(\tau, \tau'', att(Y)) \wedge \tau.t = min(\tau'.t, \tau''.t)) \vee$

$(\exists \tau'\ \tau' \in \llbracket X \rrbracket \wedge eq(\tau, \tau', att(X)) \wedge \tau.t = \tau'.t \wedge nul(\tau, att(Y)) \wedge \forall \tau'' \tau'' \in \llbracket Y \rrbracket \rightarrow \neg eq(\tau, \tau'', att(Y))) \vee$

$(\exists \tau'\ \tau' \in \llbracket Y \rrbracket \wedge eq(\tau, \tau', att(Y)) \wedge \tau.t = \tau'.t \wedge nul(\tau, att(X)) \wedge \forall \tau'' \tau'' \in \llbracket X \rrbracket \rightarrow \neg eq(\tau, \tau'', att(X)))\}$.

Select (X, Y) pairs where at least one has occurred. The timestamp of this composite event is the smaller of the two, if both have occurred, or equal to the timestamp of the one that has occurred.

The interpretation of $X \ominus Y$ is that X should have occurred but the (corresponding) Y should not have occurred. But what is the time of nonoccurrence of an event? Consider $X \ominus E[l, d]$. Here, $E[l, d]$ (corresponding to X) has not occurred if E (corresponding to X) hasn't occurred between l and d. Thus if E occurs before l, say at b, then the time of the nonoccurrence of $E[l, d]$ is b; if E doesn't occur before d, then the time of nonoccurrence of $E[l, d]$ is d. Notice that d could be ∞. The time of occurrence of the $X \ominus E[l, d]$ is the maximum of the timestamps of X and $E[l, d]$.

D_9. $\llbracket X \ominus E[l, d] \rrbracket = \{\tau | (\exists \tau'\ \tau' \in \llbracket X \rrbracket \wedge \tau.t = max(\tau'.t, d) \wedge \forall \tau''\ \tau'' \in \llbracket E[0, d] \rrbracket \rightarrow \neg eq(\tau', \tau'', cmn(X, E))) \vee (\exists \tau' \exists \tau''\ \tau' \in \llbracket X \rrbracket \wedge \tau'' \in \llbracket E[0, l] \rrbracket \wedge eq(\tau', \tau'', cmn(X, E)) \wedge \tau.t = max(\tau'.t, \tau''.t))\}$.

The definition of $\llbracket X \ominus E[c, F \oplus d] \rrbracket$ follows along the same lines except to account for the difference that the right timepoint refers to an event (F) instead of being a constant. As before, we want X if E occurs too soon (before c). We also want X if E occurs too late, in this case, after $f \oplus d$, where f is the value of F's timestamp. We will give this nonoccurrence of E the timestamp $f \oplus d$. But what if F itself hasn't occurred? Then, we won't have a value for f. But in this case, we would not want X anyway because without the occurrence of F, it is not possible to determine the appropriateness of the occurrence of E.

D_{10}. $\llbracket X \ominus E[l, F \oplus d] \rrbracket = \{\tau | (\exists \tau' \exists \tau'' \forall \tau'''\ \tau' \in \llbracket X \rrbracket \wedge \tau'' \in \llbracket F \rrbracket \wedge \tau''' \in \llbracket E[0, F \oplus d] \rrbracket \wedge eq(\tau', \tau'', cmn(X, F)) \wedge \neg eq(\tau, \tau', cmn(X, E)) \wedge \tau.t = \tau''.t \oplus d) \vee (\exists \tau' \exists \tau''\ \tau' \in \llbracket X \rrbracket \wedge \tau'' \in \llbracket E[0, l] \rrbracket \wedge eq(\tau', \tau'', cmn(X, E)) \wedge \tau.t = max(\tau'.t, \tau''.t))\}$.

D_{11}. $\llbracket X \ominus \mathsf{sum}[l, d]\ col$ of Y as $colsum$ group by $g \otimes n \rrbracket = \{\tau | \tau.t = d \wedge \exists \tau'\ \tau' \in \llbracket X \rrbracket \wedge eq(\tau, \tau', att(X)) \wedge S =$

$\{\tau''|\tau'' \in [\![Y[l,d]]\!]\} \wedge T = \mathit{sumf}(S, \mathit{col}, \mathit{colsum}, g) \wedge (\forall \tau''' \; \tau''' \in T \rightarrow \neg \mathit{eq}(\tau, \tau''', \mathit{att}(T)) \vee \neg(\tau'''.\mathit{colsum} \otimes n))\}$.

D_{12}. $[\![X \ominus \mathsf{sum}[l, F \oplus d] \; \mathit{col} \; \mathsf{of} \; Y \; \mathsf{as} \; \mathit{colsum} \; \mathsf{group\ by} \; g \otimes n]\!] = \{\tau|\tau.t = d \wedge \exists \tau' \; \tau' \in [\![X]\!] \wedge \mathit{eq}(\tau, \tau', \mathit{att}(X)) \wedge S = \{\tau''|\tau'' \in [\![Y[l, F \oplus d]]\!]\} \wedge T = \mathit{sumf}(S, \mathit{col}, \mathit{colsum}, g) \wedge (\forall \tau''' \; \tau''' \in T \rightarrow \neg \mathit{eq}(\tau, \tau''', \mathit{att}(T)) \vee \neg(\tau'''.\mathit{colsum} \otimes n))\}$.

D_{13}. $[\![X \; \mathsf{where} \; \varphi]\!] = \{\tau|\tau \in [\![X]\!] \wedge \mathit{holds}(\tau, \varphi)\}$. Select X if φ.

D_{14}–D_{17} transform complex expressions involving $\ominus$ to those where $\ominus$ has an event of the form $E[l,r]$ as its right hand side operand.

D_{14}. $[\![X \ominus (Y \sqcap Z)]\!] = [\![(X \ominus Y) \sqcup (X \ominus Z)]\!]$.

D_{15}. $[\![X \ominus (Y \sqcup Z)]\!] = [\![(X \ominus Y) \sqcap (X \ominus Z)]\!]$.

D_{16}. $[\![X \ominus (Y \ominus Z)]\!] = [\![(X \ominus Y) \sqcup (X \sqcap Z)]\!]$.

D_{17}. $[\![X \ominus (Y \; \mathsf{where} \; \varphi)]\!] = [\![(X \ominus Y) \sqcup T]\!]$. The set $T = \{\tau|\exists \tau' \; \tau' \in [\![X \sqcap Y \; \mathsf{where} \; \neg\varphi]\!] \wedge \mathit{eq}(\tau, \tau', \mathit{att}(X)) \wedge \tau.t = \tau'.t\}$.

Below, we write $N(c,r,u)$ instead of $N(x,y,c,r,u)$ as x and y do not appear on the right hand side. (They would be important in other kinds of reasoning, for example, related to group norms or norm networks.)

D_{18}. $[\![\mathsf{created} \; N(c,r,u)]\!] = [\![c]\!]$. A commitment is created when its create event occurs.

D_{19}. $[\![\mathsf{detached} \; N(c,r,u)]\!] = [\![c \sqcap r]\!]$. A commitment is detached when its create and detach events both occur.

D_{20}. $[\![\mathsf{expired} \; N(c,r,u)]\!] = [\![c \ominus r]\!]$. A commitment is expired when its create event has occurred but its detach fails to occur within the specified interval.

D_{21}. $[\![\mathsf{discharged\ commitment}(c,r,u)]\!] = [\![(c \sqcap u) \sqcup (r \sqcap u)]\!]$. A commitment is discharged when its discharge event has occurred along with either its create or detach event.

D_{22}. $[\![\mathsf{discharged\ authorization}(c,r,u)]\!] = [\![c \sqcap r \sqcap u]\!]$. An authorization is discharged when its discharge event has occurred along with its create and detach event.

D_{23}. $[\![\mathsf{discharged\ power}(c,r,u)]\!] = [\![c \sqcap r \sqcap u]\!]$. A power is discharged when its discharge event has occurred along with its create and detach event.

D_{24}. $[\![\mathsf{discharged\ prohibition}(c,r,u)]\!] = [\![(c \sqcap r) \ominus u]\!]$. A prohibition is discharged when its create and detach events occur but the violate event fails to occur.

D_{25}. $[\![\mathsf{violated\ commitment}(c,r,u)]\!] = [\![(c \sqcap r) \ominus u]\!]$. A commitment is violated when its create and detach events occur but the discharge event fails to occur.

D_{26}. $[\![\mathsf{violated\ authorization}(c,r,u)]\!] = \phi$ (the empty set). No authorization can be violated.

D_{27}. $[\![\mathsf{violated\ power}(c,r,u)]\!] = \phi$. No power can be violated.

D_{28}. $[\![\mathsf{violated\ prohibition}(c,r,u)]\!] = [\![c \sqcap r \sqcap u]\!]$. A prohibition is violated when its create, detach, and violate events all occur.

D_{29}. $[\![X]\!]_i = \{\tau|\tau \in [\![X]\!] \wedge 0 \leqslant \tau.t < i\}$. Select all X that have occurred prior to time instant i.

3.2 Properties

Stability is the idea that once an event is determined to have occurred, then at all future time instants, it should be determined to have occurred. In other words, an event that has occurred cannot later "unoccur". For example, a message that been sent cannot be unsent. Stability of events is a fundamental assumption in reasoning about distributed systems.

We would like to extend this notion of stability to norm lifecycle event instances. Thus, for example, if a prohibition instance is determined to have been violated at a time instant, then at all future instants, it should be determined violated. Stability would be highly desirable in business settings as it would give stakeholders confidence in the status of things.

A Base event is by definition stable, since the model defines its extension. However, stability for complex events, including lifecycle events, does not automatically follow from the stability of Base events. For example, imagine an event specification of the form $X \ominus E[0, 100]$. Let's say a query is run at time 50, before which some X instance has occurred but the corresponding E instance hasn't occurred. Then the query may determine that the corresponding $X \ominus E[0, 100]$ instance has occurred. However, that would be premature: the E instance could yet occur, say at time 55, and a later query would determine the $X \ominus E[0, 100]$ instance to have not occurred. In essence, we would have switched the status of the event from occurred to not occurred.

Aggregation operators also pose a challenge to stability. A sum event determined to have occurred at an instant may at future instants be determined to have not occurred as additional events occur. For example, if the sum over a number of events was required to be greater than some value, it may hit that value after observing, say, five events. However, future events may lower the sum (if the attribute which is being summed can take negative values) and cause it to dip below the required value.

We have built the semantics so that stability is guaranteed for all events and the above described scenarios do not occur. Theorem 1 states the stability requirement formally: the set defined by a query at time i (see D_{29}) should be a subset of the set defined by the query at $i+1$.

THEOREM 1. $\forall i \; [\![X]\!]_i \subseteq [\![X]\!]_{i+1}$

Proof Sketch. Let's consider a model M. The model defines $[\![E]\!]$ for Base E. Therefore, $[\![E]\!]_i \subseteq [\![E]\!]_{i+1}$. We should look at the each of queries defined in D_1–D_{28} one by one and determine that the property holds for them.

D_1. Follows from the fact that for Base E $[\![E]\!]_i \subseteq [\![E]\!]_{i+1}$. Reasoning for D_2–D_4 is essentially analogous.

D_5. If $\tau \in [\![\mathsf{sum}[l,d] \; \mathit{col} \; \mathsf{of} \; X \; \mathsf{as} \; \mathit{colsum} \; \mathsf{group\ by} \; \gamma \otimes n]\!]_i$, then all relevant X instances, that is, those that happen in $[l,d]$ have been considered. D_6 is analogous.

D_7. Follows from $[\![X]\!]_i \subseteq [\![X]\!]_{i+1}$ and $[\![Y]\!]_i \subseteq [\![Y]\!]_{i+1}$. D_8 is analogous.

D_9. Two subcases. One, $\tau \in [\![X \ominus E[l,d]]\!]_i$ and X occurs at some time k but E does not occur in $[0,d]$. In this case, $\tau.t = \mathit{max}(k,d)$. If $\tau.t = k$, then $k < i$; if $\tau.t = d$, then $d < i$. Therefore, $\tau \in [\![X \ominus E[l,d]]\!]_{i+1}$. Two, $\tau \in [\![X \ominus E[l,d]]\!]_i$ because X and $E[0,l]$ have both occurred. Since they are each stable, $\tau \in [\![X \ominus E[l,d]]\!]_{i+1}$. D_{10}–D_{12} are analogous.

D_{13}. Follow from the fact that $[\![X]\!]$ is stable.

D_{14}–D_{17}. A query of these forms reduces to a query where the right hand side of every $\ominus$ operand is a base event, lifecycle event, or aggregation event qualified by a time interval. Such expressions are stable. Their combinations with other expressions is also stable.

D_{18} Follows from the fact that $[\![c]\!]$ is stable. D_{19}–D_{28} are analogous.

Safety is a well-known correctness criterion for database queries [8]. Definition 4 defines safety formally.

DEFINITION 4. *A query Q is* safe *if and only if given any possible model M with finite extensions for Base events, the extension of Q relative to M, $[\![Q]\!]$, is finite.*

Negation-like operators such as $\ominus$ have the potential to compromise safety if their usage is not restricted adequately. For example, imagine that we had a unary negation operator $\ominus_u$ and the create clause for some commitment were simply $\ominus_u E$ (assume E is Base). This would amount to considering created infinitely many commitment instances, one for each E instance that is *not* present in $[\![E]\!]$. A technique that is commonly employed to avoid such conclusions is to guard such negation-like operators, as we do in Custard: $\ominus$ is a binary operator, the extension of whose left operand circumscribes the extension of its right operand. Theorem 2 states the theorem and the proof sketch illustrates the foregoing discussion.

THEOREM 2. *All Custard queries are safe.*

Proof sketch. In essence, any Custard query maps to finitely many applications of the query definitions D_1–D_{29}. Specifically, a query maps to a binary tree of height h where the leaf nodes are Base events and the root is the query itself. We must show that every query at every height $k (0 \leqslant k \leqslant h)$ has a finite extension if its children have finite extensions.

The proof is by induction on the height of the tree. The queries at the leaves represent the base (in the induction sense) case. We know that they have finite extensions because the model defines finite extensions for Base events. Assume finiteness for every query at height k and show finiteness for every query at height $k+1$. For brevity, we illustrate only the crucial cases that involve $\ominus$. Suppose a query at $k+1$ is $X \ominus E[l,d]$. By the inductive hypothesis, we know that both X and $E[l,d]$ have finite extensions. According to the definition of $[\![X \ominus E[l,d]]\!]$ (D_9), there are two subcases to consider, corresponding to the disjunction. In both cases though, we are selecting tuples from finite extensions of X and E (specifically, from $E[0,d]$ and $E[0,l]$). Hence, $[\![X \ominus E[l,d]]\!]$ is finite. The other cases involving $\ominus$ are analogous.

4. IMPLEMENTATION

We implemented a Custard compiler in Java using the Eclipse XText language definition and parsing library. The compiler reads in a file containing an event schema definition (such as the one in Listing 1) and norm specifications and creates two files, one containing the SQL table creation statements corresponding to the schema and another containing the SQL queries, one for each lifecycle event for each norm. Since there are several dialects of SQL, we picked one that is widely used, namely, MySQL. Listing 7 shows some of the table creation statements generated for the event schema in Listing 1. For better readability, the timestamp attribute is stamp (instead of t).

Listing 7: Generated SQL Create Table statements.

```
CREATE TABLE SentCred (
  hID  VARCHAR(10), tpID  VARCHAR(10), discID
      VARCHAR(10), credentials  VARCHAR(10),
  stamp  DATETIME,
  PRIMARY KEY(discID)
);

CREATE TABLE ReqData (
  hID  VARCHAR(10), tpID  VARCHAR(10), discID
      VARCHAR(10), reqID  VARCHAR(10), request
      VARCHAR(10),
  stamp  DATETIME,
  PRIMARY KEY(reqID)
);

CREATE TABLE Accessed (
  hID  VARCHAR(10), tpID  VARCHAR(10), reqID
      VARCHAR(10), response  VARCHAR(10),
  stamp  DATETIME,
  PRIMARY KEY(reqID)
);
```

For the authorization specification in Listing 3, the compiler generates four SQL queries corresponding to the created, expired, detached, and discharged instances (recall that in our model, authorizations can't be violated). Listing 8 shows the query that returns the created instances of the authorization at time NOW (the current time). In other words, it shows the SQL equivalent of $[\![\texttt{created}\ \mathit{DisclosureAuth}]\!]_{NOW}$. In the current implementation all queries are automatically evaluated for NOW; however, we are working on an extension where the user could input a time value. This would allow the user to run retrospective queries such as "how many instances of this authorization were created two months ago?" and hypothetical queries such as "given the current state of the database, how many instances of DisclosureCom commitments will be violated a month from now?"

Listing 8: Generated SQL for created instances of DisclosureAuth.

```
SELECT hID, tpID, discID, credentials, stamp
FROM (SELECT hID, tpID, discID, credentials,
    stamp
     FROM SentCred) AS Query0
WHERE stamp < NOW();
```

Listing 8 contains a nested query. The query is simple and could, in fact, be easily rewritten without the nesting. Listing 9, which shows the query for the discharged instances of the authorization, is far more complex, and contains several levels of unavoidable nesting. Such a query would be practically impossible to write by hand. This demonstrates the significant practical benefits of Custard.

Listing 9: Generated SQL for the discharged instances. The SQL DATETIME values '1000-01-01 00:00:00' and '9999-12-31 23:59:59' correspond to the 0th and the infinitely distant time instants, respectively, in our implementation. The unit of time is day.

```
SELECT
    hID, tpID, discID, credentials, reqID,
        response, stamp
FROM
    (SELECT
        hID, tpID, discID, credentials, reqID,
            response,
            GREATEST(Query21.stamp,
                Query28.stamp3) AS stamp
    FROM
        (SELECT
        hID, tpID, discID, credentials, stamp
    FROM
        SentCred) AS Query21
    NATURAL JOIN (SELECT
        hID, tpID, reqID, response, stamp AS
            stamp3
    FROM
        (SELECT
        hID, tpID, reqID, response,
           GREATEST(Query30.stamp,
               Query32.stamp4) AS stamp
    FROM
        (SELECT
        hID, tpID, reqID, response, discID,
            request,
            GREATEST(Query22.stamp,
                Query34.stamp5) AS stamp
    FROM
        (SELECT
        hID, tpID, reqID, response, stamp
    FROM
        Accessed) AS Query22
    NATURAL JOIN (SELECT
        hID, tpID, discID, reqID, request, stamp
            AS stamp5
    FROM
        (SELECT
        hID, tpID, discID, reqID, request, stamp
    FROM
        ReqData) AS Query23) AS Query34
    WHERE
        Query34.stamp5 + INTERVAL 0 DAY <=
            Query22.stamp
            AND Query22.stamp < '9999-12-31
                23:59:59') AS Query30
    NATURAL JOIN (SELECT
        hID, tpID, reqID, response, stamp AS
            stamp4
    FROM
        (SELECT
        hID, tpID, reqID, response, discID,
            request,
            GREATEST(Query22.stamp,
                Query36.stamp6) AS stamp
    FROM
        (SELECT
        hID, tpID, reqID, response, stamp
    FROM
        Accessed) AS Query22
    NATURAL JOIN (SELECT
        hID, tpID, discID, reqID, request, stamp
            AS stamp6
    FROM
        (SELECT
        hID, tpID, discID, reqID, request, stamp
    FROM
        ReqData) AS Query24) AS Query36
    WHERE
        '1000-01-01 00:00:00' <= Query22.stamp
            AND Query22.stamp < Query36.stamp6 +
                INTERVAL 10 DAY) AS Query31) AS
                Query32) AS Query25) AS Query28)
                AS Query26
WHERE
    stamp < NOW();
```

5. DISCUSSION

Custard is a language for specifying norms over low-level information schemas. We described its semantics and proved important properties. We also illustrated Custard with examples from healthcare. Custard supports semantically complex features that would be important in real world settings, such as the absence of events, nesting, and aggregation. Custard's novelty lies not only in that it raises norm specifications to the level of information schemas but also in the fact that it describes a general approach to formalizing norm lifecycles: others may use our basic formalization approach even when they formalize a norm's lifecycle differently (for example, considering an authorization as violated when the consequent occurs without the antecedent having occurred). The implementation of Custard to generate SQL queries is proof of its value. The generated queries even for even a simple norm turn out to enormously complex, running in tens of lines. Custard saves the programmer the effort of writing these queries by hand. We discuss below the relevant strands of literature.

Commitments. Custard follows a recent trend of more explicit information modeling in commitments [18, 5]. It is specifically informed by advances reported in Cupid [5], which presents an information-based language for commitments. Whereas Custard adopts Cupid's basic style and approach, it goes significantly beyond Cupid in expressiveness. Cupid supports only commitments and does not support aggregation. Cupid formulates the query extensions in terms of relational algebra whereas Custard defines them in terms of the TRC, which yields cleaner and more direct set-based formulations. The formulation and proof of stability is novel to Custard.

Custard leverages work on first-order event-based representations of commitments [26, 23, 15]. These often emphasize different aspects, for example, reasoning about commitment operations, richer commitment content, and deadlines. Custard attempts to bring together these concerns in a single expressive language.

Institutions and Norms. Norms are widely studied in multiagent systems from different perspectives. Important overlapping themes relevant to Custard concern the modeling of institutions and contracts [16, 3, 13], norm reasoning and conflicts [25, 9, 19], monitoring and reasoning about agent compliance with norms [24, 1, 17], and programming norm-aware adaptive agents [6, 14, 2, 15, 24, 12, 4]. Custard could be enhanced to incorporate some of the techniques studied in this literature, for instance, to support defeasible reasoning and degrees of norm compliance. Custard could also be enhanced with an agent-oriented API to support runtime monitoring of norms and their incorporation in the agent deliberation cycle.

Representations of norms vary widely. Custard models norms as directed expectations between agents. Generally, work on norms (excluding commitments, which are mostly treated as directed) does not model them explicitly as expectations between agents, as we do in Custard. For example, the norm representations of Artikis et al. [3] are essentially single-agent. Modgil et al. [17] have the notion of the *target* agents of the norms; however, it is not possible to distinguish between the expector and expectee in their representation. Further, in some of their examples, the target set contains only one agent. In general, single-agent formulations could be understood as the special case where the expector implicitly is the institution itself. Some norm formulations are devoid of agents, for example, King et al.'s [13].

Future Work. Custard represents an initial effort to represent norms in an information-oriented framework. It opens up many interesting directions of work. One, formalizing norms in higher-level database logics such as 4QL, which has been applied to settings of collaborative agents [7]. Two, extending the reasoning to settings where there is uncertainty associated with the occurrence of events [22]. Three, studying the expressiveness of Custard by encoding a variety of norm patterns that arise in real-life settings and, where necessary, extending Custard and creating macros that capture complex oft-repeating patterns. Four, devising a tool-supported methodology for writing Custard specifications. In the present paper, we did not dwell on well-formedness criteria for specifications or how we would come up Custard specifications given requirements or a low-level information schema. Techniques from traditional information modeling would be relevant. Five, relating Custard specifications to interaction specifications and formulating consistency protocols for ensuring that multiple Custard stores remain adequately synchronized.

REFERENCES

[1] N. Alechina, N. Bulling, M. Dastani, and B. Logan. Practical run-time norm enforcement with bounded lookahead. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 443–451. IFAAMAS, 2015.

[2] N. Alechina, M. Dastani, and B. Logan. Programming norm-aware agents. In *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS'12)*, pages 1057–1064, Valencia, 2012. IFAAMAS.

[3] A. Artikis, M. J. Sergot, and J. V. Pitt. Specifying norm-governed computational societies. *ACM Transactions on Computational Logic*, 10(1), 2009.

[4] M. Baldoni, C. Baroglio, and F. Capuzzimati. A commitment-based infrastructure for programming socio-technical systems. *ACM Transactions on Internet Technologies*, 14(4):23:1–23:23, Dec. 2014.

[5] A. K. Chopra and M. P. Singh. Cupid: Commitments in relational algebra. In *Proceedings of the AAAI*, pages 2052–2059, 2015.

[6] F. Dignum. Autonomous agents with norms. *Artificial Intelligence and Law*, 7(1):69–79, 1999.

[7] B. Dunin-Kęplicz and A. Strachocka. Tractable inquiry in information-rich environments. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, pages 281–300, 2015.

[8] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, Redwood City, California, 2nd edition, 1994.

[9] K. V. Hindriks and M. B. V. Riemsdijk. A real-time semantics for norms with deadlines. In *Proceedings of the 12th International Conference on Autonomous agents and Multiagent Systems*, pages 507–514, St. Paul, MN, USA, 2013. IFAAMAS.

[10] HL7. Consent directive use cases. http://wiki.hl7.org/index.php?title=Consent_Directive_Use_Cases.

[11] A. J. I. Jones and M. J. Sergot. On the characterisation of law and computer systems: The normative systems perspective. In J.-J. C. Meyer and R. J. Wieringa, editors, *Deontic Logic in Computer Science: Normative System Specification*, chapter 12, pages 275–307. John Wiley and Sons, Chichester, UK, 1993.

[12] Ö. Kafalı, A. Günay, and P. Yolum. GOSU: Computing goal support with commitments in multiagent systems. In *Proceedings of 21st European Conference on Artificial Intelligence*, pages 477–482, 2014.

[13] T. C. King, T. Li, M. D. Vos, V. Dignum, C. M. Jonker, J. Padget, and M. B. van Riemsdijk. A framework for institutions governing institutions. In *Proceedings of the Fourteenth International Conference on Autonomous Agents and Multiagent Systems*, pages 473–481. IFAAMAS, 2015.

[14] F. Meneguzzi and M. Luck. Norm-based behaviour modification in BDI agents. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 177–184. IFAAMAS, 2009.

[15] F. Meneguzzi, P. R. Telang, and M. P. Singh. A first-order formalization of commitments and goals for planning. In *Proceedings of the 23rd Conference on Artificial Intelligence (AAAI)*, pages 697–703, Bellevue, Washington, July 2013. AAAI Press.

[16] F. R. Meneguzzi, S. Miles, M. Luck, C. Holt, M. Smith, N. Oren, N. Faci, M. Kollingbaum, and S. Modgil. Electronic contracting in aircraft aftercare: A case study. In *Proceedings of the Seventh International Conference on Autonomous Agents and Multiagent Systems*, pages 63–70, 2008.

[17] S. Modgil, N. Oren, N. Faci, F. Meneguzzi, S. Miles, and M. Luck. Monitoring compliance with E-contracts and norms. *Artificial Intelligence and Law*, 23(2):161–196, 2015.

[18] M. Montali, D. Calvanese, and G. D. Giacomo. Verification of data-aware commitment-based multiagent system. In *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems*, pages 157–164, Paris, May 2014. IFAAMAS.

[19] A. Rotolo, G. Governatori, and G. Sartor. Deontic defeasible reasoning in legal interpretation: Two options for modelling interpretive arguments. In *Proceedings of the 15th International Conference on Artificial Intelligence and Law*, pages 99–108. ACM, 2015.

[20] M. P. Singh. Norms as a basis for governing sociotechnical systems. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 5(1):21:1–21:23, Dec. 2013.

[21] M. P. Singh. Cybersecurity as an application domain for multiagent systems. In *Proceedings of the 14th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS)*, pages 1207–1212, Istanbul, May 2015. IFAAMAS. Blue Sky Ideas Track.

[22] A. Skarlatidis, A. Artikis, J. Filippou, and G. Paliouras. A probabilistic logic programming event calculus. *Theory and Practice of Logic Programming*, 15(02):213–245, 2015.

[23] P. Torroni, F. Chesani, P. Mello, and M. Montali. Social commitments in time: Satisfied or compensated. In *Proceedings of the 7th International Workshop on Declarative Agent Languages and Technologies (DALT)*, volume 5948 of *LNCS*, pages 228–243. Springer, 2009.

[24] M. B. van Riemsdijk, L. A. Dennis, M. Fisher, and K. V. Hindriks. Agent reasoning for norm compliance: A semantic approach. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multiagent Systems*, pages 499–506. IFAAMAS, 2013.

[25] W. W. Vasconcelos, M. J. Kollingbaum, and T. J. Norman. Normative conflict resolution in multi-agent systems. *Autonomous Agents and Multi-Agent Systems*, 19(2):124–152, 2009.

[26] P. Yolum and M. P. Singh. Flexible protocol specification and execution: Applying event calculus planning using commitments. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 527–534. ACM Press, 2002.