Rule-Based OWL Reasoning for Specific Embedded Devices

Christian Seitz and René Schönfelder

Siemens AG, Corporate Technology
Intelligent Systems and Control
81739 Munich, Germany
{ch.seitz,rene.schoenfelder}@siemens.com

Abstract. Ontologies have been used for formal representation of knowledge for many years now. One possible knowledge representation language for ontologies is the OWL 2 Web Ontology Language, informally OWL 2. The OWL specification includes the definition of variants of OWL, with different levels of expressiveness. OWL DL and OWL Lite are based on Description Logics, for which sound and complete reasoners exits. Unfortunately, all these reasoners are too complex for embedded systems. But since evaluation of ontologies on these resource constrained devices becomes more and more necessary (e.g. for diagnostics) we developed an OWL reasoner for embedded devices. We use the OWL 2 sub language OWL 2 RL, which can be implemented using rule-based reasoning engines. In this paper we present our used embedded hardware, the implemented reasoning component, and results regarding performance and memory consumption.

1 Introduction

Ontologies have been used for formal representation of knowledge for many years now. An ontology is an engineering artifact consisting of a vocabulary used to describe some domain. Additional constraints capture additional knowledge about the domain. One possible knowledge representation language for ontologies is the OWL 2 Web Ontology Language, informally OWL 2. The OWL specification includes the definition of variants of OWL, with different levels of expressiveness. To answer queries over ontology classes and instances some reasoning mechanism is needed. OWL DL and OWL Lite are based on Description Logics, for which sound and complete reasoners exits. Unfortunately, all these reasoners are too complex for embedded systems. But since evaluation of ontologies on these resource constrained devices becomes more and more necessary (e.g. for diagnostics) we developed an OWL reasoner for embedded devices.

A possible use case for embedded reasoning is industrial diagnosis, for example in a car. A car has several system elements like the engine and tires. These elements have physical characteristics, e.g. a tire has a pressure and the engine has a temperature and specific fuel efficiency. Sensor nodes can measure these physical characteristics and deliver the measurement values to a control unit. An ontology specifies all known problems and how to detect them. To create such an ontology, a predefined language is used. With the help of the ontology and the measured data, the software can find the root

L. Aroyo et al. (Eds.): ISWC 2011, Part II, LNCS 7032, pp. 237–252, 2011.

cause of the error and may give suggestions how to fix it. To realize these possibilities, in many cases it is important that the software is executable on limited hardware like an embedded system. This work answers the questions of whether reasoning on embedded hardware is possible and how it can be implemented.

The paper is organized as follows. The next section presents already existing approaches for embedded reasoning. This is followed by a detailed explanation of our activities to use an existing DL reasoner on embedded hardware. After this, our rule-based approach is introduced. We present an architecture and the reasoning process in the next section. In the evaluation section, we show the results of the evaluation process. The paper concludes with a summary and a future outlook.

2 Related Work

Various implementations of OWL reasoners exist, e.g. Pellet, FaCT++ and RacerPro. But the memory requirements for installation and at runtime are quite high. Some of them are limited to use on desktop systems or servers only. In the following, we present some approaches that focus on embedded hardware.

SweetRules [5] pioneered rule-based implementation of DL with rule engines. SweetRules does not implement OWL 2, but it supports inferencing in Description Logic Programs subset of DL via translation of first DAML, then OWL 1, into rule engines (Jess/CLIPS).

Bossam[15] is a RETE-based example of a DL reasoner. Bossam is based on a forward chaining production rule engine, which only needs 750Kb runtime memory. In 2007 Bossam has been performance-tuned and released in a new version. The meta-reasoning approach of Bossam was to be changed to a translation-based approach. The results of this work were never released.

One further embedded reasoner is Pocket KRHyper[9]. The core of the system is a first order theorem prover and model generator based on the hyper tableaux calculus. But the development of Pocket KRHyper stopped years ago. This reasoner is not up to date and no support is provided. Documentation on how exactly KRHyper was designed and implemented is also not available.

Another embedded reasoner is μOR[1]. It is a lightweight OWL description logic reasoner for Biomedical Engineering. It was developed for resource-constrained devices in order to enrich them with knowledge processing and reasoning capabilities. To express semantic queries efficiently, the team of μOR has developed SCENT, which is a Semantic Device Language for N-Triples. This approach is similar to but different from ours. μOR is more complex and comes with more overhead. Also it is only designed for OWL Lite and not for OWL 2. For this reason, only a few parts are further pursued in this work.

The first popular approach to emulate a reasoner using CLIPS is described in [13]. Here the object oriented extension of CLIPS, called COOL, was used to build a reasoner for the OWL1 Lite[23] sub-language. This reasoner, called O-DEVICE, is a knowledge base system for reasoning and querying OWL ontologies by implementing RDF/OWL

entailments in the form of production rules in order to apply the formal semantics of the language. O-Device is an OWL 1 reasoner which is very powerful and has a lot of features. Its complexity makes it hard to optimize this reasoner for the use on embedded hardware.

The authors in [14] describe an OWL 2 RL reasoner based on Jena and Pellet, both of which are based on Java.

As described in [12], another approach exists to dynamically perform reasoning depending on the specified query. The authors create rules dynamically for the given ontology. In the case of the deterministic end state of OWL 2 RL reasoning, the complexity of this approach resembles our approach. Therefore, the worst-case memory usage is equal to the first approach because in a specially designed case it can be possible to draw every possible conclusion of the ontology.

3 Reasoning on an Embedded Device with Standard DL Reasoners

There are already a lot of implemented DL reasoners. In this section, we analyze whether the most used reasoners can be executed on embedded hardware. As an embedded system, we use a Gumstix Verdex Pro [6], as shown in Figure 1. We opt for the Gumstix because of its modular hardware architecture and its compact size and weight. Additionally, it is best suited for industrial applications because of its supported operating temperature up to 85 °C. The Gumstix can also be used in vibrating environments which often occurs if the embedded device is e.g. attached to motors or moving machines. The key features of this stick are: Marvell PXA270 CPU with XScale @ 400 MHz, 64 MB RAM, and 16 MB Flash. The Gumstix was designed to run with a stand alone Linux root image. Additional software can be downloaded or compiled using the OpenEmbedded framework.

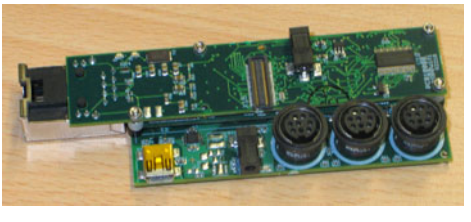

Fig. 1. Gumstix Verdex Pro

In the following, the compatibility of the DL reasoners Pellet, Fact++, and CEL with the Gumstix is analyzed.

3.1 Pellet

Pellet is an open source OWL 2 reasoner for Java. Pellet supports OWL 2 profiles including OWL 2 EL. It incorporates optimizations for nominals, conjunctive query

answering, and incremental reasoning [16]. For these tests, Pellet 2.0.2 was used. Pellet is a very powerfully reasoner which uses the tableau reasoning algorithm[7], supports consistency checking, SWRL[17], and DL Safe Rules.

Because of its complexity, Pellet is rather resource intensive. It needs at least 512 MB of RAM at startup plus memory for the Java environment. Anyhow we tried to adapt Pellet to our embedded system. The startup was modified so that Pellet only may use 40 MB of RAM at startup. Unfortunately, the Java virtual machine on the Gumstix is not fully compatible with current Sun Java and Pellet also needs these unsupported functions. Thus, at the moment there is no possibility to reason with the Pellet reasoner on the Gumstix Verdex Pro.

There are other Java DL reasoner, like HOOLET, which cannot be executed on our hardware for the same reasons like Pellet.

3.2 FaCT++

FaCT++ is an open source OWL-DL reasoner written in C++ [20]. For this evaluation the FaCT++ version 1.3.0 is used. After compiling FaCT++ on the embedded hardware three possibilities exist to interact with FaCT++: *(i)* FaCT++ as an HTTP DIG reasoner, *(ii)* FaCT++ as an OWL reasoner with HTTP interface, *(iii)* Standalone FaCT++ with a Lisp-like interface.

First we describe the connection via DIG interface [3], which is a standardized XML interface to Description Logic systems. The DIG language is an XML based representation of ontological entities such as classes, properties, and individuals, and also axioms such as subclass axioms, disjoint axioms, and equivalent class axioms. Unfortunately, there is no useful tool for creating these files for the embedded hardware platform. For that reason the DIG Interface is not considered for our approach.

The next possibility to connect with FaCT++ is the OWL API[8]. The OWL API is a Java API for creating, manipulating and serializing OWL ontologies. Since the OWL API is implemented in Java, the Java native interface (JNI) is necessary. But Java is not completely supported, and the OWL API is not applicable.

The third way to use FaCT++ is as a stand-alone version. For this, an ontology in a special format has to be created, which can be done by an online OWL Ontology converter, e.g. [22]. This procedure works well on a normal x86 computer, on the embedded hardware there are I/O problems while reading the input data. The problem is caused by the ARM architecture.

Since none of the three approaches work, FaCT++ cannot be used for reasoning on embedded systems.

3.3 CEL

CEL [2] is a polynomial-time classifier written in Allegro Common Lisp, which is a closed source, commercial Common Lisp development system and not available for the ARM architecture. On that account CEL unfortunately cannot run on the Gumstix hardware until a version for a free common lisp implementation is published.

4 Rule-Based OWL Reasoning

In the last section we analyzed various reasoners. Unfortunately, non of the above mentioned reasoner can currently be executed on the chosen embedded hardware. One reason is the complexity of the underlying description logic. Therefore, we decided to concentrate on an OWL DL subset to make it tractable.

Fortunately, with the advent of OWL 2 new profiles were introduced by the W3C. Currently the following profiles are defined: OWL 2 EL, OWL 2 QL and OWL 2 RL. OWL 2 EL serves polynomial time algorithms for standard reasoning tasks especially for applications which need very large ontologies. It is based on the description logic EL++. All known approaches for EL reasoning are $O(n^4)$[11]. OWL 2 QL handles conjunctive queries to be answered in LogSpace. It is designed for lightweight ontologies with a large number of individuals. OWL 2 RL is the profile which is used for the reasoner in this paper. It enables polynomial time reasoning using rule engines and operating directly on RDF triples[24]. The complexity of the RL sub language is $O(n^2)$ for standard reasoning. The mathematical proof for that can be done similar to [10]. Therefore OWL 2 RL is the best documented profile. Since it is processable with a standard rule engine, we decided to implement this subset for embedded reasoning.

4.1 CLIPS

In this paper a rule-based system for inferencing and querying OWL ontologies is considered. For this, CLIPS [19] the well-known production rule engine is used because of the small size of CLIPS and its programming language which makes it possible to port it to almost any embedded system. Additionally, it is fast, efficient, and open source.

CLIPS is based on a fact database and production rules. When the conditions of a rule match the existing facts, the rule is placed in the conflict set. A conflict resolution mechanism selects a rule for firing its action which may alter the fact database. Rule condition matching is performed incrementally using the RETE[4] algorithm.

4.2 Rule-Based Reasoning

The concept of our rule-based reasoner is based on a concept-ontology and an instance-ontology which should be reasoned with and a query which should be answered. The rules for the reasoning process can be found in [24]. They are subdivided into 6 parts: *i* the Semantics of Equality, *ii* the Semantics of Axioms about Properties, *iii* the Semantics of Classes, *iv* the Semantics of Class Axioms, *v* the Semantics of Data types, and *vi* the Semantics of Schema Vocabulary. The rules are described by the W3C in the following style:

$Name$	If	$Then$
eq-sym	$T(?x, owl:sameAs, ?y)$	$T(?y, owl:sameAs, ?x)$

Every rule has a name, some if-conditions, and some then-parts. It is also possible to have no if-conditions. This means that the rule should be executed at program start. For

the embedded reasoning process, we transformed all of the 80 rules of the W3C to the CLIPS syntax. In CLIPS the rule from above is expressed with the following syntax:

```
(defrule eq-sym
    (. ?x owl:sameAs ?y)
=>
    (assert (. ?y owl:sameAs ?x))
)
```

4.3 Reasoning Component

We use CLIPS as a key element for our embedded reasoning component. Nevertheless, additional modules are necessary. The complete architecture of the reasoning component is shown in Figure 2. The reasoning component is written in C++ which uses CLIPS functions internally. In order to get the OWL 2 RL functions into our system, the rules defined in [24] by the W3C were converted into CLIPS rules and are stored in the file Rules.clp.

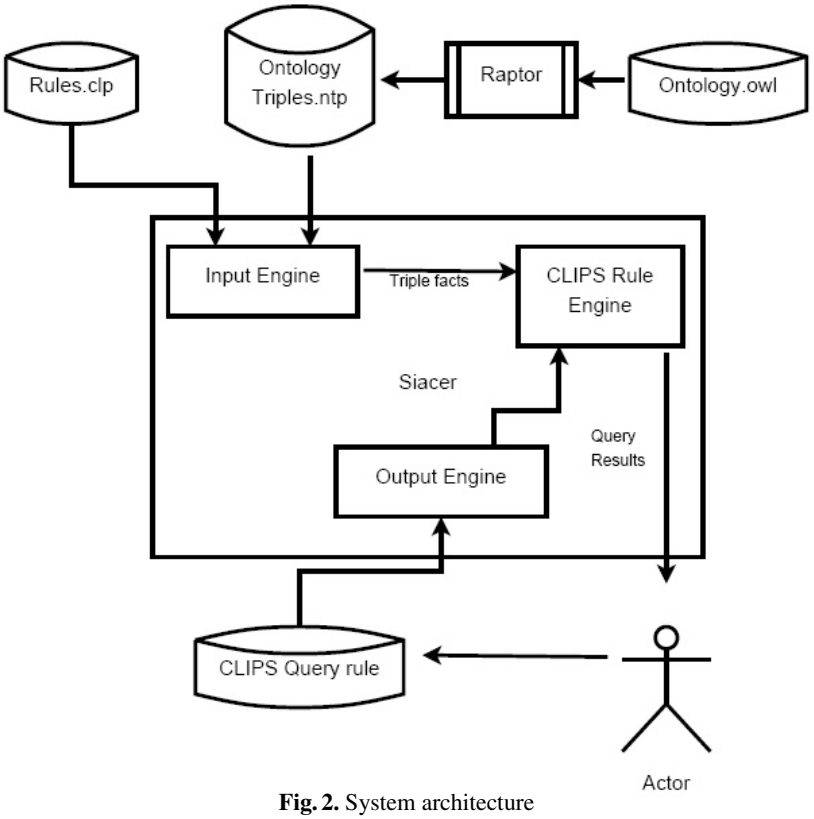

Fig. 2. System architecture

Another important element which must be processed is the ontology. We assume the OWL ontology is available in an XML based format. This OWL file is inserted in the tool Raptor[18]. In this step the ontology is checked for correctness and converted into a format called ntriples[26]. In this format facts are saved as collections of triples with subject, predicate, and object. Together with the rules and the facts CLIPS performs the reasoning. This means that the facts are input data for the OWL RL rules. When all possible facts are added, the reasoning process is finished and a complete materialized knowledge base is created. This means that no new information can be drawn of the existing information and the reasoning process is finished. This knowledge base is based on OWL 2 RL which is included in the rules. The only exception is the semantics of data types. This information causes massive performance problems, because these rules generate too many facts, which finally overflows the memory. Therefore, we only specify which literal is of which data type. This is done by a separate rule.

The test whether a literal is defined from a type which is not contained in the value space is also performed during parsing the OWL RDF file to an ntriples file using the tool Raptor. If there is any syntactical data type error in the ontology, raptor will find it at parsing time. To interact with this knowledge base an interface is necessary to enable the insertion of queries. Executing queries is only possible using a further rule. This rule must be manually created and contains CLIPS code. The syntax of the query is not conform to other knowledge based query languages like SPARQL[25]. Furthermore the rule which contains the query must exist before the reasoning begins.

4.4 Queries

To send a query to the system the simplest way is to create a new rule in CLIPS which fires if the query conditions are passed. For example to identify which student takes the math course the query rule looks like:

```
(defrule abfrage
   (. ?x rdf:type Student)
   (. ?x takesCourse math)
=>
   (printout t "Query 1: " ?x  crlf)
)
```

That prints out appropriate facts that pass the conditions. We are currently working on a SPARQL - CLIPS transformation that allows specifying standard SPARQL queries to our system.

4.5 Formal Analysis

When reasoning is performed it is important to observe the worst-case memory usage. Embedded systems do not have something like swap space. Therefore the complete calculation has to fit in the RAM. For example the reasoner needs 80 MB of memory for a full materialization and only 30 MB for a dynamical result calculation, for an

example query. In this case it is possible to run this calculation dynamically on a system with only 60 MB of free memory. But if the calculated results are cached the size of memory will raise up the 80 MB because the possible answers are the same as in a fully materialized knowledge base and it cannot be assumed that some requests will never be part of the knowledge base. Therefore an approach with caching the data comes with no benefit, although it can save memory. But this memory cannot be used for anything because it might be needed for further requests. The situation is different if the results are not cached after calculating. Here only the worst case memory usage in a single query is important.

To compare the first approach in which all calculations are performed a priori and the second approach which calculates the results only when they are requested, a theoretical analysis is necessary.

When looking at the first approach, the worst case memory usage depends on the given ontology. Since the ontology is completely materialized the memory consumption is constantly maximal. There are no relevant differences in memory consumption depending on the requests. In the second approach the worst-case memory usage after the reasoning process is the same as in the first approach. This means that there is no difference for memory usage if the reasoning is performed a priori or dynamically after a request.

To prove this hypothesis we apply graph theory and use a strongly connected graph. A directed graph is called strongly connected if there is a path from each vertex in the graph to every other vertex. We create the graph from the rules provided by the W3C [24]. The vertices of the graph are OWL 2 RL statements like `subClassOf`, `sameAs`, or `range`. The edges are defined by the rules. If a rule transforms an OWL statement in another an edge in the graph is drawn. To keep the graph as simple as possible only statements are considered which are in the `If` and `Else` part of rules. If a statement appears only in one side it cannot trigger other statements and it is not added by other statements and can be ignored. Statements with the same expression in the `If` and `Else` condition in one rule, are also ignored because this does not affect the connectivity of the graph. For illustration, a sample part of the graph, defined by the rule $prp - rng$:

$Name$	If	$Then$
prp-rng	$T(?p, rdfs:range, ?c)$ $T(?x, ?p, ?y)$	$T(?y, rdf:type, ?c)$

This rule defines the following node relationship in a graph:

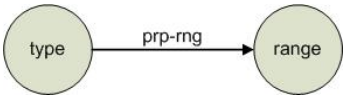

Fig. 3. Representation of the OWL 2 RL rules in a graph

For all OWL rules, the graph is successively built. The resulting graph is strongly connected, which means no information is insignificant for answering a query. Therefore for answering a request nearly all rules and facts are needed. Exceptions are rules which check the consistency of the ontology. The else statement of these rules contains only $false$. To get a correct answer these rules can be ignored theoretically but it is not recommended because they ensure that the ontology is correct. This shows that the second approach needs in the long run generally the same memory amount as the first approach.

There is only a theoretical case in which the worst-case memory usage of the second approach is smaller. When it occurs that the graph is subdivided into two or more subgraphs the worst case memory usage for a given request is the fully materialized subgraph. In this case it can be assured that in no case more memory than the largest fully materialized subgraph is needed and the remaining memory could be used. It is a very unlikely case in which the efficiency of the reasoning could be increased.

5 Evaluation

In this section the evaluation of our rule-based reasoning approach is presented. At first the evaluation environment is introduced. After this, the evaluation results are shown. Generally, we focused on performance and memory usage of the embedded system in contrast to implementing the solution on a system without hardware restrictions.

5.1 Evaluation Environment

The performance of our system has been tested with the Lehigh University Benchmark (LUBM). We use LUBM, because the results can be compared to the results of other reasoners. It is a benchmark developed to evaluate the performance of several Semantic Web repositories in a standardized way. LUBM contains a university domain ontology (Univ-Bench), customizable synthetic data, and a set of test queries. This ontology contains 309 triples and an additional data generator called UBA exists. This tool generates syntactic OWL data based on the Univ-Bench ontology. We only consider the OWL 2 RL part of the ontology for our evaluation purposes.

For our use cases we identified a number of data sets from a few hundred up to a maximum of 25.000 and we started tests with three different benchmark files. Every benchmark contains 309 concept triples. After the reasoning step, some queries from [21] are tested on the data set to determine if the reasoning was correct.

During the tests, the performance of the systems, the memory from the CLIPS reasoner is observed. The unit for the CPU time is mT [million Ticks]. One million ticks are equal to one second calculation time in the cpu and only the time when the CPU works on this process is measured. The time information in the diagrams in the following section is specified in CPU-seconds.

The situation for the memory measurement is similar. Analyzing the memory usage of a reasoning process with and without calculating the CLIPS memory usage shows that the measurement itself needs less than 10KB of memory.

As primary benchmark setup the University0_0 is taken. This is called "bench1" and contains 1657 class instances and 6896 property instances. Together with University0_1 there are 2984 class instances and 12260 property instances, which is called "bench2". This and University1_1 have 4430 class instances and 18246 property instances and called "bench3".

After the reasoning step, the queries 1, 2 and 5 from [21] tested on the data set.

For comparing our results with PC based reasoning, we use an Intel Pentium 4 @ 3200 Mhz and hyper threading, 1 GB of RAM, 4 GB of swap space and Linux (Ubuntu 9.10).

5.2 Evaluation Results

In the following, the performance and memory analysis is presented.

Performance Analysis. Figure 4 shows the complete runtime of the reasoner at the workstation (t_W) and the Gumstix (t_G). The graphs are nearly linear. In fact the runtime can be approximated on the workstation by

$$t_W(x) = 1.9 \cdot 10^{-6}x^2 - 0.016x + 100,$$

in which t_W is the runtime and x the number of triples. On the Gumstix the runtime t_G can be approximated by

$$t_G(x) = 3.4 \cdot 10^{-5}x^2 - 0.4x + 2279.$$

The complexity of the reasoner is polynomial. Therefore the approximated runtime on the Gumstix can be calculated from the time on the workstation by

$$t_G(t_W) = 2.8 \cdot 10^{-3} \cdot t_W^2 + 13 \cdot t_W - 9.8.$$

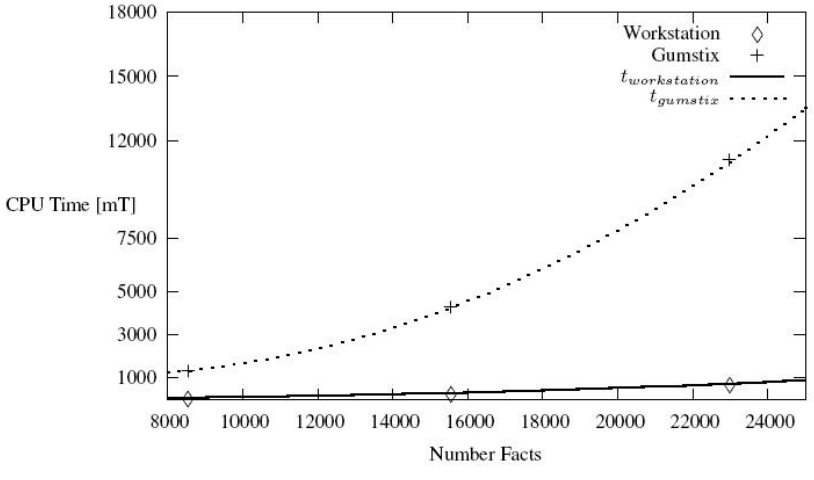

Fig. 4. CPU Time of the rule-based reasoning approach

An execution on the workstation is therefore approximately 12 to 14 times faster than on the Gumstix. This is acceptable considering that the workstation has an eight times higher clock frequency.

For the further evaluation five different measuring points are defined: (1) after all facts are loaded, (2) after the processing, (3) after the first query, (4) after the second query, and (5) after the third query. These points are depicted in the Figures. Most time is needed for the processing, see Figure 5. Here all facts are calculated and all rules have fired. The executions of the queries normally does not need much time, the first query nevertheless needs still a long time. The explanation for this is that CLIPS internally builds some kind of trees or hash tables to find the facts quicker.

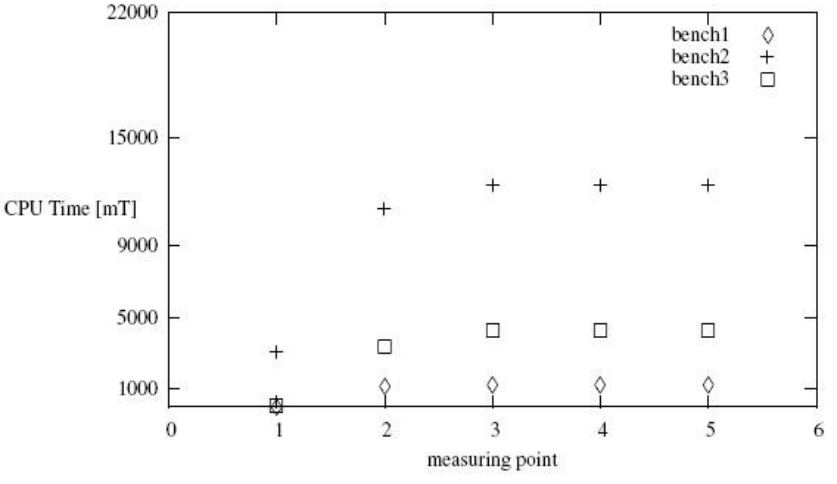

Fig. 5. CPU-Time on Gumstix

Memory Analysis. More important than the performance evaluation is the memory consumption of our approach. The amount of time, the reasoning process takes plays often only a minor role, because it can be done in the background and real time processing is currently not our focus. The memory consumption is therefore more important because it is a criterion for exclusion. If the ontology is too large, it cannot be processed at all. Therefore we need to know how our approach behaves.

In order to reduce the memory usage of CLIPS its memory allocation system was inspected. CLIPS has an integrated garbage collection. This allocates and de-allocates numerous types of data structures during runtime and only reserves new memory if it is actually needed.

Additionally, CLIPS has a special function which tries explicitly to release all memory which is currently not needed. When this function is called after the reasoning process, it can release some memory.

Therefore CLIPS can be forced to use memory economically. This is activated by a flag. If it is enabled, CLIPS will not save information about the pretty printing of facts.

This disables only formatting and can be deactivated without problems. Activating the memory release function saves about 5.8% of memory for all facts and improves our system additionally.

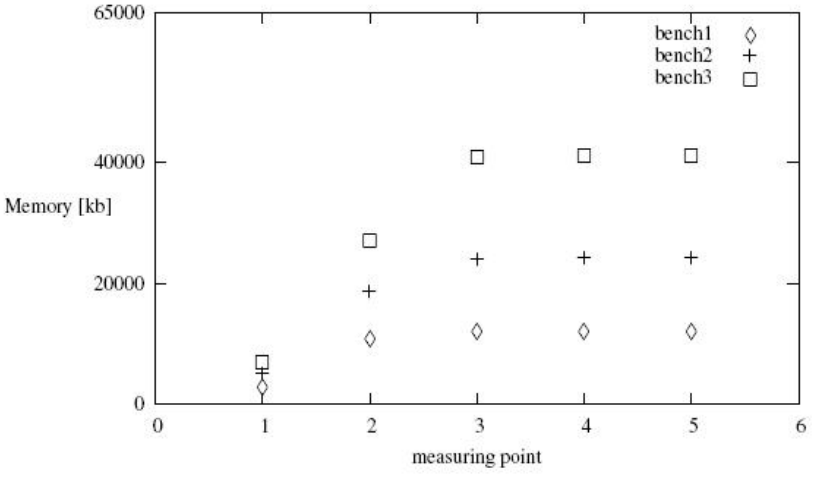

Fig. 6. Memory of rule-based reasoning approach

In Figure 6 the memory usage of the Gumstix can be seen. To calculate the approximate memory usage the formula

$$mem_{poly}(x) = 4.2 \cdot 10^{-5}x^2 + 0.69x + 3459,$$

or for only a few data sets:

$$mem_{lin}(x) = 2x - 5565,$$

in which mem is the size of needed memory in kilo bytes and x the number of triples. It is possible to calculate how many triples can be reasoned with a specified memory size. For this approximation the following formula holds:

$$triples(x) = -5.3 \cdot 10^{-6}x^2 + 0.79x - 357.$$

This is very important for us, since we plan to implement various industrial applications and with this information we can tailor our needed hardware accordingly.

5.3 Evaluation Summary

In the end of this section we provide some detailed facts of our evaluation. This is summarized in Table 1.

Table 1. Detailed evaluation results

Number of facts	CPU time System II	Mem System II	CPU time System I	Mem System I
8864	8.5 mT	7470 kb	80.24 mT	11194 kb
15557	27.54 mT	12686 kb	214.46 mT	18932 kb
22991	59.24 mT	18463 kb	437.89 mT	27482 kb

On the right hand side the data for our System I is shown. We made some additional performance tuning, e.g. compiler options, rule rewriting. This data is shown on the left side, entitled with System II. The performance gain is considerable. The calculation time has been reduced to about 12% from the original calculation time, the memory usage is reduced to about 67%.

Finally, we compared the embedded with a desktop approach. A test ontology took us 8.5 seconds on the embedded hardware. If we use Fact++, the ontology takes about 6 seconds to process the ontology. Thus, the implementation is only a bit slower on the embedded hardware, which is a really good result.

6 Embedded Reasoning Applications

We apply the above presented reasoning approach in some sample applications. Key concept is a digital product memory, which stores product relevant data of the complete product life cycle. With the reasoning approach we work with the stored data, analyze them, infer new date and perform appropriate actions.

6.1 Predictive Maintenance

A possible benefit of a product memory is the detection of technical issues when the product is already deployed in the field. Since the product memory contains a lot of sensor data, an analysis is useful for diagnostics and predictive maintenance.

An example is an industrial robot that moves a work piece from one machine to another. The robot is equipped with an acceleration sensor. While the robot moves the work pieces, the sensor values are continuously (every 20 ms) recorded in the digital product memory of the robot. On basis of this data, an evaluation of the robots product memory can detect, if the robot still works properly or whether an execution has occurred. A more sophisticated method is not only to detect errors if they have occurred, but to detect slight changes in the normal behavior. An anomaly detector should report if new measured data of certain sensors have a different pattern or are beyond the normal sensor values. The basis for an abnormality detector are sensor values and a domain ontology which specifies normal behavior. With the embedded reasoner we try to perform a mapping of sensor value to machine states. On this basis it can be decided, if the current state is an allowed one or whether an alarm must be triggered.

6.2 Decentralized Manufacturing Control

The need of products which are tailored to customers' needs, results in a reduction of the lot size and implies a more flexible production and the associated processes. In

the course of an increased diversification the changeover time will be a critical cost factor. This essentially needed flexibility is hard to realize with traditional central control architectures that can be found in nowadays automation systems. One solution is a decentralized production control, done by the product itself. The goal is to operate autonomous working stations and all data that is needed to assemble the product is kept on the product in the digital product memory. If a product enters the vicinity of a working station the necessary machine configuration information is sent from the product memory to the machine and the station accomplishes the necessary tasks.

Thus, if a product is assembled in multiple steps, the necessary data is written to the product memory when the order is entered into the order system. The data contains the description of the single production steps with all its parameters, e.g. the position of bore holes or welds, the used materials, the size or the color. The memory is read by the machine (e. g. via RFID) and the machine is parameterized and set up accordingly. If a production step is finished, the product itself is responsible for the routing to the next station. Depending on its weight and shape either an automated guided vehicle or a conveyer belt can be used. This inference step is done by our embedded reasoning component and is therefore an essential step for next generation automation systems.

6.3 Situation Recognition in Assisted Living

Due to the dramatic growth of elderly population, we additionally aim at research of near-future systems providing elderly people a safe and comfort life during daily living. The people have the possibility to stay either at home or still being mobile and could be relatively healthy or having some physical disabilities or medical liabilities. The diversity and breadth of these scenarios and realistic approaches make this target challenging, assuming the use of various medical devices, different home and mobile systems, heterogeneous and data-rich environments.

A core functionality of such assisted living systems is the conclusion of knowledge about the activities of the user and the current situation in the environment from low-level sensor data and to plan the appropriate short-term and long-term reaction. This reaction on the situations and activities to be recognized are based on situational models that have to keep reasoning system safe for people using it and preserve the relevancy of reactions to the situations respectively. Typical reaction aims at recognizing and/or preventing an urgency situation, defined in form of situation rules within ontology-based situation understanding system.

A small embedded device which can be easily worn on the wrist was chosen. Sensors are needed that monitor the health state of a person. Thus, the product memory becomes a patient memory. It records the vital signs and activities of the senior, does some basic evaluation and executes finally necessary tasks, e.g. informing the patient or sending text messages to doctors.

Additional rules specify the interaction with the environment, because other objects in the smart home environment may be equipped with a product memory as well. This eases the detection of certain situations, e.g. when the temperature of a kettle changes, maybe tea is prepared. Such activities and interdependencies can be optimally expressed with rules.

7 Conclusion

This paper shows that it is currently not possible to use a standard OWL reasoner on the specified embedded system, because existing OWL reasoner are too resource intensive and difficult to port to embedded and source restricted architectures. Therefore, we use a rule-based approach to achieve OWL reasoning on embedded devices. We apply OWL 2 RL rules to the rule engine CLIPS to accomplish the OWL behavior. We integrate the rule engine in an embedded architecture to enable ontology processing in an comfortable and easy way. Several methods for reducing calculation time and memory consumption were reviewed and selected methods were implemented. The created reasoner is compatible with OWL 2 RL with the exception of the semantics of data types which are deleted due to performance reasons. Performance evaluations and the results of our approach are very satisfying. Additionally, we are able to calculate the necessary amount of memory for a given number of facts.

In the future, we will extend our embedded reasoning system. To improve the expressive power of the reasoner, it can be extended for reasoning with OWL 2 EL using the work from [11]. Although this will increase the calculation time and the complexity of reasoning, more complex problems can be addressed.

References

1. Ali, S., Kiefer, S.: μOR – A Micro OWL DL Reasoner for Ambient Intelligent Devices. In: Abdennadher, N., Petcu, D. (eds.) GPC 2009. LNCS, vol. 5529, pp. 305–316. Springer, Heidelberg (2009)
2. Baader, F., Lutz, C., Suntisrivaraporn, B.: CEL — A polynomial-time reasoner for life science ontologies. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 287–291. Springer, Heidelberg (2006)
3. Bechhofer, S.: The DIG description logic interface: DIG/1.1. Tech. rep., University of Manchester (2003)
4. Forgy, C.L.: Rete: A fast algorithm for the many pattern/many object pattern match problem. Department of Computer Science, Carnegie-Mellon University, Pittsburgh (2003)
5. Grosof, B., Dean, M., Ganjugunte, S., Tabet, S., Neogy, C.: Sweetrules homepage (2005), `http://sweetrules.semwebcentral.org/`
6. Gumstix: Gumstix website (2010), `http://www.gumstix.com`
7. Hähnle, R.: Tableaux and Related Methods. Handbook of Automated Reasoning (2001)
8. Horridge, M.: OWL2 API (2010), `http://owlapi.sourceforge.net/`
9. Kleemann, T., Sinner, A.: KRHyper - in your pocket. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 452–457. Springer, Heidelberg (2005)
10. Krötzsch, M.: Efficient inferencing for OWL EL. In: Janhunen, T., Niemelä, I. (eds.) JELIA 2010. LNCS, vol. 6341, pp. 234–246. Springer, Heidelberg (2010)
11. Krötzsch, M.: Efficient inferencing for the description logic underlying OWL EL. Institut AIFB, KIT, Karlsruhe (2010)
12. Krötzsch, M., ul Mehdi, A., Rudolph, S.: Orel: Database-driven reasoning for OWL 2 profiles. In: Int. Workshop on Description Logics (2010)
13. Meditskos, G., Bassilades, N.: A rule-based object-oriented OWL reasoner. In: IEEE Transactions on Knowledge and Data Engineering (2008)
14. Meditskos, G., Bassiliades, N.: DLEJena: A practical forward-chaining OWL 2 RL reasoner combining Jena and Pellet. Web Semantics 8(1), 89–94 (2010)

15. Jang, M., Sohn, J.-C.: Bossam: An Extended Rule Engine for OWL Inferencing. In: Antoniou, G., Boley, H. (eds.) RuleML 2004. LNCS, vol. 3323, pp. 128–138. Springer, Heidelberg (2004)
16. Parsia, B., Sirin, E.: Pellet: An OWL DL Reasoner. In: Third International Semantic Web Conference-Poster (2004)
17. Parsia, B., Sirin, E., Grau, B.C., Ruckhaus, E., Hewlett, D.: Cautiously approaching SWRL. Tech. rep., University of Maryland (2005)
18. Raptor: Raptor website (2010), http://librdf.org/raptor/
19. Riley, G.: CLIPS (2010), http://clipsrules.sourceforge.net/
20. Tsarkov, D., Horrocks, I.: FaCT++ Description Logic Reasoner: System Description. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 292–297. Springer, Heidelberg (2006)
21. University, L.: LUBM website (2010), http://swat.cse.lehigh.edu/projects/lubm/query.htm
22. Volz, R.: FactConverter (2010), http://phoebus.cs.man.ac.uk:9999/OWL/Converter
23. World Wide Web Consortium (W3C): OWL-Lite (2010), http://www.w3.org/TR/2004/REC-owl-features-20040210/#s3
24. World Wide Web Consortium (W3C): OWL profiles (2010), http://www.w3.org/TR/owl2-profiles/
25. World Wide Web Consortium (W3C): SPARQL (2010), http://www.w3.org/TR/rdf-sparql-query/
26. World Wide Web Consortium (W3C) OWL Working Group: nTriples Format (2010), http://www.w3.org/TR/rdf-testcases/#ntriples