Rafael H. Bordini, Louise A. Dennis and Yves Lespérance (eds)
Pre-proceedings of the 7th International Workshop on Engineering Multi-Agent Systems (EMAS 2019)
13th-14th May 2019, Montreal, Canada
cgi.csc.liv.ac.uk/~lad/emas2019/accepted/EMAS2019_paper_34.pdf

Hercule: Reasoning about Norms over Unstructured Events

Samuel H. Christie V[1], Amit K. Chopra[1], and Munindar P. Singh[2]

[1] Lancaster University, Lancaster LA1 4WA UK
{samuel.christie,amit.chopra}@lancaster.ac.uk
[2] North Carolina State University, Raleigh NC 27607, USA
mpsingh@ncsu.edu

Abstract. Existing work on representing and reasoning about norms has largely focused on a relational information model. However, the relational model is rigid and able to store only information and events that match a predefined schema, or is simply not available in some IT settings. We propose a new approach for realizing norms based on a document-store model. In particular, we provide a language for specifying norms, and an implementation that realizes this language and supports reasoning about norms stored as unstructured documents in CouchDB.

1 Introduction

Sociotechnical systems are multiagent systems that involve both social agents or *principals* and technical elements such as computing systems. A norm is an expectation about how agents in a social context ought to interact. One example of a norm from healthcare is of a prohibition against a healthcare provider disclosing private health information without consent. Protecting private information is traditionally done by humans with knowledge of the relevant regulations and policies. However, these social expectations can be formalized and reasoned about computationally, enabling the development of socially aware software agents capable of performing the same functions.

Abstractly specifying norms as regulations rather than rigidly enforcing them via regimentation and automated systems preserves agent autonomy and flexibility, enabling more robust and reliable systems. By autonomously reasoning about norms, an agent can better handle exceptional circumstances. For example, many electronic health record systems have access controls in place to prevent personnel from accessing patient files unrelated to their work in accordance with the prohibition against unauthorized disclosure. However, in the case of an emergency, a doctor or nurse can override the protections, recording the reason and their identity. After the emergency is over, the records can be reviewed to determine if the access was warranted, and sanctions imposed if it was not. This approach preserves privacy, but not at the risk of a patient's life.

Since norms are not directly enforced, they are effective only to the extent that agents are aware of and act on them. The agents must not only know

that certain norms exist, but also how to apply them. For example, a healthcare provider should not only know they ought to delete private data when the patient requests it, but also be aware of such requests and the resulting obligation to delete the data. Similarly, a patient should be able to identify and avoid providers that violate commitments. Thus, agents need to be able to compute the *state* of a norm based on observed events.

Reasoning about norms and using them to inform agent behavior is a well-established idea in multiagent systems. Developments in engineering socially-aware and norm aware agents include agent programming frameworks with social reasoning capabilities [3], software engineering methodologies [1, 6], planning-based [11, 12], and BDI [2] approaches to reasoning. Derakhshan et al. described an architecture using a rules engine to reason over events, but only at a high level without details about storage or format, or an example implementation [8]. Criado et al. developed a BDI-compatible framework for reasoning about constitutive norms, and demonstrated that it helped the agents better understand facts about the organization, and actions they could and should take in a given situation, supporting agent autonomy [7].

Custard [5] and Cupid [4] are both prior approaches to representing and reasoning about norms using a relational model. However, the relational approach has several limitations. For instance, most relational systems expect that the number and names of the columns (that is, attributes) themselves remain constant. Furthermore, there are environments in which the relational model is undesirable or even unavailable. For example, the Hyperledger Fabric blockchain framework (in its current version) provides only a LevelDB based key-value store, and a CouchDB-based document store. Enabling normative reasoning on blockchain platforms will be important if more business relationships are instantiated as "smart contracts" and in fact higher-level declarative specifications of normative relationships may be essential to making such contracts viable [10].

In such document stores, documents do not (necessarily) have a schema; that they may have fields added or removed at any time. This is what we mean by unstructured data—data without a predefined schema or guaranteed structure, as is commonly found in document stores. The lack of structure usually means that the database does not have the kind of indexing required for arbitrary and efficient join queries. Instead, more complex queries can be computed using highly efficient and parallel map and reduce operations.

Our motivation to build a normative computation service on top of document stores such as CouchDB is motivated by the possibility of efficiency and better scaling to larger data sets, as well as simply the ability to process existing data that may be stored in such systems. Using an unstructured event model has minimal impact on the norm specification language, and instead primarily affects the communication between an agent and the reasoner. As will be demonstrated, the difference is primarily that the unstructured approach does not require an event schema, but does require an enactment identifier to associate events that need to be processed together.

1.1 Contributions

We examine the conceptualization of a normative system in terms of the document data model, provide a norm specification language specialized for the model, and develop a prototype service for reasoning about norms using events in a document store.

1.2 Case Study

The examples in this paper are drawn from the setting of health care and clinical trials. Specifically, patient consent for participation in clinical trials. Many healthcare providers (HCPs) have electronic health record (EHR) data about their patients that could be useful in research, but privacy regulations prevent them from sharing that data outside of the organization, unless specifically authorized by the patient. Getting a patient's approval and securely sharing data with external researchers is a complex problem, but for illustration purposes we will be using a simplified construction.

The requirements are as follows:

1. **Storage**: After a patient visits an HCP, the HCP should store the data so that the patient's doctor can access it at any time.
2. **Destruction**: The HCP should destroy a patient's data upon request by the patient
3. **Sharing**: An agent authorized by the patient to access their data should receive requested data in a timely fashion, unless access was revoked before the request was made.
4. **Privacy**: No other agent should be able to access the documents except the patient, their doctor, and those authorized by the patient.

2 Social Norms

For clarity, we use terminology and semantics based on the existing literature on norms [14]. A norm is a directed expectation between two agents, the *subject* and *object*. Each norm has an *antecedent*—the condition under which it takes effect—and a *consequent*—the condition under which it is fully satisfied. A specific *instance* of a norm can exist in one of several states: *created*, when the norm is instantiated for particular agents, *detached* when the antecedent holds, *discharged* when the consequent holds, and *violated* when the norm is detached but cannot be discharged, because time limits have passed.

In a **commitment**, the subject or *creditor* commits to the object or *debtor* that they will ensure the consequent holds when the antecedent is true.

An **authorization** is a norm describing that the object allows the subject to bring about the consequent when the antecedent is true. A friend or relative can be authorized to access a patient's health records (consequent) if the patient submits a release form (antecedent).

A **prohibition** is a norm that forbids the subject from bringing about the consequent if the antecedent holds. A healthcare provider is prohibited from

sharing patient data with an agent (consequent) if the patient has not granted access (antecedent).

Although various authors have defined other kinds of norms and norm states, the above is sufficient for our purposes of demonstrating basic reasoning about norms.

3 Technical Framework

Hercule is a service for both specifying and running a normative system. Specifically, the offline portion of Hercule uses norm specifications written in a domain-specific language to generate queries. The online portion then uses those queries to compute current norm states based on events reported to it by an agent.

3.1 Architecture

At a high level, the architecture of a multiagent system using Hercule for normative reasoning should look something like the following diagram.

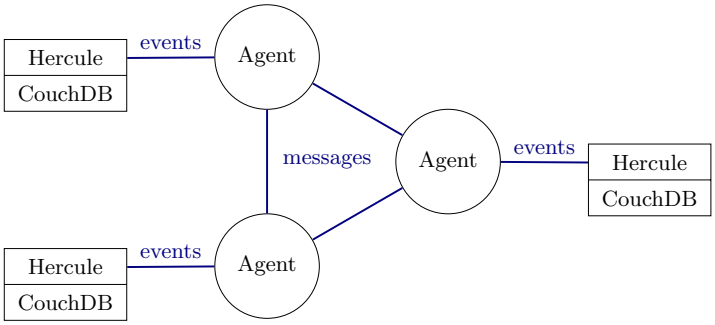


Fig. 1. Architecture diagram for a multiagent system using Hercule

Hercule does not provide any of the other features necessary for building or operating a multiagent system, such as the messaging and communication between agents. Each Hercule instance operates independently, providing reasoning capabilities to its agent based on the information it is provided, from the perspective of that agent. The Hercule instances do not share any data, but operate exclusively on information known by their agent.

Hercule computes norm states from *events* which are reported to it by the agent, and then stored internally in an encapsulated document store. These events can have any structure, as they are not constrained by a schema. Whereas a system specification in Hercule may define norms and their states in terms of fields on an event, these identified fields do not constrain the stored events in any way. Also, each field can contain any type of data, though some types are more amenable to comparison than others.

An instance of Hercule does not have direct access to the environment or messages received by the agent; it is simply a reasoning tool, not a fully-featured

MAS implementation framework. As such, relevant events must be reported to Hercule by the agent. Decoupling events from messages gives the MAS developer full control over integration, adding Hercule reasoning to agents in an existing system on an as-needed basis.

However, decoupling means that event replication is not automatic between agents. Rather, each agent must independently observe or receive any relevant events and report them to its Hercule instance. Relying only on each agent's local perspective naturally avoids problems due to scaling and over-sharing of information.

While the proposed architecture is the simplest and best suited to scalable multiagent systems, there is nothing preventing Hercule from being used as a shared resource among multiple agents, or treated as an artifact in the environment. The current implementation assumes for simplicity that the agent submitting events is honest, which is reasonable in a single-agent case; it would deceive no one but itself. If used as a service by multiple agents, either the assumption of honesty must hold or access controls should be implemented to prevent agents from submitting events using false identities. Timestamp falsification is already protected against, because Hercule handles all time internally anyway.

3.2 Syntax

The syntax used by Hercule is loosely inspired by Custard and Cupid. Table 1 shows the formal syntax that we adopt for Hercule, taken almost directly from the parser specification of our implementation.

An example specification in Hercule of the norms described in Section 1.2 is given in Listing 1:

Listing 1. Norm specifications in Hercule

```
context Consultation(patient, doctor, hospital) {
 commitment StoreData(hospital->patient):
  created: patient.Visit{date}
  detached: doctor.Record{patient, item}
  discharged: hospital.Store{item}

 commitment DestroyData(hospital->patient):
  created: hospital.Store{item}
  detached: patient.RequestDeletion{item}
  discharged: hospital.Deleted{item}

 authorization Access(patient->recipient, item):
  created:
   patient.GrantAccess{recipient, item} @ t
  detached:
   recipient.RequestAccess{item} @ t2 > t
   except patient.RevokeAccess{recipient,item} @ [t,t2]
  discharged:
   hospital.Shared{item, recipient} @ [t2, t2+10]
```

```
 prohibition Disclosure(patient->hospital):
  created: hospital.Store{patient, item}
  violated:
   hospital.Shared{item, recipient}
   except Access(patient->recipient, item):detached
}
```

Listing 1 specifies a *context* named *Consultation* containing four norms: a commitment by the hospital to store patient data (*StoreData*), a commitment by the hospital to destroy data upon request (*DestroyData*), an authorization by the patient to allow a recipient to access their health data (*Access*), and a prohibition against a hospital disclosing information to an unauthorized agent (*Disclosure*).

A context is a collection of related norms that conceptually captures the setting in which the norms are defined, and when instantiated as an *enactment* provides a scope for storing event instances and their attributes. Any names such as event attributes in the same context refer to the same information; different contexts may reuse names with different meanings. In addition, each context identifies a set of roles that must be bound to initiate an enactment. In *Consultation*, PATIENT, DOCTOR, and HOSPITAL must all be identified.

Within each context are one or more *norm* definitions. The first is a commitment named *StoreData*. Whereas Hercule does use special semantics for well-known norm kinds such as commitments, authorizations, and prohibitions, any name can be used for the kind of norm. *StoreData* is a commitment from HOSPITAL to PATIENT; that is, HOSPITAL is the debtor and thus accountable for the commitment, and PATIENT is the creditor. If it is violated, HOSPITAL is at fault. No consequences immediately or necessarily apply, but other agents can observe the fault and adjust their behavior accordingly.

Within each norm are defined one or more *states*. As with norms, there are default relationships between the well-known states but any name can be used. For example, a norm cannot be detached until it is created, regardless of the type of norm. However, new states such as *active* or *canceled* can easily be defined. Each state contains an expression used to select matching enactments from the database. The basic expression is an *event* expression. The first event expression, in the created state of *StoreData*, specifies an event named *Visit* which is created by PATIENT and contains an attribute named `date`. Thus, any enactment containing a Visit event matching this description is considered as creating an instance of *StoreData*. More complex expressions can be formed using operators such as *and*, *or*, and *except*.

Event expressions can be extended by time expressions, appended with the '@' symbol, which either label the time at which an event occurs (as in Access.created, which occurs at time t) or constrain it. The time expression may perform a simple comparison, as in $t_2 > t_1$, or restrict it to an interval, such as $[t_2, t_2 + 10]$.

Using these concepts, we can interpret the specification above as follows: *StoreData* is defined as a *commitment* with three states: *created*, *detached*, and *discharged*. *StoreData* is created when a patient visits the doctor—that is, when

the agent whose Hercule instance we are considering reports a *Visit* event containing the patient and the date of the visit. *StoreData* is detached when the doctor produces a record of the visit and reports the *Record* event with attributes describing both which patient the record is for, and what item is being recorded. Finally, *StoreData* is discharged when the hospital permanently stores the item, as indicated when the hospital reports a *Store* event with the item as an attribute.

DestroyData is also defined as a commitment, and is created when the hospital stores patient data, using the same *Store* event as above. *DestroyData* is then detached when the patient requests that their data be deleted with the *RequestDeletion* event, and discharged when the hospital deletes the data as indicated by a *Deleted* event.

Access is defined as an authorization, and created when the patient grants another agent access to their data. This occurs when the patient reports the *GrantAccess* event with the designated recipient and the item they are authorized to access as attributes. *Access* is detached when the recipient requests the data by reporting the *RequestAccess* event, so long as they make the request before the patient revokes their access by reporting the *RevokeAccess* event. Finally, *Access* is discharged when the hospital shares the data with the recipient within a certain amount of time.

The last norm in this context, *Disclosure*, is a prohibition created when the hospital stores patient data. *Disclosure* is violated if the hospital shares the data with an agent that is not authorized to access it. *Disclosure* does not reference any new events, and uses a reference to the *Access* norm for concision.

3.3 Semantics

Semantically, Hercule is flexible, and enables the designer to define custom norms using any name, with custom states that may have any desired logical relationship. There is no restriction on the names that can be used for a norm's kind or its states.

However, to reduce repetition and make norm design easier, several common norm types are treated as special cases, with predefined states and relationships between them. These predefined states and relationships are based on established definitions from the literature [13, 14].

The predefined kinds of norms are the commitment, authorization, and prohibition. Each of these norms has four states: created, detached, discharged, and violated. The designer can define additional states as needed.

For all norms, the created state is assumed to be provided by the designer. The other states are automatically extended to save repetition and guard against false positives. For example, the detached state, if provided, is modified to depend on the created state. That is, an enactment will not be considered to detach an instance of the norm unless it also creates an instance.

For commitments, the created and detached states must be provided by the designer. The discharged state is then extended so that a commitment can only be discharged if it is already created or detached. The violated state is unique,

Table 1. Syntax of Hercule.

Context	⟶	'context' Name '(' roles:NameList ')' '{' Norm+ '}'
NameList	⟶	first ',' rest:NameList | Name
Norm	⟶	Kind Name '(' Relationship (',' ParamList)? '):' State+
Name	⟶	String
Kind	⟶	String
Relationship	⟶	Name '->' Name
State	⟶	Name ':' EventConjunction
EventConjunction	⟶	EventDisjunction ('and' | 'except') EventConjunction | EventDisjunction
EventDisjunction	⟶	EventUnary 'or' EventDisjunction | EventUnary
EventUnary	⟶	('not' | 'unless') EventClause | EventClause
EventClause	⟶	'(' EventConjunction ')' | Event | Reference
Event	⟶	Role '.' Name '' ParamList '' TimeClause?
Reference	⟶	Name '(' Relationship (',' ParamList)? '):' State
ParamList	⟶	Param (',' ParamList)?
Param	⟶	Name Match?
Match	⟶	':' Name
TimeClause	⟶	'@' TimeComp | '@' '[' TimeExpr ',' TimeExpr ']'
TimeComp	⟶	TimeExpr ('>' | '<' | '==') TimeComp | TimeExpr
TimeExpr	⟶	TimeTerm ('+' | '-') TimeExpr | TimeTerm
TimeTerm	⟶	TimeFactor ('*' | '/') TimeTerm | TimeFactor
TimeFactor	⟶	'(' TimeExpr ')' | Time
Time	⟶	Integer | Name

in that it may be explicitly specified instead of a discharge clause. If a violated state is provided explicitly, it is guarded so that a commitment cannot be violated unless it is created. If a violated state is not provided, it is automatically generated so that a commitment is violated if it is detached but not discharged.

Authorizations are simpler, in that a discharge can only follow detachment, and violation is never provided explicitly but always derived from the discharged state.

Conversely, prohibitions do not expect an explicit discharge, but derive it from the violated state.

These relationships are summarized in Figure 2.

4 Representation

4.1 Unstructured Data

Events in Hercule are stored as *unstructured data*; that is, data that does not have a predefined and fixed schema. To be relevant to a specific norm an event will need fields with certain names, but there is no requirement that each event have any particular field, and fields can be added as needed.

Whereas there are many document store databases, we implement Hercule on top of CouchDB [15], as a representative unstructured database with a map-reduce query system. As a document store, CouchDB stores objects or documents, instead of rows in a table with a schema. These documents are key-value dictionaries, where the values may themselves be complex data such as nested documents. CouchDB, for example, uses the JSON data format for its documents. An example document is given in Listing 2. Using complex formats for the documents allows the database to exchange them directly with an application. However, because a given document could have any set of keys—that is, there are no fixed schema—CouchDB does not automatically index for each key the way a relational database would index each column. Instead it provides the massively parallel *map* and *reduce* operations to enable a user to generate their own indexes or, as CouchDB names them, *views* of the data. The map function is applied to each document in the database, and can emit one or more output documents that are saved in a separate collection. The logic it uses can be as complicated as necessary, as the map functions are implemented using the full JavaScript programming language. However, the map process is designed to work in parallel, and so must operate on each document in isolation—it cannot join or compare fields across documents, as one might do in a complex SQL query. Optionally, a reduce function may combine all of the result documents to compute something like a count, sum, or average over all the results.

In CouchDB, the JavaScript map and reduce functions are stored as strings in documents alongside the other data in the database, in documents called *design documents*. Any time a design document is modified, the database detects the

Fig. 2. Norm Semantics

Commitment:
- detached = detached ∧ created
- discharged = (created ∧ discharged) ∨ (detached ∧ discharged)
- violated = detached ∧ violated
  or
- violated = detached ∧ ¬discharged

Authorization:
- discharged = detached ∧ discharged
- violated = detached ∧ ¬discharged

Prohibition:
- discharged = detached ∧ ¬violated

change and recomputes the associated view. The first time a view is queried, it batch processes all of the documents in the database. Subsequent queries reuse the previous results, only applying the map and reduce functions to documents that have not been processed yet.

As a service for reasoning about norm states, Hercule generates map and reduce functions from norm specifications, so that the norm states can simply be queried as needed.

4.2 Context and Enactments

The documents that Hercule stores are *enactments*, or instances of the social context. Each enactment identifies the agents performing each of the roles in the social context. An enactment includes one or more events that monotonically bind information attributes.

For example, the *Consultation* context specified above in Listing 1 contains all of the norms and information related to the healthcare privacy setting.

An instance of this context might look like the example enactment document given in Listing 2.

Listing 2. An example enactment document.

```
{
  "_id": "7c4f054dc8740698c93a9452e856cc87",
  "patient": "P",
  "doctor": "D",
  "hospital": "H",
  "date": "2018-11-16",
  "item": "Diagnosis",
  "Visit": {
    "$by": "P",
    "date": "2018-11-16",
    "$time": 1,
  },
  "Record": {
    "$by": "D",
    "patient": "P",
    "item": "Diagnosis"
    "$time": 2,
  },
  "Store": {
    "$by": "H",
    "item": "Diagnosis"
    "$time": 3,
  }
}
```

Listing 2 clearly illustrates several of the important features of the way enactments are represented. First, it contains a unique identifier, in this case the _id field. Second, it has a set of bindings for each of the roles declared in the context specification. For example, patient is bound to P, and doctor to D. Finally, each

of the events that have occurred in the enactment are represented as a single object attached as a top-level property.

Note that all of the events necessary for determining the state of a norm must be stored in the same context document, because map and reduce operate only on individual documents in isolation. Correlation of information across documents would require additional processing after querying the database.

Also, Hercule does not automatically identify new enactments or match events to them. Instead, the agent must generate an enactment ID, and attach it to all events it deems part of that enactment.

4.3 Events

Events are stored as top-level properties of enactment documents, which can be easily matched against, and each event is itself an object containing information attributes.

Listing 3 focuses on a single event instance.

Listing 3. Example event object.

```
"Store": {
  "$by": "H",
  "item": "Diagnosis",
  "$time": 1
}
```

The information payload of the event is copied into the parent enactment document, because the information is monotonically bound and cannot be contradicted or overwritten within a single enactment. In this way, an attribute can contribute to the state of a norm, regardless of the event the attribute came from.

Time of occurrence is an important and distinctive attribute of an event. Without time it would be impossible to have a concept of events, as there would be no frame of reference for detecting change [9]. In Hercule, the timestamp of each event is represented simply as an integer for ease of manipulation and comparison, and is injected by the service when it is reported to ensure a consistent concept of time.

4.4 Norm Instances

Norm instances are not reified objects stored in the database, but rather expectations derived from stored events. Thus, a norm instance is realized in Hercule as a map-reduce process that collects all of the relevant events and then computes the resulting state from them.

Some norms are defined in terms of the states of other norms. For example, the Disclosure prohibition described in Listing 1 refers to the Access authorization, which is itself a norm. Unfortunately, the map-reduce implementation provided by CouchDB cannot directly reference another view or query, so Hercule compiles the referenced logical expression into a single query.

5 Implementation

A proof of concept for Hercule has been implemented in JavaScript using the `pegjs` parser generator, and the `astring` code generator.

In CouchDB, the map-reduce model is provided via *views* [16]: indexes over the data computed by JavaScript map and reduce functions stored in *design documents*.

Hercule processes a context specification to produce a design document for each norm, with one view for each state. When loaded into the database, each view is applied to the data to produce a collection of matching norm instances. These collections can then be queried to discover the current states of various norm instances and inform agent behavior, e.g., by detecting past violations by a potential provider. The specific agent behavior in response to norm states is outside Hercule's scope.

Hercule can operate both on and offline. This flexibility results from the architecture as a simple interface to CouchDB, computing a view from unmodified events. If used offline, CouchDB will perform the map-reduce computations in an efficient batch operation, while online usage will appropriately operate incrementally on any recent changes to the data. The computation does not modify any of the documents, so either the data or the norm specification can be updated and the query results recalculated at any time without repercussion. Though we have not implemented it, CouchDB also provides a change notification API, so that Hercule could easily be extended to push notifications of norm state changes to an agent as they occur in real time, instead of waiting for the agent to query the latest state.

Listing 4 shows part of the design document generated by Hercule for the Consultation context specification given in Listing 1.

Listing 4. StoreData design document (generated from Listing 1)

```
"StoreData": {
 "language": "javascript",
 "views": {
  "created": {
   "map":
    "function (doc) {
      doc.Visit && emit(doc)
     }"
  },
  "detached": {
   "map":
    "function (doc) {
      doc.Visit && doc.Record && emit(doc)
     }"
  },
  "discharged": {
   "map":
    "function (doc) {
      (doc.Visit && doc.Store
```

```
        || doc.Visit && doc.Record && doc.Store)
      && emit(doc)
     }"
  },
  "violated": {
   "map":
    "function (doc) {
      doc.Visit
     && (doc.Visit && doc.Record
     && !(doc.Visit && doc.Store
           || doc.Visit && doc.Record && doc.Store))
     && emit(doc)
     }"
  }
 }
}
```

Listing 4 contains the code generated for the StoreData commitment.

As a design document, *StoreData* consists of two keys, language and views. Each view has the name of the state as its key, and a single map function implementing the query logic. Each map function is applied to every document in the database, via the parameter doc, to produce one or more results via emit.

Each map function contains a single Boolean expression testing whether a given doc matches the specified norm state. For example, the created function in *StoreData* simply emits all documents that contain the *Visit* event.

According to the semantics of a commitment, the detached state requires the norm to already be in the created state. For simplicity, Hercule compiles the two conditions into a single expression. Thus, the detached function checks that both *Visit* and *Record* have occurred. Similarly, discharged requires that either created or detached be true. Finally, violated need not be explicitly specified, but rather is generated automatically—if created and detached are true but discharged is not, the commitment is violated.

Listing 5. Access design document

```
"Access": {
 "language": "javascript",
 "views": {
  "created": {
   "map":
    "function (doc) {
      doc.GrantAccess && doc.GrantAccess.$time == doc.t
     && emit(doc)
     }"
  },
  "detached": {
   "map":
    "function (doc) {
      doc.GrantAccess
     && doc.GrantAccess.$time == doc.t
```

```
      && (doc.RequestAccess && doc.t2 > doc.t)
      && !(doc.RevokeAccess
           && (doc.RevokeAccess.$time > doc.t
           && doc.RevokeAccess.$time < doc.t2))
      && emit(doc)
     }"
  },
  "discharged": {
   "map":
    "function (doc) {
      doc.GrantAccess
      && doc.GrantAccess.$time == doc.t
      && (doc.RequestAccess && doc.t2 > doc.t)
      && !(doc.RevokeAccess
           && (doc.RevokeAccess.$time > doc.t
           && doc.RevokeAccess.$time < doc.t2))
      && (doc.Shared
      && (doc.Shared.$time > doc.t2
      && doc.Shared.$time < doc.t2 + 10))
      && emit(doc)
     }"
  },
  "violated": {
   "map":
    "function (doc) {
      false && emit(doc)
     }"
  }
 }
}
```

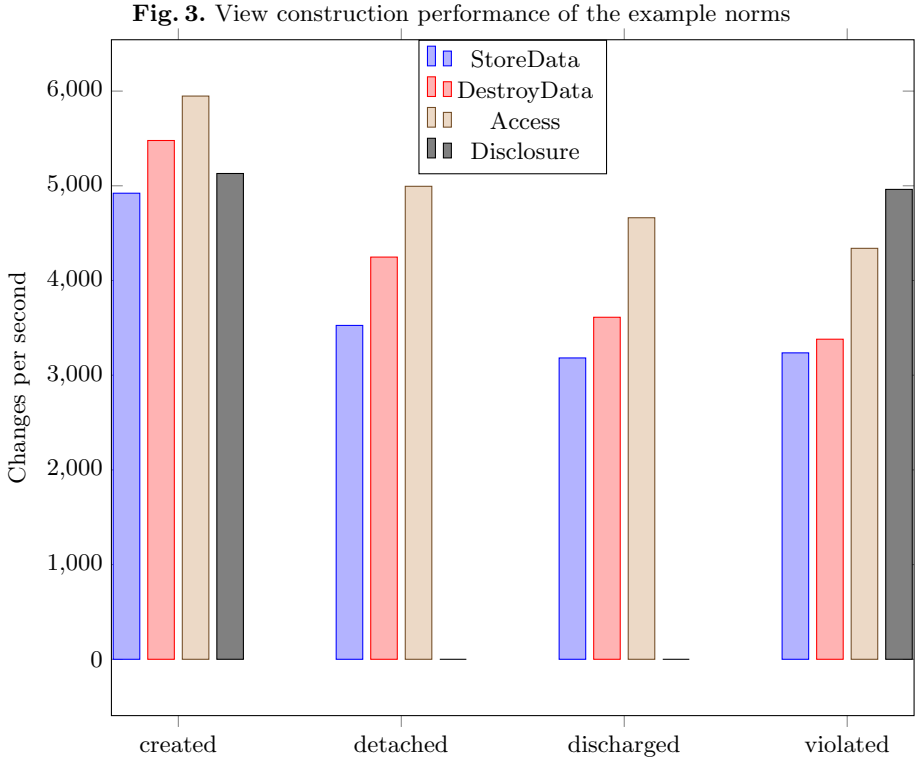
Rather significantly, the Access norm includes time constraints on the events. The compiled view function checks that the event attributes match those in the document, and that the timestamp is within the specified range.

6 Performance

To estimate the practicality of this system, we measured the time it takes for CouchDB to produce views for the example norms using randomly generated enactments. We first filled the database with lots of random enactments and then triggered the view generation process, sampling periodically as it processed to get a report from CouchDB of how many "changes per second" the database was performing. In this case, changes mean documents generated by the map function to populate the view. Thus, we can use the changes-per-second measure as an estimate for how quickly the map-reduce approach can process incoming events. We used a large number of pregenerated enactments so that the processing time would take long enough to actually sample multiple times.

Each enactment was generated based on a uniformly-distributed random number from 0 to 9. Enactments with higher numbers progressed further, and included more events. A bar plot of the results is displayed in Figure 3.

Fig. 3. View construction performance of the example norms



CouchDB does not generate a view until it is first retrieved. This data was collected by creating a new database to prevent caching and data reuse, loading 200,000 randomly generated enactments, and probing the *changes per second* statistic multiple times while the database created the view. The number of enactments was arbitrary, selected to achieve a reasonable minimum number of performance samples. This result was obtained from a multiprocess, single node CouchDB installation running in docker on a laptop.

The measured throughput ranges from 3-6k changes per second across up to 8 parallel tasks. For each norm, *created* is faster than *detached*, which is faster than *discharged*. This seems reasonable, given that each state depends on the previous ones, and therefore subsumes all of their logic. However, the later norms have higher throughput in a given state than the earlier ones. One possible explanation is that because of the way the enactments are generated, with lower odds of later events such as *GrantAccess* occurring, *Access* was able to quickly reject a larger percentage of the enactments as not having the required events, and consequently not spend further time checking the values or writing the enactments to the view. One exception is *Disclosure*'s *created* state, which

is identical to that of *DestroyData*, except that it requires an additional patient attribute on the *Store* event and so slightly slower.

These results indicate that performance is affected by the complexity of the generated code, and is not limited by write speed. How norms are written by the designer and optimized by the the query generator can have an effect on performance, but should not negatively impact most use cases.

The results correspond to a throughput of three thousand norm-state changes per second. That could be one event and three thousand norms, or three thousand events and a single norm. The more complex norms were more efficient, because their states were not triggered as frequently. This seems more than sufficient for a practical real-time norm processing environment. Even VisaNet, the largest transaction processing system in the world, handles on average about 1736 transactions per second worldwide [17]. Plus, Hercule only computes the states for norms that the agents using it actually query, not necessarily all of the norms it has specifications for.

7 Discussion

Hercule demonstrates a possible approach to realizing normative reasoning over events and information stored in scalable, document-based databases.

A significant disadvantage compared to relational databases is the difficulty of composing or joining queries. In the current implementation, there is no way to reason about norms affected by events that occur across multiple interactions, since they are stored in separate documents.

To some extent this limitation can be circumvented using a thicker client architecture. Instead of leaving all of the querying and reasoning up to the database, the Hercule service could act as an intermediary that receives higher level queries from an agent, and computes the answer using information from multiple database queries.

7.1 Future Work

Extending Hercule to support communication protocol specifications and directly ingest messages would help with automatically identifying enactments, and storing events in the correct enactment. It could also simplify adopting Hercule in an existing multiagent system, and reduce duplication in specifications.

Acknowledgments

We thank the reviewers for providing useful comments. Chopra and Christie were supported by EPSRC grant EP/N027965/1 (Turtles). Singh thanks the US Department of Defense for partial support under the Science of Security Lablet.

References

1. Ajmeri, N., Guo, H., Murukannaiah, P.K., Singh, M.P.: Arnor: Modeling social intelligence via norms to engineer privacy-aware personal agents. In: Proceedings of the 16th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 230–238. IFAAMAS, São Paulo (May 2017)
2. Alrawagfeh, W., Meneguzzi, F.: Utilizing permission norms in BDI practical normative reasoning. In: Coordination, Organizations, Institutions, and Norms in Agent Systems X - COIN 2014 International Workshops, COIN@AAMAS, Paris, France, May 6, 2014, COIN@PRICAI, Gold Coast, QLD, Australia, December 4, 2014, Revised Selected Papers. pp. 1–18. Springer (2014). https://doi.org/10.1007/978-3-319-25420-3_1, https://doi.org/10.1007/978-3-319-25420-3_1
3. Baldoni, M., Baroglio, C., Capuzzimati, F., Micalizio, R.: Commitment-based agent interaction in jacamo+. Fundamenta Informaticae **159**(1-2), 1–33 (2018). https://doi.org/10.3233/FI-2018-1656, https://doi.org/10.3233/FI-2018-1656
4. Chopra, A.K., Singh, M.P.: Cupid: Commitments in relational algebra. In: Proceedings of the 29th Conference on Artificial Intelligence (AAAI). pp. 2052–2059. AAAI Press, Austin, Texas (Jan 2015)
5. Chopra, A.K., Singh, M.P.: Custard: Computing norm states over information stores. In: Proceedings of the 15th International Conference on Autonomous Agents and MultiAgent Systems (AAMAS). pp. 1096–1105. IFAAMAS, Singapore (May 2016)
6. Chopra, A.K., Singh, M.P.: From social machines to social protocols: Software engineering foundations for sociotechnical systems. In: Proceedings of the 25th International World Wide Web Conference. pp. 903–914. ACM, Montréal (Apr 2016)
7. Criado, N., Argente, E., Noriega, P., Botti, V.J.: Reasoning about constitutive norms in BDI agents. Logic Journal of the IGPL **22**(1), 66–93 (2014). https://doi.org/10.1093/jigpal/jzt035, https://doi.org/10.1093/jigpal/jzt035
8. Derakhshan, F., Bench-Capon, T., McBurney, P.: Dynamic assignment of roles, rights and responsibilities in normative multi-agent systems. Journal of Logic and Computation **22**, 1–18 (2012), published online. doi: 10.1093/logcom/exr027
9. Kamp, H.: Events, instants, and temporal reference. In: Bauerle, R., Schwarze, C., von Stechow, A. (eds.) Semantics from Different Points of View, pp. 376–417. Springer, Berlin (1979)
10. Magazzeni, D., McBurney, P., Nash, W.: Validation and verification of smart contracts: A research agenda. IEEE Computer **50**(9), 50–57 (Sep 2017)
11. Meneguzzi, F., Magnaguagno, M.C., Singh, M.P., Telang, P.R., Yorke-Smith, N.: Goco: Planning expressive commitment protocols. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **32**(4), 459–502 (Jul 2018)
12. Panagiotidi, S., Vázquez-Salceda, J.: Towards practical normative agents: A framework and an implementation for norm-aware planning. In: Coordination, Organizations, Institutions, and Norms in Agent System VII, COIN 2011 International Workshops, COIN@AAMAS 2011, Taipei, Taiwan, May 3, 2011, COIN@WI-IAT 2011, Lyon, France, August 22, 2011, Revised Selected Papers. pp. 93–109 (2011). https://doi.org/10.1007/978-3-642-35545-5_6, https://doi.org/10.1007/978-3-642-35545-5_6
13. Singh, M.P.: An ontology for commitments in multiagent systems: Toward a unification of normative concepts. Artificial Intelligence and Law **7**(1), 97–113 (Mar 1999)

14. Singh, M.P.: Norms as a basis for governing sociotechnical systems. ACM Transactions on Intelligent Systems and Technology (TIST) **5**(1), 21:1–21:23 (Dec 2013)
15. The Apache Software Foundation: Couchdb (Dec 2018), http://couchdb.apache.org/
16. The Apache Software Foundation: Guide to views (2018), https://docs.couchdb.org/en/stable/ddocs/views/index.html
17. Vermeulen, J.: Visanet – handling 100,000 transactions per minute (2016), https://web.archive.org/web/20190330124843/https://mybroadband.co.za/news/security/190348-visanet-handling-100000-transactions-per-minute.html