2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

ENRE: A Tool Framework for Extensible eNtity Relation Extraction

Wuxia Jin
Xi'an Jiaotong University
wx_jin@stu.xjtu.edu.cn

Yuanfang Cai, Rick Kazman
Drexel University, University of Hawaii
yfcai@cs.drexel.edu, kazman@hawaii.edu

Qinghua Zheng, Di Cui, Ting Liu
Xi'an Jiaotong University
qhzheng,cuidi,tingliu@mail.xjtu.edu.cn

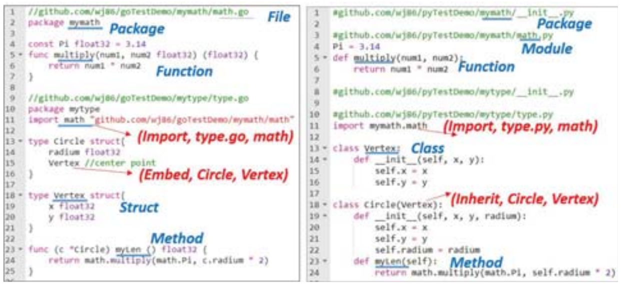
***Abstract*—Understanding the dependencies among code entities is fundamental to many software analysis tools and techniques. However, with the emergence of new programming languages and paradigms, the increasingly common practice of writing systems in multiple languages, and the increasing popularity of dynamic languages, no existing framework can reliably extract this information. That is, no tools exist to accurately extract dependencies from systems written in multiple and dynamic languages. To address this problem, we have designed and implemented the Extensible eNtity Relation Extraction (ENRE) framework. ENRE supports the extraction of entities and their dependencies from systems written in multiple languages, enables the customization of dependencies of interest to the user, and makes implicit dependencies explicit. To demonstrate feasibility of this framework, we developed two ENRE instances for analyzing Python and Golang programs. Our experiments on 12 Python and Golang projects demonstrated the effectiveness and flexibility of ENRE. By comparing with a commercial static analysis tool, we show that we can extract dependencies from Golang programs which are not supported by existing tools and we can reveal implicit dependencies in Python.**
(Demo Video: https://youtu.be/BfXp5bb1yqc)

***Index Terms*—Entity relation extraction; Implicit dependency; Python; Golang**

I. Introduction

Understanding source code entities—such as packages, functions, and classes—and their relations—such as *Import*, *Extend* and *Call*—is critical for many kinds of software analyses, ranging from basic coupling and cohesion measures, to architecture recovery [1], quality evaluation [2] [3], and malware analysis [4]. Modern software systems are often implemented using diverse programming languages to achieve better performance or quicker time-to-market. The dynamic and diverse nature of these systems presents challenges to existing static analysis tools:

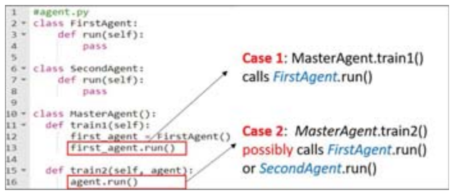
1) **How to support multiple languages and easily extend this support to accommodate new languages?** Different programming languages have different types of entities and relations. As shown in Figure 1, in Python, entities include package, module, class, function, method member, variable, etc. Relations include "Import", "Inherit", "Call", etc. The same relation can exist between different types of entities, e.g., in the case of "`import x`" or "`from x import y`", x can a package or a module, and y can be a module, class, function, etc. The concrete types of entities are different in different languages. As shown in Figure 1, Golang does not



```
//github.com/wj86/goTestDemo/mymath/math.go
package mymath

const Pi float32 = 3.14
func multiply(num1, num2 float32) (float32) {
    return num1 * num2
}

//github.com/wj86/goTestDemo/mytype/type.go
package mytype
import math "github.com/wj86/goTestDemo/mymath/math"

type Circle struct{
    radium float32
    Vertex //center point
}

type Vertex struct{
    x float32
    y float32
}

func (c *Circle) myLen () float32 {
    return math.multiply(math.Pi, c.radium * 2)
}
```

```
#github.com/wj86/pyTestDemo/mymath/__init__.py

#github.com/wj86/pyTestDemo/mymath/math.py
Pi = 3.14
def multiply(num1, num2):
    return num1 * num2

#github.com/wj86/pyTestDemo/mytype/__init__.py

#github.com/wj86/pyTestDemo/mytype/type.py
import mymath.math

class Vertex:
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Circle(Vertex):
    def __init__(self, x, y, radium):
        self.x = x
        self.y = y
        self.radium = radium
    def myLen(self):
        return math.multiply(math.Pi, self.radium * 2)
```

Fig. 1: Entity and relation diversity in Golang and Python



```
#agent.py
class FirstAgent:
    def run(self):
        pass

class SecondAgent:
    def run(self):
        pass

class MasterAgent():
  def train1(self):
      first_agent = FirstAgent()
      first_agent.run()

  def train2(self, agent):
      agent.run()
```

Case 1: MasterAgent.train1() calls FirstAgent.run()
Case 2: MasterAgent.train2() possibly calls FirstAgent.run() or SecondAgent.run()

Fig. 2: Call relations which Understand does not resolve

have *class* while Python does. A new programming language may have additional types of entities and relations. Therefore, we need a framework that is general enough to accommodate different types of entities and relations.

2) **How to identify implicit relations among entities**? Many languages, such as Python and Javascript, are dynamically typed. Many relations among entities are implicit and not detectable by traditional static analysis tools. As illustrated in Figure 2, both function `train1()` and `train2()` call a method named `run()`, which can belong to either class `FirstAgent` or `SecondAgent`. When we use Understand [5], one of most widely used commercial reverse-engineering tools, to extract such dependencies, we observed that it ignores the relation from `train2()` to either `FirstAgent` or `SecondAgent`, due to the fact that it cannot resolve which classes the function `run`—this can only be determined at runtime. This presents a problem for any analysis that relies on coupling relations: the potentially strong dependencies between `train2` to the other two classes are ignored.

To address these challenges, we have created a frame-

TABLE I: Tool features for code dependency analysis

Tool	Multi-Relation	Multi-Language	Implicit-Relation	Unified-Representation	Open-Source
Code2Graph	✗	✗	–	✗	–
Rexdep	✗	✓	✗	✗	✓
Doxygen	✗	✓	✗	✗	✓
Understand	✓	✓	✗	–	✗
ENRE	✓	✓	✓	✓	✓

work which: 1) **supports multiple entities and relations**, 2) **supports multiple programming languages**, 3) **manifests dynamic, implicit relations**, 4) **provides a unified representation**, and 5) **is open-source**.

Many static analysis tools have been proposed for extracting entities and their relations; we list some examples in Table I. Code2graph [6] provides *Call* relation extraction between function entities from Python. Rexdep [7] extracts *Import* relations at the file and package level, but does not identify relations between classes or functions. Doxygen [8] can generate documentation from source code. It supports building UML diagrams with *Extend* (or Inherit) relations between classes. Both Rexdep and Doxygen can process multiple languages, but can not extract multiple relations between code entities. SciTool's Understand [5] can extract most entities and relations and supports multiple languages, but ignores relations among dynamic entities. Also, since it is not open source, it can not be (easily) extended to process new languages.

In this paper we propose the **Extensible eNtity Relation Extraction** (**ENRE**[1]) framework to extract entities and relations from different programming languages. ENRE satisfies all requirements in Table I. Our contributions are as follows:
1) We define a unified entity and relation representation, which is the foundation of a generic framework to process different languages. In particular, we define *Dynamic relation* which makes implicit relations in dynamic languages explicit.
2) We propose the ENRE framework to flexibly support entity and relation extraction for new language analysis by extending just two framework components.
3) We have instantiated the ENRE framework for Python and Golang. We begin with these two languages because Python is a popular dynamic language, and Golang is a relatively new language that is not supported by most existing tools.

To demonstrate feasibility and validity, we have extracted entities and relations from 5 Golang and 7 Python projects using ENRE. In addition we have conducted an accuracy check by manually comparing the results with dependencies extracted from Understand for 2 small Python projects (noting that Understand does not support Golang at all).

II. The ENRE Framework

ENRE is designed as a general, extensible framework for extracting entities and relations from programming languages. We first define unified entity and relation representations, and then introduce the framework and components of ENRE. Finally, we describe ENRE's implementation.

A. Unified Entity Relation Representation (UERR)

Unified representations of entities and relations are the foundation of our general framework. Users can define concrete entities and relations for different languages by extending predefined base entities and relations.

1) Entity: A code entity is an object with a given name or identifier. We define 5 base types of code entities: *Package* (code folder), *File*, *Class* (e.g. class, interface, struct), *Function* (e.g. function, method member) and *Variable* (e.g. variable member, local or global variable), as shown in Figure 1. An entity has the following attributes: `id`, `name`, `shortName`, `containerId`, `childrenIdList`.

2) Entity Relation: Entity relations are relationships between code entities. There is a relation from $entity_i$ to $entity_j$ when $entity_i$ depends on $entity_j$ to complete its functionality. We define 8 types of relations: *Import*, *Extend* (e.g. Implement, Inherit), *Call*, *Set* (a function modifies a variable), *Use* (a function reads or uses a variable), *Parameter Type* (a function takes a class type as a parameter), *Return Type* (a function returns a class type), and *Dynamic relation*. A relation has the following attributes: `sourceId`, `destId`, `type`, `weight`. For instance, a *Call* relation has `sourceId` as the id of the calling entity, `destId` as id of the called entity, `type` as "Call", `weight` as the number of the call instances between the two entities. We now elaborate how to define and extract the 8th relation, *dynamic (implicit) relations*, as illustrated in Figure 2.

Dynamic relations are relations that can only be precisely resolved at run-time. The difference between static and dynamic relations are mainly manifested in *Call* relations. We define two dynamic dependencies for *Call* relations when resolving a called identity, such as `object.m()`, where the type of *object* cannot be statically determined.
1) Internal Case: The type of a callee is instantiated in the code scope visible for the caller, as illustrated in Figure 2: `first_agent` is a local object created at Line 12. In this case, we detect that `MasterAgent.train1()` calls `FirstAgent.run()`, and resolve the class type of the object through its instantiation statement.
2) External Case: The type of a callee is instantiated beyond the code scope visible for the caller. The class object is passed into the caller as a parameter. Based on the Duck typing[2], we resolve the callee as the method member of the classes which define this method member. As shown in Case 2 in Figure 2, we identify two possible call relations: `MasterAgent.train2()` possibly calls `FirstAgent.run()` or `SecondAgent.run()`.

B. ENRE Components

Figure 3 illustrates the ENRE framework. To process a project written in a new programming language, a user should extend the framework and implement a few concrete components. We call a dependency extraction program extended from ENRE as an *ENRE engine*. The input of an ENRE

[1]https://github.com/jinwuxia/ENRE

[2]https://en.wikipedia.org/wiki/Duck_typing

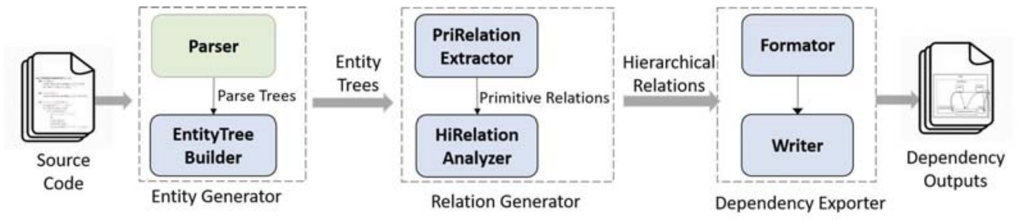


Fig. 3: Framework of ENRE

engine is *Source Code* that will be analyzed. ENRE processes the source code and outputs dependency relations. Currently ENRE provides JSON, XML and DOT output, and can be extended to support other output formats.

We design ENRE using a Pipe-And-Filter pattern, so that each component can be replaced without influencing others. It consists of three parts:

1) Entity Generator. This component translates *Source Code* into *Entity Trees*, an intermediate representation. It includes two components:

a) *Parser*. We use Antlr [9] as the underlying *Parser* that conducts lexical and syntactic analysis of source code. It generates *Parse Trees* using the grammar rules of a given language.

b) *EntityTree Builder*. This component traverses the *Parse Trees* to build *Entity Trees*. Each node in an *Entity Tree* is a concrete entity extending the base entities defined in UERR. Each node can have a parent node and multiple child nodes. For example, if file f_j is defined within package p_i, the parent of f_j is p_i, and p_i has a child f_j. Nodes in *Entity Trees* are annotated with relation-relevant semantics, such as "import" or "function call" information. This component is language *dependent*; an analyst needs to identify and implement concrete entities from parse trees for new languages.

2) Relation Generator. This component extracts and analyzes relations from *Entity Trees*, using two components:

a) *PriRelation Extractor*. This component extracts entity relations by analyzing the annotations in *Entity Trees*. For example, f_1 node contains a function call annotation like $v.f_2()$. This component will search and resolve names v and f_2, deciding which entity this name f_2 is paired with. After processing all annotations, it generates all *Primitive Relations* which are direct relations in code. This component is language-dependent because different languages have different semantics for name scoping and resolution.

b) *HiRelation Analyzer*. According to hierarchical scope, this component processes *Primitive Relations* and generates *Hierarchical Relations*, e.g. which packages contain which files, which files contain which functions, etc.

3) Dependency Exporter. This transforms the relations extracted from the previous two components into a specified format, using two components:

a) *Formator*. This defines data models to wrap entity relations. The *Formator* can use popular data models such as *GraphViz DOT* or customized ones.

b) *Writer*. This outputs the formatted entities and relations into file types like .CSV, .XML or .JSON.

To create an engine for a given programming language, the user only needs to extend **EntityTree Builder** and **PriRelation Extractor** to accommodate different entity and relation types. The user can also extend the language-irrelevant components, e.g. to customize a different output format.

C. Implementation

We have implemented the ENRE framework and two ENRE engines to explore feasibility. In the *EntityTree Builder* component and *PriRelation Extractor*, we provide interfaces for extensions to analyze different languages. We also implemented other language-independent components, we implemented 3 data models in *Formator*—2 customized models and a *Graphviz DOT* model—and we implemented XML, JSON and DOT file *Writer*s.

We have implemented ENRE engines to process Python code and Golang respectively by extending *EntityTree Builder* and *PriRelation Extractor*. For Golang, we have extracted various entities and 7 types of relations except Dynamic relation, since Golang is statically typed. For Python, we have extracted entities and 6 types of relations, omitting *Return Type* and *Parameter Type* relations. The two relations are associated with the passed or returned types which are determined dynamically in Python.

III. Preliminary Evaluation

As our framework is the first to support *general* dependency extraction, the objective of our initial evaluation is to test its ability to support multiple languages.

A. Accuracy Verification for ENRE

To verify the accuracy of ENRE tool, we used the ENRE engines and processed 5 Golang and 5 Python projects of different scales as subjects. Tables II and III list the number of concrete entities and unique entity relations extracted using the two ENRE engines, and their execution times. Related data and output can be found at *https://github.com/wj86/ENRE-data*.

B. Sanity Check for ENRE

To the best of our knowledge, ENRE is the first framework supporting Golang dependency extraction, and we cannot find a comparable tool to assess its accuracy. For Python, we use the dependencies extracted by Understand [5] as a benchmark

TABLE II: Summary of ENRE results in Golang projects

Concrete Entity	Beego	Hugo	Dep	Gogs	Aws-sdk-go
Package	37	67	213	144	453
File	205	382	629	1,029	1,364
Struct	265	327	1,790	2,425	13,410
Alias	69	59	596	796	97
Interface	31	57	54	146	221
Function	766	1,188	5,394	7,562	3,846
Method Member	1,290	1,293	1,661	5,182	83,841
Entity Relation					
Import	95	530	225	389	3,358
Extend	27	67	60	170	45
Call	660	1,295	9,869	11,005	7,619
Set	3,875	6,501	14,783	24,078	100,437
Use	7,150	10,208	28,942	46,997	206,890
Parameter Type	290	462	2,594	3,336	34,044
Return Type	384	542	715	2,116	57,294
ExecutionTime	27.06sec	45.17sec	4.35min	7.03min	7.53min

TABLE III: Summary of ENRE results in Python Projects

Concrete Entity	Voc	Scrapy	Django	Tensorflow-python	Tensorflow-models
Package	9	32	567	47	143
File	182	291	2,524	1,216	1,523
Class	610	614	6,048	2,545	1,360
Function	102	243	1,514	3,671	3,852
Method Member	2,655	2,959	25,690	21,293	7,406
Entity Relation					
Import	368	804	573	8	1,836
Extend	266	192	1,467	390	402
Set	647	3,804	28,383	64,383	41,814
Use	2,378	6,221	36,331	78,332	57,957
Call	1,118	1,509	2,951	8,421	8,430
DynamicCall(Internal)	176	1,307	5,580	6,874	2,885
DynamicCall(External)	4,423	5,952	295,881	191,440	32,968
ExecutionTime	13.98sec	15.43sec	1.90min	3.64min	2.30min

to assess ENRE's accuracy. We chose 5 entity types and 3 common relations for accuracy verification on 2 Python projects. The results obtained by Understand and ENRE are shown in Table IV. To find the roots causes for the differences we manually inspect the source code of each project.

For entities, Table IV shows ENRE gets the same number of Package or File entities as Understand, but ENRE extracted fewer functions and more method members, because 1) ENRE only counts the outermost classes (or functions), ignoring their inner ones, since inner ones are often for encapsulation. Understand counts both inner and outermost ones, hence generates more functions. 2) ENRE always counts `__init__` methods for class instantiation whether explicitly defined or not. Understand only counts the explicitly defined ones, hence generates fewer method members than ENRE.

TABLE IV: Result summary by ENRE and Understand

	python-fire		python-patterns	
	ENRE	Understand	ENRE	Understand
Concrete Entity				
Package	6	6	8	8
File	41	41	69	69
Class	58	58	160	162
Function	64	71	33	41
Method Member	300	255	497	412
Entity Relation				
Import	63	63	56	56
Extend	18	18	39	39
Call	299	299	220	220
DynamicCall(Internal)	29	6	86	16
DynamicCall(External)	53	-	365	-

For relations, Table IV shows that ENRE extracts the same number of *Import* and *Extend* relations as Understand. But ENRE extracts additional dynamic call relations. Understand resolves partial *Dynamic Internal Call* relations only when the object variable is `self` (that is, the callee is like `self.m()`).

Overall our evaluation shows that our extraction is quite accurate for Python but goes beyond what Understand can extract in terms of dynamic relations. We are in the process of conducting more evaluations for verification. To our knowledge, no existing tools are available to extract Golang dependencies. We have manually verified the ENRE accuracy on smaller Golang projects (the results are included in the previous github URL). Table II shows ENRE can analyze large-scale Golang projects, and we will manually inspect these Golang projects to further validate ENRE.

IV. Conclusion

This paper introduces a general, extensible code entity and relation extraction framework, ENRE. Users can extend this framework to extract dependencies in different programming languages.We have implemented two ENRE engines to process Python and Golang to test the feasibility and efficiency of ENRE. To our best knowledge, ENRE is the first framework that supports Golang dependency extraction, and a comparison with Python dependencies extracted from Understand verified that implicit dependencies missed by Understand can be detected by ENRE.

ACKNOWLEDGMENT

This work was supported by National Key RD Program of China (2016YFB1000903), National Natural Science Foundation of China (61632015, 61772408, U1766215, 61721002, 61532015, 61833015), Ministry of Education Innovation Research Team (IRT_17R86).

References

[1] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidovic, and R. Kroeger, "Comparing software architecture recovery techniques using accurate dependencies," in *Proceedings of the 37th Intl Conference on Software Engineering*, vol. 2, 2015, pp. 69–78.
[2] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th Intl Conference on Software Engineering*, 2016, pp. 499–510.
[3] W. Jin, T. Liu, Y. Qu, Q. Zheng, D. Cui, and J. Chi, "Dynamic structure measurement for distributed software," *Software Quality Journal*, vol. 26, no. 3, pp. 1119–1145, 2018.
[4] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Transactions on Information Forensics and Security*, vol. 13, no. 8, pp. 1890–1905, Aug 2018.
[5] "Understand," *https://scitools.com/features/*.
[6] G. Gharibi, R. Tripathi, and Y. Lee, "Code2graph: automatic generation of static call graphs for python source code," in *Proceedings of the 33rd Intl Conference on Automated Software Engineering*, 2018, pp. 880–883.
[7] "rexdep," *https://github.com/itchyny/rexdep*.
[8] "Doxygen," *http://www.doxygen.nl*.
[9] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.