

Article

Plausible Description Logic Programs for Stream Reasoning

Adrian Groza * and Ioan Alfred Letia

Department of Computer Science, Technical University of Cluj-Napoca, Memorandumului 28, Cluj-Napoca 400391, Romania; E-Mail: letia@cs-gw.utcluj.ro

* Author to whom correspondence should be addressed; adrian.groza@cs.utcluj.ro;
Tel.: +40-264-401-446.

Received: 9 August 2012; in revised form: 9 September 2012 / Accepted: 25 September 2012 /
Published: 17 October 2012

Abstract: Sensor networks are estimated to drive the formation of the future Internet, with stream reasoning responsible for analysing sensor data. Stream reasoning is defined as real time logical reasoning on large, noisy, heterogeneous data streams, aiming to support the decision process of large numbers of concurrent querying agents. In this research we exploited non-monotonic rule-based systems for handling inconsistent or incomplete information and also ontologies to deal with heterogeneity. Data is aggregated from distributed streams in real time and plausible rules fire when new data is available. The advantages of lazy evaluation on data streams were investigated in this study, with the help of a prototype developed in Haskell.

Keywords: stream reasoning; description logic; plausible logic; lazy evaluation; sensors; Haskell

1. Introduction

Sensor networks are gradually becoming ubiquitous in the industrial world [1], whilst they are estimated to drive the formation of the future Internet, by 2015 [2]. The value of the Sensor Web is related to the capacity to aggregate, analyse and interpret this new source of knowledge. Currently, there is a lack of systems designed to manage rapidly changing information at the semantic level [3]. The solution given by data-stream management systems (DSMS) is limited mainly by the incapacity to perform complex reasoning tasks, and semantic technologies are seen as the main technical instrumentation available to deal with the current problems [4].

Stream reasoning is defined as real time logical reasoning on huge, possibly infinite, noisy data streams, aiming to support the decision process of large numbers of concurrent querying agents. In order to handle blocking operators on infinite streams (like min, mean, average, sort), the reasoning process is restricted to a certain window of concern within the stream, whilst the previous information is discarded [5]. This strategy is applicable only for situations where recent data have higher relevance (e.g., average water debit in the last 10 minutes). In some reasoning tasks, tuples in different streams that are far apart need to be joined arbitrarily.

Stream reasoning adopts the continuous processing model, where reasoning goals are continuously evaluated against a dynamic knowledge base. This leads to the concept of transient queries, opposite to the persistent queries in a database.

Typical applications of stream reasoning are: traffic monitoring, urban computing, patient monitoring, weather monitoring from satellite data, monitoring financial transactions [3] or stock market. Real time events analysis is conducted in domains like seismic incidents, flu outbreaks, or tsunami alert based on a wide range of sensor networks starting from the RFID (Radio-Frequency Identification) technology to the Twitter data flow [6]. Decisions should be taken based on plausible events. Waiting to have complete confirmation of an event might be too risky.

To summarise, the problem is given by the inability of the current stream reasoning systems (i) to perform semantic reasoning on changing information from heterogeneous sensors and (ii) to support decision in case of incomplete information.

Our solution is based on three technical components: description logic (DL), plausible logic (PL) and functional programming. Description logic is used to deal with heterogeneity and to fill the gap between low level sensor data and high level knowledge requested by decision takers. Plausible Logic, being non-monotonic, is used for handling incomplete information and its explicit priorities are used to solve inconsistencies. The Haskell data structures are used to model infinite streams based on the lazy evaluation mechanism of Haskell. Also, functional programming is used to process streams on the fly according to given aggregation policies.

Note that in the current approach, the rule-based system is not on top of the ontology level, as envisaged by the Semantic Web stack. Instead, they are at the same level with part of the ontology being translated into strict rules. Consequently, after the translation, all the reasoning on streams is performed within the plausible logic framework. The main advantage rests in the time needed, given the superior efficiency of plausible logic [7] compared to description logic [8].

This paper is an extended version of [9]. The first contribution regards the integration of plausible rules with description logic. As a second contribution, we describe the use of Haskell data structures and the lazy evaluation mechanism for continuous reasoning on infinite streams.

We exploit non-monotonic rule-based systems for handling inconsistent or incomplete information and also ontologies to deal with heterogeneity. Data is aggregated from distributed streams in real time, and plausible rules fire when new data is available. This study investigates the advantages of lazy evaluation on data streams as well.

Section 2 evidentiates distinctive features of our approach compared to related work. Section 3 introduces the technical instrumentation, which is based on plausible logic and description logic.

Section 4 presents the developed stream management system. A running scenario is illustrated in Section 5. Finally, Section 6 summarizes the advantages and possible improvements of our solution.

2. Related Work

Stream integration is considered an ongoing challenge for the stream management systems [2,3,10,11], with several tools available to perform stream reasoning.

DyKnow [12] introduces the knowledge processing language KPL to specify knowledge processing applications on streams. We exploit the Haskell stream operators to handle streams and the list comprehension for querying these streams. The SPARQL algebra is extended in [13] with time windows and pattern matching for stream processing. In our approach we exploit the existing list comprehension and pattern matching in Haskell, aiming at the same goal of RDF streams processing. Comparing with C-SPARQL, Haskell provides capabilities to aggregate streams before performing queries against them. Etalis tool performs reasoning tasks over streaming events with respect to background knowledge encapsulated as Prolog rules [14]. In our case, the background knowledge is obtained from ontologies, translated as strict rules in order to reason over a unified space.

The need to consider revision in stream processing is also addressed in [15]. A rule-based algorithm is developed to handle different situations in which event revision should be activated. In our case, the defeasible semantics of plausible logic is enacted to block the derivation of complex events in case of new contradictory information. Consequently, the knowledge engineer would be responsible to define what a plausible event is, and what consequences cannot be retracted even if its premises have been proved false after revision, by modelling them as strict rules. One possible line of research, which harmonises these two complementary approaches, would be to develop a logic for stream processing.

The strength of plausibility of the consequents is given by the superiority relation among rules. One idea of computing the degree of plausibility is to exploit specific plausible reasoning patterns like *epagoge*: “If A is true, then B is true. B is true. Therefore, A becomes more plausible”, “If A is true, then B is true. A is false. Therefore, B becomes less plausible”, or “If A is true, then B becomes more plausible. B is true. Therefore, A becomes more plausible.”

The research conducted here can be integrated into the larger context of Semantic Sensor Web, where challenges like abstraction level, data fusion, application development [16] are addressed by several research projects like Aspire [17] or Sensei [18]. By encapsulating domain knowledge as description logic programs, the level of abstraction can be adapted for the current application by importing a more refined ontology into DLP. From a different perspective, the ontology used for stream processing acts as a summator [19]: instead of storing all the input data, the incoming items are classified and only this abstract knowledge is stored as instances. One advantage is that the system facilitates business intelligence through the set of semantic queries that can be addressed against the classified knowledge. Examples of such queries are “how many dairy products have been sold since yesterday” or “what quantity of man’s wear was sold in the last month”, where the level of refinement is given by the exploited ontology.

3. Integrating Plausible Rules with Ontologies

This section starts with the formalisation of plausible logic responsible for performing reasoning on streams. Then, it introduces description logic for modelling ontologies. Finally, it formalises the translation of ontologies into strict rules in PL. We consider that sensor data are available as instances of a sensor ontology.

3.1. Plausible Logic

As a non-monotonic formalism, plausible logic (PL) deals with the problem of deriving conclusions in case of incomplete or uncertain information. Plausible logic is considered an improvement of defeasible logic [20,21].

A clause in PL $\vee a_1, a_2, \dots, a_n$ is the disjunction of positive or negative atoms a_i . If both an atom and its negation appear, the clause is a tautology. A *contingent* clause is a clause which is neither empty nor a tautology [20].

Definition 1 A plausible description of a situation is a tuple $PD = (Ax, R_p, R_d, \succ)$, where Ax is a set of contingent clauses, called axioms, characterising the aspects of the situation that are certain, R_p is a set of plausible rules, R_d is a set of defeater rules, and \succ is a priority relation on $R_p \cup R_d$.

A plausible theory is computed from a plausible description by deriving the set R_s of strict rules from the definite facts Ax . Thus, a plausible knowledge base consists of strict rules (\rightarrow), plausible rules (\Rightarrow), defeater (warning) rules (\dashv), and a priority relation on the rules (\succ). Strict rules are rules in the classical sense (that is, whenever the premises are indisputable), then so is the conclusion. An atomic fact is represented by a strict rule with an empty antecedent. The plausible rule $a_i \Rightarrow c$ means that if all the antecedents a_i are proved and all the evidence against the consequent c has been defeated, then c can be deduced. The plausible conclusion c can be defeated by contrary evidence.

The only use of defeaters is to prevent some conclusions, as in “if the buyer is a regular one and he has a short delay for paying, we might not ask for penalties”. This rule does not provide sufficient evidence to support a “non-penalty” conclusion, but it is strong enough to prevent the derivation of the penalty consequent. The priority relation \succ allows the representation of preferences among non-strict rules.

Decisive plausible logic consists of a plausible theory and a proof function $P(\lambda f,)$ that, given a proof algorithm λ and a formula f in conjunctive normal form, returns +1 if λf was proved, -1 if there is no proof for λf , or 0 when λf is undecidable due to looping. Plausible Logic has five proof algorithms $\{\mu, \alpha, \pi, \beta, \gamma\}$, in which one is monotonic and four are non-monotonic: μ monotonic, strict, like classical logic; $\alpha = \beta \wedge \pi$; π plausible, propagating-ambiguity; β plausible, blocking-ambiguity; and $\gamma = \pi \vee \beta$.

Example 1 Consider the following rules for triggering an alarm system in case the temperature rises above 0°C in a refrigerator room.

$r_1 : \text{sensor}(x), \text{measures}(x, y), y > 4 \rightarrow \text{highTemperatureAlarm}.$

$r_2 : \text{sensor}(x), \text{measures}(x, y), y > 0 \Rightarrow \text{highTemperatureAlarm}.$

$r_3 : \text{inside}(x) \Rightarrow \neg \text{highTemperatureAlarm}.$

$r_4 : \text{opendoor} \dashv \neg \text{highTemperatureAlarm}.$

$r_4 > r_3$

The rule r_1 is a strict one: whenever the temperature y measured by the sensor x is above 4 °C the alarm starts automatically. The rule r_2 is a plausible one: if the measured temperature is above 0 °C, there is a reason to support the decision to trigger the alarm. The algorithm analyses if there are other reasons supporting the opposite conclusion or trying to defeat it. Rule r_3 is such a rule. If both rules can be fired, we have an ambiguity. In case of the propagating-ambiguity strategy, both consequences are derived by the system. In case of the blocking-ambiguity strategy, no conclusion would be valid. Note that the rule r_4 is a defeater that, in case of activation, blocks the derivation of the conclusion $\neg highTemperatureAlarm$. If active, r_4 being stronger than r_3 is able to block the corresponding decision. At this moment the consequence of the rule r_2 is no longer attacked by any valid rule. This mechanism corresponds to the argumentative semantics characterising the family of the defeasible logics [22].

3.2. Description Logic

In the description logic ALC , concepts are built using the set of constructors formed by negation, conjunction, disjunction, value restriction, and existential restriction, as Table 1 bears out [23]. Here, C and D represent concept descriptions, whilst r and s role names. In this study we used the extension of ALC with transitivity (R_+) on roles and role hierarchy (H), known as $ALCR_+H$ or SH . The syntax of SH is defined by the following grammar:

$$C, D ::= \top | A | \neg C | C \sqcap D | \exists r.C | \forall r.C | r_+ | r \sqsubseteq s \tag{1}$$

where A represents an atomic concept and \top the top level concept.

The semantics is defined based on an interpretation $I = (\Delta^I, \cdot^I)$, where the domain Δ^I of I contains a non-empty set of individuals, whilst the interpretation function \cdot^I maps each concept name C to a set of individuals $C^I \subseteq \Delta^I$ and each role r to a binary relation $r^I \subseteq \Delta^I \times \Delta^I$. The second column of Table 1 illustrates the extension of \cdot^I to arbitrary concepts.

Table 1. Syntax and Semantics of SH concepts.

Syntax	Semantics
$\neg C$	$\Delta^I \setminus C^I$
$C \sqcap D$	$C^I \cap D^I$
$C \sqcup D$	$C^I \cup D^I$
$\exists r.C$	$\{x \in \Delta^I \exists y : (x, y) \in r^I \wedge y \in C^I\}$
$\forall r.C$	$\{x \in \Delta^I \forall y : (x, y) \in r^I \rightarrow y \in C^I\}$
$r \sqsubseteq s$	$\{x \in \Delta^I \forall y : (x, y) \in r^I \rightarrow (x, y) \in s^I\}$
r_+	$\{x \in \Delta^I \forall y, z : (x, y) \in r^I \text{ and } (y, z) \in r^I \rightarrow (x, z) \in r^I\}$

Definition 2 A concept C is satisfiable if there exists an interpretation I such that $C^I \neq \emptyset$. The concept D subsumes the concept C ($C \sqsubseteq D$) if $C^I \subseteq D^I$ for all interpretations I .

Definition 3 An *ABox* is a finite set of concept assertions $a : C$ or role assertions $(a, b) : r$, where C represents a concept, r a role, and a and b are two instances. Usually, the unique name assumption holds within the same *ABox*. A *TBox* is a finite set of terminological axioms of the form $C \equiv D$ or $C \sqsubseteq D$.

Example 2 Consider the domain $\Delta = \{s_1, 0.1^\circ C\}$. Let the terminology:

$$\begin{aligned} TBox &= \{Sensor \sqsubseteq \exists hasAccuracy.MeasureUnit, WirelessSensor \sqsubseteq Sensor\} \\ ABox &= \{s_1 : WirelessSensor, 0.1^\circ C : MeasureUnit, (s_1, 0.1^\circ C) : hasAccuracy\} \end{aligned}$$

The interpretation function treats the individual s_1 as a wireless sensor with the measurement accuracy of $0.1^\circ C$, where the value is interpreted as an instance of the class *MeasureUnit*. Based on the second axiom in the *TBox*, the subsumption reasoning service of the DL derives s_1 as an instance of the *Sensor* concept.

3.3. Translating from DL to Plausible Logic Programs

This section exploits the work in [24] in order to translate description logic based on ontologies into plausible logic axioms. Facing the challenge to reason on huge amount of noise and heterogeneous data, the DL fragment corresponding to Horn clauses, known as Description Logic Programs [25], can be a suitable choice.

Conjunctions and universal restrictions in the right hand side of inclusion axioms are converted into rule heads (T_h classes), whilst conjunction, disjunction and existential restriction appearing in the left-hand side are translated into rule bodies (T_b classes). Figure 1 presents the mapping function \mathcal{T} from DL to strict rules in a plausible knowledge base, where A, C and D are concepts such that $A, C \in T_b$, $D \in T_h$, A is an atomic concept, X, Y and Z are variables, and P and Q are roles.

Figure 1. Mapping from DL ontologies into strict rules.

$$\begin{aligned} \mathcal{T}(C \sqsubseteq D) &= T_b(C, X) \rightarrow T_h(D, X) \\ \mathcal{T}(\top \sqsubseteq \forall P.D) &= P(X, Y) \rightarrow T_h(D, Y) \\ \mathcal{T}(\top \sqsubseteq \forall P^-.D) &= P(X, Y) \rightarrow T_h(D, X) \\ \mathcal{T}(a : D) &= T_h(D, a) \\ \mathcal{T}((a, b) : P) &= P(a, b) \\ \mathcal{T}(P \sqsubseteq Q) &= P(X, Y) \rightarrow Q(X, Y) \\ \mathcal{T}(P^+ \sqsubseteq P) &= P(X, Y) \wedge P(Y, Z) \\ &\rightarrow P(X, Z) \end{aligned}$$

where

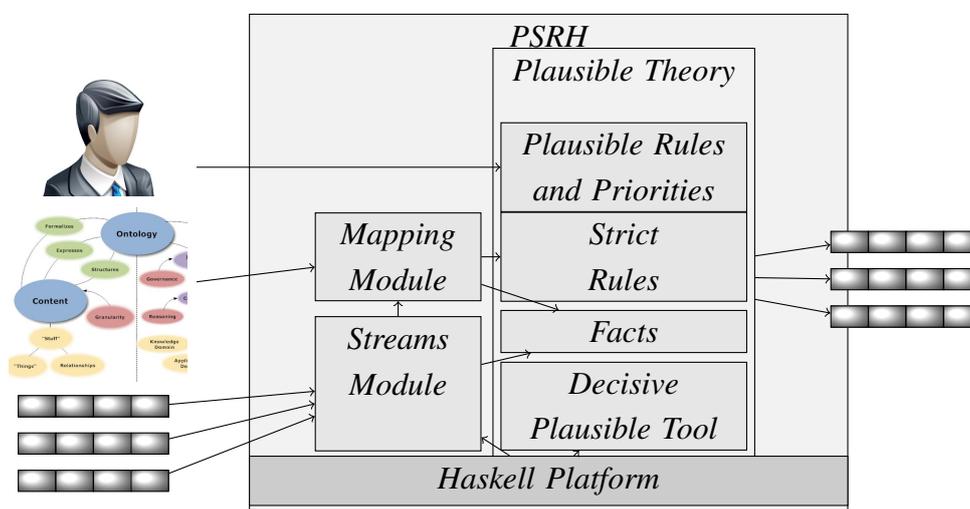
$$\begin{aligned} T_h(A, X) &= A(X) \\ T_h(C \sqcap D, X) &= T_h(C, X) \wedge T_h(D, X) \\ T_h(\forall R.C) &= R(X, Y) \rightarrow T_h(C, Y) \\ T_b(A, X) &= A(X) \\ T_b(C \sqcap D, X) &= T_b(C, X) \wedge T_b(D, X) \\ T_b(C \sqcup D, X) &= T_b(C, X) \vee T_b(D, X) \\ T_h(\exists R.C) &= R(X, Y) \rightarrow T_b(C, Y) \end{aligned}$$

If C is subsumed by D , then each individual X in C is also an instance of the class D . The axiom $\top \sqsubseteq \forall P.D$ says that, with certainty, all the roles P point towards individuals of type D . It is translated in second line of Figure 1 as: if there is a relation P between X and Y , the element Y should be of type D . Similar semantics is applied for the inverse role P^- , where the domain and range are interchanged. If a is interpreted as a D concept, then a is of type D . The role assertion $(a, b) : P$ is interpreted as the two individuals a and b being in relation P . The mapping of role subsumption $P \sqsubseteq Q$ outputs that given two elements X and Y in relation P , they are linked by the more general relation Q . The transitivity property of roles $P^+ \sqsubseteq P$ says that if the instance X is related to Y by the role P and if Y is related to Z by the same role, then it is necessary that X is related to Z by the same role P .

4. Data Stream Management System in Haskell

This section details the architecture of the PSRH (Plausible Stream Reasoning in Haskell) system (Figure 2). For each problem, a domain expert is responsible to define the priorities and the plausible rules in order to handle contradictory data. The mapping module translates the available ontologies into facts and strict rules. The stream module provides a collection of functions for manipulating the input data streams. The relevant sensor-based measurements are stored as facts in the plausible theory. The decisive plausible tool is continuously queried in order to support plausible decisions in real-time. Each module described in the following paragraphs is built on top of the Haskell platform.

Figure 2. The architecture of the PSRH (Plausible Stream Reasoning in Haskell).



4.1. The Haskell Platform

The advantages that Haskell brings in this landscape—lazy evaluation and implicit parallelism—are significant features when dealing with huge data streams that are parallel in nature. The parallel performing of reasoning tasks is of significant importance in order to provide answers in due time [3]. The Haskell’s polymorphism allows to write generic code to process streams, which is particularly useful due to the different usages of the same data stream. The absence of side effects means that the

order of expression evaluation is of no importance, which is extremely desirable in the context of data streams coming from different sources.

One challenge in answering many continuous queries in real-time is query optimisation. Allowing equational reasoning, the platform can be exploited for automatic program and optimising queries. A premise specified as lazy is matched only when its variables are anticipating to participate in the answer to a pending conclusion or query.

The continuous semantics of data streams assumes that: (i) *streams are volatile*—they are consumed on the fly and not stored forever; and (ii) *continuous processing*—queries are registered and produce answers continuously [5]. In our case, the rules are triggered continuously in order to produce streams of consequents. The stateless feature of pure Haskell facilitates the conceptual model of networks of stream reasoners as envisaged in [26], where data is processed on the fly, without being stored.

The lazy evaluation in Haskell provides answers perpetually, when the queries are executed against infinite streams. One does not have to specify the time steps when the query should be executed. By default, the tuples are consumed when they become available, and only in case they contribute to a query answer.

The computational efficiency is supported by the fact that a function is not forced to wait for a data to arrive—the possible computations are executed instead. Moreover, one can use the about-to-come data by borrowing it from the future, as long as no function tries to change its value. The non-strict semantics of Haskell allows the functions not to produce errors that can be avoided. Consequently, some noise data can be avoided, without disturbing the computations.

There is no constraint on the nature of data fed by a stream. The functions can be applied on RDF streams as follows: A triple object is created by the *triple* function

```
data Triple = triple !Node !Node !Node
triple      :: Subject -> Predicate -> Object -> Triple
```

Definition 4 An RDF stream is an infinite list of tuples of the form $\langle subj, pred, obj \rangle$ annotated with their time stamps τ .

$$type RDFStream = [((subj, pred, obj), \tau)]$$

Example 3 An RDF stream of auction bids comprises the bidder agent, its action (sell or buy), and the bid value:

$$[(a_1, sell, 30\$), 14.32), ((a_2, sell, 28\$), 14.34), ((a_3, buy, 26\$), 14.35), ((a_1, sell, 27\$), 14.36)]$$

In the first tuple the agent a_1 wants to sell a specific item for 30\$, the offer being made at the time step 14.32. The stream is implemented with the infinite list data structure in Haskell.

4.2. Streams Module

Table 2 illustrates the operators provided by Haskell to manipulate infinite streams. The basic operators allow to construct infinite streams or to extract elements from the input stream. The high order functions *map*, *inter*, *scan*, and *transp* allow to apply different transformations on the input streams.

Table 2. Stream operators in Haskell (*S* stands for the *Stream* datatype).

Type	Function	Signature
Basic	constructor	$\langle : \rangle :: a \rightarrow S a \rightarrow S a$
	extract the first element	$head :: S a \rightarrow a$
	extracts the sequence following the stream's head	$tail :: S a \rightarrow S a$
	takes a stream and returns all its finite prefixes	$inits :: S a \rightarrow S ([a])$
	takes a stream and returns all its suffixes	$tails :: S a \rightarrow S (S a)$
Transformation	applies a function over all elements	$map :: (a \rightarrow b) \rightarrow S a \rightarrow S b$
	interleaves 2 streams	$inter :: Stream a \rightarrow Stream a \rightarrow S a$
	yields a stream of successive reduced values	$scan :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow S b \rightarrow S a$
	computes the transposition of a stream of streams	$transp :: S (S a) \rightarrow S (S a)$
Building streams	repeated applications of a function	$iterate :: (a \rightarrow a) \rightarrow a \rightarrow S a$
	constant streams	$repeat :: a \rightarrow S a$
	returns the infinite repetition of a set of values	$cycle :: [a] \rightarrow S a$
Extracting sublists	takes the first elements	$take :: Int \rightarrow S a \rightarrow [a]$
	drops the first elements	$drop :: Int \rightarrow S a \rightarrow S a$
	returns the longest prefix for which p holds	$takeWhile :: (a \rightarrow Bool) \rightarrow S a \rightarrow [a]$
	returns the suffix remaining after takeWhile	$dropWhile :: (a \rightarrow Bool) \rightarrow S a \rightarrow S a$
	removes elements that do not satisfy p	$filter :: a \rightarrow Bool \rightarrow S a \rightarrow S a$
Index	returns the element of the stream at index n	$!! :: S a \rightarrow Int \rightarrow a$
	returns the index of the first element equal with a	$elemIndex :: Eq a \Rightarrow a \rightarrow S a \rightarrow Int$
	returns the index of the first element satisfying p	$findIndex :: (a \rightarrow Bool) \rightarrow S a \rightarrow Int$
Aggregation	returns a list of corresponding pairs from 2 streams	$zip :: S a \rightarrow S b \rightarrow S (a,b)$
	combines two streams based on a given function	$ZipWith :: (a \rightarrow b \rightarrow c) \rightarrow S a \rightarrow S b \rightarrow S c$

For instance, the *map* function converts each element within a stream, according to a given conversion function. The conversion takes place when the converted values are required by other function. Based on a given operation, the *scan* function aggregates the elements within a stream into a single value. For computing at each step the sum of a stream of transactional data, the following expression can be used:

$$scan + 0 [2, 4, 5, 3, \dots]$$

It provides as output the infinite stream $[0, 2, 6, 11, 14, \dots]$, where the current value sums all the previously ones.

The aggregation functions combine elements from two input streams in order to generate a continuous output stream. Consider one wants to add the corresponding values from two financial data streams s_1 and s_2 , expressed by two different currencies:

$$zipWith + s_1 (map conversion s_2)$$

where the conversion function is applied on each element from s_2 .

The aggregation of two streams takes place according to an aggregation policy, depending on the time or the configuration of the new tuples. The policy is a function provided as the first input argument for the high order function *zipWith*, which has three input parameters and one output:

$$zipWith policy inStream1 inStream2 outStream$$

The elements of the input streams are combined according to the *policy*. The *zipWith* function continuously produces the output stream *outStream* with the elements from the input streams *inStream1* and *inStream2* aggregated based on the given policy. Similarly, generating new streams can be done based on a policy *f*, as in: *iterate f x = [x, fx, f(fx), ...]*. An incoming stream can be dynamically split into two streams, based on a predicate *p*.

4.3. The Mapping Module

Two sources of knowledge are exploited to reason on data collected by the sensors. On the one hand, one needs detailed information about sensors, measurements domain and units, or accuracy. On the other hand, domain specific axioms are exploited when reasoning on a specific scenario.

A partial view of the sensor ontology is formalised in DL in Figure 3. Figure 4 graphically illustrates the *TBox* and *ABox* of the ontology. The sensor s_1 is an instance of the class *ActiveRDF* and it measures temperature with an accuracy of 0.5 °C. The current temperature is 6 °C, and the measurement frequency is six observations per minute (Figure 4). Noting that *Temperature* is a *PhysicalQuality* (axiom 9 in Figure 3), there is a role *measure* between the sensor s_1 and the temperature value 6°C, as axiom 1 defines. The corresponding RDF stream for the sensor s_1 looks like:

$$[((s_1, \text{measures}, 4^\circ\text{C}), 8.30'10''), ((s_1, \text{measures}, 5^\circ\text{C}), 8.30'20''), ((s_1, \text{measures}, 6^\circ\text{C}), 8.30'30'')]$$

where at each 10 seconds the measured value increases with one degree Celsius, from 4 °C to 6 °C.

The ontology is translated into strict rules based on the conceptual instrumentation introduced in Section 3.3, in order to reason only within plausible logic. The resulted strict rules and facts appear in Figure 5. Observe that the terminology is considered static, whilst assertions may vary in time in Figure 5.

Rapid developments of the sensor technology raises the problem of continuously updating the sensor ontology. The system is able to handle this situation by treating the ontology as a stream of description logic axioms. When applying the high order function *map* on the transformation function \mathcal{T} , each axiom in the description logic is converted to strict rules as soon as it appears:

$$\text{map } \mathcal{T} [A \sqsubseteq B, C \sqsubseteq \forall r.D, \dots]$$

outputs the infinite list:

$$[r_1 : A(X) \rightarrow B(X), r_2 : C(X), r(X, Y) \rightarrow D(Y), \dots]$$

The main advantage consists in the possibility to dynamically include new background knowledge in the system.

Figure 3. Partial view of the sensor ontology.

<i>TBox</i> (assumed static)	1:	<i>Sensor</i>	$\sqsubseteq \forall \text{measure. PhysicalQuality}$
	2:	<i>Sensor</i>	$\sqsubseteq \forall \text{hasLatency. Time}$
	3:	<i>Sensor</i>	$\sqsubseteq \forall \text{hasLocation. Location}$
	4:	<i>Sensor</i>	$\sqsubseteq \forall \text{hasFrequency. Frequency}$
	5:	<i>Sensor</i>	$\sqsubseteq \forall \text{hasAccuracy. MeasureUnit}$
	6:	<i>WirelessSensor</i>	$\sqsubseteq \text{Sensor}$
	7:	<i>RFIDSensor</i>	$\sqsubseteq \text{WirelessSensor}$
	8:	<i>ActiveRFID</i>	$\sqsubseteq \text{RFIDSensor}$
	9:	<i>Temperature</i>	$\sqsubseteq \text{PhysicalQuality}$
<i>ABox</i> (at time 8.30'30")	10:	$s_1 : \text{Sensor}$	
	11:	$(s_1, 6^\circ C) : \text{measures}$	

Figure 4. Graphical view of the sensor ontology.

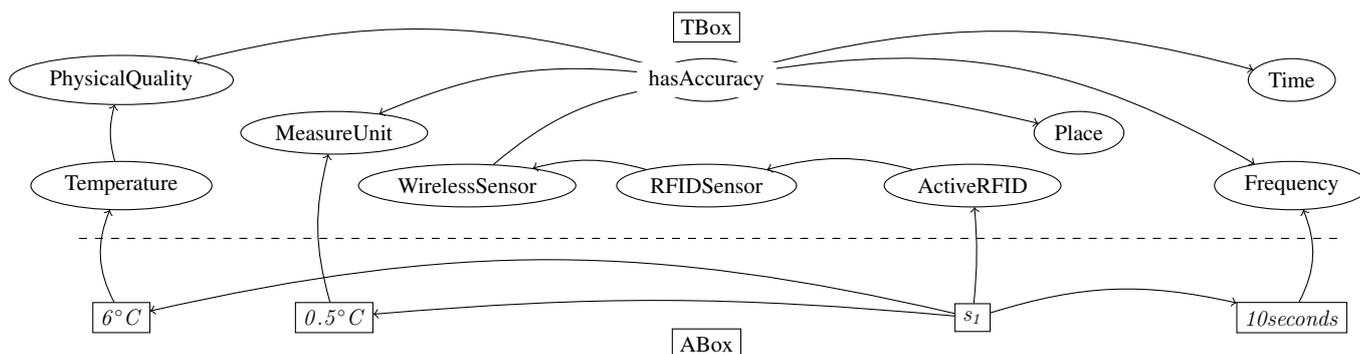


Figure 5. Translating the sensor ontology.

<i>Strict Rules</i>	1':	$\text{Sensor}(X), \text{Measures}(X, Y)$	$\rightarrow \text{PhysicalQuality}(Y)$
	2':	$\text{Sensor}(X), \text{HasLatency}(X, Y)$	$\rightarrow \text{Time}(Y)$
	3':	$\text{Sensor}(X), \text{HasLocation}(X, Y)$	$\rightarrow \text{Location}(Y)$
	4':	$\text{Sensor}(X), \text{HasFrequency}(X, Y)$	$\rightarrow \text{Frequency}(Y)$
	5':	$\text{Sensor}(X), \text{HasAccuracy}(X, Y)$	$\rightarrow \text{MeasureUnit}(Y)$
	6':	$\text{WirelessSensor}(X)$	$\rightarrow \text{Sensor}(X)$
	7':	$\text{RFIDSensor}(X)$	$\rightarrow \text{WirelessSensor}(X)$
	8':	$\text{ActiveRFID}(X)$	$\rightarrow \text{WirelessSensor}(X)$
	9':	$\text{Temperature}(X)$	$\rightarrow \text{PhysicalQuality}(X)$
<i>Facts</i> (at time 8.30'30")	10':	$\text{sensor}(s_1)$	
	11':	$\text{measures}(s_1, 6^\circ C)$	

4.4. Efficiency Remarks

The system incorporates the Decisive Plausible Logic tool [27] A Haskell glue module that exports functions requesting proofs [20] is used to make the connection with the other modules. The efficiency is mandatory when one needs to reason on huge data in real time. The efficiency of the proposed solution is based on the following vectors:

- The implementation of a family of defeasible logic is polynomial [7]. Plausible logic, being a particular case of defeasible reasoning, belongs to this efficiency class. The extensive experiments in [7] have proved that the family of defeasible reasoning tools that we integrated in our framework can deal with knowledge bases up to hundreds of thousands of rules, with a theoretical complexity of $O(N \log N)$. The cpu time for proving a conclusion was 0.4 seconds in case of 4200 contradictory rules, 3.83 seconds for 42,000 rules, and 21.15 seconds for 210,000 rules. This cpu time includes the time consumed by the garbage collector in Haskell [7].
- Under the assumption that the domain is static, the translation from description logic axioms into strict rules can be performed offline, before taking real-time decisions. The resulted Description Logic Programs are sub-fragments of Horn logics and their complexity is polynomial, as reported in [8].
- Only few general ontologies and few medical ontologies have this size. Most of the domain ontologies do not reach the comparable size of 4200 axioms. Depending on the time constraints for the given problem, a smaller ontology can be imported instead of a refined one.
- The plausible rules and preferences added by the human expert also do not go up to thousands of rules. Given the experience from the expert systems domain, few of the existing commercial expert systems do contain thousands of rules [28].
- The possibility to select the current inference algorithm among $\{\mu, \alpha, \pi, \beta, \gamma\}$ can also be exploited to adjust the reasoning task to the complexity of the current real-time decision.

5. Running Scenario

The scenario regards supporting real-time supply chain decisions based on RFID streams. Consider the stock management of an online shop. RFID sensors are used to count the items entering the warehouse from two locations. The items leave the warehouse from exit points, corresponding to three output streams. Monitoring an item like *Milk* implies monitoring several subcategories as *WholeMilk* and *LowFatMilk*. The retailer sells a specific item f_{m_1} of whole milk, and two types of low fat milk sm_1 and sm_2 . Some peak periods are associated to each commercialised item.

This background knowledge is formalised in Figure 6. The corresponding strict rules are depicted in the upper part of the Figure 7. During peak periods for an item, the usual supply action is blocked by the defeater r_{11} . The plausible rule r_{10} says that if the milk stock Y is below the alert threshold c_1 , the *NormalSupply* action should be executed. *NormalSupply* assures a stock value of c_2 . Instead, the *PeakSupply* action is derived by the rule r_{11} .

Figure 6. Domain knowledge sample for milk monitoring.

<i>TBox</i>	<i>Milk</i> \sqsubseteq <i>Item</i>
	<i>Item</i> $\sqsubseteq \forall HasPeak.Time$
	<i>WholeMilk</i> \sqsubseteq <i>Milk</i>
	<i>LowFatMilk</i> \sqsubseteq <i>Milk</i>
<i>ABox</i>	<i>fm</i> ₁ : <i>WholeMilk</i>
	<i>sm</i> ₁ : <i>LowFatMilk</i>
	<i>sm</i> ₂ : <i>LowFatMilk</i>

Figure 7. Plausible knowledge base.

<i>Strict Rules</i>	<i>r</i> ₁ : <i>Milk</i> (<i>X</i>) \rightarrow <i>Item</i> (<i>X</i>)
	<i>r</i> ₂ : <i>Item</i> (<i>X</i>), <i>HasPeak</i> (<i>X</i> , <i>Y</i>) \rightarrow <i>Time</i> (<i>Y</i>)
	<i>r</i> ₃ : <i>WholeMilk</i> (<i>X</i>) \rightarrow <i>Milk</i> (<i>X</i>)
	<i>r</i> ₄ : <i>LowFatMilk</i> (<i>X</i>) \rightarrow <i>Milk</i> (<i>X</i>)
<i>Facts</i>	<i>f</i> ₁ : <i>WholeMilk</i> (<i>fm</i> ₁)
	<i>f</i> ₂ : <i>LowFatMilk</i> (<i>sm</i> ₁)
	<i>f</i> ₃ : <i>LowFatMilk</i> (<i>sm</i> ₂)
<i>Plausible Rules</i>	<i>r</i> ₁₀ : <i>Milk</i> (<i>X</i>), <i>Stock</i> (<i>X</i> , <i>Y</i>), <i>Less</i> (<i>Y</i> , <i>c</i> ₁) \Rightarrow <i>NormalSupply</i> (<i>X</i> , <i>c</i> ₂)
	<i>r</i> ₁₂ : <i>Milk</i> (<i>X</i>), <i>Stock</i> (<i>X</i> , <i>Y</i>), <i>Less</i> (<i>Y</i> , <i>c</i> ₁), <i>hasPeak</i> (<i>X</i> , <i>Z</i>), <i>now</i> (<i>Z</i>) \Rightarrow <i>PeakSupply</i> (<i>X</i> , <i>c</i> ₃)
	<i>r</i> ₁₃ : <i>AlternItem</i> (<i>X</i> , <i>Z</i>), <i>Milk</i> (<i>X</i>), <i>Stock</i> (<i>Z</i> , <i>Y</i>), <i>Greater</i> (<i>Y</i> , <i>c</i> ₄) $\Rightarrow \neg PeakSupply$ (<i>X</i> , <i>c</i> ₃)
	<i>r</i> ₁₄ : <i>LastMeasurement</i> (<i>S</i> , <i>Y</i>), <i>hasLatency</i> (<i>S</i> , <i>Z</i>), <i>Greater</i> (<i>Y</i> , <i>Z</i>) \Rightarrow <i>BrokenSensor</i> (<i>S</i>)
<i>Defeaters</i>	<i>r</i> ₁₅ : <i>BrokenSensor</i> (<i>S</i>), <i>measures</i> (<i>S</i> , <i>X</i>) \rightarrow <i>Stock</i> (<i>X</i> , _)
	<i>r</i> ₁₁ : <i>hasPeak</i> (<i>X</i> , <i>Y</i>) \rightarrow <i>NormalSupply</i> (<i>X</i> , <i>c</i> ₂)
<i>Priorities</i>	<i>r</i> ₁₁ \succ <i>r</i> ₁₀

If there is an alternative item *Z* for the *Milk* product and the stock of the alternative is larger than the threshold *c*₄, this implies not to supply the higher quantity *c*₂ (the rule *r*₁₂). Whether the action is executed or not depends on the priority relation between the rules *r*₁₂ and *r*₁₃,

The sensor related information can be integrated when reasoning. If the sensor *S* seems not to function according to the specifications in the ontology, it is plausible to be broken (the rule *r*₁₄). A broken sensor defeats the stock information asserted in the knowledge base related to the measured item (the defeater *r*₁₅).

The merchandise flow is simulated by generating infinite input and output streams. Assuming that the function *randomItem* :: [*Item*]- \rightarrow *Item* based on the list of available items returns a random item. The infinite output stream for the payment point *out*₁ would be:

$$out_1 = (randomItem\ l) : out_1$$

where *l* is a list with the available items in the simulation. Consider the following RDF stream *s*₁ of sold items (*item sold price*) and the associated time of measurement, where the predicate *sold* and the *price* value are removed for clarity reasons:

$$[(sm_1, -, -), 1], ((m_1, -, -), 2), ((fm_1, -, -), 3), ((m_2, -, -), 4), ((m_3, -, -), 5), \\ ((sm_2, -, -), 6), ((m_4, -, -), 7), \dots]$$

The *updateStock* function continuously computes the current stocks based on the s_1 stream. Based on the fact f_1 and the rule r_3 , one can conclude that fm_1 is a milk item. Similarly, based on the facts f_2 and f_3 , the rule r_4 categorises the instances sm_1 and sm_2 as milk items. The filter function is used to monitor each milk item, either low fat or not:

$$milkItems = filter\ milk\ (map\ first\ \circ\ first\ s_1)$$

Here, the predicate *milk* returns true if the input is of type *Milk* according to the rules r_3 or r_4 . The *map* function is used to select only the element *item* from the tuples $((item, -, -), time)$ from the stream s_1 : the composition *first* \circ *first* is used to extract the first element in the first tuple.

The stream *milkItems* collects all the items of type milk every time an item occurs. Based on s_1 , the *milkItems* is:

$$[(sm_1, -, -), 1], ((fm_1, -, -), 3), ((sm_2, -, -), 6), \dots]$$

The *updateStock* :: *Item* \rightarrow *Stream* \rightarrow *Int* function is activated to compute the available stock for a specific category of products. Consider the current stock for milk is 102 and the threshold c_1 for triggering the alarm is 100. Assume the function *updateStock* is called with the first input parameter *milk* and the second parameter *milkItems*. At time 1, sm_1 being low fat milk, identified as a subtype of milk, the stock is updated at the value 101. At instance 3, fm_1 being fat milk, identified also as a subtype of milk, the stock is updated at the value 100. At time step 6, sm_2 , the stock value reaches $c_1 = 99$. At this moment, the predicate *Less*(99, c_1) becomes valid.

Consequently, the rule r_{10} in Figure 7 is plausibly activated. The algorithm checks whether any defeater or stronger rule can block the derivation of the conclusion of the rule r_{10} . If no one blocks it, the action *normalSupply* for *milk* of value c_2 is executed. If, for instance in case of a peak period, the defeater r_{11} is active, since r_{11} is stronger than the rule r_{10} , it successfully blocks the derivation of the action *normalSupply*. Instead, the consequent of the rule r_{12} will be executed.

Thus, by combining ontological knowledge with plausible rules, one can reason with generic products (*Milk*), even if the streams report data regarding instances of specific products (*WholeMilk* and *LowFatMilk*). The benefit here consists in minimising the number of business rules that should be added within the system.

The current implementation was tested only on the above proof of concept scenario. The scenario involves a small number of axioms from the sensor ontology and from the milk-domain ontology, and also few plausible rules manually constructed to test the plausible reasoning mechanism. For the simulated ten streams of items with a time delay of 1 s for each item, the decision was taken in real time.

6. Conclusions

The proposed semantic based stream management system is characterised by: (i) continuous situation awareness and capability to handle theoretically infinite data streams due to the lazy evaluation mechanism; (ii) aggregating heterogeneous sensors based on the ontologies translated as strict rules; (iii) handling noise and contradictory information inherently in the context of many sensors, due to

the plausible reasoning mechanism. The system represents a step towards building real-time stream processors for knowledge-rich applications.

With streams being approximate, omniscient rationality is not assumed when performing reasoning tasks on streams. Consequently, we argue that plausible reasoning for real time decision making is adequate. One particularity of our system consists of applying an efficient non-monotonic rule based system [7] when reasoning on gradually occurring stream data. The inference is based on several algorithms, which is in line with the proof layers defined in the Semantic Web stack. Moreover, all the Haskell language is available to extend or adapt the existing code. The efficiency of data driven computation in functional reactive programming is supported by the lazy evaluation mechanism, which allows to use values before they can be known.

In order to apply our PSRH system to a different domain, three tasks are necessary:

1. to translate the domain specific ontologies into strict rules, which is automatically performed by the mapping module;
2. to design plausible rules and priorities by a domain expert;
3. to import the most adequate sensor ontology for the current problem (for instance SWEET or W3S SN Ontology).

During the translation part, some knowledge from the ontology may be lost. The axioms stating a subclass of a complex class which is a disjunction cannot be translated into PL, as in $CheapMilk \sqsubseteq LowFatMilk \sqcup NoBrandMilk$. Also, subclasses of a complex class expression which is existential quantified cannot be translated, such as $Sensor \sqsubseteq \exists hasPart.Battery$. Our solution allows this limitation of expressivity in order to perform reasoning in real time within the efficient plausible logic framework.

The current implementation was tested only on the proof of concept scenario described in Section 5. More extensive experiments are needed to test the large scale efficiency and scalability of the proposed system. At the moment, we are able to support our solution based on the results reported in [7] and based on the reduced complexity of description logic programs [8].

Acknowledgments

We are grateful to the anonymous reviewers for their useful comments. The work has been co-funded by the Sectoral Operational Programme Human Resources Development 2007–2013 of the Romanian Ministry of Labour, Family and Social Protection through the Financial Agreement POSDRU/89/1.5/S/62557.

References

1. Christin, D.; Mogre, P.S.; Hollick, M. Survey on wireless sensor network technologies for industrial automation: The security and quality of service perspectives. *Future Internet* **2010**, *2*, 96–125.
2. Le-Phuoc, D.; Parreira, J.X.; Hausenblas, M.; Hauswirth, M. *Unifying Stream Data and Linked Open Data*; Technical Report for Digital Enterprise Research Institute (DERI): Galway, Ireland, 2010.

3. Valle, E.D.; Ceri, S.; van Harmelen, F.; Fensel, D. It's a streaming world! Reasoning upon rapidly changing information. *IEEE Intell. Syst.* **2009**, *24*, 83–89.
4. Granitzer, M.; Sabol, V.; Onn, K.W.; Lukose, D.; Tochtermann, K. Ontology alignment—A survey with focus on visually supported semi-automatic techniques. *Future Internet* **2010**, *2*, 238–258.
5. Barbieri, D.; Braga, D.; Ceri, S.; Valle, E.D.; Grossniklaus, M. Incremental reasoning on streams and rich background knowledge. *Lect. Notes Comput. Sci.* **2010**, *6088*, 1–15.
6. Savage, N. Twitter as medium and message. *Commun. ACM* **2011**, *54*, 18–20.
7. Maher, M.J.; Rock, A.; Antoniou, G.; Billington, D.; Miller, T. Efficient defeasible reasoning systems. *Int. J. Artif. Intell. Tools* **2001**, *10*, 483–501.
8. Krötzsch, M.; Rudolph, S.; Hitzler, P. Complexity boundaries for horn description logics. In *Proceedings of the 22nd national conference on Artificial intelligence*, 22–26 July 2012, Toronto, Canada, 2007; AAAI Press: Palo Alto, CA, USA; pp. 452–457.
9. Letia, I.A.; Groza, A. *Description Plausible Logic Programs for Stream Reasoning*; Filipe, J., Fred, A.L.N., Eds.; SciTePress: Setubal, Portugal, 2012; pp. 560–566.
10. Calbimonte, J.P.; Corcho, Ó.; Gray, A.J.G. Enabling ontology-based access to streaming data sources. *Lect. Notes Comput. Sci.* **2010**, *6496*, 96–111.
11. Palopoli, L.; Terracina, G.; Ursino, D. A plausibility description logic for handling information sources with heterogeneous data representation formats. *Ann. Math. Artif. Intell.* **2003**, *39*, 385–430.
12. Heintz, F.; Kvarnström, J.; Doherty, P. Stream reasoning in DyKnow: A knowledge processing middleware system. Presented at 1st International Workshop on Stream Reasoning, Heraklion, Crete, Greece, 31 May 2009.
13. Bolles, A.; Grawunder, M.; Jacobi, J. Streaming SPARQL extending SPARQL to process data streams. *Lect. Notes Comput. Sci.* **2008**, *5021*, 448–462.
14. Anicic, D.; Fodor, P.; Rudolph, S.; Stühmer, R.; Stojanovic, N.; Studer, R. A rule-based language for complex event processing and reasoning. *Lect. Notes Comput. Sci.* **2010**, *6333*, 42–57.
15. Anicic, D.; Rudolph, S.; Fodor, P.; Stojanovic, N. Retractable complex event processing and stream reasoning. *Lect. Notes Comput. Sci.* **2011**, *6826*, 122–137.
16. Corcho, Ó.; Garcia-Castro, R. Five challenges for the Semantic Sensor Web. *Semant. Web* **2010**, *1*, 121–125.
17. ASPIRE Project (Advanced Sensors and lightweight Programmable middleware for Innovative Rfid Enterprise applications). Available online: <http://www.fp7-aspire.eu/> (accessed on 15 October 2012).
18. Sensei Project. Available online: <http://www.ict-sensei.org/Sensei090422/> (accessed on 15 October 2012).
19. Ruta, M.; Colucci, S.; Scioscia, F.; Sciascio, E.D.; Donini, F.M. Finding commonalities in RFID semantic streams. *Procedia Comput. Sci.* **2011**, *5*, 857–864.
20. Rock, A. *Implementation of Decisive Plausible Logic*; Technical Report for School of Information and Communication Technology, Griffith University: South Brisbane, Australia, 2010.
21. Billington, D.; Rock, A. Propositional plausible logic: Introduction and implementation. *Stud. Log.* **2001**, *67*, 243–269.

22. Governatori, G.; Maher, M.J.; Antoniou, G.; Billington, D. Argumentation semantics for defeasible logic. *J. Log. Comput.* **2004**, *14*, 675–702.
23. Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; Patel-Schneider, P. *The Description Logic Handbook: Theory, Implementation and Applications*; Cambridge University Press: New York, NY, USA, 2003.
24. Gomez, S.A.; Chesnevar, C.I.; Simari, G.R. A defeasible logic programming approach to the integration of rules and ontologies. *J. Comput. Sci. Technol.* **2010**, *10*, 74–80.
25. Grosz, B.N.; Horrocks, I.; Volz, R.; Decker, S. Description logic programs: Combining logic programs with description logic. In *Proceedings of the 12th international conference on World Wide Web*, Budapest, Hungary, 20–24 May 2003; ACM Digital Library: New York, NY, USA; pp. 48–57.
26. Stuckenschmidt, H.; Ceri, S.; Valle, E.D.; van Harmelen, F. Towards expressive stream reasoning. In *Semantic Challenges in Sensor Networks*; Aberer, K., Gal, A., Hauswirth, M., Sattler, K.U., Sheth, A.P., Eds.; Schloss Dagstuhl—Leibniz-Zentrum fuer Informatik: Dagstuhl, Germany, 2010.
27. Decisive Plausible Logic Tool. Available online: <http://www.ict.griffith.edu.au/arock/DPL/> (accessed on 15 October 2012).
28. De Hoog, R. *Expert Systems—Past, Present, and Future*; Technical Report for Metis, University of Amsterdam: Amsterdam, the Netherlands, 2003.

© 2012 by the authors; licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution license (<http://creativecommons.org/licenses/by/3.0/>).