Embedded Rule-based Reasoning for Digital Product Memories

Christian Seitz, Steffen Lamparter, Thorsten Schöler, and Michael Pirker
Siemens AG, Corporate Technology
Autonomous Systems
81739 Munich, Germany
[ch.seitz|steffen.lamparter|thorsten.schoeler|michael.pirker]@siemens.com

Abstract

A Digital Product Memory provides a digital diary of the complete product life cycle that is embedded in the product itself using smart wireless sensor technology. The data is hereby gathered by recording relevant ambient parameters in digital form. In this paper, we present the architecture and cost-efficient implementation of an autonomous digital product memory that generates and interprets its diary using rule-based reasoning methods. As we assume an open, heterogeneous sensor infrastructure, we rely on standard syntax and semantics provided by the Web Ontology Language OWL. The digital product memory collects and provides data using the OWL fragment OWL 2 RL which can be processed with standard rule engines. As rule engine we use CLIPS on embedded hardware and exemplify the application of the digital product memory e. g. for predictive maintenance.

Introduction

Today, RFID (Radio Frequency Identification) is used to identify a wide range of work pieces or individual products for tracking their movements through the logistics chain. For future purposes the idea of storing only a single ID must be extended to a Digital Product Memory. This memory stores information of the complete product life cycle and is embedded in the product itself. But the product memory is not only a passive data storage, it is also able to monitor and control its environment e. g. by communicating with other products or sending commands to manufacturing devices. Integrating digital memories to products results in shorter product and innovation cycles, more complex logistics chains, and a product-driven production process. Finally, the digital product memory opens up new dimensions for protection from product piracy, consumer protection and product liability.

A key problem of digital product memories is their cross-domain nature, i. e. new relevant data must be added to the memory from various stake holders during the complete product life cycle and the memory has to be interpretable by the product memory in an integrated way. Integrating data from heterogeneous sources requires a data exchange format with standard syntax and semantics. In order to enable the product memory to analyze the data and trigger appropriate events, the memory has to be machine-interpretable.

This paper introduces a flexible approach for accumulating and analyzing digital product memories. We use the Web Ontology Language (OWL) (Motik, Patel-Schneider, and Parsia 2009) for providing a standard syntax/semantics that can be interpreted automatically by the product memory. Since data is successively added to the product memory an event-driven interpretation method is appropriate and we therefore use production rules for the reasoning task. Our contribution is based on embedded hardware (sensor mote), encompasses software modules for integrating data into a product memory and contains components to reason about this product memory data.

The paper is organized as follows. The next section presents the requirements which are necessary for product memory applications. This is followed by a detailed explanation of already existing approaches. After this, the architecture is presented and all components are explained in detail. The next section explains our implementation. This section is followed by example applications. The paper concludes with a summary and an outlook.

Requirements

This section describes general and implementation specific requirements for our embedded reasoning system.

General Requirements

The goal of our research is to develop a comprehensive cross-domain approach for creating digital product memories. We assume a distributed, heterogeneous sensor infrastructure or other relevant information sources that can be accessed by a product memory which is based on embedded hardware.

We aim to store all relevant data on the product. It is not appropriate to swap relevant data to external data bases. The product memory data must be accessed at any time, which cannot be guaranteed if the data is swapped out, because of non-permanent communication possibilities. However, historical data, which is not necessary for future purposes, e. g. interpretation processes, can be transferred to data bases and may still be accessible via links to external resources.

Implementation Requirements

R1 The solution should not depend on a specific application domain, because the product memory must be accessed during the complete product life cycle. Our main application field is the factory automation domain. We focus on integrating sensor values into the product memory, which result during the production process. For this purposes the product memory needs to be autonomous, i. e. the product memory is not filled by other entities, e. g. programmable logic controller. Therefore, the product memory consists of an own controller that requests the relevant data from the environment.

R2 To specify which product memory data is relevant, an expressive formalism is needed. But the syntax needs to be very compact, because a fast processing with restricted resources is necessary. Therefore an XML representation for the specification is not appropriate whereas XML for representing product data is possible.

R3 Additionally, the collected product memory data must be interpreted. Therefore, an event-based approach is useful. Entering new data in the product memory may cause new actions. These actions must be executes at once, without initiating additional queries.

R4 The controller of the product memory should also be able to analyze the product memory data. Since the product memory contains a huge amount of data and the product memory is attached to the product it is not appropriate that basic data interpretation is done by external services.

R5 The controller of the product memory should be implemented as efficient as possible. Therefore, the usage of interpreted programming languages with high memory consumption shall not be used without intense performance evaluation.

R6 The hardware for the product memory must be affordable or in reasonable relation to the price of the product. For extremely cheap products, a product memory is beyond reality. But we believe in product memories for valuable goods (price $>$ 1000 \$) in about five years. For that reason the price of the product memory hardware must drop to a few dollars.

R7 The product memory hardware must contain communication interfaces, because a product memory must be able to interact with its environment.

The single requirements are picked up in the rest of the paper to check if they are met.

Related Work

Generally, there are two major streams of work related to our approach. On the one hand, there is work on methods and implementations for OWL reasoning and rule-based systems. In this area substantial work has been done in recent years – also particularly for bridging the gap between rule-based and description logic-based system. In this context, a lot of proposals (Cadoli et al. 1996; Levy and Rousset 1996; Motik, Sattler, and Studer 2005; Franconi and Tessaris 2004; Rosati 2005) have been put forward, how description logics can be expressed with rule-based systems in order to provide more efficient reasoning algorithms. Together with standard OWL reasoners such as Pellet (Sirin et al. 2007) these stream of approaches provides an important starting point for our work. However, the implementation of these reasoning systems target personal computers rather than embedded devices and thus cannot be easily migrated to resource-constrained devices as typically huge amounts of memory and additional software is required.

On the other hand, there is a stream of work with the goal of implementing reasoners for embedded devices. Safdar and Ali present with μOR (Ali and Kiefer 2009) a lightweight micro OWL description logic reasoning system for resource-constrained devices. While the system is lightweight and very suitable for embedded devices, it supports only a very restricted fragment of OWL-Lite. As OWL-Lite itself is already rather inexpressive, the approach is not sufficient for our product memory scenario. The authors of (Meditskos and Bassiliades 2008) use the object-oriented extension of a production system for reasoning and querying OWL ontologies. Their approach is based on a transformation procedure of OWL ontologies into an Object-Oriented schema and the application of inference production rules over the generated objects in order to implement the various semantics of OWL. While the approach uses also the CLIPS rules engine for reasoning with OWL, the used OWL-to-Objects-mapping leads to an unnecessary high overhead that restricts its application. The rule engine Bossam (Jang and Sohn 2004) is especially developed for OWL reasoning. Semantic web features like URI referencing are supported and cooperation among multiple Bossam instances is also possible. While an expressive logical fragment is supported, the Java-based implementation restricts the migration of the tool to an embedded device.

Architecture

Based on the existing work discussed in the previous section, we address our requirements by means of the following architecture.

Figure 1 shows the generic architecture of a single product memory. It consists of four major components: (*i*) a *Communication Interface* to send and receive information; (*ii*) a *Product Memory Agent* which coordinates the communication activities; (*iii*) a *Knowledge Base* containing all information stored on the memory; (*iv*) and finally a *Rule Engine* for evaluating the stored information and inferring required actions. In the following we discuss each of the components in more detail.

Communication Interface

The Communication Interface is responsible for receiving data (e. g. from manufacturing machines) as well as for sending data to outside components (e. g. in order to adjust a machine or to report to the manufacturing execution system). The Communication Interface has to be adapted to the concrete applications area. For instance, in industry automation or logistics scenarios, RFID-based communication might be most promising since the corresponding infrastructure is often already in place. In general, a wide range of different

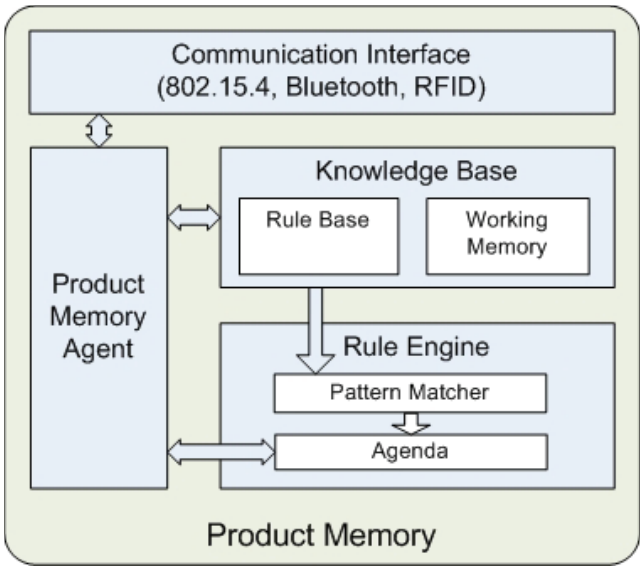

Figure 1: Generic Product Memory Architecture

technologies and protocols can be supported. Since according to Requirement *R2* the representation has to be compact, an efficient encoding of e. g. XML documents has to be applied (Schneider and Kamiya 2008). Sending and receiving activities are controlled by the Product Memory Agent.

Product Memory Agent

The Product Memory Agent is responsible for forwarding information to the knowledge base once it is received by the Communication Interface. Currently, we assume that the input data is already semantically annotated in a standardized way.[1] As a language with standardized syntax and semantics we rely on a restricted fragment of the W3C Web Ontology Language OWL 2 (Motik, Patel-Schneider, and Parsia 2009) called *OWL 2 RL profile*.

This profile is based on Description Logic Programs (Grosof et al. 2003) and pD* (ter Horst 2005). The major difference of the profile in terms of expressiveness compared to OWL 2 DL are a limited set of supported axioms (no disjoint unions of classes, no reflexive object property axioms, no negative property assertions) as well as restrictions on the use of certain constructs (e. g. no existential quantification on the right side of an axiom allowed). Although not fully expressive, the fragment has two major advantages with respect to embedded applications: First, all standard reasoning tasks (such as checking for ontology consistency, class expression satisfiability, class expression subsumption, and instance checking) are tractable, i. e. they can be solved by a deterministic algorithm in polynomial time which is absolutely crucial for the real world embedded systems. Second, OWL 2 RL can be implemented using traditional (forward-chaining) rule-based systems (Meditskos and Bassiliades 2008; Motik et al. 2009) and we can thus easily realize a reactive behavior, i. e. knowledge base updates may directly trigger new actions (meets requirement *R3*).

[1]Note that direct sensor connections to the product memory requires to add functionality for handling raw sensor data to the Product Memory Agent.

constructor	DL	CLIPS rules
class instance	$a : C$	$C(a)$
role instance	$(a, b) : R$	$R(a, b)$
class inclusion	$C \sqsubseteq D$	$\forall x.C(x) \rightarrow D(x)$
role inclusion	$R^+ \sqsubseteq R$	$\forall x.y.z(R(x, y) \wedge R(y, z) \rightarrow R(x, z))$
complement	$\neg C$	$\neg C(x)$
conjunction	$C_1 \sqcap \ldots \sqcap C_n$	$C_1 \wedge \ldots \wedge C_n$
disjunction	$C_1 \sqcup \ldots \sqcup C_n$	$C_1 \vee \ldots \vee C_n$
existential quantification	$\exists R.C$	$\exists y.(R(x, y) \wedge C(y))$
universal quantification	$\forall R.C$	$\forall y.(R(x, y) \rightarrow C(y))$
...	...	...

Table 1: Translation from description logics to CLIPS rules. For a full list refer to (Motik et al. 2009).

The Product Memory Agent therefore has to translate the received data into rules and facts before storing them in the Knowledge Base. The translation is exemplified in Table 1. The translation can be done axiom by axiom and thus exhibits a polynomial complexity. In addition to this transformation, the agent is able to initialize the Knowledge Base and to remove a set of facts from the Knowledge Base.

Knowledge Base

The Knowledge Base contains two sets of data: The *Working Memory* contains the facts about the current state of the world. A fact over a n-ary predicate Q is an expression $Q(a_1, \ldots, a_n)$ where a_i is a constant. The *Rule Base* stores a finite set of rules that determine the behavior of the Product Memory Agent and thus also that of the product memory itself. A rule is simply an expression of the form $B_1, \ldots, B_n \rightarrow A_1, \ldots A_k$ with $k \geq 1, n \geq 0$, which can be expressed more informally $\text{IF} B_1, \ldots, B_n \text{THEN} A_1, \ldots A_k$. A_j and B_i represent literals of the form $(\neg)Q(x_1, \ldots, x_m)$ with $m \geq 0$. Actions that are executed by the Product Memory Agent can be attached to the literals A_j in the rule head (consequent). These actions could be communication activities or changes in the knowledge base.

Rule Engine

Each time a change in the Knowledge Base occurs, the *Pattern Matcher* within the Rule Engine verifies whether the current facts in the Working Memory fulfill the conditions listed in the body of a rule (antecedent). This is repeated for all rules until no rule fires anymore. In case actions are attached to the consequences of a fired rule, they are collected in an *Agenda.* This Agenda then triggers the corresponding methods of the agent implementation. Note that an obvious limitation of the current OWL 2 RL translation to a forward chaining rule language is the possible loss of the declarative model since the sequence of rule processing might change

the results. In real applications declarative models could be important – particularly if different not coordinated sets of rules are added to the product memory by different parties.

In the next section we show how the presented architecture of a product memory can be realized using existing hardware and software modules.

Implementation

This section describes briefly the implementation of our first prototype of a digital product memory.

The hardware basis of the prototype is a Crossbow Imote2 module as shown in figure 2. The Imote2.NET is an advanced wireless sensor node platform. It is built around the low-power PXA271 XScale CPU and also integrates an 802.15.4 compliant radio (Crossbow 2009). It is endowed with 32MB SDRAM and 32MB of FLASH memory, which is sufficient for our first version of a digital product memory. The currently used hardware is still too expensive to realize a product memory today. But we think that in a few years the price of such sensor modules is appropriate, see requirement *R6*.

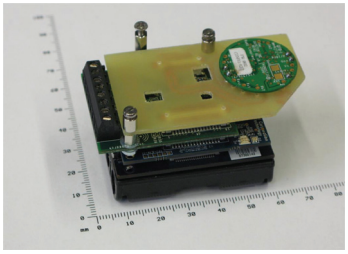

Figure 2: Crossbow Imote2 sensor node with RFID module - hardware prototype for a digital product memory

The programming environment for the xBow Imote2 is Microsoft's .NET Micro Framework with C# as supported programming language. This may at first be a contradiction to requirement *R5*, because C# is an interpreted programming language. The .NET Micro Framework benefits from it's tailoring to small embedded devices with lower resource and memory requirements. Furthermore it does not require a dedicated operating system (like Windows CE). Therefore, the absence of the operating system and the optimized .NET Tiny Common Language Runtime (TinyCLR) provides a good compromise for resource-efficiency whilst offering a modern programming environment.

A small number rule engines (nxBRE, Drools.net) is available for C#, but the Micro Framework provides in contrast to the .NET Framework a limited API, e. g. no generic data types are available. Thus, some additional effort is necessary to deploy a rule engine to the xBow sensor mote. We were faced with the decision, whether we port a C# rule engine to the Micro Framework or whether we integrate a rule engine in native code. We analyzed the effort and decided to integrate the rule engine CLIPS (CLIPS 2009; Riley 1991) in the .NET Micro Framework TinyCLR. CLIPS is a production rule engine written in C, based on the Rete algorithm (Forgy 1982). Today, it is one of the most widely used expert system tools because it is fast, efficient and its source code and binaries are available in the public domain. Additionally, the CLIPS rule language is very compact, which saves memory (see requirement *R2*).

Our solution integrates CLIPS as a native (C language) library into the .NET Micro Framework TinyCLR, similar to native device drivers. CLIPS itself is wrapped via the Façade pattern (Gamma et al. 1995) to provide a consistent and easy-to-use object-oriented STL[2] C++ API. The STL C++ API provides all necessary CLIPS function calls for rule-based reasoning and rule engine management. On top of the STL C++ API, the .NET Micro Framework Interopation mechanism is used to provide access from C# to the C++ CLIPS API.

In order to improve the user-interaction with the Imote2, it was upgraded with an additional Skyetek RFID (Radio Frequency Identification) module (SkyeTek 2009). It provides a low-power, high performance, and cost effective platform and is a self-contained multi-protocol 13.56 MHz module. The Imote2 in combination with the Skyetek module is called in the following siTag (smart industrial tag).

This siTag contains the software modules which are introduced in the architecture section and is supposed to be attached to valuable products. However, the product memory must be configured with rules and initial facts. Since many participants interact with the product memory during the product life cycle it is not possible to store all relevant rules initially. Some participants are not willing to hand their rules to other partner. Not to mention, an increasing amount of rules influences the execution time as well as the memory consumption in a negative way.

Interaction takes place via wireless radio technologies. In order to add new rules or facts to the knowledge base the RFID subsystem is used, because RFID systems are widely used in the supply chain. Rules and facts can be stored on multiple RFID transponders. But there is the constraint, that a rule must fit on a single transponder. The maximum memory capacity of supported transponders is 112 bytes. Due to the slim rule syntax of CLIPS, most rules fit on a single transponder. To increase the maximum rule size, a binary compression is also possible but not necessary for our purposes.

In order to add new rules or facts, one or more transponders must be moved into communication range of the RFID reader. The Product Memory Agent on the siTag can check whether new data is available on the transponder and can shift the new data to the knowledge base of the siTag . Additionally, a special Command Transponder exists. With such a transponder, rule engine management commands can be issued to CLIPS. This is necessary to e. g. clear the rule memory, delete certain facts or perform a complete re-initialize. These special transponders contain a special header in order to recognize the data as management commands.

In addition the RFID-based data exchange it is also possible to use the 802.15.4 radio interface. This enables that two siTags can exchange rules, facts or also product memory data, see requirement *R7*.

[2]Standard Template Library

Applications

In the following, two example applications for our rule-based reasoning approach in the context of digital product memories are presented.

Aggregation of Product Memory Data

In order to get a digital product memory filled with relevant data, it must be a priori specified, which data will become part of the memory. The product memory is active, i. e. the product memory agent makes requests to the environment for certain data (see requirement *R1*). We use a rule-based approach to specify the content of the product memory, e. g.:

$(Temperature(Product) > 25.0\ °C) \rightarrow$
enter $Humidity\_value(Environment)$ *in product memory*

This rule simply specifies that the product memory agent has to provide the memory with an additional humidity value. The product memory agent needs to communicate with an infrastructure, which is explained in detail in (Seitz, Schöler, and Neidig 2009).

With such rules the product memory will be successively filled. Additionally, there are rules to analyze the product memory, e. g.:

$(Temperature(Product) > 25.0\ °C \wedge$
$Humidity(Environment) > 90\ \% \rightarrow$
send ALARM Message

By means of such rules, complex situations can be recognized and necessary actions can be executed by the product memory agent.

Predictive Maintenance

The benefit of a digital product memory is, that the data can be used for the detection of technical issues when the product is already deployed in the field. Since the digital memory contains a lot of sensor data, an analysis is useful for diagnostics and condition-based or predictive maintenance, see requirement *R4*. The second application of the digital product memory is about detection of anomalies in acceleration data.

An anomaly detector should report if newly measured data of certain sensors have a different pattern or are beyond the normal sensor values. The basis for an abnormality detector are time series of sensor values or other data sources (e. g. from databases), which are elements of the product memory. A given time series which describes the normal behavior is the basis for creating a model. In our case this model is specified with rules. In a detection phase of an anomaly detector it is checked whether the rules for normal behavior are fulfilled or an exception has occurred. This detection step should be as fast as possible – in many cases even real-time processing is necessary.

The algorithm we use is based on the work of (Salvador and Chan 2005).

The algorithm transforms the data of the digital product memory into a multi-dimensional feature space, which results in a trajectory, see figure 3 on the left side. This trajectory is covered with shapes. We use a three-dimensional

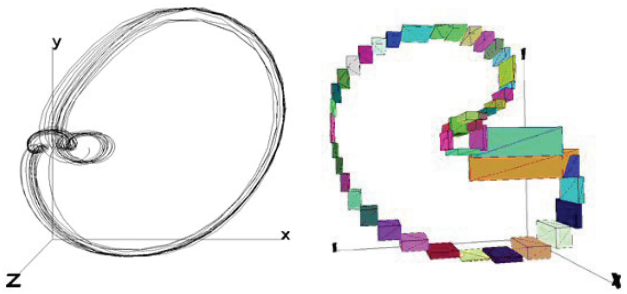

Figure 3: Transformation of a time series in a feature space

feature space and boxes as shapes, see figure 3 on the right side. When we assume that a box is not rotated in the feature space, a box is determined by two points in a three-dimensional space. Each box in the transformed feature space is represented by one rule. The following gives an example in a pseudo rule language:

$(data.x < box_i.x_{min} \vee data.x > box_i.x_{max} \vee$
$data.x < box_{i+1}.x_{min} \vee data.x > box_{i+1}.x_{max}) \rightarrow$
$exception = TRUE$

An anomaly is detected if new data is not within these shapes. The size and the location of the shapes are transformed into rules. Thus, an anomaly test can be achieved by using the rule engine as described in the architecture section. To use the rule engine as an anomaly detector, the rules need to be added to the rule base and new data needs to be committed to the working memory. Afterwards the rule engine checks if any rule is violated, which will trigger an exception.

To differ outliers from abnormal behavior other rules are activated and this results in positive output, e. g.

$(exception = TRUE) \rightarrow exceptionCounter$++
$(exceptionCounter > threshold) \rightarrow$
$anormalityDetection = TRUE$

The algorithm is highly sensitive and a small rule set is sufficient to detect abnormal behavior. Therefore, it is perfectly suited to be implemented on resource-constraint devices like our sensor nodes.

Conclusion and Future Work

In this paper we presented an embedded reasoning approach for digital product memories. A product memory provides a digital diary of the complete product life cycle. We presented an architecture which enables autonomous product memories, i. e. the product memory determines which data becomes part of the memory. In order to realize reactive behavior, we use a rule-based reasoning approach based on the CLIPS rule engine. The communication with the product digital memory is encoded with OWL 2 RL. We implemented applications on a Crossbow Imote2 sensor node.

We are currently analyzing the performance of our approach in combination with the used hardware. Additionally, we are testing alternative hardware to find the optimum, regarding price and performance. For the future we

plan to extend the expressiveness of our approach beyond OWL 2 RL and extend the supported logical fragment to ELP (Kroetzsch, Rudolph, and Hitzler 2008). ELP provides additional modeling constructs, it can be mapped to a rule language, and it keeps reasoning still tractable. The rule-based approach can be enhanced with temporal reasoning, allowing declarative event correlation, e. g. in causality or in time. This will enable the product memory to detect certain situations more easily and helps to implement event-driven business activity monitoring with digital product memories. We also plan to apply our embedded reasoning architecture to other scenarios such as Ambient Assisted Living (AAL) applications. The basic idea is to support elderly people in their homes with small intelligent devices.

Acknowledgment

This research was funded in part by the German Federal Ministry of Education and Research under grant number 01 IA 08002 G. The responsibility for this publication lies with the authors.

References

Ali, S., and Kiefer, S. 2009. μor — a micro owl dl reasoner for ambient intelligent devices. In *GPC '09: Proceedings of the 4th International Conference on Advances in Grid and Pervasive Computing*, 305–316. Berlin, Heidelberg: Springer-Verlag.

Cadoli, M.; Donini, F. M.; Liberatore, P.; and Schaerf, M. 1996. Comparing space efficiency of propositional knowledge representation formalisms. In *In Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96*, 364–373.

CLIPS. 2009. Rule Engine. `http://clipsrules.sourceforge.net/`, Accessed 5.10.2009.

Crossbow. 2009. Crossbow Imote2 Datasheet. `http://www.xbow.com/Products/Product_pdf_files/Wire% -less_pdf/Imote2.NET_ED_Datasheet.pdf`, Accessed 5.10.2009.

Forgy, C. L. 1982. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19:17–37.

Franconi, E., and Tessaris, S. 2004. Rules and queries with ontologies: Unified logical framework. In *In Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR-04)*, 50–60. Springer.

Gamma, E.; Helm, R.; Johnson, R. E.; and Vlissides, J. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley.

Grosof, B. N.; Horrocks, I.; Volz, R.; and Decker, S. 2003. Description logic programs: combining logic programs with description logic. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, 48–57. New York, NY, USA: ACM Press.

Jang, M., and Sohn, J.-C. 2004. Bossam: An extended rule engine for owl inferencing. volume 3323/2004, 128–138. Hiroshima: Springer Berlin / Heidelberg.

Kroetzsch, M.; Rudolph, S.; and Hitzler, P. 2008. ELP: Tractable Rules for OWL 2. In Sheth, A.; Staab, S.; Dean, M.; Paolucci, M.; Maynard, D.; Finin, T.; and Thirunarayan, K., eds., *Proceedings of the 7th International Semantic Web Conference (ISWC 2008)*, volume 5318 of *LNCS*, 649–664. Springer.

Levy, A. Y., and Rousset, M.-C. 1996. Carin: A representation language combining horn rules and description logics. 323–327.

Meditskos, G., and Bassiliades, N. 2008. A rule-based object-oriented owl reasoner. *IEEE Trans. on Knowl. and Data Eng.* 20(3):397–410.

Motik, B.; Grau, B. C.; Horrocks, I.; Zhe Wu, A. F.; and Lutz, C. 2009. Owl 2 web ontology language profiles. `http://www.w3.org/TR/owl2-profiles/#OWL_2_RL`. W3C Proposed Recommendation.

Motik, B.; Patel-Schneider, P. F.; and Parsia, B. 2009. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. `http://www.w3.org/TR/2009/PR-owl2-syntax-20090922/`. W3C Proposed Recommendation.

Motik, B.; Sattler, U.; and Studer, R. 2005. Query answering for owl-dl with rules. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web* 3(1):41–60.

Riley, G. 1991. Clips: An expert system building tool. In *In Proceedings of the Technology 2001 Conference*.

Rosati, R. 2005. On the decidability and complexity of integrating ontologies and rules. *Journal of Web Semantics* 3(1):41–60.

Salvador, S., and Chan, P. 2005. Learning states and rules for detecting anomalies in time series. *Applied Intelligence* 23(3):241–255.

Schneider, J., and Kamiya, T. 2008. Efficient XML Interchange (EXI) Format 1.0. `http://www.w3.org/TR/2008/WD-exi-20080919/`. Working Draft.

Seitz, C.; Schöler, T.; and Neidig, J. 2009. An agent-based sensor middleware for generating and interpreting digital product memories. In *AAMAS '09: Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*.

Sirin, E.; Parsia, B.; Grau, B. C.; Kalyanpur, A.; and Katz, Y. 2007. Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web* 5(2):51–53.

SkyeTek. 2009. Web site. `http://www.skyetek.com/ProductsServices/EmbeddedRFID-Readers/SkyeModuleM1mini/tabid/338/Default.aspx`, Accessed 12.10.2009.

ter Horst, H. J. 2005. Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary. *Journal of Web Semantics* 3(2-3):79–115.