Verifying Algorithm Traces and Fault Reason Determining Using Ontology Reasoning *

Oleg Sychev[1[0000-0002-7296-2538]], Mikhail Denisov[1[0000-0002-1216-610X]], and Anton Anikin[1[0000-0003-0661-4284]]

Volgograd State Technical University, Lenin Ave, 28, Volgograd, 400005, Russia
oasychev@gmail.com

Abstract. Domain ontologies that can solve various tasks using its concepts and determine fault reason for students' answers may serve as a good basis for creating for a testing system with extensive explanatory abilities. But capabilities of modern ontological reasoners may not be enough for this kind of task. In this study, we developed and tested an ontology able to build execution trace for the given algorithm containing sequences and alternatives and find fault reasons for incorrect traces. The study also showed problems with the used reasoner that hinder the developed ontology from becoming fully effective.

Keywords: Ontology · Reasoning · Error detection · Algorithm · Execution trace.

1 Introduction and Related Work

Ontology-based models and formal reasoning are used for knowledge representation in different domains for a wide range of applications. Ontology Driven Software Engineering (ODSE) approach [5] implies using ontology models for various aspects of the software engineering, (e.g. for modeling different parts of systems, software products, modules, algorithms).

The interesting and relevant topic within this aspect is algorithms' and programs' analysis and synthesis using ontology-based models [7]. Ontologies allow capturing program components and their relations as a formal model. The reasoning rules defined within ontologies using inference engines (or reasoners) allow solving a wide range of tasks from system modeling to software generation. These tasks include algorithmic structures analysis (e.g., canonical loop analysis to recognize whether a loop in the program code is in a canonical form), data access pattern analysis; alias, dependence, and static code analysis; program synthesis using an ontology-based algorithm description and formal reasoning rules defined for the specific programming language. For example, in [1] software requirement specifications (SRS) are defined using an ontology that becomes the

* The reported study was funded by RFBR, project number 20-07-00764.
Copyright ©2020 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR-WS.org/Vol-2721/paper495.pdf

conceptual model of the developed software and is mapped to the entities in the program code. This allows automating SRS detection and updating. The article [3] describes an ontology combining control-flow graph representation with the static trace of the program for source code analysis, showing a trend in supplementing the knowledge about the program structure with the knowledge about the program execution.

So ontology-based reasoning is a relevant approach for many tasks both in software engineering as a professional activity and in software engineering learning (e.g., to detect and explain the issues in the program code, to check the conformity of the task and conceptual model with the program implementation, etc). Ontologies are used in the learning process for different domains to generate and grade questions [2,6].

According to [2], the simplicity of the questions is one of the main limitations of existing works in question generation that must be addressed in further research. However, while for simple questions checking factual knowledge question generation and reasoning the correct answer are practically the same, for more complex questions, (e.g. building a program trace) question generation and question solving become different problems. Using automatically-generated questions is not practical without automating finding the correct answers, and reasoning on an ontology, capturing domain-related knowledge, is a relevant approach to this task.

Using complex questions targeting high cognitive levels also allows generating non-trivial feedback that is also an important problem in question generation. Explicitly defining logical constraints for a correct answer to the given question allows detecting the broken constraints if a student's answer is incorrect, making possible showing explanations of the domain laws that the student violated in their answer.

In our research, we aim at creating a system for generating questions, reasoning the correct answers to them using domain-dependent formal models like ontologies, determine fault reasons in students' answers and explaining them to correct misunderstandings of important domain concepts. One of the first steps requires studying whether modern ontological reasoners are capable of performing the necessary tasks. When teaching introductory programming courses, one of the complex tasks involving learning concepts of control structures is building program trace [4]. In this study, we attempted to develop an ontology for building algorithm traces and finding fault reasons for mistakes in students' traces to improve students understanding of how the algorithm is executed.

2 Method

An algorithm is represented as a tree of algorithmic structures: sequences, alternatives, and loops. Its trace is a sequence of acts of their execution in time; simple actions are represented as acts of their executions while complex algorithmic structures containing nested actions have different acts for the beginning and finishing of their execution. The trace also must contain the values of control

conditions for alternatives and loops.

The input of the developed ontology includes:

– An algorithm tree consisting of algorithmic structures and simple statements.

– The set of unordered execution acts to be connected by the `next_act` to embody the correct trace. Acts executing expressions have `expr_value` datatype property holding the result.

– The student's trace as `student_next` relations. Student's trace and correct trace share as many acts as possible.

The developed SWRL rules perform the following steps. First, the correct trace is built. The rules doing that are called positive. These rules look for a correct act in the partially built trace to connect the next act to. The connected act gets one of the subclasses of `correct_act` class, showing the reason why the act should be there (e.g., `AltBranchBegin`, `SequenceEnd`).

Second, the additional properties `parent_of` and `corresponding_end` are computed to easier navigating the correct trace (see fig. 1). Similar properties are calculated for the student's trace. Indices for acts in both traces are calculated too.

Third, the negative rules determine student's mistakes and fault reasons. Fault reason detection is performed by rules that infer subclasses of `Erroneous` class showing the fault reason like `CorrespondingEndMismatch`, `ExtraAct`, `DuplicateOfAct`, `MissingAct`, `DisplacedAct`, `BranchOfFalseCondition`, `NoBranchWhenConditionIsTrue`, etc.

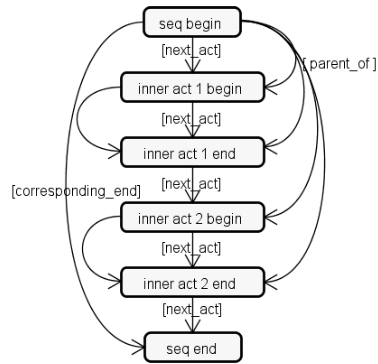

Fig. 1. Trace structure for two actions (1, 2) nested to a sequence (seq)

After the reasoning, the ontology is supplemented with the correct trace and facts about the types of student mistakes. This information can be extracted and shown to the students in a human-readable way using message templates for each fault-reason class, for example, "Act A can't be executed after the end of Act B because sequence B contains act A and each act of a sequence execution include executing each its element once" or "Branch A can't be executed at this place because the condition B is false" or "Act A must be executed at this place because a do-while loop body must be executed at least once."

3 Results

The current revision of the ontology[1] includes: 74 SWRL rules (28 positive, 28 negative, 18 helper rules), 66 classes (24 representing algorithm elements, 7 -

[1] https://github.com/den1s0v/c_owl

trace acts, 18 - mistakes, 27 - explanations for correct acts), and 30 properties.

The ontology supports sequences, alternatives, and loops of different types. It guarantees that at least one student's act is marked incorrect if there are any mistakes, even if all mistakes are omissions of correct acts: that makes providing detailed information about the mistakes' positions possible. Simultaneous reasoning over multiple traces injected into the same ontology is supported and can be used to improve the overall performance when processing multiple answers to the same question.

The developed SWRL rules require using SWRL built-ins like `swrlb:add()`, `swrlb:notEqual()`. Suitable reasoners include Pellet 2.3 and Pellet 3. Other popular reasoners like HermiT do not support these built-ins.

The ontology passed 33 tests consisting of short traces with annotated mistakes. A single run of the ontology with one trace takes about 1 second in Stanford Protégé Editor invoking Pellet 2.3 plugin and about 5 seconds while running Pellet 2.3 as a standalone process. The same ontology using Stardog with Pellet 3 cannot complete the inference in minutes.

A few issues of SWRL implementation in Pellet 2 and 3 were found.

– The problems with `DifferentFrom`: when no facts related to individuals distinction was provided a rule that uses `DifferentFrom` never fires; in Pellet 2 it sometimes causes "Inconsistent Ontology" error or silently truncates some of the inference results. This problem effectively stops the ontology from working. So it's better to avoid using `DifferentFrom` or `SameAs` unless you explicitly specify the information about the differences of individuals.

– In Stardog, using `SameAs` in the SWRL rule causes a parsing exception.

– In Stardog, the unification of "scalar" variables works only for binding; when matching with an already bound variable, it allows any values. With rule (1), Stardog will connect all possible pairs of `Product` instances, regardless of whether their prices are equal, instead of selecting only instances that match at a price. This behavior was only observed when `has_price` is a DatatypeProperty.

$$Product(?a), Product(?b), has_price(?a, ?p), has_price(?b, ?p) \rightarrow \\ \rightarrow has_same_price(?a, ?b) \qquad (1)$$

– Including transitive properties has a significant impact on the reasoning time if they are inferred. Two of the determined mistakes - `ExtraAct` and `MissingAct` - require looking up any number of acts ahead that is impossible without using transitive properties. As a workaround, we created several duplicates of the rules, each with a fixed length of the lookup (up to 6 acts).

The developed ontology differs from the ontology-based program analysis system [3] in that it performs dynamic analysis building the program trace and uses SWRL rules instead of Jena. The kinds of mistakes detected by analysis are different because our ontology is aimed at supporting teaching and it compares student-built trace with the ontology-built one instead of seeking errors in the given code.

Our work differs from other ontologies used for question generation and solving in that it can be used in developing a question-generating system aimed at

more complex questions targeting higher cognitive levels that makes reasoning the correct answer for the generated questions non-trivial as a question-solver and grading block. It also allows for generating explanatory feedback.

The proposed approach is limited to closed-ended questions because it requires that every possible mistake in the answer can be a result of violating a specific domain law. The developed ontology solves and finds mistakes for questions requiring ordering of a given set of program-trace strings. Open-ended questions are limited to the lower-lever feedback for now [6].

4 Conclusion and Future Work

During our research, we developed an ontology able to build correct traces for the given algorithm consisting of sequences, alternatives, and loops and find mistakes in the given traces with enough information to generate explanatory feedback. This shows that developing an ontology-based system for generating and solving questions and providing explanatory feedback to students about their mistakes is possible. However, some problems of Pellet reasoner prevent efficient implementation of the rules and limit the number of mistakes that can be detected.

Future work will concern adding support for procedure calls, including recursive calls, and developing the tutor system using the created ontology and logical rules.

References

1. Bhatia, M.P.S., Kumar, A., Beniwal, R., Malik, T.: Ontology driven software development for automatic detection and updation of software requirement specifications. Journal of Discrete Mathematical Sciences and Cryptography **23**(1), 197–208 (Jan 2020)
2. Kurdi, G., Leo, J., Parsia, B., Sattler, U., Al-Emari, S.: A systematic review of automatic question generation for educational purposes. International Journal of Artificial Intelligence in Education **30**(1), 121–204 (Nov 2019)
3. Pattipati, D.K., Nasre, R., Puligundla, S.K.: OPAL: An extensible framework for ontology-based program analysis. Software: Practice and Experience **50**(8), 1425–1462 (Mar 2020)
4. Risha, Z., Brusilovsky, P.: Making it smart: Converting static code into an interactive trace table. Sixth SPLICE Workshop "Building an Infrastructure for Computer Science Education Research and Practice at Scale" at ACM Learning at Scale 2020 (Aug 2020)
5. Strmečki, D., Magdalenić, I., Kermek, D.: An overview on the use of ontologies in software engineering. Journal of Computer Science **12**(12), 597–610 (Dec 2016)
6. Sychev, O., Anikin, A., Prokudin, A.: Automatic grading and hinting in open-ended text questions. Cognitive Systems Research **59**, 264–272 (Jan 2020)
7. Zhao, Y., Chen, G., Liao, C., Shen, X.: Towards Ontology-Based Program Analysis. In: ECOOP 2016. Leibniz International Proceedings in Informatics, vol. 56, pp. 26:1–26:25 (Jul 2016)