Embedded $\mathcal{EL}^+$ Reasoning on Programmable Logic Controllers

Stephan Grimm, Michael Watzke, Thomas Hubauer, Falco Cescolini

Siemens AG, Corporate Technology, Munich, Germany

Abstract. Many industrial use cases, such as machine diagnostics, can benefit from *embedded reasoning*, the task of running knowledge-based reasoning techniques on embedded controllers as widely used in industrial automation. However, due to the memory and CPU restrictions of embedded devices like programmable logic controllers (PLCs), state-of-the-art reasoning tools and methods cannot be easily migrated to industrial automation environments. In this paper, we describe an approach to porting lightweight OWL 2 EL reasoning to a PLC platform to run in an industrial automation environment. We report on initial runtime experiments carried out on a prototypical implementation of a PLC-based $\mathcal{EL}^+$-reasoner in the context of a use case about turbine diagnostics.

1 Motivation

Embedded controllers are extensively used in industrial environments for operating and monitoring technical machinery. They can be placed near the underlying machine's sensors and actuators to perform local computation tasks on-site with quick reaction times. Much of the computation required for process control or condition monitoring on industrial facilities is thus performed on computational devices that are embedded in the surrounding field in a decentralized manner, which is in contrast to the central computation performed on PCs in areas such as business applications. When embedded controllers run knowledge-based systems that apply the computational techniques of automated reasoning, we speak of *embedded reasoning*.

One particular use case for embedded reasoning is on-site diagnostics of industrial facilities based on diagnostic knowledge models for technical devices. There, embedded controllers can perform reasoning on sensor data in the context of diagnostic background knowledge to detect a machine's faulty behavior or to trace the causes thereof. Single diagnostic results reasoned over local device models can be combined to yield an overall diagnosis for the whole facility under investigation. An example scenario is reactive diagnostics for steam and gas turbines in electrical power plants, where early detection of anomalies in running a plant helps avoid costly downtimes and repair of turbine machinery.

In contrast to business applications that run on PCs, knowledge-based modeling and reasoning techniques, such as those used in the Semantic Web, are not readily available in an automation environment. Embedded controllers are

subject to tight limitations on computational power or memory size and often run proprietary operating systems for which no standard reasoning tools are available, as reported in [11]. There, it was also observed that standard implementations of state-of-the-art reasoning algorithms designed for a PC environment cannot easily be migrated to embedded controllers due to their hardware restrictions. Hence, there is the need for porting reasoning techniques to embedded environments.

In the area of industrial automation, programmable logic controllers (PLCs) are the most widespread type of embedded controller used for many real-time automation tasks. Unlike general purpose computers, PLCs are designed for control of industrial machinery and employ a processing scheme based on fixed length execution time cycles to meet realtime requirements. On the other hand, the Web Ontology Language (OWL) and its underlying reasoning services are well suited for representing structural diagnostic models of technical devices and their dependencies. Especially the OWL 2 profiles, which provide restricted language variants with faster computation, are well suited for use in resource-constrained embedded systems. Hence, it would be beneficial to port (lightweight) OWL 2 reasoning to PLCs in order to provide for on-site processing of knowledge models in the context of machine diagnostics and many other industrial use cases.

The implementation of state-of-the-art OWL reasoning on a PLC platform, however, has various difficulties. On the one hand, the typical methods used for OWL reasoning, such as tableaux or consequence-driven procedures, are designed to use dynamic data structures like tree models or concept mappings that are continuously expanded, whereas on PLCs only static memory management is available, and restricted to very limited size. Moreover, the dynamic runtime behavior of standard reasoning algorithms operating on a set of tableau- or consequence-like rules over an indeterminate number of iterations does not fit the strictly cyclic processing pattern that PLCs follow.

In this paper, we present an approach to porting consequence-driven ontology classification in the OWL 2 EL language profile, as e.g. described in [1, 9], to PLC platforms in the context of an industrial diagnostics use case. We propose an architecture for using PLCs in combination with standard OWL tools hosted on a PC environment, and we show how to overcome the architectural discrepancies between standard implementation platforms for such procedures and the PLC world. In particular, we present the use of a compact and efficient axiom representation based on fixed length data structures, and we introduce an interruption-safe saturation mechanism that fits a PLC's cyclic processing paradigm. We also report on initial runtime experiments carried out on our prototypical implementation of a PLC-based embedded OWL 2 EL reasoner covering the description logic $\mathcal{EL}^+$ to show the feasibility of our approach. This also includes a formulation of a turbine diagnostic problem in terms of an $\mathcal{EL}^+$ ontology and the use of classification for deriving diagnoses. By this, we contribute to making Semantic Web reasoning technology available on industrial automation platforms dominated by the PLC paradigm.

2 Preliminaries

2.1 Programmable Logic Controllers

In the automation domain, *programmable logic controllers* (PLCs) are a flexible means for solving a control problem. In contrast to control wiring contactors and relays, a PLC uses a specific program to realize a control task. PLCs are the central part of today's automation solutions for control of machines or plants. They are networked to central and/or distributed I/O modules connected to sensors and actuators, display devices for monitoring and operation, as well as a programming device or SW development environment for programming and configuration [5].

In order to behave like hardwired logics consisting of contactors and relays, where logical operations are effectively executed in parallel, a PLC executes its sequential user program cyclically. Within each execution cycle, a PLC first stores a snapshot of its current input signals in a process image input table in the CPU memory. The CPU executes the user program step-by-step, one command at a time. During command execution, signal states are taken from the process image input table, processed and stored in the process image output table. At the end of an execution cycle, when the user program has finished, a PLC sets output signals according to their values in the process image output table. The shorter the cycle time of the PLC user program, the more frequently input signals are read and output signals are set. PLC commands can be clustered into binary logic operations (e.g. AND, OR), memory functions (e.g. bit assignment), move functions (e.g. register load/store from/to memory), counters and timers [4].

The work presented in this paper is based on Siemens' SIMATIC automation system. The SIMATIC automation system is a family of different products [5] built around the SIMATIC S7 controller. There are several PLC types available, addressing a range of performance and availability requirements. Usually, a power supply unit, CPU module and I/O modules are installed in a mounting rack and form a *station*. The SIMATIC DP distributed I/O system allows for input/output modules to be installed nearby a machine, connected to a PLC by means of a PROFIBUS network. SIMATIC HMI (Human Machine Interface) provides a number of different products for visualization, ranging from simple text displays to interactive touch screen panels. SIMATIC NET provides networking of all SIMATIC stations for data exchange, programming and configuration. OPC[1] is an interface standard for communication between several stations or to PCs. STEP 7 is the standard tool for configuration and programming of the user program in any of the available languages (Ladder Logic (LAD, a representation similar to relay logic diagrams), Function Block Diagram (FBD), Statement List (STL, an Assembler-like language) or Structured Control Language (SCL, a high-level language similar to Pascal)).

In addition to hardware-based PLCs, SIMATIC WinAC provides software that emulates a PLC on a standard Industrial PC (IPC) running Microsoft Windows.

[1] `http://www.opcfoundation.org/`

2.2 Ontologies and the $\mathcal{EL}^+$ Description Logic

In the area of knowledge-based systems and the Semantic Web, *ontologies* are the key artifact for representing and reasoning about knowledge of a specific domain, such as turbine machinery. Compared to subsymbolic approaches for capturing domain knowledge, ontologies allow for an explicit representation of domain concepts and their interrelations as well as for automated reasoning based on the clear semantics of logic. In this work, we build on the prominent *OWL Web Ontology Language* which is semantically founded on the description logic (DL) formalism [2]. In its second version, OWL comes with a number of so-called profiles that offer language variants with reduced expressivity and better computational properties. In particular, we use (a part of) OWL 2 EL [13], which is based on the tractable description logic $\mathcal{EL}^+$ [1]. Research has shown that the OWL 2 EL profile is especially well suited for representing diagnostic models, including the well-known SNOMED-CT[2] and GALEN[3] medical ontologies.

The basic building blocks for representing knowledge in $\mathcal{EL}^+$ are (atomic) *concepts*, such as *GasTurbine*, and *roles*, such as *hasComponent*. An $\mathcal{EL}^+$ *ontology* $\mathcal{O}$ is a set of concept inclusion axioms of the form $\mathit{GasTurbine} \sqsubseteq \mathit{Turbine}$ (stating that every *GasTurbine* is a *Turbine*) and role inclusion axioms of the form $\mathit{directlyControls} \circ \mathit{hasSubComponent} \sqsubseteq \mathit{controls}$ (stating that control is propagated over the partonomy of a system). The symbol $\sqsubseteq$ can be read as logical implication. The sets $\mathsf{N}_C^{\mathcal{O}}$ and $\mathsf{N}_r^{\mathcal{O}}$ denote the concept and role names that occur in an ontology $\mathcal{O}$ and form its *signature*. Complex concepts can be composed from simpler concepts and roles using the concept constructors $\sqcap$ and $\exists$. An example for a complex $\mathcal{EL}^+$ concept inclusion axiom is $\mathit{GasTurbine} \sqsubseteq \exists\, \mathit{hasComponent}.\mathit{Combustor}$, stating the fact that any gas turbine has a combustor as its component. For further details on $\mathcal{EL}^+$, see [2, 13].

The main features that $\mathcal{EL}^+$ lacks compared to OWL 2 EL are concept disjointness by means of the bottom concept $\bot$, and nominals $\{o\}$, which allow expression of facts about particular instances of a concept. The main reasoning task in $\mathcal{EL}^+$ (and a central reasoning task for description logics in general) is called *classification*, it builds up a complete inferred subsumption hierarchy of all atomic concepts mentioned in an ontology $\mathcal{O}$. This can be understood as deriving deductively all concept subsumptions $A \sqsubseteq B$ that are a logical consequence of $\mathcal{O}$, expressed formally by $\mathcal{O} \models A \sqsubseteq B$, for $A, B \in \mathsf{N}_C^{\mathcal{O}}$

2.3 Consequence-driven Reasoning in $\mathcal{EL}^+$

Classification of $\mathcal{EL}^+$ ontologies can be realized efficiently (i. e., in polynomial runtime) using a so-called consequence-driven reasoning approach, as described in [1] or [9]. Since our embedded reasoning method for PLCs also employs a consequence-driven approach, we give an introduction to consequence-driven reasoning for $\mathcal{EL}^+$, following the approach from [1], in this section.

[2] `http://www.nlm.nih.gov/research/umls/Snomed/snomed_main.html`
[3] `http://www.openclinical.org/prj_galen.html`

$$\textbf{(CR1)}\ \frac{C \sqsubseteq C_1}{C \sqsubseteq D}\ [C_1 \sqsubseteq D \in \mathcal{O}] \quad \textbf{(CR2)}\ \frac{C \sqsubseteq C_1 \qquad C \sqsubseteq C_2}{C \sqsubseteq D}\ [C_1 \sqcap C_2 \sqsubseteq D \in \mathcal{O}]$$

$$\textbf{(CR3)}\ \frac{C \sqsubseteq C_1}{C \sqsubseteq \exists\, r.D}\ [C_1 \sqsubseteq \exists\, r.D \in \mathcal{O}] \quad \textbf{(CR4)}\ \frac{C \sqsubseteq \exists\, r.C_1 \qquad C_1 \sqsubseteq C_2}{C \sqsubseteq D}\ [\exists\, r.C_2 \sqsubseteq D \in \mathcal{O}]$$

$$\textbf{(CR5)}\ \frac{C \sqsubseteq \exists\, r.D}{C \sqsubseteq \exists\, s.D}\ [r \sqsubseteq s \in \mathcal{O}] \quad \textbf{(CR6)}\ \frac{C \sqsubseteq \exists\, r_1.C_1 \qquad C_1 \sqsubseteq \exists\, r_2.D}{C \sqsubseteq \exists\, s.D}\ [r_1 \circ r_2 \sqsubseteq s \in \mathcal{O}]$$

Fig. 1. Completion rules for $\mathcal{EL}^+$

In a preprocessing step, the input ontology $\mathcal{O}$ is normalized into a semantically equivalent ontology $\|\mathcal{O}\|$ which contains only axioms of the form **(NF1)** $C_1 \sqsubseteq D$, **(NF2)** $C_1 \sqcap C_2 \sqsubseteq D$, **(NF3)** $C_1 \sqsubseteq \exists\, r.D$, **(NF4)** $\exists\, r.C_2 \sqsubseteq D$, **(NF5)** $r \sqsubseteq s$, and **(NF6)** $r_1 \circ r_2 \sqsubseteq s$ for atomic concepts C_1, C_2 and D, and roles $r, 2, r_1$, and r_2. Next, starting from the trivially true tautologies $C \sqsubseteq \top$ and $C \sqsubseteq C$ (for each concept name C in $\|\mathcal{O}\|$), the consequence-driven classification algorithm derives additional valid subsumptions that are also logical consequences of $\|\mathcal{O}\|$ based on the information stated explicitly in $\|\mathcal{O}\|$. This is done based on a set of so-called completion rules shown in Figure 1, which can be understood as follows: If the premise(s) above the horizontal line are known to be consequences of $\|\mathcal{O}\|$ and the axiom in square brackets is contained in the ontology $\|\mathcal{O}\|$, then the conclusion below the horizontal line is a valid consequence of $\|\mathcal{O}\|$ as well. Note that there is exactly one completion rule for each type of normalized axiom; the application of completion rule **(CR*i*)** is therefore guarded by the presence of an appropriate **(NF*i*)**-axiom in $\|\mathcal{O}\|$. When the completion process terminates (i. e. no more rules are applicable), all valid subsumptions have been derived.

The procedure outlined above is typically realized based on two mappings $S : \mathsf{N}_C^{\mathcal{O}} \mapsto \mathcal{P}(\mathsf{N}_C^{\mathcal{O}})$ and $R : \mathsf{N}_r^{\mathcal{O}} \mapsto \mathcal{P}(\mathsf{N}_C^{\mathcal{O}} \times \mathsf{N}_C^{\mathcal{O}})$ which make explicit the subsumptions derived so far[4]. For this, observe that the premises and conclusions of the completion rules only consist of two types of axioms, namely $C \sqsubseteq D$ and $C \sqsubseteq \exists\, r.D$. These axioms are represented in the mappings as follows: $C \sqsubseteq D$ corresponds to a mapping entry $D \in S(C)$, and $C \sqsubseteq \exists\, r.D$ is represented by $\{C, D\} \in R(r)$. Following the intuition given above, these mappings are initialized by $S(C) = \{C, \top\}$ for each concept name C in $\|\mathcal{O}\|$ and $R(r) = \emptyset$ for each role name r in $\|\mathcal{O}\|$. Testing the premises in the rules from Figure 1 then corresponds to lookups in the mappings S and R. Analogously, the conclusions correspond to the addition of elements to the respective mapping. The classification result of the original input ontology $\mathcal{O}$ can directly be read from S after termination.

The algorithm presented above has been implemented successfully in the CEL[5] reasoner. Naturally, such a practical implementation requires additional

[4] $\mathcal{P}(\cdot)$ denotes the powerset operator

[5] `http://lat.inf.tu-dresden.de/systems/cel/`

considerations for minimizing memory usage and maximizing performance; in [3], an overview of the respective decisions made for the CEL system is given. Most recent developments for consequence-driven OWL 2 EL classification are reported in [9] for the ELK[6] reasoner system. One central optimization, the representation of entities using integer numbers, has also been employed for our embedded implementation.

2.4 Related Approaches to Embedded Reasoning

Although there are a variety of approaches for reasoning on embedded (or mobile) devices, we are not aware of any other research on implementing reasoning algorithms for PLCs or similarly constrained devices. The Pocket KRHyper system[7], for example, realizes DL reasoning via a hyper tableau calculus for first-order logic [12]. It is implemented as a Java 2 Mobile Edition[8] application and thereby overcomes any need to address hardware issues. On the downside, KRHyper can only be used on systems for which a Java Virtual Machine is available; PLCs and other automation systems do not fall into this category and their cycle-based paradigm makes an easy port unlikely. More closely related to the approach presented in this paper is [11], whose authors propose a RETE-based OWL 2 RL reasoner for an embedded system with a 400 MHz CPU and 64 MB of RAM that features a Linux operating system. Although standard DL reasoners such as Pellet[9] or FaCT++[10] cannot be used sensibly in this environment either, the availability of a standard operating system permits the authors to use standard programming paradigms. Interestingly, the authors also considered the possibility of using consequence-driven reasoning (namely CEL) on their platform; this was impractical due to the lack of an Allegro Common Lisp environment. In a very similar fashion, Bossam[11] uses the RETE algorithm to realize an OWL reasoner with a small memory footprint [8], but it requires a Java Virtual Machine just like the Pocket KRHyper system.

3 Industrial Application of Embedded $\mathcal{EL}^+$ Reasoning

In this section, we describe the use of embedded $\mathcal{EL}^+$ reasoning for industrial diagnostics, and we sketch a suitable architecture for such reasoning in automation environments.

3.1 Diagnostics of Technical Devices with Embedded Reasoning

The main motivation for our research on embedding reasoning on PLCs is industrial diagnostics. In particular, we consider a use case involving reactive diagnos-

[6] http://code.google.com/p/elk-reasoner/
[7] http://mobilereasoner.sourceforge.net/
[8] http://www.oracle.com/technetwork/java/javame/index.html
[9] http://clarkparsia.com/pellet/
[10] http://owl.man.ac.uk/factplusplus/
[11] http://bossam.wordpress.com/

tic reasoning for steam and gas turbines. In the scenario considered here, a power generation command and control center (CCC) is responsible for a large number of plants scattered all over the globe. Each plant typically comprises one to three turbines which are equipped with several hundred sensors each, mounted to the numerous components of the turbine. All sensors typically provide measurements at a rate between 0.1 and 1.0 Hertz, sending their data to the CCC. Engineers at the CCC have the task of monitoring the turbines, identifying faults, and taking reactive measures for preventing subsequent damage.

To illustrate how diagnostic knowledge in the turbine domain can be represented using $\mathcal{EL}^+$ description logic axioms, consider the expert statement *"fan blade vibrations and fluctuations in the combustion chamber's temperature indicate a can flame failure"*. Based on a diagnostic meta model that relates the unknown *FaultMode* that a certain *System operatesIn* to the observable *Symptoms* it *shows* (see Figure 2), the above statement can be formalized as follows:

$$\begin{gathered} \mathit{System} \sqcap \exists\, \mathit{isInfluencedBy}.(\mathit{Fan} \sqcap \exists\, \mathit{shows}.\mathit{Vibrations}) \sqcap \\ \exists\, \mathit{isInfluencedBy}.(\mathit{CombChamber} \sqcap \exists\, \mathit{shows}.\mathit{TempFluctuations}) \\ \sqsubseteq\ \exists\, \mathit{operatesIn}.\mathit{CanFlameFailure} \end{gathered}$$

To understand how classification can be employed for diagnostics in this case, assume that the compositional model of the turbine also contains the axioms $\mathit{Turbine} \sqsubseteq \exists\, \mathit{hasComponent}.\mathit{Fan}$ and $\mathit{Turbine} \sqsubseteq \exists\, \mathit{hasComponent}.\mathit{CombChamber}$ describing the turbine components, and the role inclusion axiom $\mathit{hasComponent} \sqsubseteq \mathit{isInfluencedBy}$ stating that subcomponents influence their supercomponents. This static knowledge about a turbine is complemented by additional axioms based on sensor measurements: If the vibration sensor mounted at the fan hub detects vibrations exceeding a threshold, a corresponding axiom $\mathit{Fan} \sqsubseteq \exists\, \mathit{shows}.\mathit{Vibrations}$ is added. Analogously, temperature fluctuations at the combustion chamber lead to the addition of the axiom $\mathit{CombChamber} \sqsubseteq \exists\, \mathit{shows}.\mathit{TempFluctuations}$. Once this knowledge about currently observed symptoms has been added, the resulting ontology entails the axiom $\mathit{Turbine} \sqsubseteq \exists\, \mathit{operatesIn}.\mathit{CanFlameFailure}$ (among others).[12] By triggering the classification process either regularly based on a timer or after every addition of a symptom, a diagnostic system can therefore determine the state of the turbine based on a formal model of the system and its

[12] Since $\mathcal{EL}^+$ does not support Aboxes, we must encode facts as terminological axioms. A more natural modelling approach using Abox axioms could be realized in $\mathcal{EL}^{++}$.

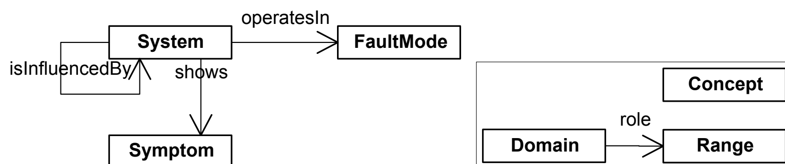

Fig. 2. A basic diagnostic model formalized in $\mathcal{EL}^+$ (c.f. [6]).

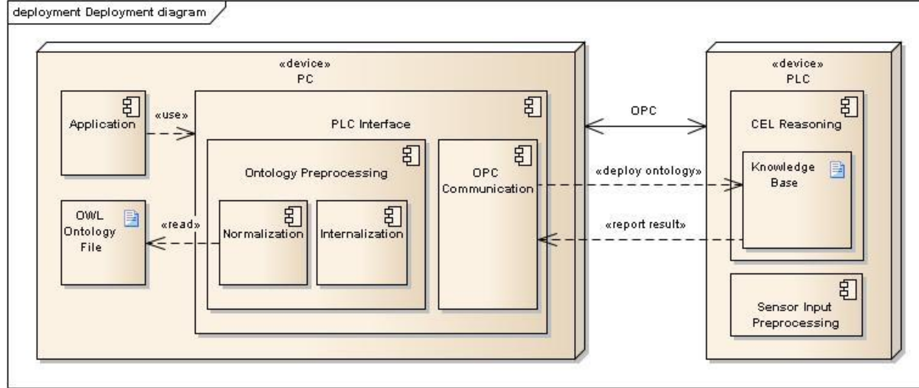

Fig. 3. An architecture for embedded reasoning on PLCs.

diagnoses. If a faulty situation is detected, this can be signaled to the operator at the CCC, but also be used as input for other diagnostic components in a hierarchical setting.

Similar models are used in existing solutions such as [7, 10], which typically assume that standard computer systems can be used for evaluating the models, e.g. by moving all data to a central store. To address the special restrictions posed by a PLC environment, we present a dedicated approach to embedded diagnostic reasoning, as follows.

3.2 An Architecture for Embedded $\mathcal{EL}^+$ Reasoning

Figure 3 shows an architecture for embedding $\mathcal{EL}^+$ reasoning in a PLC. Its core part is the *CEL Reasoning* component, running on a PLC, which implements consequence-driven $\mathcal{EL}^+$ reasoning based on the CEL approach from [1]. The *Knowledge Base* subcomponent contains both application-specific background knowledge as well as axioms reflecting current PLC sensor input.

The *CEL Reasoning* component is based on a very compact ontology representation suitable for resource-constrained embedded devices, such as a PLC, which are limited in computing power and memory. In order to be deployed on a PLC, any OWL ontology has to be converted to a compact ontology representation in a preprocessing step during development. For this purpose, a *PLC Interface* component running on a PC provides for ontology preprocessing as well as OPC-based communication between the PC and the PLC. The *Ontology Preprocessing* subcomponent reads a use case specific OWL ontology from a file. Next, its *Normalization* subcomponent normalizes ontology axioms to syntactically match the axiom types **(NFi)** used in the derivation rules of the CEL algorithm. In a subsequent ontology pre-processing step, the *Internalization* subcomponent maps ontology literals (concept names, role names) to corresponding integer numbers, which are used as indices in an array-based, compact ontol-

ogy representation on the PLC. Via OPC communication, the normalized and internalized ontology is finally deployed on the PLC.

During runtime, an application running on a PC, or alternatively a use case specific user program running on another superordinate PLC, may trigger the CEL Reasoning component on the PLC by sending an appropriate signal across the OPC communication link. Before the reasoning process, a *Sensor Input Processing* component of the PLC reads input signals from sensors, maps them to respective ontology axioms and adds them to the knowledge base. Accordingly, axioms that are no longer valid due to changes in sensor signals are being removed. When the reasoning process has been finished (indicated by an appropriate flag), the application or user program is reported relevant parts of the reasoning result, depending on the use case to be realized.

4 Consequence-driven $\mathcal{EL}^+$ Reasoning on a PLC

In this section, we describe aspects of the implementation of consequence-driven $\mathcal{EL}^+$ reasoning in PLC environments, where restrictions preclude a direct implementation of the CEL algorithm as described in [1, 3].

4.1 Difficulties with Reasoning Algorithms in PLC Environments

PLCs have very limited computing power and working memory capacity compared to contemporary desktop PCs. This, and their architectural design tailored to automation tasks, imposes various difficulties on the task of porting algorithms for efficient embedded OWL reasoning to a PLC environment.

One difficulty is the lack of dynamic data structures and memory allocation mechanisms. DL reasoning methods like tableaux or consequence-driven saturation procedures dynamically expand their data structures during the construction of tree-like models or derivation structures and partly also use backtracking to discard previously computed intermediate results. The CEL algorithm, in particular, operates on the mappings S and R, which are dynamically expanded by derived axioms during ontology classification. PLC platforms, however, only support static memory management based on a fixed size memory block scheme. Therefore, the consumption of PLC memory needs to follow a strategy of efficient memory layout based on upper bounds for the number of axioms that can potentially be derived in S and R, such that the very limited overall memory is not quickly exceeded by allocating sparsely populated blocks.

Another difficulty is rooted in the cyclic processing paradigm of PLCs. The repeated processing of a memory image of input signals within a fixed time slice is contrary to algorithms that expand their results non-deterministically within processing times that depend on intermediate results. The CEL algorithm, in particular, has two sources of dynamic expansion of results. Firstly, the application of the derivation rules **(CR*i*)** is triggered by both original axioms in the ontology and derived consequences in S and R, while the sequence of their application has an influence on the overall number of derivation steps required

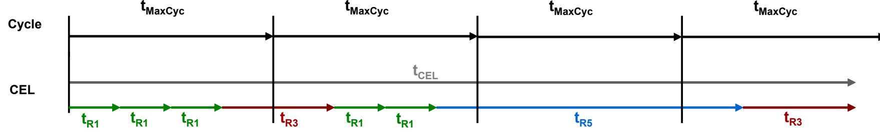

Fig. 4. Time cycle problems with PLCs.

for classification. Hence, the number of rule applications is highly dynamic and cannot be determined in advance. Secondly, the time required for performing the various rules is also dependent on the previously computed consequences in S and R and differs from rule to rule. A rule can either be performed very quickly compared to the maximum cycle time of the PLC, or its runtime can even exceed this limit, in which case its interruption would lead to a PLC error state. Figure 4 depicts a cyclic view of the CEL algorithm, comparing the PLC's cyclic processing pattern (t_{MaxCyc}) to the overall classification time (t_{CEL}) composed of the runtimes for individual rule applications (t_{Ri}). The overall runtime t_{CEL} needs to fit into the cyclic time pattern given by t_{MaxCyc}, such that no application of a rule **(CR*i*)** with duration t_{Ri} exceeds the maximum cycle length. However, an intricate rule like **(CR*5*)** might not fit in a single cycle.

In the following sections, we describe our approaches to overcome these difficulties.

4.2 Compact Axiom Representation

The restricted memory available on PLCs requires a compact representation of axioms in the ontology $\|\mathcal{O}\|$ based on an integer encoding of concept and role names, which can then be used for efficient index-based memory array access. We achieve this by exploiting the fact that the normalized axioms in $\|\mathcal{O}\|$ are of one of the forms **(NFi)**, all of which require at most three class/role names as their parameters. Thus, we can encode any normalized axiom $\alpha \in \|\mathcal{O}\|$ as a four-tuple

$$\alpha = (\mathsf{T}, p_1, p_2, p_3),$$

where T encodes the type of normal form axiom (1-6) and the p_i are the integer numbers for concept and role names obtained by simply enumerating the elements in $\mathsf{N}_C^{\mathcal{O}}$ and $\mathsf{N}_r^{\mathcal{O}}$, e.g. in a lexicographic ordering with regard to their string name representation. Although the different positions of such an axiom tuple could be encoded with variable bit lengths depending on the number of concept and role names, we have chosen a representation that uses a two byte integer value per position for a total of eight bytes for an axiom, since such a fixed size encoding scheme provides for more efficient access. Based on this axiom encoding, we represent the ontology $\|\mathcal{O}\|$ on the PLC side as a fixed length array whose size of $8 \cdot \#\|\mathcal{O}\|$ bytes is determined during the preprocessing phase.

We represent the classification results stored in the mappings S and R as fixed length bit arrays $S[i][j]$ and $R[i][j][k]$, while i, j range over concept name

and k over role name index numbers. We have to assume that, in the worst case, all possible axioms of the forms $A \sqsubseteq B$ and $A \sqsubseteq \exists r.B$ are being derived, which provides the following upper bounds for the lengths of the arrays.

$$l_S = (\#\mathsf{N}_C^{\mathcal{O}})^2 \,, \qquad l_R = (\#\mathsf{N}_C^{\mathcal{O}})^2 \cdot \#\mathsf{N}_r^{\mathcal{O}}$$

Thus, the overall memory required for representing the reasoning results in S and R is given by $\lceil (l_S + l_R)/8 \rceil$ in bytes. Efficient access to these arrays through the indices i, j, k also provides good performance for the frequent checks on S and R in the rules **(CRi)**.

While PC-based reasoners can optionally use such compact memory representations for optimization, they are necessary for porting reasoning algorithms to a PLC to handle the above mentioned memory issues.

4.3 Interruption-safe $\mathcal{EL}^+$ Saturation

To address the difficulty about the PLC's cyclic processing paradigm, our objective is to safely fit the whole required computation time t_{CEL} for classification into the periodic cycles without running into error states by exceeding t_{MaxCyc} in a single rule application. To this end, we have implemented a time-out mechanism that uses a PLC's integrated interruption features for program-triggered termination of the current processing cycle. Figure 5 shows a timeline for the behavior of this interruption mechanism.

Shortly before the end of the cycle time t_{MaxCyc} is reached while processing a derivation rule, our mechanism initiates an artificial interrupt that saves the current state of rule processing to be reentered at the beginning of the next cycle. The state consists of the index variables i, j, k that serve to iterate over the structures S and R for checking a rule's applicability, as well as the currently processed axiom $\alpha \in \|\mathcal{O}\|$ and the type of rule **(CR*i*)** whose processing is to be reentered. In this way, we abstract from the cyclic processing constraints of a PLC, spreading the required overall calculation time t_{CEL} over an arbitrary sequence of cycles. In Section 5, we will show that the overhead for saving and reentering rule processing states is negligible for reasonable values of t_{MaxCyc}.

Notice that this interruption mechanism allows us to dynamically add diagnostic $\mathcal{EL}^+$-reasoning to any control program installed on a PLC whenever

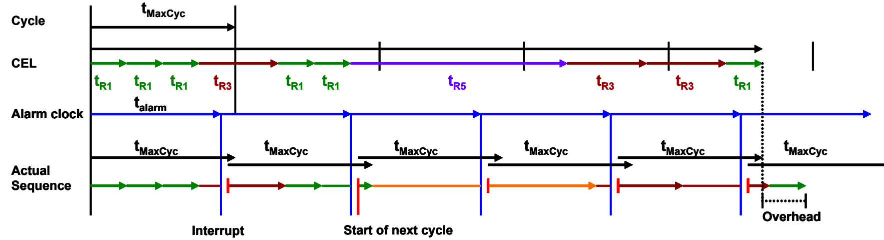

Fig. 5. Cycle time interruption mechanism.

the maximum cycle length determined by the control task is not fully exploited. The rest of t_{MaxCyc} not used for control tasks can then be utilized for diagnostic reasoning for many pre-installed PLCs in the automation field.

5 Evaluation

In this section, we report on first experimental results that we carried out to evaluate our embedded reasoning approach. For this purpose, we first describe our experimental setting, before we report on performance experiments.

5.1 Evaluation Setting

The hardware setting used for our evaluation consists of a SIMATIC IPC427C industrial PC with various sensors and a WinCC HMI interface attached to it via PROFINET. It is equipped with an Intel Core2 Duo U9300 1.2GHz processor and 956 MB RAM and runs Microsoft Windows XP SP2 as its operating system. During the SIMATIC WinAC RTX installation, the Ardence RTX 8.1 real time kernel is installed which adds real time capabilities to the OS. On this basis, the Software PLC SIMATIC WinLC (Windows Logic Controller) RTX v4.4.1 SP1 performs the tasks of a S7-300 or S7-400 PLC in our setting. The WinLC software provides the full functionality of an S7-300 or S7-400 PLC, although the processing time depends on the actual CPU used. It hosts our prototypical PLC-based $\mathcal{EL}^+$-reasoner implemented in SCL following the architecture and implementation features described in the sections 3 and 4 . For OPC communication we use the Siemens OPC Server v7.0 and a Java-based PC-client.

As test data we have used a set of $\mathcal{EL}^+$ ontologies that stem from the turbine diagnostics use case described in Section 3. They are listed in Table 1 with their number of concepts, roles and axioms. The base ontology $\mathcal{O}_{tur}$ is a local diagnostic model that was used to capture the causal relationships between symptoms and faults for certain parts of turbine machinery in the $\mathcal{EL}^+$ formalism. Since in $\mathcal{O}_{tur}$ not all the $\mathcal{EL}^+$ constructs are used, however, we have produced a modified version $\mathcal{O}^+_{tur}$, which contains some additional axioms to cover the full expressivity of $\mathcal{EL}^+$ (the missing construct was conjunction $\sqcap$). As a result, classification of $\mathcal{O}^+_{tur}$ triggers all the derivation rules **(CRi)** at least once and thus comprises a suitable test data ontology for our system. To scale to larger test data, we extended the ontologies $\mathcal{O}_{tur}$ and $\mathcal{O}^+_{tur}$ to multiplied versions that contain multiple renamed copies of the original axioms, indicated by a factor in their name (e.g., $\mathcal{O}_{tur10}$ contains 10 copies of the original axiom set in $\mathcal{O}_{tur}$). The largest ontologies $\mathcal{O}_{tur22}$ and $\mathcal{O}^+_{tur16}$ have been chosen such that their classification uses up all of the PLC's total memory of 4MB. In this way we get runtime results for the case of maximal memory usage as an upper bound for answer times.

5.2 Performance Experiments

Correctness Tests. To ensure correctness of our implementation, we performed a back-to-back test of our PLC-based $\mathcal{EL}^+$ reasoner compared to a standard

Table 1. Detailed overview of ontologies, their memory consumption and runtimes.

	Ontology				memory in kByte				runtime in ms	
	$\mathcal{O}$	$\#\mathsf{N}_C^{\mathcal{O}}$	$\#\mathsf{N}_r^{\mathcal{O}}$	$\#\lVert\mathcal{O}\rVert$	$\lVert\mathcal{O}\rVert$	S	R	total %	t_{CEL}	t_{total}
1	$\mathcal{O}_{tur}$	28	4	49	0.38	0.10	0.38	0.02%	29	250
2	$\mathcal{O}_{tur5}$	136	20	245	1.91	2.26	45.16	1.18%	773	641
3	$\mathcal{O}_{tur10}$	271	40	490	3.83	8.97	358.60	9.02%	1774	2156
4	$\mathcal{O}_{tur22}$	595	88	1078	8.42	43.22	3803.00	94.00%	8848	9141
5	$\mathcal{O}_{tur}^{+}$	34	7	77	0.60	0.14	0.99	0.03%	113	375
6	$\mathcal{O}_{tur3}^{+}$	100	21	231	1.80	1.22	25.63	0.68%	976	1188
7	$\mathcal{O}_{tur5}^{+}$	166	35	385	3.00	3.36	117.73	2.99%	2815	2937
8	$\mathcal{O}_{tur16}^{+}$	529	112	1232	9.63	34.16	3825.95	94.36%	29586	29781

reasoner available on a PC platform. For this purpose, we used the Java-based JCEL[13] reasoner as a reference system in order to compare the classification hierarchy output by our system through the structure S to that produced by JCEL. We noticed no differences in the output of the two systems for our turbine diagnostics ontologies or for some $\mathcal{EL}^+$ ontologies available on the web. Although this is not a 100% test, we argue that these tests strongly support the correctness of our implementation, especially since the ontology $\mathcal{O}_{tur}^{+}$ is modified such that it covers all features of $\mathcal{EL}^+$ and triggers all the derivation rules (**CR1**) - (**CR6**), which ensures a certain coverage of the underlying formalism's constructs.

Runtime Performance and Scalability Test. Runtime and scalability performance requirements for embedded reasoning very much depend on the particular use case in question. For the diagnostics use case that we consider here, neither the ontologies get typically very large as any single PLC does only have to reason over the local diagnostic model for its surrounding machinery, nor the answer times for retrieving diagnoses need to be in real time due to the offline nature of the diagnostic task. Therefore, we varied the ontology size in our experiments in the lower ranges (compared to typical benchmark ontologies) and we accepted answer times for retrieving diagnoses within seconds or even minutes.

Notice that a direct comparison of classification times with those of PC-based systems, such as JCEL, does not provide useful insights on performance issues for PLC-based reasoning, since today's standard PCs are much more powerful than PLCs in terms of CPU and memory. Instead, we are interested in the memory consumption and runtime behavior of our PLC-based reasoning approach in the light of the requirements of our diagnostics use case. In particular, we are interested in bringing embedded reasoning onto existing PLC hardware pre-installed in the automation field alongside the control tasks that these PLCs already run. To this end, we report on investigations about memory consumption, optimal cycle length and answer times for diagnoses in the following.

[13] `http://jcel.sourceforge.net/`

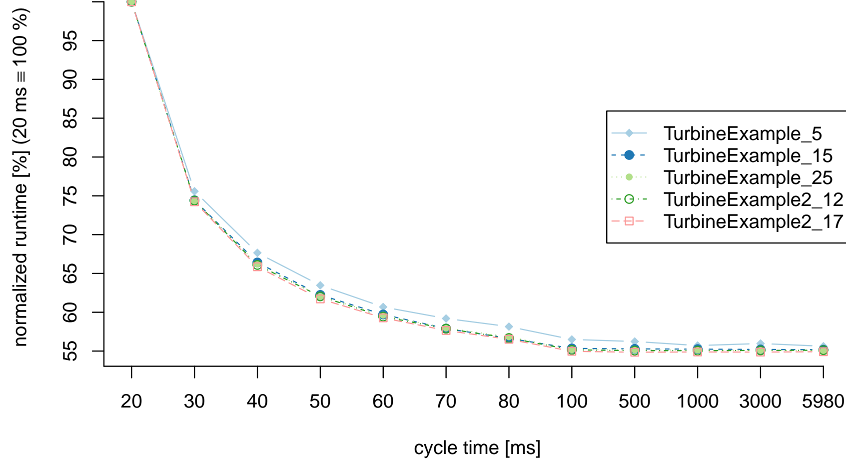

Fig. 6. Cycle Time Evaluation.

Memory Consumption. The memory required for classifying an ontology $\mathcal{O}$ in the PLC is given by the size and signature of $\mathcal{O}$, while $\mathsf{N}_C^{\mathcal{O}}$ and $\mathsf{N}_r^{\mathcal{O}}$ determine the size of the data arrays for the mappings S and R. Table 1 shows the sizes of these memory components for all the test ontologies used. It also shows the percentage of the PLC's total memory used for classification. It can be seen that most of the memory used is reserved for the classification results stored in S and R, while the representation of the ontologies' axioms occupy only a small amount, for all but the smallest ontologies. Classification of the single original and modified diagnostic models $\mathcal{O}_{tur}$ and $\mathcal{O}_{tur}^{+}$ requires only modest memory, while for their multiplied versions a polynomial increase can be observed due to the respective growth of the arrays for S and R. Only for the largest artificially increased models of over a thousand axioms does memory consumption reach critical values. For the expected sizes of local diagnostic models, it should thus be feasible to run diagnostic reasoning on a pre-installed PLC in addition to the automation tasks it already runs, without consuming too much of its memory.

Cycle Time Optimization. User programs performed on pre-installed PLCs often do not require the possible maximum cycle time allowed by the process to be controlled but run much faster. By exploiting such potential process control idle times, additional tasks like diagnostic reasoning can be run within a PLC's execution cycle at no cost of critical CPU power if the cycle length for reasoning is chosen small enough to fit the idle time.

To this end, we investigated the overhead that comes with small execution cycle times for $\mathcal{EL}^+$ reasoning in our approach, where any interruption of rule execution requires the saving and reentering of the current execution state. The diagram in Figure 6 shows the relative classification times in relation to different cycle lengths for all the test ontologies. (The longest classification time corresponds to 100%.) It can be seen that the overhead for rule interruption is only significant for very small cycle lengths less than about 100ms. Beyond this threshold the overhead becomes negligible for all the test ontologies considered. This suggests that from a cycle time of 100ms or greater, PLC-based $\mathcal{EL}^+$-reasoning runs in an optimal mode without wasting a significant amount of execution time due to interruption handling. Since this is a rather low number, we are hopeful to encounter many cases for adding $\mathcal{EL}^+$-based diagnostic reasoning to pre-installed PLCs, exploiting potential process control idle times.

Runtime Performance. As for runtime measurement, Table 1 shows the time for actual classification on the PLC (t_{CEL}) and total time (t_{total}) including OPC communication between the PLC and PC (ontology deployment + result reporting). Classification times for the original diagnostic model $\mathcal{O}_{tur}$ as well as for the modified $\mathcal{O}^+_{tur}$ are within milliseconds and are thus rather fast. Also the multiplied models of up to five or even ten copies are being classified within a few seconds. For the larger samples $\mathcal{O}_{tur22}$ and $\mathcal{O}^+_{tur16}$, classification requires several seconds up to half a minute. Since these two ontologies already fill the maximum memory of the PLC, we can say that the answer times in our approach lie within feasible bounds for the task of offline diagnostics. The OPC-based communication causes a significant relative overhead for small ontologies and becomes negligible for larger models.

6 Conclusion

In this paper, we have presented an approach to porting OWL 2 EL reasoning to PLCs, a type of embedded controller prevalent in industrial automation. We have described how deficiencies about the differing paradigms of PLCs and PC-hosted reasoning algorithms can be overcome by a compact memory representation strategy and an interruption-safe variant of the CEL [1] classification procedure suitable for PLCs. Furthermore, we have reported on promising initial experimental results carried out on a prototypical implementation of a PLC-based $\mathcal{EL}^+$ reasoner in the context of a use case about diagnostics for turbine machinery, in which we prove the feasibility of our approach in terms of memory consumption and answer times for retrieving diagnoses.

A particular finding we have made here is that attempts of porting reasoning procedures to strongly restricted embedded devices could greatly benefit from avoiding too much dynamics and flexibility in the data structures used, ideally keeping them as static as possible. In this work, we could successfully employ low-degree polynomial upper bounds on the maximum size of derivation results that reside in memory at a single time. For tableau-style algorithms, whose tree-like models appear to be more difficult to handle, the separate calculation of single tableau branches at once goes in this direction.

To the best of our knowledge, this is the first endeavor of bringing description logic reasoning to the PLC-dominated industrial automation field. Building on this enabling step of providing a platform for embedded reasoning, we intend to utilize this platform in our use case of turbine diagnostics in forthcoming field tests and experiments to have an impact on the maintenance of power plants in the energy sector. Next to machine diagnostics, we see many other potential use cases for embedded reasoning in industry, such as component verification in industrial engineering or the support of condition monitoring and control tasks, that benefit from exploiting an explicit representation of expert knowledge models brought close to industrial machinery.

This research was funded in part by the German Federal Ministry of Education and Research under grant number 01IA11001.

References

1. F. Baader, S. Brandt, and C. Lutz. Pushing the EL envelope. In *Proc. of IJCAI 2005*, pages 364–369. Professional Book Center, 2005.
2. F. Baader, D. Calvanese, D.L. McGuinness, D. Nardi, and P.F. Patel-Schneider, editors. *The Description Logic Handbook*. Cambridge University Press, 2003.
3. F. Baader, C. Lutz, and B. Suntisrivaraporn. Efficient reasoning in EL+. In *Proc. of DL 2006*, volume 189 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.
4. H. Berger. *Automating with STEP 7 in STL and SCL: programmable controllers SIMATIC S7-300/400*. John Wiley & Sons, 2007.
5. H. Berger. *Automating with SIMATIC: Controllers, Software, Programming, Data Communication Operator Control and Process Monitoring*. Publicis, 2009.
6. T.M. Hubauer, S. Grimm, S. Lamparter, and M. Roshchin. A diagnostics framework based on abductive description logic reasoning. In *Proc. of ICIT 2012*. IEEE Computer Society, March 19-21 2012.
7. T.M. Hubauer, C. Legat, and C. Seitz. Empowering adaptive manufacturing with interactive diagnostics: A multi-agent approach. In *Proc. of PAAMS 2011*, volume 88 of *Adv. in Intelligent and Soft-Computing*, pages 47–56. Springer, 2011.
8. Minsu Jang and Joo-Chan Sohn. Bossam: An extended rule engine for OWL inferencing. In *Proc. of RuleML 2004*, volume 3323 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2004.
9. Y. Kazakov, M. Krötzsch, and F. Simančík. Concurrent classification of $\mathcal{EL}$ ontologies. In *Proc. 10th Int. Semantic Web Conf. (ISWC'11)*, volume 7032 of *LNCS*, pages 305–320. Springer, 2011.
10. C. Legat, T.M. Hubauer, and C. Seitz. Integrated diagnosis for adaptive service-oriented manufacturing control with autonomous products. In *Proc. of IESM 2011*, pages 1363–1372. International Institute for Innovation, Industrial Engineering and Entrepreneurship (I^4e^2), 2011.
11. C. Seitz and R. Schönfelder. Rule-based OWL reasoning for specific embedded devices. In *Proc. of ISWC 2011*, volume 7032 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2011.
12. A. Sinner and T. Kleemann. KRHyper – in your pocket. In *Proc. of CADE 2005*, volume 3632 of *Lecture Notes in Computer Science*, pages 452–457. Springer, 2005.
13. W3C OWL Working Group. *OWL 2 Web Ontology Language: Profiles*. W3C Recommendation, October 27 2009.