Event-Driven Rule-Based Reasoning using EYE

Ben De Meester[1], Dörthe Arndt[1], Pieter Bonte[2], Jabran Bhatti[3], Wim Dereuddre[3], Ruben Verborgh[1], Femke Ongenae[2], Filip De Turck[2], Erik Mannens[1], and Rik Van de Walle[1]

[1] Ghent University – iMinds – Multimedia Lab, Belgium
{ben.demeester, doerthe.arndt}@ugent.be
[2] IBCN research group, INTEC, Ghent University – iMinds, Belgium
{pieter.bonte, femke.ongenae, filip.deturck}@intec.ugent.be
[3] Televic Healthcare, Belgium
{j.bhatti, w.dereuddre}@televic.com

Abstract. Ontologies and reasoning algorithms are considered a promising approach to create decision making applications. Rule-based reasoning systems have the advantage that rule sets can be managed and applied separately, which facilitates the custom configuration of those systems. However, current implementations of rule-based reasoning systems usually introduce a trade-off between expressiveness and performance, which either deteriorates the configurability of the application, or limits its performance in an event-driven system. In this paper, we devise an event-driven rule-based reasoning system that preserves its expressiveness. We devise an automatic nurse call system that is able to handle hard time constraints without limiting the possibilities of the reasoner, and list the encountered problems together with their suggested solutions. We achieve reasonable performance in small-scale environments when evaluating this system using N3 rules and the EYE reasoner. We however observe that a large dynamic database limits the performance of the system, because of the file-based nature of the EYE reasoner. As long as no in-memory reasoning is supported, the performance of the resulting system cannot compete with the state of the art. However, the linear scaling of the proposed expressive solution is promising.

Keywords: event-driven, EYE, N3, rule-based reasoning, stream reasoning

1 Introduction

This paper handles the building of an automatic nurse call system in a hospital. When assistance is needed (e.g., when a patient calls for a nurse, or a monitoring machine launches a call) the most suited nurse needs to be assigned to provide for the needed assistance. The definition of "most suited" depends on the context, e.g., the trust relationship between the nurse and the patient, the competences of the available nurses, and the current locations of these nurses (among others). Moreover, this "most suited" definition can be different for each hospital.

This decision making application thus needs to separate the execution strategy from the execution engine, to enable ease of configurability. Such an execution strategy is usually drafted as a decision tree, i.e., a sequence of "if this then that". To design

automatically executable decision trees, reasoning engines can be used. These reasoning engines do not hard-code the execution strategy, but use external descriptions to define this execution strategy, and thus provide for the needed separation. As such, execution strategies could be devised and adjusted in different environments, without the need to re-implement the entire application. To describe the context, ontologies can be used, as they formally specify a shared conceptualization [8]. Complex systems that handle real-life scenarios can thus be built by reasoning over these ontologies.

The nurse call system is an event-based system that should handle multiple events per minute (e.g., calls are being made, nurses move, shifts change, etc.), and the system has a hard time constraint, i.e., answers should always be given within a certain time.

The contribution of this paper is that we specifically devise a reasoning platform that handles changing data in a timely manner, without sacrificing the ease of changing the execution strategy. The relevant technologies and important modules of such a system will be identified and specified in this paper, and encountered problems will be tackled.

In Section 2, we analyze the use case for which this system was designed. In Section 3, we discuss relevant technologies and the state of the art, and draw conclusions based on the use case and the related work that contributed to the design decisions for the architecture of the system. In Section 4, we describe this architecture both conceptually and via a proof-of-concept implementation which is then evaluated in Section 5. Afterwards, conclusions are drawn in Section 6.

2 Use Case

The nurse call system implies implementing decision trees that are easily configurable, and taking the machine-interpretable context into account. Previous work includes an ontology built specifically to describe the context factors used for this use case, e.g., the factors that attribute to the definition of the "most suited nurse" [11]. This ontology is called ACCIO, and its default prefix is `accio`[4]. ACCIO's reasoning complexity is $\mathcal{SHIOQ}(\mathcal{D})$. Below, we list the functional and non-functional requirements of the reasoning platform.

Consistent state The platform must be able to update its knowledge base when an event is triggered, and keep this knowledge base consistent.

Scalable The platform must cope with data sets until 50 000 relevant triples (i.e., triples necessary to be included for the reasoning to be correct).

Expressive The platform must not only support the complexity of the ontology, but also allow decision trees to be configured in varying complexity. This eases configurability.

Configurability The platform must have the ability to change these decision trees at configuration time.

Real-time The platform must respond to any event within 5 seconds.

An analysis of the use case leads to the specifications listed below:

---

[4] Also see `http://users.intec.ugent.be/pieter.bonte/ontology/accio.html`

1. Only a subset of the data is susceptible to change when an event is triggered, and this subset can be defined at initialization time. This subset is quite small – around 15% – and is clearly defined, as only a couple of classes and predicates introduce dynamic data[5].
2. There are events that trigger a state change before the decision trees are visited, e.g., a nurse walks from a certain location to another implies updating the nurse's location in the knowledge base.
3. A state change can be the output of the decision trees. For example, when a nurse is busy with a certain patient, and he or she changes location, consensus is needed whether his or her status changes to *free*, to another status, or not change at all. This consensus could be different for different hospitals. These state changes should also be persisted in the knowledge base, as they possibly influence next reasoning runs.

Conclusions that can be drawn from these observations are as follows:

1. Preprocessing the static data at initialization time can give a significant performance gain, as this preprocessed optimized static data set can be reused every time an event is triggered.
2. Preliminary status changes need to be done programmatically to avoid conflicting data states. For example, when a nurse moves from the hallway to a patient's room, the current state of the reasoning system signifies that the nurse is in the hallway, whilst the event data signifies that the nurse is in a patient's room. Without updating the current state before visiting the decision trees, the reasoner would reason about the nurse being at two conflicting locations at the same time.
3. The second update needs to be done via reasoning. For example, when a patient makes a call, that call is assigned to a certain nurse (via the decision trees). This assignment should be persisted by, e.g., changing the status of the nurse to `accio:assigned`. However, this should remain configurable, as the state of the nurse might influence the decision trees (that are also configurable). The distinction between reasoning results that should be persisted and those that shouldn't is depending on the use case, but can be described by rules. For example, all reasoning results that involve a nurse's state change should be persisted. This can be intercepted by devising a rule that outputs all triples with predicate `accio:hasStatus` and type `accio:Person` (this involves reasoning, as this rule also covers the status of nurses and doctors, which cannot be solved generically by programming).

3 Related Work

The related work will be split up in three parts, to elaborate on ontology reasoning (Subsection 3.1), implementing decision trees using rules (Subsection 3.2), and stream reasoning (Subsection 3.3). We draw conclusions in Subsection 3.4

[5] This ratio is irrespective of, e.g., the amount of nurses or patients in a hospital, thus, irrespective of the initialized ABox, and was calculated by dividing the amount of triples of those classes and/or having those predicates with the total amount of triples in the database.

3.1 Ontologies and reasoning

The ACCIO ontology is described in the Web Ontology Language (OWL) [11], the ontology description standard as defined by the World Wide Web Consortium (W3C) [4].

OWL 2 profiles exist to trade expressiveness for performance, named OWL EL (Existential quantification Language), OWL QL (Query Language), and OWL RL (Rule Language). OWL EL is useful in applications utilizing an ontology that contains a large number of properties and/or classes. An example of a performant OWL EL reasoner is ELK [9]. OWL QL is created for applications where query answering is the main task, and OWL RL provides scalable reasoning without sacrificing too much expressive power. Due to the use case, the OWL RL profile seems most applicable.

3.2 Rule-based reasoning

Rule-based reasoning platforms have been used to aid in decision making use cases for quite some time [7]. Since multiple reasoning systems exist, the execution strategy defined as rules is completely independent of the actual reasoning platform. Moreover, as both decision trees and rule-based reasoning follow the same kind of thinking (i.e., a sequence of 'if this then that' steps), writing rules to specify decision trees comes very natural. The used engine should be very expressive, to maximize configurability. In rule-based reasoning engines, OWL-specific constructs need to be specified in rules[6].

Notation3 (N3) is a Semantic Web logic. Being a superset of Turtle – a serialization format of RDF data [3] – it is capable of describing everything using triples, but also capable of describing rules to be executed on those triples. N3 differentiates itself from other rule languages because of its expressiveness. For example, in N3 it is possible to create rules in the consequence, and to use built-ins. The N3 logic has monotonicity of entailment, which means that the hypotheses of any derived fact may be freely extended with additional assumptions, which is an important property when reasoning about a changing knowledge base.

The expressiveness of rule-based reasoning engines depends on which logic the underlying programming language supports, and on the inherent expressiveness of the rule language. RDFox is a triple store that supports arbitrary Datalog rules over RDF triples [10]. FuXi[7] supports the N3 syntax but it is a Datalog engine, which means that FuXi does not support all N3's expressitivity. The EYE reasoner [12] is a reasoning engine that uses an optimized resolution principle, supporting forward and backward reasoning. It is written in Prolog and supports, among others, all built-in predicates defined in the Prolog ISO standard. Backward reasoning with new variables in the head of a rule and list predicates are a useful plus when dealing with OWL ontologies[8]. This is rather difficult in Datalog. As such, EYE is more expressive than RDFox or FuXi, whilst being more performant than other N3 reasoners [12].

[6] See `http://www.w3.org/TR/owl2-profiles/#OWL_2_RL` for the rule specification of OWL RL, which can serve as a base of implementing the needed OWL constructs. However, the ACCIO ontology requires more expressiveness than only OWL RL.

[7] `https://code.google.com/p/fuxi/`

[8] For example, to implement `unionOf`, see `http://www.w3.org/TR/owl2-profiles/#cls-uni`

3.3 Stream reasoning

Complex Event Processing (CEP) is a research domain that handles the processing of real-time information, i.e., stream processing [6]. CEP detects situations of interest based on events, however, this is not sufficient for the use case at hand, as CEP cannot combine the event data stream with background knowledge, and CEP does not support reasoning. A Data Stream Management System (DSMS) handles data streams and enables installing queries on top of these data streams. Continuous SPARQL (C-SPARQL) is an example of a semantic DSMS [2].

Stream reasoning is the task of reasoning over streaming data (e.g., RDF triples) in a time window with respect to background knowledge (e.g., formalized in an ontology) [5]. Stream reasoning thus unifies general reasoning and stream processing whilst preserving the order of the events. ETALIS is a rule-based stream reasoning engine that can reason over RDFS ontologies [1].

3.4 Conclusions

When reviewing the ACCIO ontology and the decision trees of the use case, we can conclude that a lot of expressiveness is needed. For example, one of the decision tree nodes is about deciding which nurse to call, based on the location. This involves arithmetic calculation using event data (i.e., calculate the euclidean distance between the patient and the current location of the nurse). Another example is scoped negation. When a nurse redirects a call (e.g., the nurse indicates that he/she will not answer the call), another nurse needs to be assigned, using the same decision tree, but except the nurse who redirected the call. To make sure these decision trees can be configured easily, we require a lot of expressiveness of the rule language. As such, the combination of the N3 language with the EYE reasoner seems a good fit for the task at hand, even though the EYE reasoner is not optimized to support event-based data. Current stream reasoners are not fit for the task as their expressiveness falls short. Also, the throughput of events is a lot slower than for stream reasoners, in the order of multiple events per minute, instead of multiple events per second.

The most important drawback of the EYE reasoner is that it is a file-based reasoner. EYE processes all necessary ontologies, data, and rule files, and uses a query file that filters the output of the reasoning result. This query file allows us to direct the reasoning process to only return the relevant results. For example, when we reason about the state change of a nurse (e.g., from *free* to *with patient*), the reasoner only returns the status change. This avoids unwanted conflicting states. It accepts data from local files, as well as from remote files (i.e., using URIs). For supporting the necessary OWL constructs, we use already defined OWL-RL rules[9], however, more complex constructs are also supported. The fact that the EYE reasoner is a single process complicates the implementation of an event-based rule system. Whereas most other reasoning engines can keep data in memory, the EYE reasoner needs to re-read and parse all data from file every time reasoning is requested. The larger the amount of data, the more time will be

[9] `http://www.w3.org/TR/owl2-profiles/#Reasoning_in_OWL_2_RL_and_RDF_Graphs_using_Rules`

wasted on this reading and parsing data from file (i.e., Disk I/O time). The EYE reasoner however can export the parsed data as Prolog bytecode, enabling us to skip the parsing step, and thus improving the Disk I/O time.

4 System Description

In Figure 1, the overall architecture of our system is shown. The static elements that the system comprises are elaborated on in Subsection 4.1. A dynamic view is discussed in Subsection 4.2. For each component, Figure 1 shows which elements are used (e.g., databases and/or rule files). The cog-icon shows how a component is mainly a coded component, whereas a document-icon shows how this component is mainly a reasoning component that works with a rule file. Two databases are used: a static and a dynamic database. The dynamic database is annotated with an asterisk (∗). Note that there is only one static and one dynamic database, and the replication of the icons in the figure is just to denote how each component makes use of which database. A grayed out element signifies that that element is being updated in that component, thus, this system separates between modules using the data (*Decider*) and modules changing the data (*PreUpdater* and *PostUpdater*). The numbers from `(0)` till `(6)` show the flow of the dynamic system. Number `(0)` only needs to be executed when the static data changes (e.g., when the ontology changes or when the system is installed in another hospital), numbers `(1)–(6)` are executed in order when an event comes in.

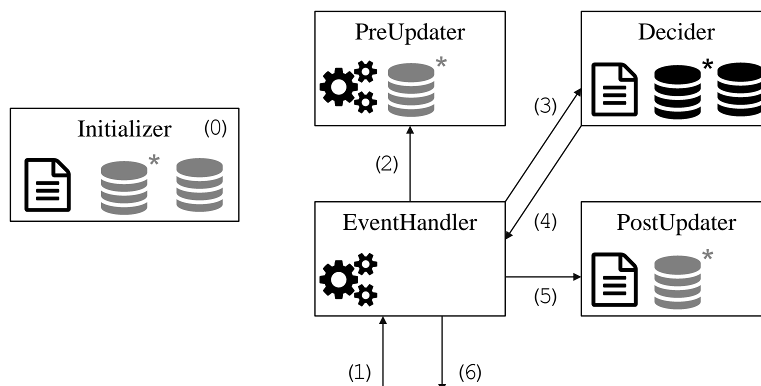

Fig. 1. The overall architecture of the system

4.1 Architecture

The devised system consists of five components, the *Initializer*, the *EventHandler*, the *PreUpdater*, the *Decider* and the *PostUpdater*.

Initializer The Initializer only executes once at initialization time. This component is responsible for the separation between static data and dynamic data, and to preprocess the static data. The rule file of this component thus only needs to change when the separation between static and dynamic data changes, or when the static data changes. This is typically the case when an extra (dynamic) field is added to the used ontology. Also note that the static data not only comprises the static data in the ABox, but also the entire TBox, and – in this case – the rules that are used by the ontology. The monotonicity of the N3 logic allows us to preprocess and reason about the static data without negative side-effects.

EventHandler The EventHandler is a customly written component for each individual use case. It handles the incoming events, and handles the output of the Decider. For example, after the Decider returns that *Nurse A* needs to be assigned to a certain patient call, the EventHandler needs to make the actual call to *Nurse A*, for example by pushing a notification to his or her smartphone.

PreUpdater The PreUpdater is a component that needs to check whether dynamic data needs to be updated due to the incoming event. For example, when a certain nurse changes his or her location, the internal state of the system needs to be updated accordingly. The processing in this component cannot be done via rules as an event can imply inconsistent states, e.g., when a nurse changes location, two conflicting locations are available in the PreUpdater, one from the nurse's previous state, and one from the event data.

Decider The Decider is the core reasoning component of the system. This component uses the information of the current state of the database(s), and the new information of the incoming event to infer new facts. At least a subset of the rules used by this decision component is configurable for specific environments.

PostUpdater The PostUpdater uses a rule file to update the current state of the dynamic data with the result of the Decider component and the handled event. By doing this using a reasoning process, and not hard-coded, we allow this component to remain configurable. As the EYE reasoner filters the output of the result, we can ask only for relevant results. For example, when a reasoning process changes the status of a nurse, the current statuses of the nurses are fed to the reasoning process, together with the event and the rule files, and the output is the updated statuses of the nurses. This way, conflicts are avoided.

4.2 Processing requirements

After the initialization phase `(0)`, the system has successfully split up the static data from the dynamic data and preprocessed the static data. In our case, this preprocessing means that the EYE reasoner converts the static N3 data into compiled Prolog, which means that all static data is already parsed and indexed when an event comes in.

When an event arrives `(1)`, the EventHandler first executes the PreUpdater `(2)` to make sure all dynamic data is first updated in the dynamic database. Hereafter, the Decider component is executed `(3)`, and the result is returned `(4)`. This is the only

result necessary to start executing the actual functions of the system (e.g., sending a notification to *Nurse A*), but is also necessary to update the dynamic database (e.g., assigning the status of *Nurse A* to *with patient*). After the dynamic database is updated by the PostUpdater `(5)`, and all actions are executed, the final result is returned (e.g., a success code), and the next event can be processed.

Multiple Initialized EYE instances There is a big downside with previously described architecture: due to the file-based nature of the EYE reasoner, every reasoning run involves reading in the static and dynamic data. This implies a huge overhead compared with the actual reasoning times. As a first step to improve on this problem, we observed that the EYE process has two ways of reading in data, i.e., using local files or via an HTTP connection. If multiple data files (or URIs) are used, the EYE process parses each of them in order.

When an EYE process reads data from an HTTP connection, it streams all data from the HTTP connection to a local temporary file, and halts until the connection is closed. After this connection is closed, the temporary file is read in, and the EYE service executes the reasoning run.

Thus, we can devise a methodology where we present the EYE process with two data inputs: a local file with the (large amount of) static data and a remote file with the (relatively small) dynamic data. We thus initialize an EYE process by letting it read all static data into memory, and afterwards block it on the HTTP connection. When an event comes in, only the dynamic data needs to be read into memory by the EYE process, thus gaining a great amount of Disk I/O time otherwise consumed by reading the static data into memory first.

Secondly, we can observe that this static data that is being read into memory on beforehand by definition does not change. Thus, initializing multiple EYE instances with this static data will not have any effect on the reasoning results, as long as the updated dynamic data is being consumed after a new event is triggered.

Streaming framework Intializing multiple EYE instances enables us to reason on streaming data using time windows. Being event-driven, this framework is already capable of reasoning on data streams. However, currently, the data has been summarized on beforehand and returned as events (e.g., not every location update of a nurse is triggered, only when a nurse changes room).

To use this framework for actual stream reasoning, we can pipe the data stream to the HTTP connections of all initialized EYE instances. If we have a pool of four EYE instances, and close off one HTTP connection every $500ms$, we effectively have created a streaming rule-based reasoning system with a time window of $4 \cdot 500ms = 2s$, where reasoning is executed every $500ms$. This implies that the reasoning results of concurrent instances should not conflict with each other. As at any given time, there are at least three out of four EYE instances running, it is not possible to feed the reasoning of the last finished EYE instance to these already running EYE instances.

5 Evaluation

We evaluated the proposed system in terms of scalability using simulated data, based on real-life situations, as deducted from user studies elaborated on in [11]. The simulated data consists of a ward in a hospital, and the data was scaled by increasing the amount of wards gradually to fill the ABox with more data. Afterwards, a use-case specific scenario was run 35 times, 3 of these runs being warm-up runs. All experiments were run on the same technology stack[10]. We ran the experiment with four variations, to compare between the effect of splitting up the static data with the dynamic data, and the effect of initializing multiple EYE instances concurrently. Four concurrent EYE processes were started, as the used hardware consisted of a CPU with four cores.

Figure 2 shows the execution times for the scalability evaluation. By gradually increasing the amount of wards, and thus the dynamic data, we see how the execution time of the current system linearly increases with the size of the ABox, and has a total execution time of about 5.5*s* for the largest dynamic database, i.e., fifty wards or about 50 000 triples in total. Of these 5.5*s*, about 0.6*s* is the actual decision making time. The remaining 4.9*s* is mostly Disk I/O, and updating time via reasoning. Splitting the data in static and dynamic parts improves the Disk I/O time for fifty wards with about 0.8*s*.

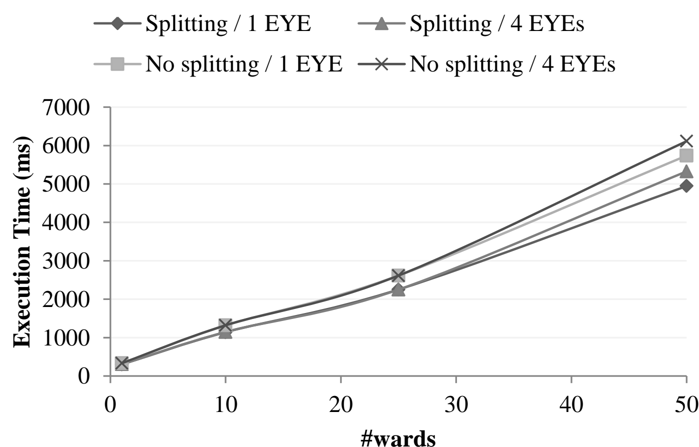

Fig. 2. Scaling the amount of hospital wards introduces a linear rise in execution time. We achieve a total execution times of about 5.5*s* for fifty wards. Splitting the data into static and dynamic parts improves the execution time with about 0.8*s*. Average reasoning times are consistently 11.33% of the total execution time, e.g., the actual decision making time for fifty wards takes about 0.6*s*.

When we investigate the timings of the individual components (averaged over all variations), we notice that the PostUpdater has a significant portion of the total execution

[10] Hardware: Intel(R) Xeon(R) E5620@2.40GHz CPU with 12 GB RAM. Software: Debian "Wheezy", EYE 7995 and SWI-Prolog 6.6.6

time: about $3.7s$, or 70% of the total execution time (as compared to about $0.1s$ execution time for the programmatic PreUpdater). About $3.2s$ of the PostUpdater execution time is reasoning time, the other $0.5s$ is Disk I/O time. This is certainly a point of improvement, as the large reasoning times of the PostUpdater compared to the reasoning times of the Decider (about $0.8s$) hint that the rule set of the PostUpdater could be optimized.

Taking a closer look into the reasoning times and Disk I/O times of the Decider (see Figure 3), we see how splitting up the static from the dynamic data does improve the Disk I/O times (from about $1.3s$ to about 0.5). We however also see that the reasoning times of EYE instances in a pool are larger than the reasoning times of the individually initialized EYE instances. This is probably due to the fact that concurrent EYE instances hog the CPU more, as there are more instances running concurrently.

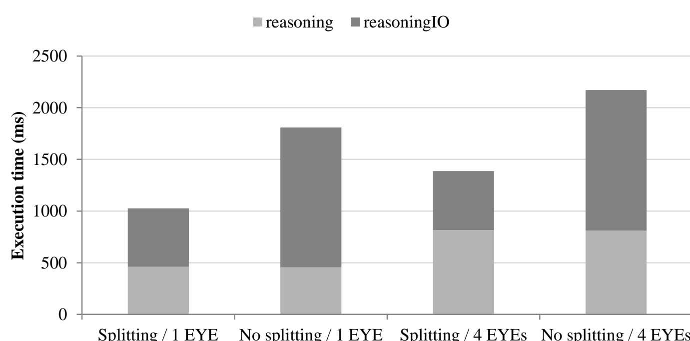

Fig. 3. Timings of the Decider for fifty wards show how splitting up the data positively impacts the Disk I/O time, but running multiple instances concurrently negatively impacts the reasoning time.

One could thus argue that splitting the data into static and dynamic parts, and only initializing one EYE instance is the fastest solution. This is the case when new events would only be triggered when all previous events are handled. Then, the single EYE instance would process the previous event and then immediately start reasoning about the next event. However, if multiple events could be triggered concurrently, they would block, and be executed in order by the single EYE instance. Thus, the handling of the first event would be finished after $5.5s$, the second event would be handled after $11s$, etc. Using multiple EYE instances solves this bursting behavior partially, as multiple events can be handled concurrently. However, the amount of concurrently initialized EYE instances limits the bursting capabilities of the system. For example, if a system with four concurrent EYE instances is initialized, and eight events are triggered at the same time, the first four events would be finished after $5.5s$, and the latter four events would be handled after $11s$. Initializing multiple concurrent EYE instances thus improves the bursting behavior of the proposed system, but its applicability is limited and depends on the rate of change of the environment.

6 Conclusion and Future Work

Reasoning applications have been successfully used for decision making problems. However, current solutions usually introduce a trade-off between expressiveness and performance. In this paper, we described a functional rule-based reasoning system using the N3 logic and the EYE reasoner. This system has been applied to a nurse call system. The system consists of an Initializer that separates static from dynamic data and preprocesses the static data to improve performance, a PreUpdater that programmatically updates the incoming event-data to avoid reasoning conflicts, a Decider that does the actual decision making, a PostUpdater that updates the dynamic database using the output of the Decider using a reasoning cycle, and an EventHandler to orchestrate all the components.

We can conclude that the analysis of the TBox and ABox can give valuable insights into constructing an event-driven system using rule-based reasoning. In our case, a large part of the ABox was in fact static data, and the large TBox is by definition static, so we could split up the data into a large piece of static data that can be optimized and preprocessed on beforehand, and a relatively small piece of dynamic data. By instantiating multiple reasoning instances concurrently, we degrade the performance slightly, but we gain a much better bursting behavior. Updating data by reasoning over updating rules is much slower than updating data programmatically, but allows for better (and in this case, necessary) configurability. We however have to conclude that although the current system does perform within reasonable times for the given use case with a small-scale data set, and supports a high level of expressitivity, its performance is very poor compared with the state of the art. The file-based nature of the EYE reasoner is contradictory to an event-based system. Optimizations such as splitting up static and dynamic data help, but adjustments to the core of the EYE reasoner are much needed to achieve comparable reasoning times. We however need to take into account that in this system, a lot of expressiveness is kept, and that the decision tree rules cannot be optimized, as they need to be easily configured.

Future work is improving the PostUpdater rule set to improve its reasoning time and improve the Disk I/O times by adjusting the EYE service to cope with in-memory data instead of only file-based data. The linear correlation between ABox data and reasoning time is promising to achieve good results for larger data sets. The strategy as explained in Subsection 4.2 could also be implemented to adapt this framework for continuous stream reasoning.

Acknowledgements The research activities described in this paper were funded by Ghent University, iMinds, the IWT Flanders, the FWO-Flanders, and the European Union, in the context of the project “ORCA”, which is a collaboration by Televic Healthcare, Internet-Based Communication Networks and Services (IBCN), and Multimedia Lab (MMLab).

References

1. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: Ep-sparql: a unified language for event processing and stream reasoning. In: Proceedings of the 20th international conference on World

Wide Web. pp. 635–644. ACM (2011), http://www.wwwconference.org/proceedings/www2011/proceedings/p635.pdf
2. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th international conference on World Wide Web. pp. 1061–1062. ACM, New York, USA (2009), http://dx.doi.org/10.1145/1526709.1526856
3. Beckett, D., Berners-Lee, T., Prud'hommeaux, E., Carothers, G.: Turtle - Terse RDF Triple Language. Tech. rep., World Wide Web Consortium (W3C) (Feb 2014), http://www.w3.org/TR/turtle/, accessed June 22nd, 2015
4. Bock, C., Fokoue, A., Haase, P., Hoekstra, R., Horrocks, I., Ruttenberg, A., Sattler, U., Smith, M.: OWL 2 Web Ontology Language. Tech. rep., World Wide Web Consortium (W3C) (Dec 2012), http://www.w3.org/TR/owl2-syntax/, accessed June 22nd, 2015
5. Della Valle, E., Ceri, S., Van Harmelen, F., Fensel, D.: It's a streaming world! reasoning upon rapidly changing information. IEEE Intelligent Systems 24(6), 83–89 (Nov 2009), http://doi.ieeecomputersociety.org/10.1109/MIS.2009.125
6. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co., Greenwich, CT, USA, 1st edn. (2010)
7. Georgeff, M.P., Ingrand, F.F.: Decision-making in an embedded reasoning system. In: Proceedings of the 11th Internation Joint Conferences on Artifical Intelligence (IJCAI-89). Australian Artificial Intelligence Institute, Detroit, Michigan, USA (1989), http://ijcai.org/Past%20Proceedings/IJCAI-89-VOL-2/PDF/020.pdf
8. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge acquisition 5(2), 199–220 (Jun 1993), http://www.sciencedirect.com/science/article/pii/S1042814383710083
9. Kazakov, Y., Krötzsch, M., Simančík, F.: The incredible ELK – from polynomial procedures to efficient reasoning with EL ontologies. Journal of Automated Reasoning 63(1), 1–61 (Jun 2014), http://link.springer.com/article/10.1007/s10817-013-9296-3
10. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In: Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence. vol. 1, pp. 129–137. Association for the Advancement of Artificial Intelligence (AAAI), The AAAI Press, Québec, Canada (Jul 2014), https://krr-nas.cs.ox.ac.uk/2014/AAAI/RDFox/paper.pdf
11. Ongenae, F., Bleumers, L., Sulmon, N., Verstraete, M., Van Gils, M., Jacobs, A., De Zutter, S., Verhoeve, P., Ackaert, A., De Turck, F.: Participatory design of a continuous care ontology: towards a user-driven ontology engineering methodology. In: Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD). Paris, France (October 2011)
12. Verborgh, R., De Roo, J.: Drawing Conclusions from Linked Data on the Web: The EYE Reasoner. IEEE Software 32(5), 23–27 (May 2015), http://online.qmags.com/ISW0515?cid=3244717&eid=19361&pg=25