Synthesis of Logic Interpretations

Jian Xiang, John Knight, Kevin Sullivan
Department of Computer Science
University of Virginia
Charlottesville, VA USA
{Jian,Knight,Sullivan}@cs.virginia.edu

***Abstract*— Logic interpretations define the relationships between real-world entities and their logic representations. We introduce an explicit structure for documenting interpretations based on real-world types. As we have demonstrated previously, the benefits of using real-world types can be considerable, but the effort required might deter programmers. In this paper, we introduce a semi-automated mechanism that: (a) synthesizes candidate real-world types, (b) infers real-world type bindings for program variables, and (c) synthesizes candidate type rules from verified or trusted programs. We illustrate the synthesis approach using an open-source project for which we have previously developed a complete real-world type system.**

***Keywords*— Logic interpretation, real-world types, software reliability**

I. Introduction

The notations that are used for defining software are formal languages. High-level languages, assembly languages and machine languages are *all* formal and all have the property that, as formal languages, they have no inherent real-world meaning. For any statement in a formal language to be anything other than a purely syntactic entity, an *interpretation* has to be added to the logic. The interpretation defines the intended meaning in the real world of elements of the logic. In doing so, the interpretation exposes the logic to constraints and invariants that derive from the real world, such as the laws of physics. To be valid, the logic must conform to these constraints and invariants, and so they provide a rich opportunity for error checking of the software.

In practice, the interpretation of a software system is *always* present, but usually documented in an ad hoc, informal and sometimes implicit manner using casual techniques such as "descriptive" comments, "meaningful" identifiers, and design documents. We note that the execution of software is unchanged by the replacement of identifiers with random strings, and the removal of comments and design documents. The logic is unaffected by these changes, but human understanding of what the logic does is mostly destroyed.

The importance of interpretation can be seen when one notes that many software systems, especially cyber-physical systems, interact with the real world. Such systems are often safety critical, and assurance of their correct operation depends on the interaction being complete and correct. Unless the interaction is documented carefully, doubt in the correctness of the effects of such systems is inevitable.

In previous work, we showed that an explicit interpretation provides a new capability for detecting software faults, and we demonstrated its performance [21]. An interpretation allows automated detection of faults that result from misuse of real-world entities or violate real-world constraints. In case studies, analyses revealed both unrecognized faults and faults that had been reported as bugs in real systems.

In this paper we make two contributions: (a) we introduce a preliminary explicit content and structure for an interpretation, and (b) we introduce a semi-automatic *synthesis* system for creating draft interpretations from available system assets. Making the interpretation explicit eliminates the ad hoc and informal form that is used currently. The effort in constructing an interpretation might be considerable, and so semi-automatic synthesis is designed to assist engineers and reduce that effort.

The explicit documentation of the interpretation of logic provides three advantages: (a) it informs the software design of the entities that the software will affect enabling better design choices, (b) it documents essential reference materials in a centralized and well-defined form allowing rigorous examination of the logic for correctness and completeness by human inspection, and (c) the real-world constraints and invariants that the interpretation exposes can be checked providing a new mechanism for detecting software faults.

To investigate the utility of the synthesis system concepts presented in this paper, we have developed a prototype that we applied to Java. The prototype analyzes Java source programs based on naming conventions, and presents the synthesized candidate artifacts to the programmer ranked by preliminary suggestions. We conducted a case study on a moderate-sized project for which separately we have developed a complete real-world type system. The results show that most of the real-world types needed in the project can be located in the synthesized candidates as can the type rules and bindings.

In the next Section, we summarize the interpretation concept and introduce a structure for interpretations based on real-world types. In Sections III through VII we discuss the concept of real-world types and the synthesis of interpretations, real-world types, real-world type bindings, and real-world type rules. In Section VIII we present a case study assessment of the synthesis mechanism, and in Section IX we review related work. Finally, in Section X we present our conclusions.

II. Concept of Interpretation

An interpretation defines the real-world meaning of an element of logic. For example, an integer variable in an avionics program might be used to represent the actual altitude of an aircraft. Within the logic of the software, the variable is merely an integer. The role of the interpretation is to reveal everything about the *actual* altitude of the aircraft.

Figure 1 illustrates this idea. The system design process starts with a problem to be solved in the real world and develops a concrete solution in logic. The interpretation provides the real-world details of an entity in the logic.

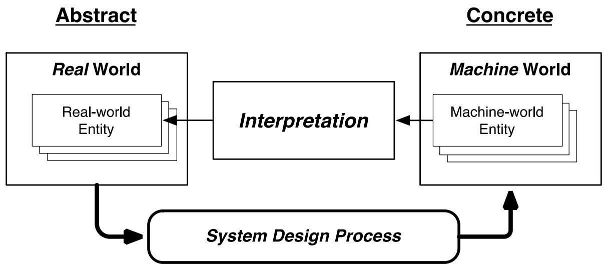

Fig. 1. Interpretation as an abstraction from concrete to abstract.

Without an explicit interpretation, important characteristics of real-world entities, such as units and dimensions, and associated real-world constraints, such as not mixing units, are stated and enforced either in ad-hoc ways or not at all. In addition, crucial relationships between logic representations and real-world entities, such as accuracy of sensed values, remain under specified. The result is that programs tend to treat representations in logic as if they were isomorphic to the real-world entities. This practice leads to the introduction of faults into systems due to unrecognized discrepancies, and executions end up violating rules inherited from the real world [15].

As an example, consider again the altitude of an aircraft and the representation of altitude in avionics software. Aircraft altitude is measured in units (feet, meters, etc.) and has the fundamental physical dimension of length. The meaning of altitude depends on the measurement origin and direction, i.e., the frame of reference. Whatever value the logic representation has, a sensor will have determined that value, and so the logic value will be of limited precision and accuracy, and will be the value when the sensor sample was taken, not the "current" value. With an explicit interpretation, units and dimensions can be checked in logic expressions, as can consistency of attributes such as measurement origin and orientation. Programming languages have been enhanced to permit unit and dimensional analysis in the past but not to support the comprehensive notion of interpretation [10, 12].

Defining the content and structure of an effective and complete interpretation is a significant challenge, and in this paper we define a preliminary form based on the concept of a *real-world type* system [21].

III. The Real World Type Concept

A real-world type is the real-world analog of a type in a formal language. A real-world type defines the values that a physical entity in the *real world* of that type can have and the operations in which it can engage. More generally, a real-world type *system* documents: (a) the real-world attributes associated with the types, (b) the type rules that define allowable operations on entities of the various types, (c) the machine representations for entities of the real-world types, and (d) the relations between real-world entities and their machine representation. Our preliminary structure for an explicit interpretation is a set of real-world types. A real-world type system and its connections to an application system of interest are shown in Figure 2.

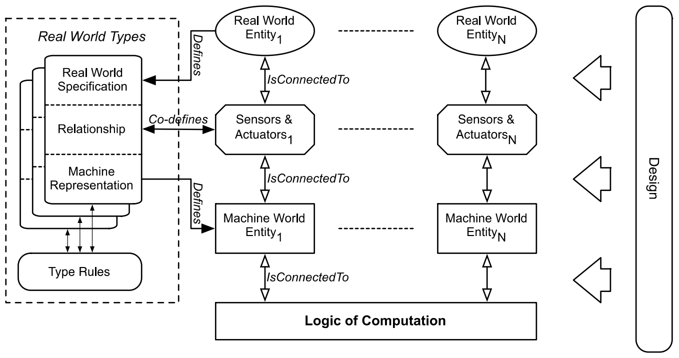

Fig. 2. Real-world type system structure

Real-world types are designed to facilitate design in software engineering *from* the real world *to* the machine world so as to enable all relevant aspects of the real world to be considered in developing a computer system. As such, a real-world type system is not expressible in its entirety within a mathematical framework. In general, real-world types cannot be expressed conveniently in the type systems of modern programming languages, because of the large number of distinct attributes that real-world types typically include.

IV. Synthesis of Interpretations

The benefits of an interpretation using our prototype interpretation structure can be considerable. Nevertheless, the effort involved in defining the interpretation for a particular application could be significant. The entities in programs that can have real-world types are variables, constants, functions and expressions. To create an interpretation based on real-world types, users have to: (1) develop real-world type definitions, (2) define the bindings between program entities and the necessary types, and (3) define the associated type rules.

An overview of the synthesis mechanism is shown in Figure 3. Starting with a variety of assets including the target application's sources, other application documents, references (including ontologies, dictionaries, and other natural-language information), the synthesis mechanism: (1) extracts candidate real-world type definitions from the subject software, (2) infers candidate type bindings from program statements for which developers have high confidence, and from defined type inference rules, and (3) extracts candidate type rules from a verified or otherwise trusted program. The synthesis mechanism produces *candidate* artifacts that are potentially incomplete, inconsistent, of no value, or otherwise deficient. For example, developers might have used different identifier naming conventions. In order to determine which of the candidates is of value, the synthesis phase is followed by inspection, selection, and completion by software engineers and domain experts.

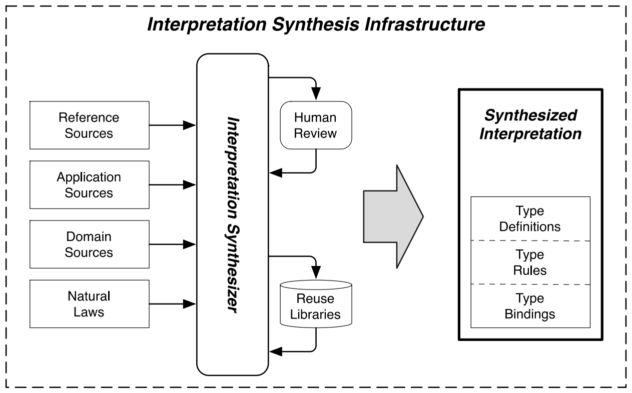

Fig. 3. Real-world type synthesis system overview

V. Synthesis of Real-World Types

A. Type Synthesis Process

Synthesizing a real-world type necessitates recovery of details of a complex, composite entity. A real-world type consists of: (1) a name that identifies the type, (2) a logic entity that is relevant to the computation, and (3) one or more attributes that describe the associated real-world characteristics. Significant challenges to identifying these structures include:

- Distinguishing accurately between the three major elements of the composite entity.
- Determining whether the list of attributes is complete and accurate in the sense that all relevant real-world characteristics are documented fully.
- Composing information about the same type that is spread across multiple sources where there are slight variations in the information, such as text that includes both singular and plural instances of a term.

For purposes of synthesis, we hypothesize that much of the necessary information is encoded in the likely existing "informal" interpretation, i.e., identifiers, comments and other documents that programmers prepare. Programmers usually follow widely known and commonly adopted naming conventions. They follow the same (often implicit) grammatical rules for names of program elements that are structurally the same. These rules indicate the role of each term in a name. For example, method names are often constructed from verbs that are followed by nouns, while class names are frequently sequences of nouns.

By processing program source text, the synthesizer leverages these naming conventions to generate a set of terms and linkages between the terms. Along with a set of reference sources, these materials are used to create or select from a library a set of candidate type definitions automatically. The synthesizer then supports a human-analysis phase in which useful type definitions are selected from the candidates.

The steps followed by the synthesizer are illustrated in Figure 4. The individual steps operate as follows:

- **Source Parser**. The source parser parses the program source code and locates all of the identifiers in use.

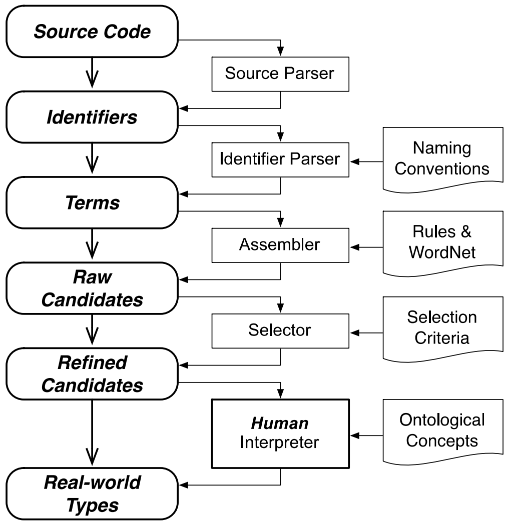

Fig. 4. Synthesis of candidates for real-world types

- **Identifier Parser**. The identifier parser parses the identifiers using a grammar based on the naming conventions, such as camelCase and underscores, and then, for each identifier, produces a list of the words and acronyms present within the identifier. We refer to these words and acronyms as *terms.*
- **Assembler**. For each term located in an identifier, the WordNet lexical database is consulted to determine whether the term is a noun [20]. Nouns are referred to as *major terms* and other terms as *associated* terms. For each major term, a list, the *term list*, of associated terms that occurred with the major term in an identifier is computed to form a *term entry,* a {major term, term list} pair. Terms in the term list could be nouns, and so a single identifier could yield multiple term entries. Multiple term entries for the same major term are combined, and the frequency of occurrence of the term across all identifiers is computed. Finally, the frequencies with which each associated term in the term list occurred in the same identifier as the major term are computed, and the entries are combined into a set of raw type candidates.

  WordNet's lemma is used to normalize every major term, i.e., plural and abbreviation forms of the same term are merged. The report sums the frequencies of each major term, and frequencies of each term in the term list of a major term.
- **Selector**. The list of raw candidates synthesized by the Assembler contains a list of major terms, and a list of associated terms for each major term. The Selector prioritizes the candidates based on a selection criterion. The selection criterion is not fixed and several possible criteria could be used.
- **Interpreter**. The role of the Interpreter is to apply human insight to the candidate types. Programmers and domain experts can review the prioritized list of candidates and select important major terms as type names. The associated terms of a major term suggest real-world

attributes, and, again, programmers and domain experts can refer to domain models, ontologies, personal experience, and application specific information to elaborate candidate types.

B. *Type Synthesis Example*

The following code snippet comes from the project used in the case study discussed in Section VIII:

```
timeToTOC  = altToTOC/plan.getAircraft()
                          .getClimbRate()/60;
altToBOD   = altitude - to.getElevation();
```

The **source parser** extracts nine identifiers in total. Five of the identifiers come from the first statement and the other four identifiers come from the second statement.

The **identifier parser** splits these identifiers into terms:

```
timeToTOC         =>    time, to, TOC
altToTOC          =>    alt, to, TOC
plan              =>    plan
getAircraft       =>    get, aircraft
getClimbRate      =>    get, climb, rate
altToBOD          =>    alt, to, BOD
altitude          =>    altitude
to                =>    to
getElevation      =>    get, elevation
```

The **assembler** identifies the major terms and constructs the term lists. The major term is shown to the left of colon and the term list to the right:

```
time, to, TOC     => time        : to, TOC
alt, to, TOC      => alt         : to, TOC
plan              => plan        : NONE
get, aircraft     => aircraft    : get
get, climb, rate  => rate        : get, climb
alt, to, BOD      => alt         : to, BOD
altitude          => altitude    : NONE
to                => NONE
get, elevation       => elevation   : get
```

The results for a single major term are accumulated, and the frequency of occurrence of the major term and the frequencies of occurrence of the associated terms are determined:

```
time(1)        : to(1), TOC(1)
alt(2)         : to(2), TOC(1), BOD(1)
altitude(1)    :
plan(1)        :
aircraft(1)    : get(1)
rate(1)        : get(1), climb(1)
elevation(1)   : get(1)
```

Finally, normalized forms of the same major term are coalesced based on likely abbreviations, plurals, etc.:

```
time(1)        : to(1), TOC(1)
altitude(3)    : to(2), TOC(1), BOD(1)
plan(1)        :
aircraft(1)    : get(1)
rate(1)        : get(1), climb(1)
elevation(1)   : get(1)
```

Here, the term `alt` is an abbreviation of `altitude`, so the entry for `alt` is merged into the entry for `altitude`.

The **selector** sorts the results of the assembler using a changeable criterion, frequency in this example, to produce the list of type candidates:

```
altitude(3)    : to(2), TOC(1), BOD(1)
aircraft(1)    : get(1)
elevation(1)   : get(1)
plan(1)        :
rate(1)        : get(1), climb(1)
time(1)        : to(1), TOC(1)
```

The **interpreter** (a human) then constructs a final list of type candidates. The associated terms for each major term can help suggest real-world attributes:

```
Type_1                  :  time
  Possible attribute    :  NONE
Type_2                  :  altitude
  Possible attribute    :  reference point
Type_3                  :  climb_rate
  Possible attribute    :  direction of movement
Type 4                  :  elevation
  Possible attribute    :  NONE
```

The associated term `BOD` (bottom of decrease) appears with the major term `altitude`. The term `BOD` refers to the lowest `altitude`, which could be either the local ground or mean sea level. A real-world attribute `reference point` is useful for the type `altitude`. Similarly, the major term `rate` has an associated term `climb`. The term `climb` implies the direction of the movement, and so for the type `climb_rate`, `direction of movement` is likely one of the real-world attributes.

VI. SYNTHESIS OF REAL-WORLD TYPE BINDINGS

The primary principle upon which candidate bindings are synthesized is *inference*. Inference has to be "seeded" by an initial set of bindings created by developers, and those bindings are then propagated algorithmically using a set of inference concepts and an associated inference process.

A. *Binding Synthesis Concepts*

The prototype supports three types of inference:

- **Parameter inference**. Parameter type bindings in method declarations are propagated to arguments in method invocations.
- **Return statement inference**. Types bound to return values are propagated to method signatures.
- **Assignment inference**. Type bindings in assignment statements are propagated from one side to the other.

Figure 5 illustrates the parameter inference approach. The individual steps are as follows:

- **Trusted method selection**. Users select a list of methods that they trust, i.e., methods of which all invocations are assumed to be coded correctly.
- **Invocation location**. All invocations of the methods in the list are located.
- **Parameter and argument location**. A parser produces an abstract syntax tree for the program. For method declarations, the parser retrieves parameters and their real-world types. For method invocations, the parser locates the arguments so that the binder can process them in the next step.
- **Binding**. The parameter types are bound to the corresponding arguments. If an argument has been bound previously to an inconsistent real-world type, an error message is issued.

As an example, consider the following method declaration:

```
void setLatitude(double l){
  this.lat = l;}
```

The parser determines that the real-world type for parameter `l` is `latitude_geocentric` as set by the user. Two invocations are located in the source files:

```
double lat1, lat2 = 0;
setLatitude(lat1);
setLatitude(lat2);
```

The type `latitude_geocentric` is bound to variables `lat1` and `lat2`. These bindings will persist so that subsequent analyses will consider `lat1` and `lat2` as variables of known type `latitude_geocentric`.

For return statements, a real-world type might have been bound to the return value of the method. If this is the case and if the method declaration contains only one return statement, then the return type of the method signature will be bound to the same type.

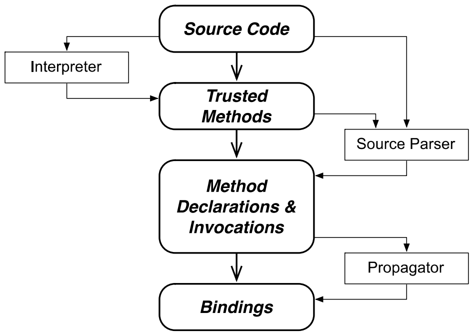

Fig. 5. Type binding inference from parameters to arguments

Assignment inference can exploit many different heuristics, and the prototype synthesis mechanism uses two. In the first, if either side of an assignment has a real-world type binding but the other does not, then the unbound side is bound to the same type as the bound side.

The second inference heuristic is based on a simple pattern matching approach. The algorithm accumulates details of assignment statements in which the right-hand side of each assignment is bound to the same single type and the left-hand side is unbound except for a single instance. If the total number of such assignments exceeds a threshold, then all of the left-hand sides are bound to the type of the single bound left-hand side. Clearly, a wide variety of machine learning techniques could be used to improve assignment inference.

B. Binding Synthesis Process

The binding inference process combines seeding steps with inference steps in an attempt to bind as many program entities as possible. The process is organized into three stages: (1) the *field* stage, (2) the *method* stage, and (3) the *local variable* stage. Each stage focuses on a single type of program element and combines seeding with use of one or more inference concept. The process is iterative and inference steps are repeated in sequence until no new bindings are generated.

Field stage. In this stage, developers seed bindings to fields in class definitions. Classes frequently contain "get" and "set" methods for these fields, and types can be bound to these methods if the field is bound. After developers bind real-world types to all of the fields in class definitions, return statement inference and assignment inference can be invoked. For example, consider this class:

```
public class location{
  double latitude;
  private double getLatitude(){
     return latitude;
  }
  private void setLatitude(double lat){
     latitude = lat;
  }
     ...}
```

Developers might bind the type `latitude_geocentric` to the variable `latitude`. Return statement inference would generate bindings for return values of methods similar to `getlatitude`, and assignment inference would generate bindings for variable similar to `lat` in methods similar to `setLatitude`.

Method stage. In this stage, developers seed real-world type bindings to parameters of method declarations. For example, consider this method signature:

```
public double distanceTo
  (double lat1, double lon1,
  double lat2, double lon2)
```

Bindings can be seeded for variables `lat1`, `lat2`, `lon1` and `lon2`. After binding types to method parameters, parameter inference can be used to generate bindings for method invocations throughout the application.

Local variable stage. In this stage, developers seed type bindings for local variables. After seeding a small set of bindings, assignment inference and return statement inference can be used to generate more bindings. For example, consider this code snippet:

```
getLocalLatitude(){
  double lat1 = 0.0;
  double lat3 = lat1;
     ...
     return lat3}
```

If the developers bind the real-world type `latitude_geocentric` to variable `lat1` and then apply inference, the type `latitude_geocentric` will be bound to the variable `lat3` through assignment inference, and the type `latitude_geocentric` will be bound to the method return value of `getLocalLatitude` by return statement inference.

VII. Synthesis of Real-World Type Rules

The third phase in the synthesis of the real-world type system is to recover the type rules, i.e., to determine the legal operations involving variables of the various real-world types and the types of the results of those operations. The phase is based on the hypothesis that, for an existing program which has been developed and verified carefully, the implied use of real-world types in the program is largely correct. Thus, inferring rules from such a program is likely to be successful. Also, many general templates based on unary operators (such as negation) and binary operators (such as addition) need to be instantiated frequently.

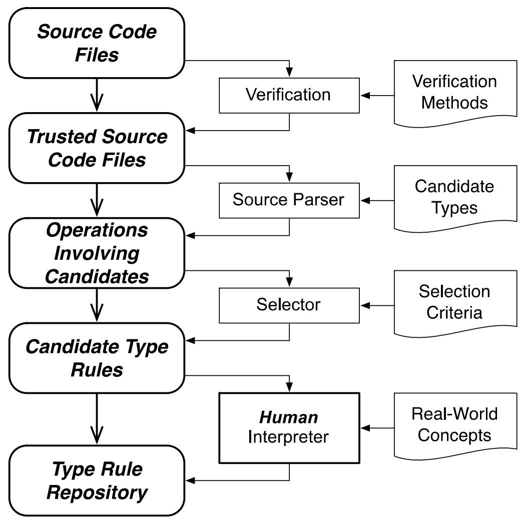

Fig. 6. Synthesizing real-world type rules

The mechanism in our prototype proceeds in three steps shown in Figure 6:

- **Verification**. The verification step checks the available source files and selects those for which there is reasonable assurance of adequate verification.
- **Selector**. Candidate type rules are formed by collecting details of all operations involving entities with real-world types including the operation, the real-world types used, and the frequencies of each particular combination.
- **Interpreter**. As in type synthesis, programmers and domain experts review the candidate type rules and select those considered valid.

VIII. CASE STUDY

In order to get insights into the utility and performance of real-world type synthesis, we applied the mechanisms to an open-source software project for which we have previously and separately developed a complete real-world type system with no automated support. The subject of study was a project called the *Kelpie Flight Planner* [13]. Having an existing complete example real-world type system, we were able to compare the results of the synthesis mechanism with the human-generated system. The authors have no association with the Kelpie Flight Planner project other than using it in this research.

The Kelpie Flight Planner is a moderate-sized, open-source Java project based on Flightgear, a flight simulator [6]. The program uses the airport and navaid databases of Flightgear to determine routes between airports based on user inputs. Results are presented using a sophisticated graphical interface. The program is 13,884 lines long, is organized as 10 packages, and is contained in 126 source files. The human-generated, real-world type system for the planner project contains 35 real-world types, 97 real-world type rules, and 255 real-world type bindings.

A. *Synthesis Of Type Candidates*

The total number of identifiers in the project is 28,754. Most classes in the project have less than 200 identifiers, and a few have more than 2,000. The identifier parser produces 45,585 terms in total, and so the average number of terms per identifier is approximately 1.59. 9,352 identifiers were parsed into two terms, 2,839 identifiers were parsed into three terms, and 582 identifiers were parsed into lists with more than three terms.

The first stage of the assembler searches for identifiers whose lists of terms contain a noun (major term). Of the 28,754 identifiers that were identified, 20,140 had a noun within their lists. These identifiers contained a total of 30,358 terms. The second stage of the assembler merges the lists of terms for identifiers that possess the same major term. 676 different terms were determined to be nouns and constitute the final set of major terms. Thus, the *draft* set of real-world type candidates has 676 entries. The third and final stage of the assembler coalesces the set of candidates using WordNet's lemma. After this stage, the *final* set of candidates had 528 entries.

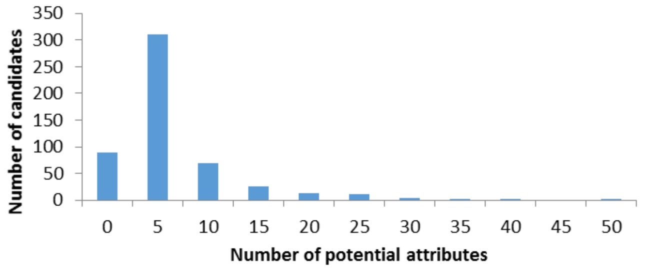

Fig. 7. Number of potential attributes for candidates

Every candidate in the set is a potential real-world type. The major term in the candidate usually leads to the primary meaning or type name. The associated terms that appear with a major term are considered as potential *real-world attributes* for the real-world type. Figure 7 shows how many potential real-world attributes each candidate in the final set could have.

Beginning with the set of candidates, the selector applies two selection criteria: (1) the frequency of major terms, and (2) the number of possible attributes a major term has.

For criterion 1, the selector sorts the list of candidates by the frequency of major terms, and then reduces the set by cutting all terms with frequencies below a selectable threshold. Table 1 shows the results of applying the first criterion with different thresholds. The first column is the threshold values, and the second column the number of terms having frequencies more than the threshold value. The third column is the average frequency of the terms with frequencies above the threshold, and the fourth column shows the average number of potential attributes the terms have.

TABLE I. SELECTION BASED ON FREQUENCY OF MAJOR TERMS

Threshold values	Number of terms	Average frequency	Average number of attributes
200	29	341	20.2
100	77	181	13.6
50	142	150	10.5
30	217	111	8.2
20	240	103	7.7

For criterion 2, the selector sorts the list of candidates by the number of attributes each term has and eliminates terms whose number of potential attributes are below a selectable a threshold.

Table 2 shows the results of applying the second criterion with different thresholds.

TABLE II. SELECTION BASED ON NUMBER OF POTENTIAL ATTRIBUTES

Threshold values	Number of terms	Average number of attributes	Average frequency (of major terms)
20	21	26.9	302
10	73	17.1	191
5	146	11.8	128
3	248	8.3	89
2	331	6.7	71

B. *Analysis*

In the synthesis mechanism, the interpreter follows the selector. In the interpreter, human judgment is used to choose actual real-world types of interest from the set of candidates. The expectation is that human insight will allow rapid selection and subsequent synthesis of appropriate syntactic structures.

In this case study, rather than operating the interpreter phase we compared the set of candidates with the actual real-world type definitions developed separately. The latter were treated as a "gold" set for purposes of evaluation.

Table III shows the assessment of the candidate set based on selector criterion 1 (frequency of occurrence of the major term) together with our assessment of the connections between the set of candidates and the real-world types that we identified.

TABLE III. CONNECTIONS BETWEEN CANDIDATES AND KNOWN REAL-WORLD TYPES BASED ON CRITERION 1

Threshold values	Number of terms	Important major terms	Real-world types that can be formed
200	29	2	11
100	77	7	18
50	142	15	21
30	217	21	25

For purposes of this analysis, we define a term to be *important* if, in our opinion, the term could lead directly to a real-world type definition. The first column and second column are repeated from Table 1. The third column shows how many important terms are in the subsets. The last column shows how many real-world types could be formed from the important major terms. The number in this column is typically larger than the number in the third column, because one major term might be used to construct more than one real-world type. For example, the term `lat` can be used to define real-world types `geocentric_latitude`, `geodetic_latitude` and others.

With a threshold value of 50 (frequency), we were able to form 21 real-world types. The total number of real-world types in the application is 35, and so we conclude that a large proportion of the actual real-world types could be formed from the candidates. With the lower threshold, application and domain experts will have to review more terms, but more real-world types could probably be formed.

Table IV shows the assessment of the candidate set based on selector criterion 2 (number of potential attributes of the major term) together with our assessment the connections between the set of candidates and the real-world types that we identified.

The first column and second column are repeated from Table 2. With a threshold value of 5, we could construct 20 real-world types and 23 when the threshold was set to 3.

TABLE IV. CONNECTIONS BETWEEN CANDIDATES AND KNOWN REAL-WORLD TYPES BASED ON CRITERION 2

Threshold values	Number of terms	Important major terms	Real-world types that can be formed
20	21	2	11
10	73	5	15
5	142	10	20
3	217	15	23

The results shown in Tables III and IV suggest that reviewing approximately 80 terms from the candidate set can reveal a substantial fraction the useful set of actual real-world types.

C. *Synthesis Of Real-World Type Bindings*

Table V describes data collected at each stage.

TABLE V. PERFORMANCE OF TYPE INFERENCE

Process stage	Bindings seeded by developers	Bindings synthesized by inference	Total number of bindings
Field	28	64	92
Method	64	65	221
Locate variable	30	4	255

In the table, the first column shows the stage of the bindings synthesis process. The second column shows the number of bindings we seeded acting as developers. The third column shows the number of type bindings automatically generated by inference. The fourth column shows the total number of bindings after inference at that stage.

In this study, we maximized the possibility of parameter inference, i.e., if the parameters were bound with appropriate real-world types, these types were propagated to all arguments. For assignment inference, we listed eight assignments that we trust as permitted assignment for inference.

In the field stage, we add 28 real-world type bindings to all suitable fields, and inference then generated 64 type bindings, mainly from return statement inference and assignment inference. In the method stage, we seeded 64 type bindings to suitable parameters in different method declarations, and then inference produced 65 bindings for arguments in various method invocations, primarily from parameter inference. In the local-variables stage, we seeded 30 type bindings to local variables, and 4 bindings were generated by assignment inference.

The project required 255 bindings in total, and 133 of these were generated automatically, i.e., 52%. This fraction suggests that the combination of some human effort and various forms of inference can yield reasonable performance in binding program elements to real-world types.

D. *Synthesis Of Real-World Type Rules*

We determined the number of real-world type rules that could be synthesized from every class. The necessary type rules were extracted from a small minority of the files. Most of the project files yielded less than 20 rules, but a few yielded more than 50 rules each.

In our hand-build, real-world type system for the Kelpie Flight Planner project, we created a total of 97 real-world type rules. The synthesizer mechanism extracted candidate rules from every class, and the total number extracted from individual files was 260. Many of these candidate rules are the same, and they can be reused in different methods of one class or methods across different classes.

IX. Related Work

Concept location is related to the mapping of real-world entities to software [1, 7, 8, 11, 17, 18] but with the goal of improving program understanding. The use of parts of speech of terms in identifiers has been investigated as a means to extract information from the source code [2, 14] and to extract domain models [1, 18]. Extensions to support dimensional and unit analysis have been developed for several programming languages [10, 12] but without the general notion of documenting logic interpretations.

Powerful extensions to the basic notion of type have been developed, in particular in the form of pluggable type systems [5, 9, 16]. Pluggable type systems are designed to provide greater flexibility in type mechanisms. However, the resulting flexibility remains within the mathematical framework of machine logic, and does not address the notion of deriving and exploiting type information from the real world.

Type providers in F# extend the language so that the compiler can generate both new types and new code that leverage these types based on the structure of real-world data services [19]. Type providers in Idris extend this idea to languages with dependent types [3]. Neither F# or Idris address the general issue of defining logic interpretations.

The Autocert system develops source-code constraints based on analysis of real-world information contained in Simulink models [4]. The checking that this enables is, in part, based on real-world information but does not enable the comprehensive checking that real-world types enable.

X. Conclusion

Preliminary results indicate that explicit interpretations based on real-world types provide a useful complement to available techniques for analysis of critical software systems. Despite the potential, the effort required to create the interpretation for an application might be a deterrent.

We have introduced a system for aiding developers in the creation of interpretations. In an initial case study, we have shown that significant candidate material can be synthesized from informal documentation if programmers follow common rules for forming that documentation.

References

[1] S. Abebe, and P. Tonella. “Towards the extraction of domain concepts from the identifiers,” In Proceedings of the 18th Working Conference on Reverse Engineering (WCRE), 2011, pp. 77–86.

[2] D. Binkley, M. Hearn, and D. Lawrie. “Improving identifier informativeness using part of speech information,” in Proceedings of the 8th Working Conference on Mining Software Repositories (MSR), ACM, New York, NY, USA, 2011, pp. 203–206

[3] D. Christiansen. “Dependent type providers”. Proceedings: 9th ACM workshop on generic programming, ACM, New York, 2013, pp.23-34.

[4] E. Denney and B. Fischer. “Annotation inference for the safety certification of automatically generated code”. Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE ’06), pages 265– 268, Tokyo, Japan, September 2006. IEEE.

[5] T. Ekman and G. Hedin. “Pluggable checking and inferencing of non-null types for Java.” J. Object Tech., 6(9):455--475, Oct. 2007.

[6] FlightGear. http://www.flightgear.org/

[7] G. Gay, S. Haiduc, A. Marcus, and T. Menzies. “On the use of relevance feedback in IR-based concept location,” in Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009). 2009, pp. 351-360.

[8] S. Grant, J. Cordy, and D. Skillicorn. “Automated concept location using independent component analysis,” In Proceedings of the 15th Working Conference onReverse Engineering (WCRE), IEEE Computer Society, Washington, DC, USA, 2008, pp.138–142.

[9] N. Haldiman, M. Denker, and O. Nierstrasz. “Practical, pluggable types,” in Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference (ICDL '07). ACM, New York, NY, USA, 2007, pp.183-204.

[10] S. Hangal, and M. Lam. “Automatic dimension inference and checking for object-oriented programs,” in Proceedings of the 31st International Conference on Software Engineering (ICSE '09). IEEE Computer Society, Washington, DC, USA, 2009, pp. 155-165.

[11] E. Hill, L. Pollock, and K. Vijay-Shanker. “Automatically capturing source code context of nl-queries for software maintenance and reuse,” In Proceedings of the 31st International Conference on Software Engineering (ICSE), 2009, pp. 232–242

[12] L. Jiang, and Z. Su. “Osprey: a practical type system for validating dimensional unit correctness of C programs,” in Proceedings of the 28th international conference on Software engineering (ICSE '06). ACM, New York, NY, USA, 2006, pp. 262-271.

[13] Kelpie Flight Planner for Flightgear. http://sourceforge.net/projects/fgflightplanner/

[14] Y. Li, H. Yang, and W. Chu. “Generating linkage between source code and evolvable domain knowledge for the ease of software evolution,” in Proceedings International Symposium on Principles of Software Evolution, 2000.

[15] Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999. National Aeronautics and Space Administration, Washington DC, November 10, 1999.

[16] A. Milanova and W. Huang. “Inference and checking of context-sensitive pluggable types,” in Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12). ACM, New York, NY, USA, 2012, Article 26。

[17] D. Poshyvanyk, A. Marcus. “Combining formal concept analysis with information retrieval for concept location in source code,” in Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC), 2007, pp.37–48.

[18] D. Ratiu, and F. Deissenboeck. “From Reality to Programs and (Not Quite) Back Again,” in Proceedings of the 15th IEEE International Conference on Program Comprehension(ICPC '07). IEEE Computer Society, Washington, DC, USA, 2007. pp. 91-102.

[19] D. Syme, K. Battocchi, K. Takeda, D. Malayeri, J. Fisher, J. Hu, T. Liu, B. McNamara, D. Quirk, M. Taveggia, W. Chae, U. Matsveyeu, and T. Petricek. “Strongly-typed language support for internet-scale information sources.” Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

[20] Wordnet. http://wordnet.princeton.edu

[21] J. Xiang, J. Knight, and K. Sullivan, “Real-world Types and Their Application”, SAFECOMP: International Conference on Computer Safety, Reliability and Security, Delft, The Netherlands, 2015.