Representing Software Process in Description Logics: An Ontology Approach for Software Process Reasoning and Verification

Edward Kabaale[2(✉)], Lian Wen[1,2], Zhe Wang[1,2], and Terry Rout[1]

[1] Institute for Integrated and Intelligent Systems, Griffith University, 170 Kessels Rd, Nathan, QLD 4111, Australia
{l.wen,z.wang,t.rout}@griffith.edu.au
[2] School of Information and Communication Technology, Griffith University, 170 Kessels Rd, Nathan, QLD 4111, Australia
e.kabaale@griffith.edu.au

Abstract. Software process is critical for producing high quality software. However, software processes are usually described in natural language which makes it difficult to verify if they have been fully or how well implemented in complex software projects. It's also hard for practitioners to implement processes from different standards and make sure they work harmonically, consistently and completely. Composition Tree (CT) notation, a Behavior Engineering approach has been successfully used to formalize software process in previous work. However, there are no reasoning tools for CT to automatically check and verify the modeled software processes. In this study we explore the synergy of software process modeling and Description Logics (DLs). Given the rich expressiveness of DLs and their efficient and automated reasoning support, DLs can be used to reason and verify software processes more effectively. We propose an algorithm for transforming CT software process model into a DL so that DL reasoning engines can be used to perform automated software process analysis. Case studies and simple examples are also given to justify the feasibility of this proposed approach.

Keywords: Software process · Composition tree · Description logics · Automatic reasoning · Process verification · Behavior engineering · Software engineering

1 Introduction

Software processes represented in international standards such as ISO/IEC 12207, ISO/IEC/29110 [5, 6] among others are explicitly described in formal documents using natural language. Compliance to the processes described in these prescriptive process models is considered as best practices to ensure successful completion of software development projects. They guide, support and advise software developers by prescribing activities, tasks and steps to be followed in quality and repeatable software production. However, these process models are commonly specified using natural language and/or graphical representations, both lacking the computational semantics

© Springer International Publishing Switzerland 2016
P.M. Clarke et al. (Eds.): SPICE 2016, CCIS 609, pp. 362–376, 2016.
DOI: 10.1007/978-3-319-38980-6_26

needed to enable their automated verification and reasoning. They consist of verbose description that seeks to accurately and completely describe the processes and activities to be followed in software development. Such comprehensive natural language based descriptions present a number of challenges for implementing organizations.

It's difficult to compare the similarity and difference between two similar processes described in different standards. For example comparing a process from ISO/IEC 12207 and its counterpart in ISO/IEC 15228. This comparison is always necessary for process understanding, correctness verification and classification. It's also crucial when designing new standards and choosing appropriate process models for efficient process definition and improvement.

Similar to the above challenge is when two processes are to be merged and integrated. There are current efforts in aligning process models in the system and software engineering community. While these two fields are working closely, their process models use different terminologies, process sets, process structures and levels of description [23]. When process models are used in isolation on the same project, they tend to be less effective and redundant which results into inconsistencies among the implemented processes. On the other hand, merged and aligned processes enable common vocabulary, single process structure, jointly planned level of prescription and effective communication across the project [23].

Thirdly, although processes described in natural language provide a recipe in overall software development guidance, they are seldom implemented and followed exactly. As software developers often need to collect data by questionnaires to validate their organizational processes against the process models. This leads to problems of ambiguity, subjectivity, inconsistency and inaccuracy in process implementation and validation [13].

Even though the Composition Tree (CT) approach has proven to be useful in formalizing, modeling and comparing software processes [2], it has some limitations. Firstly, CT models can grow big and this presents a problem to software engineers to analyze and verify such models due to human memory lapse and state explosion [22]. Secondly, its language for logic tests in the tools for Behavior Engineering is limited subsequently no reasoners currently can be used to automatically verify and reason the consistency and completeness of software process models produced. On the other hand Description Logics (DLs), a family of knowledge representation languages with well understood semantics provide means by which models can be understood by machines and therefore reasoners can be used to automatically verify and reason the consistency and completeness of the software process. Highly optimized and efficient DL reasoners such as PELLET [26], FACT ++ [27] and HERMIT [24, 25] are readily available off shelf for this purpose.

Therefore in this paper we propose an algorithm to translate CT models to a DL. We also present an example of the translation algorithm, and a case study to demonstrate the DL services in software process verification and reasoning. The rest of the paper is organized as follows; Sect. 2 introduces background information about software process modeling, Composition Trees and Description Logics while our approach forms Sect. 3. Sections 4 and 5 look at related work and conclusion respectively.

2 Background

2.1 Software Process Modeling

Software Process Modeling (SPM) refers to the activities in creating abstract representations of the methodology, design or definition of the software process [1]. SPM incorporates a representation approach and comprehensive analysis capabilities (a range of tests in the areas of consistency, completeness, and correctness). The goals of SPM are to abstract and organize software process information into well-defined models, analyze the inter-relationships among model elements and attributes, and predict the outcome of software process [1]. Process models can be analyzed, validated and simulated [1]. And if represented by a formal language with clear semantics like DL then they can also be reasoned on for inconsistencies.

Process models are of great importance in software engineering as summarized below by [3]; (i) Process models ease understanding and communication between different stakeholders in software development, (ii) Process models help in process management support and control; requiring a project specific software process and monitoring, management and coordination, (iii) They provide automated orientations for process performance and reasoning, (iv) They provide automated support; requiring automated process parts, cooperative work support, a compilation of metrics and process integrity assurance, (v) Process model enhance process improvement.

The resulting software process models from SPM can be *descriptive* or *prescriptive* [14]. *Prescriptive* process modeling defines how software development processes should be performed, including methodologies, rules, guidelines, and behavior patterns that would lead to the desired process performance. *Prescriptive* process models are used as guidelines or frameworks to organize and structure how software development activities should be performed, and in what order [1]. Thus, *prescriptive* process models can also be referred to as reference process models – see, for example ISO/IEC 12207 [5], ISO/IEC 29110 [6] among others. These models determine a set of essential, but unordered activities, which have to be completed to obtain a software product. They do not prescribe a specific life cycle. Each organization that uses the standard must instantiate the prescribed process as a specific process.

ISO/IEC 24774: 2007 *Software and Systems Engineering–life cycle management—Guideline for process description* [4] is one important example of prescriptive Process Reference Models that defines a general format for any process reference model without specifying a specific life cycle. Any organization that uses the standard must instantiate their own specific process based on the guidelines provided by the standard. This general purpose standard outlines the elements used to describe a process as; title, purpose statement, outcomes, activities and tasks [2].

- The **purpose** describes the goal of performing the process it is expressed as a high level goal for performing the process, preferably stated in a single sentence. The implementation of the process should provide measurable, tangible benefits to the stakeholders through the expected outcomes.
- The **outcomes** express the observable results expected from the successful performance of the process. Outcomes are expressed in terms of a positive, observable

objective or benefit. The list of outcomes associated with a process shall be prefaced by the text, 'As a result of successful implementation of this process:' the outcomes should be no longer than two lines of text, about twenty words. The number of outcomes for a process should fall within the range 3 to 7. Outcomes should express a single result. The use of the word 'and' or 'and/or' to conjoin clauses should be avoided. Outcomes should be written so that it should not require the implementation of a process at any capability level higher than 1 to achieve all of the outcomes, considered as a group.

- The **activities** are a list of actions that may be used to achieve the outcomes. Each activity may be further elaborated as a grouping of related lower level actions.
- The **tasks** are specific actions that may be performed to achieve an activity. Multiple related tasks are often grouped within an activity.

Despite the fact that software processes represented in international standards are well-structured and contain detailed technical description on a complex subject, their implementation is still a challenge. Their natural language representation lacks formal semantics for automated analysis, consistency checking and reasoning. It makes it difficult to rigorous analyze and compare information from the many different versions of the same standard. It may also be complicated to tailor processes to specific software development projects [2]. This situation calls for a formal modelling formalism that can produce precise, unambiguous and consistent models. In CT and DL formalisms we can find a solution to the above shortcomings of processes modeled in natural language.

2.2 Composition Trees

Behavior Engineering (BE) is a general high level graphical modeling notation [16]. BE approaches have proven successful in elimination of ambiguity and incompleteness in requirements and process models with great success [2, 17]. Just like UML diagrams are used in various phases of software design, BE approaches are used in modeling and verifying various software artefacts such as functional requirements, software architecture and traceability [17, 18], and process models [2].

The main modeling techniques of BE are Behavior Trees (BTs) and Composition Trees (CTs). Well as BTs are used in modeling dynamic system aspects [17], CTs similar to UML class diagrams, model static system aspects in terms of entities, relationships, attributes and component states [2]. CT just like BT is constructed through a careful step wise approach and later integrated into one complete tree like graphical model. The created models are more intuitive, less ambiguous and easier to read and verify than the original natural language processes [2]. The application of BE approaches and CT in particular to process modeling and verification has highlighted the importance of using a simple yet intuitive modeling notation than the more complicated UML with teens of diagrams to contend with. Being simple and easy aids communication and understanding among the different SE stakeholders with different domain backgrounds. We believe the simple created CT models can easily be reasoned on using DL reasoners for consistence further enhancing the effectiveness of this approach in process modeling.

2.3 Using Composition Trees to Model Software Processes

ISO/IEC 24774:2007, prescribes the standard elements to define a process as the title, purpose, outcomes, activities and tasks. The purpose and outcomes are more suitable static elements of a process that can easily be modelled by CT. These process elements have been successfully used to model software processes in previous works [2, 7]. More information about how to model software process using CT can be found in these studies. In this case study, we translate the Human Resource Management Process from ISO/IEC TS 33053, a draft Process Reference Model for Quality Management [28], to CT.

Process Name: *Human Resource Management*

Purpose: *The purpose of Human Resource Management is to provide the organization with necessary competent human resources and to improve their competencies, in alignment with business needs.*

Outcomes: As a result of successful implementation of this process;

1. *The competencies required by the organization to produce products and services are identified*
2. *Identified competency gaps are filled through training or recruitment*
3. *Understanding of role and activities in achieving organizational objectives in product and service provision is demonstrated by each individual*

CT Modeling

The first step as indicated already is to identify the components from the process;

HRM: *Human Resource Management*
HR: *Human Resource*
BN: *Business needs*
Gap: *Competency gaps*
TR: *Training and Recruitment*
TR: *Roles and activity*
PS: *Products and services*

The CT model above shows a more intuitive and less ambiguous process as compared to the natural language process. It's easy to establish the relationships between the different components of the process. It also makes it possible to follow the consistency of the process and makes the process available for automation.

The benefit of modelling a software process in a CT is that the graph gives an overall view of the process and it's less ambiguous and intuitive. Formal verification such as comparing two processes can be performed by using automated tools [4]. However, reasoning of processes modeled in CT is not possible because they are no mature reasoning tools for CT models currently. This has propelled us to look at knowledge representation where DLs can be used to model and reason processes efficiently there by enabling automated process analysis.

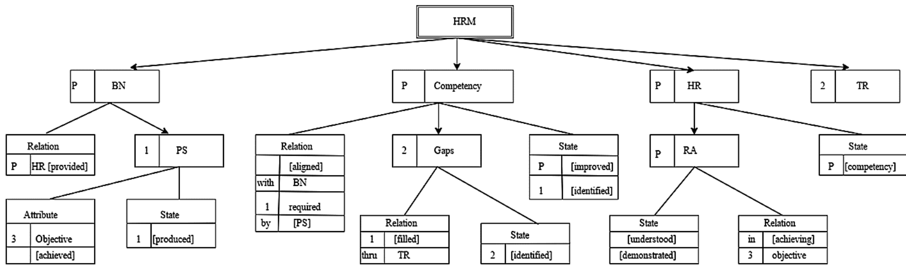

Fig. 1. The CT for Human Resource Management Process

2.4 Description Logics

Description Logics (DLs) [9] are a family of logics specifically designed to represent and reason on structured knowledge. DLs are a collection of logics that are less expressive than for example first order logic but highly structured with improved computational complexity [11]. DLs are class based representation formalisms that represent aspects of the domain of interest in terms of concepts, roles and individuals. Knowledge modeling in DL is done using Boolean constructors to build complex concepts from simple ones. Depending on the modeling needs of the domain of interest more expressive variants of DL can be obtained by adding more constructors. One of the most expressive DLs is *SROIQ* that underpins the OWL 2 a W3C standard for the Semantic Web. Some DL constructors and their syntax are shown in the table below (Table 1).

Table 1. Syntax and examples of some DL constructors.

Construct	Syntax	Example
Atomic concept	A	*Teacher*
Atomic role	P	*hasChild*
Atomic negation	$\neg A$	$\neg$*Teacher*
Conjunction	$C \sqcap D$	*Human* $\sqcap$ *Male*
Existential restriction	$\exists R.C$	$\exists$*hasChild.Male*
Value restriction	$\forall R.C$	$\forall$*hasChild.Male*
Top concept	$\top$	
Bottom concept	$\bot$	
Disjunction	$C \sqcup D$	*Father* $\sqcup$ *Mother*

A precise specification of the meaning of DL axioms is given by their model theoretical mapping via an interpretation function to the domain of interest. In the standard set-theoretic semantics of concept descriptions, concepts are interpreted as subsets of a domain of interest, and roles as binary relations over this domain. An interpretation I consists of a non-empty set Δ^I (domain of I) and a function $\cdot^I$ (the interpretation function of I) which maps each atomic concept A to a subset A^I of Δ^I, and each atomic role R to a subset R^I of $\Delta^I \times \Delta^I$ and each individual name a to an element $a^I \in \Delta^I$ [11]. A DL knowledge base is mainly composed of two main components. The Terminological

knowledge represented in the TBox (schemata) and the Assertional knowledge forming the ABox (data instances).

The *TBox* is also referred to as the ontology in DL. It defines the intensional knowledge by which a concrete world can be described. This knowledge is represented by axioms in the form of logical sentences. The TBox axioms include (i) Concept inclusions in the form of $C \sqsubseteq D$, where C and D are arbitrary concepts. *For example,* $Mother \sqsubseteq Parent$, (ii) Concept equivalence in the form of $C \equiv D$; meaning that C and D have the same instances and model. For example, $Mother \equiv Parent \sqcap Female$. Generally a TBox is a finite set of axioms of the above forms. A model of a TBox is an interpretation that satisfies all its axioms.

The *ABox* represents assertional knowledge that complies with the intensional knowledge in the TBox. There are two main types of assertions for DL systems, i.e. a concept assertion that asserts an individual belongs to a given concept in the form of $C(a)$, *for instance* $Mother(Mary)$ *means that Mary belongs to the concept of mothers;* and a role assertion that asserts two individuals are related via a given role in the form of $R(a, b)$. For example, we can express that Mary has a child called Peter as $hasChild(Mary, Peter)$. An ABox is a finite set of concept and role assertions. The interpretation function in the ABox is extended to individual names, as each individual is mapped to an element of the domain by the interpretation function.

The DL knowledge base KB is a pair of a TBox T and an ABox A, i.e. $KB = (T, A)$. An interpretation I satisfies a DL knowledge base iff it satisfies both the TBox and the ABox. A DL knowledge base KB entails a DL statement ϕ, written as $KB \vDash \phi$, iff every interpretation that satisfies KB also satisfies ϕ.

DL systems have been used to model different domains of interest such as conceptual modeling [8], natural science [20], databases [19] and above all ontological modeling. DLs underpin the logical foundation of Web Ontology Language (OWL) which is now a W3C standard for the Semantic Web [20]. Generally DL can be used to model anything that can be modelled by a directed graph [10].

Starting with the design of a TBox is a popular approach in developing DL complex ontologies, however, there is little discussion on practical ways of dealing with real world challenges that must be overcome to build, refine and maintain TBox models [12]. Nevertheless it provides the basic structure of the ontology to be developed and it is here where the conceptualization of the domain under study is enumerated. In [12] a methodology is provided for modeling DL knowledge bases based on TBox approach. The ABox complements the TBox by relating individuals to concepts and to other individuals via roles [11]. While it is possible to split the TBox from ABox in modeling DL knowledge bases for various reasons and benefits; see for example [12]. In this paper we adopt a methodology in [12] that allows the knowledge base structure to be constructed first as a TBox and later complimented with ABox at run time. This approach enables TBox maximum reasoning capabilities. We are more interested in the benefits that happen in the knowledge base in the middle of TBox and ABox as shown below [12] (Fig. 2).

3 Translating CT Diagrams to DL TBoxes

In this section we describe how a CT model can be translated to a DL TBox. In addition, using DL ABox we can enumerate the instances and properties of the components. The goal of such translation is that the resulting knowledge base (both the TBox translated from the CT model and the additional ABox) can be used for verification and automated reasoning. For instance, using the resulting knowledge base we can verify organizational processes through instance checking. The application will be discussed in detail in Sect. 4, whereas in what follows we will focus on the TBox translation.

3.1 CT to DL Translation

Our translation process is based on the following Composition Tree Axioms in Table 2, which aims to maximally preserve the structure of the tree. Our translation is different from the approach in [8, 10] but shares the same spirit. In particular, we formalize the CT model into a DL TBox (ontology) as follows:

Table 2. CT to DL translation.

CT Statement	DL Axiom
A component C has a subcomponent C_i	$C \sqsubseteq \exists hasSubComponent.C_i$
A component C has an attribute a_i with value type T_i	$C \sqsubseteq \exists a_i.T_i$
A component C has relation R with another component C'	$C \sqsubseteq \exists R.C'$
A component C has a state S	$C \sqsubseteq \exists hasState.S$
Two components C_1 and C_2 are disjoint	$C_1 \sqsubseteq \neg C_2$
A component C is constituted from all its properties (incl. subcomponents, attributes, relations to other components, and states)	$C \equiv \exists hasSubComponent.C_1 \sqcap \cdots \sqcap \exists hasSubComponent.C_n \sqcap \exists a_1.T_1 \sqcap \cdots \sqcap \exists a_m.T_m \sqcap \exists R_1.C'_1 \sqcap \cdots \sqcap \exists R_l.C'_l \sqcap \exists hasState.S_1 \sqcap \cdots \sqcap \exists hasState.S_k$

a. *Each component is represented by an atomic concept.*
b. *Each component is associated with its subcomponent through a universal role hasSubComponent.*
c. *Each state is represented by an atomic concept and each component is associated with its states through a universal role hasState.*
d. *Each attribute is represented by a role and each value type is represented by an atomic concept.*
e. *Each relation is represented by a role.*

Note that the final axiom provides a definition of the component. This assumes the CT model provide complete information about each component. Such axioms are useful for process verification as we will discuss in the following section.

3.2 Reasoning Services

DL systems provide users with various reasoning services that deduce implicit knowledge from the explicitly represented knowledge [11]. The basic reasoning service in DL systems is to test for satisfiability of a concept or a TBox. That is to test whether the conceptualization specified in the TBox has a contradiction or not. In an unsatisfiable TBox any consequence can flow logically but this will be without meaning. Testing for satisfiability is often a first step in checking whether a TBox models anything meaningful. Let C be a concept description and T a TBox. The concept C is satisfiable w.r.t. T *iff* there is a model of T in which C can be interpreted as nonempty, i.e. there exists an interpretation I such that $C^I \neq \emptyset$. A TBox is satisfiable iff it has a model. Satisfiability checking can capture CT coherence and consistency checking in our approach as follows.

CT Coherence: A *CT* is *coherent*, if it can represent all the intended information about its components from natural languages without violating any of the constraints in the CT diagram. This may help to check its composition and completeness. Exploiting the formalization in *DL*, CT coherence can be checked by checking satisfiability of the corresponding concepts in the *DL* TBox representing the CT diagram.

CT Consistency: A CT diagram is *consistent*, if it does not violate any of the constraints in the diagram. This is important as different CT can be integrated to form one integrated CT, and CT consistency check guarantees whether such integration violates any of the constraints in the initial CT diagrams; as otherwise the integration of the different CTs may make it very difficult to detect inconsistencies. By exploiting the formalization in *DL*, the consistency of an Integrated Composition Tree (ICT) diagram can be checked by checking the satisfiability of the corresponding *DL* TBox [10].

CT Consistency with Instances: Another DL reasoning service that is of importance in our case is checking the consistency of a TBox related to an ABox. As mentioned at the beginning of Sect. 3, the ABox comes from logged processes, i.e. instances of a process. Just like for consistency of a TBox, we can also test for (i) contradictions in the ABox and (ii) At run time, we can check whether the ABox is consistent with the conceptualization in the TBox.

Instance Checking: An individual a is an instance of a concept C w.r.t a knowledge base KB, iff $KB \models C(a)$. Instance checking denotes the task of testing whether a given individual is an instance of a given concept, i.e., whether $KB \models C(a)$ holds. Instance checking is the central reasoning service for information and instance retrieval from knowledge bases. Instance checking can also be used to verify whether an individual can be classified into some defined DL concepts in the knowledge base. In our case study to be presented in the next section, we demonstrate that instance checking can be used to verify whether the implemented organizational processes in the ABox confirm (instances of) to the standard processes specified out in the TBox.

ABox realization: Realization of an individual, in turn denotes the retrieval of all named concepts from the knowledge base that a given individual is an instance of. This reasoning service is also central in automated verification of organizational process models. Again, using ABox instances as a practice model for an organization we can verify the processes against the standard processes in the TBox by checking all the possible components an instance belongs to. In the section below we illustrate how DL TBox translation and how to use ABox reasoning for software process consistence, completeness and conformance checking.

4 A Case Study

In this section, we translate the CT model for Human Resource Management Process from Fig. 1 to a DL for consistency checking and other reasoning. To ensure faithful translation to a DL, we use the method in Table 2 in the previous section.

Table 3. ABox fragment for Process P

$Process(P)$	$hasSubComponent(P, Need)$
$hasSubComponent(P, JP)$	$hasSubComponent(P, Peter)$
$HR(Peter)$	$Competency(JP)$
$hasSubComponent(Need, System)$	$isProvidedBy(Need, Peter)$
$hasObjective(System, Objective)$	$hasState\ (System, State)$

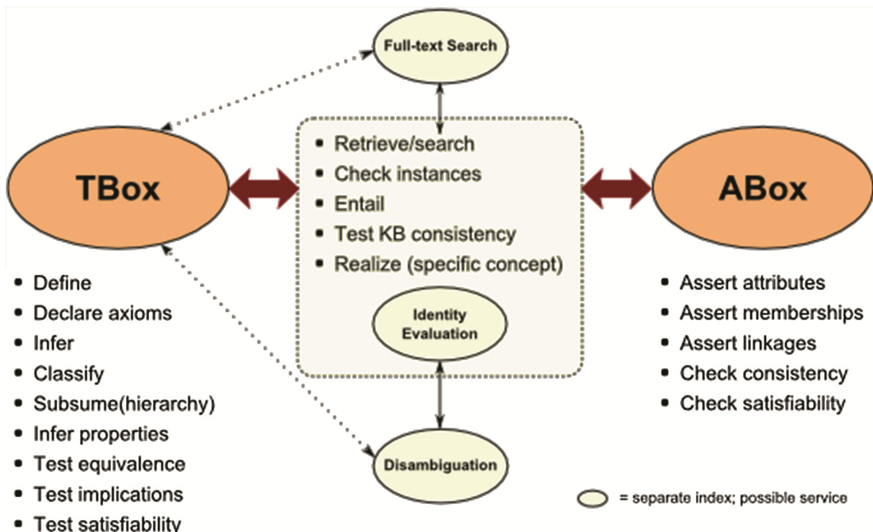

Fig. 2. Showing the benefits of TBox and ABox working together in the Knowledge

4.1 DL TBox Translation

Following the approach we introduced in the previous section, the above CT is translated to the following TBox axioms.

$$HRM \equiv \exists hasSubComponent.BN \sqcap \exists hasSubComponent.Competency \sqcap \exists hasSubComponent.HR \sqcap \exists hasSubComponent.TR \quad (1)$$

$$BN \equiv \exists hasSubComponent.PS \sqcap \exists isProvidedBy.HR^{-} \quad (2)$$

$$PS \equiv \exists hasState.Produced \sqcap \exists hasObjective.Achieved^{-} \quad (3)$$

$$Competency \equiv \exists hasSubComponent.Gaps \sqcap \exists isAlignedTo.BN \sqcap \exists isRequiredBy.PS \sqcap \exists hasState.(Improved \sqcap Identified) \quad (4)$$

$$Gaps \equiv \exists hasState.Identified \sqcap \exists isFilledThru.TR \quad (5)$$

$$HR \equiv \exists hasSubComponent.RA \sqcap \exists hasState.Competency \quad (6)$$

$$RA \equiv \exists hasState.(Understood \sqcap Demonstrated) \sqcap \exists isUsedInAchieving.Objectives \quad (7)$$

4.2 Process Verification

Organizational Process *P* is recorded for an organization that has a business need of developing an accounting software system in Java platform following the human resource process as defined in the draft Process reference model for quality management [28] as modeled above. It has a human resource (individual) *Peter* who has competency in C programming. The organizational objective is to develop an accounting software system in Java platform. This means that, there is a competency gap in the organization that can be filled through training or recruitment. This shows that *Peter* needs training in Java in order to meet the organizational object of developing the software system. This will help to align the human resource competency with the business needs.

This information can be modeled as an ABox and instance checking reasoning service can be used to check if Process *P* is HRM process described previous. Table 3 below represents the ABox fragment.

The ABox says *P* is a process and it has three subcomponents a *Need*, the individual human resource *Peter*, and the competency of Java programming, denoted *JP*. The *Need* has a subcomponent that is the *System* to be developed, and the *Need* must be provided by *Peter*, the only human resource available. The implementation of the *Sysem* has an *Objective* and a *State.*

From the above example, we can use a DL reasoner, e.g., HERMIT, to check if *HRM*(*P*) is entailed by the knowledge base consists of both the TBox (1)-(7) and the ABox in Table 3 to check for the completeness of the process. The DL reasoner can easily detect that *HRM*(*P*) is not entailed by the knowledge base because it's missing an important component, that is, *Peter* is not trained in Java to enable him have competency in using Java to produce the software system needed. Hence, by adding the training for

Java programming, denoted *JP_Training*, as an instance of the concept *TR*, i.e., *TR*(*JP_Training*), and as a subcomponent of the process *P*, i.e., *hasSubComponent*(*P*, *JP_Training*), it will help to complete the process. The training should enable Peter to produce the system, i.e., *Produced*(*State*), and achieve the objective of the system, i.e., *Achieved*(*Objective*). After all these steps, the process *P* is complete, and the new knowledge base by adding the above facts will entail *HRM*(*P*).

The above case study shows how organizations can check and reason their processes in terms of consistency and conformance with standard process reference models modeled as the TBox. Our application scenario appears to be simple, but it exemplifies the main issues of our approach in order for the reader to comprehend the steps followed in the approach. It forms a simple and intuitive theoretical basis for automated process analysis in our approach.

Compared our approach with the natural language description of software process analysis and verification, it is easy to see how simple and scalable our approach can be when dealing with the same tasks. The benefit of modelling and verifying a software process by this approach is that, it gives an overall view of the process both in graphical and formal notation and yet less ambiguous, precise and intuitive. Formal verification and reasoning can be performed using readily available automated tools off shelf [2, 26, 27].

5 Related Work

There is a common agreement that software engineering in general and software process in particular can benefit from DL (Ontology) based approaches to modeling and reasoning. However, many approaches focus on software process modeling and knowledge sharing, instead of what ontology reasoning support to use for software process consistency and verification. Recent research efforts concentrate on generic solutions for introducing different levels of abstraction and formalisation for software process [13, 15].

The approach described in [15, 30] represents the Software Process Engineering Metamodel (SPEM) in DL to give it precise semantics. This enables the application of process analysis techniques to the models created using SPEM. While the approach uses DL just like in our approach, no DL based reasoning was used. Similar to our work, is the work presented in [21], where conceptual graph theory is used to represent and compare organisational processes and the practice models in various combination. While this work has the same sprit like ours in formalising and validating software process, no reasoning services are employed to detect inconsistencies in software process. Similarly the work presented in [29] is in the same vain like ours, where ontology reasoning is used to detect inconsistencies and incompleteness in requirements specification. Specifically the use of TBox as a requirement Meta model against which requirements instances of a particular project can be checked for consistency and completeness. The checks are complimented with consistence and completeness rules written in Semantic Web Rule Language (SWRL). However, in this work it's assumed that the requirements

ontology and its instances will be provided by the same project hence its main concern is more on consistence and completeness checking than validation and verification.

The need for the use of ontology reasoning in software engineering and in particular software process has been recognised by many other researchers. However, in most cases the purpose has been to establish a common vocabulary and formalizing the software process concepts [13, 15, 30]. To our knowledge, none of the reviewed studies provides for the use of ontology reasoning in software process analysis and verification like our approach does.

6 Conclusion and Outlook

This paper presents a formal approach for software process verification and reasoning. A software process presented in natural language is firstly translated into a composition tree, and then to a description logic TBox with formal semantics. The DL presentation enables automated verification and reasoning tools to be used for process analysis. Using description logics, we modeled the Human resource process in the quality management process as a TBox against which organizational implemented processes in form of ABox at run time can be verified for consistency, completeness and compliance.

We plan to refine the modeling and apply our approach also to a wide spectrum of processes. For engineering processes. It can also be extended to other application domains and more complex scenarios. Besides this, our work can be extended in many ways. On the practical side we are working on developing an operational ontology for a given standard process against which organizational processes can be verified and reasoned for consistency and conformance. On the theoretical side, we are looking into the trade-off between expressiveness and performance as well as the limits of modeling and reasoning a software process in CT and DL, especially in comparison to other modeling and reasoning approaches for software process.

References

1. Feiler, P.H., Humphrey, W.S.: Software process development and enactment: Concepts and definitions. In: 2rd International Conference on Software Process, pp. 28–40 (1993)
2. Wen, L., Tuffley, D., Rout, T.: Using composition trees to model and compare software process. In: O'Connor, R.V., Rout, T., McCaffery, F., Dorling, A. (eds.) SPICE 2011. CCIS, vol. 155, pp. 1–15. Springer, Heidelberg (2011)
3. Curtis, B., Kellner, M., Over, J.: Process modeling. Commun. ACM **35**, 75–90 (1992)
4. ISO/IEC TR 24774. Software and systems engineering – Life cycle management – Guidelines for process description (2007)
5. ISO/IEC IEEE 12207 CD1 - revision of 12207:2008 Systems and software engineering Software life cycle processes (2014)
6. ISO/IEC TR 29110-5-1-2, Software engineering – Lifecycle profiles for Very Small Entities (VSEs): Management and engineering guide: Generic profile group: Basic profile (2011)

7. Wen, L., Rout, T.: Using composition trees to validate an entry profile of software engineering lifecycle profiles for very small entities (VSEs). In: Mas, A., Mesquida, A., Rout, T., O'Connor, R.V., Dorling, A. (eds.) SPICE 2012. CCIS, vol. 290, pp. 38–50. Springer, Heidelberg (2012)
8. Calvanese, D.: Description Logics for Conceptual Modeling. EPCL Basic Training Camp Dresden, Germany (2012)
9. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation and Applications. Cambridge University Press, Cambridge (2003)
10. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artif. Intell. **168**, 70–118 (2005)
11. Krotzsch, M., Simancik, F., Horrocks, I.: A Description Logic Primer. University of Oxford, Oxford (2013)
12. Bergman, M.: The Fundamental importance of keeping an ABox and TBox Split, AI3 Adaptive Information (2009). http://www.mkbergman.com
13. Thaddeus, S., Kasmir Raja, K.: Ontology for software Engineering Process Automation (2006). http://www.researchgate.net/publication/278241783
14. Acuna, S.T., Jusristo, N., Moreno, A.M.: A Software Process Model Handbook for Incorporating People's Capabilities XXVIII, 324 p. 90 (2005)
15. Rodriguez, D., Garcia, E., Sanchez, S., Nuzzi, C.R.: Defining software process model constraints with rules using OWL and SWRL. International Journal of Software Engineering and Knowledge Engineering, World Scientific Publishing Company (2010)
16. Behavior Engineering Web Site. http://www.behaviorengineering.org/
17. Dromey, R.G.: System Composition: Constructive Support for the Analysis and Design of Large Systems, SETE, Systems Engineering Conference, Brisbane, Australia (2005)
18. Wen, L., Dromey, R.G.: From Requirements Change to Design Change: A Formal Path. In: Proceedings of the 2nd IEEE International Conference on SEFM (2004)
19. Borgida, A., Lenzerini, M., Rosati, R.: Description logics for databases. In: [5], pp. 462– 484. Cambridge University Press (2003)
20. Horrocks, I., Kutz, O., Sattler, U.: The even more irresistible *SROIQ*. In: Doherty, P., Mylopoulos, J.,Welty, C. (eds.) Proceedings of the 10th International Conference on the Principles of Knowledge Representation and Reasoning KR, pp. 57–67. AAAI Press (2006)
21. Moor, A., Delugach, H.: Software process validation: comparing process and practice models. In: Eleventh International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2006. Conjunction with 18th Conference on Advanced Information Systems Engineering, Luxembourg (2006)
22. Yatapanage, N., Winter, K., Zafar, S.: Slicing behavior tree models for verification. In: Calude, C.S., Sassone, V. (eds.) TCS 2010. IFIP AICT, vol. 323, pp. 125–139. Springer, Heidelberg (2010)
23. Roedler, G.: An Overview of ISO/IEC/IEEE 15288, System Life Cycle Processes. Asian Pacific Council on Systems Engineering (APCOSE) Conference (2010)
24. Motik, B., Shearer, R., Horrocks, I.: A hypertableau calculus for SHIQ. In: Calvanese, D., Franconi, E., Haarslev, V., Lembo, D., Motik, B., Tessaris, S., Turhan, A.-Y. (eds.) Proceedings of the 2007 Description Logic Workshop (DL 2007) (2007)
25. Motik, B., Shearer, R., Horrocks, I.: Optimized reasoning in description logics using hypertableaux. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 67–83. Springer, Heidelberg (2007)
26. Sirin, E., Parsia, B.: Pellet system description. In: Parsia, B., Sattler, U., Toman, D. (eds.), Description Logics. CEUR Workshop Proceedings, vol. 189. CEUR-WS.org (2006)

27. Tsarkov, D., Horrocks, I.: FaCT ++ description logic reasoner: System description. In: Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), 2006. FaCT ++ download page http://owl.man.ac.uk/factplusplus/
28. ISO/IEC 33053 PDTS1 Information Technology — Process Assessment — Process reference model for quality management (2016)
29. Siegemund, K., Thomas, E., Zhao, Y., Pan, J., Assmann, U.: Towards ontology-driven requirements engineering. In: The 10th International Semantic Web Conference (ISWC2011)
30. Wang, S., Jin, L., Jin, C.: Represent software process engneering metamodel in description logic. In: Proceedings of World Academy of Science, Engineering and Technology. 11 ISSN 1307-6884 (2006)