Interpreted Formalism: Towards System Assurance and the Real-World Semantics of Software

A Dissertation Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy

Computer Science

by

Jian Xiang

August 2016

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Jian Xiang (Author)

This dissertation has been read and approved by the examining Committee:

John C. Knight (Advisor)

Jack W. Davidson (Chair)

Kevin Sullivan

Hongning Wang

Houston Wood (Minor Representative)

Accepted for the School of Engineering and Applied Science:

Craig H. Benson (Dean, School of Engineering and Applied Science)

August 2016

ABSTRACT

Software systems, especially cyber-physical systems, sense and influence real-world entities under the control of software logic in order to realize desired real-world behaviors. Such software systems are based upon three essential components: (1) a computing platform, (2) a set of physical entities with which the computing platform interacts, and (3) the relationship between the first two components. These three components seem familiar, and the third component seems trivial. In fact, the third component, the relationship, is crucial, because it defines how logical values read and produced by the computing platform will be affected by and will affect the various physical entities.

Formally, the relationship between real-world entities and a computer system's logic is the *interpretation* of the logic. Software logic is necessarily formal, but, in practice, interpretations are usually documented informally and incompletely, and programmers treat elements in software logic as if they were the real-world entities themselves. As a result, faults are introduced into systems due to unrecognized discrepancies, and executions end up violating constraints inherited from the real world. The results are software and system failures and adverse downstream consequences.

This dissertation argues that, to mitigate such risks, software engineers should produce not just traditional software, but a new engineering structure, the *interpreted formalism*. The structure combines software logic with an explicitly documented interpretation. Among other things, an interpretation documents differences that arise inevitably between real-world values and corresponding logic values. An interpreted formalism provides centralized documentation of a system’s software and its intended relationship to the real world in an analyzable form, facilitating fault detection.

An implementation of the interpretation, *real-world type*, is introduced. For a specific software system, an interpretation is composed of a set of real-world types, and an interpreted formalism is implemented as a real-world type system combined with a software system.

The pragmatics of the interpreted formalism concept are illustrated by conducting case studies on open-source software systems of different sizes. The interpreted formalism is evaluated from several viewpoints: (a) overall feasibility, (b) error detection capability, (c) effort level required, and (d) scalability. The results of the case studies suggest that (1) the interpreted formalism concept can be used on modern software systems of different sizes, (2) the technology is capable of detecting real errors that violate real-world constraints, and (3) the effort required from engineers for developing and using the interpreted formalism can be reduced greatly by an automated synthesis framework developed as part of this research.

ACKNOWLEDGEMENTS

I am very grateful to my advisor John Knight for his many contributions to the thesis and his patience and encouragement in my ability to complete the research. My life as a Ph.D. student was troubled by various kinds of problems. John never gave up on me and led me through the dark time. He brought me back into the right direction and saw me through to the end. I am greatly indebted to him.

My dissertation committee was extremely cooperative and insightful. Kevin Sullivan provided me with excellent insights, inspired me with a lot of brilliant ideas, and surprised me with his marvelous writing skills. Hongning sparked the synthesis idea which I personally like most in the thesis.

I appreciate people from the department for their help and understanding. Kevin Skadron warmed me up with his concerns. Wes Weimer guided me through the requests from the engineering school. And Jan Cornell helped me with all the administrative and financial issues.

Last and most importantly, I express my love and appreciation for my family. Dad pushed me to accomplish the research; mom supported me with her love and tolerance. My wife, Xiaoxi, totally changed my life and made me a happier, healthier, and more responsible man.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

1. INTRODUCTION

Software systems, especially cyber-physical systems, sense and change real-world entities under the control of software to realize desired real-world behavior. Such software systems are based upon three essential components: (1) a computing platform, (2) a set of physical entities with which the computing platform interacts, and (3) the relationship between the first two components. These components seem familiar, and the third component seems trivial. In fact, the third component, the relationship, is crucial, because it defines how logical values read and produced by the computing platform will be affected by and will affect the various physical entities.

Such relationships, however, are usually defined in an ad hoc manner lacking in rigor, yet a misunderstanding of the relationship could lead to serious consequences [6, 53]. Specifically, the relationships between elements in software logic and physical entities in the real world are under-specified, and programs treat elements in logic as if they were the real-world entities themselves. As a result, faults are introduced into systems due to unrecognized discrepancies, and executions end up violating constraints inherited from the real world. The results are software and system failures and adverse downstream consequences.

Formally, the relationship between real-world entities and a computer system's logic is the *interpretation* of the logic. This dissertation examines some of the challenges inherent in defining and leveraging the interpretation of software. Specifically, it describes and evaluates the mechanisms by which interpretations can be explicitly defined and utilized. Instead of developing traditional software, this work advocates developing a new explicit structure, the *interpreted formalism* that combines rigorous methods and notations for defining interpretations for software. An interpretation will include information such as details of differences between real-world values and values in logic that arise inevitably in sensor systems. The interpreted formalism provides centralized documentation of a system's software and its relationship to the real world in an analyzable form thereby facilitating fault detection.

The remainder of this chapter establishes the context of the work and previews the idea of the interpreted formalism and contributions.

1.1 PROBLEM OVERVIEW: UNINTERPRETED LOGIC

The need to define the relationships between elements in the software and entities in the real world explicitly arises because software is a logic function with no interpretation. The correctness of uninterpreted logic is not comprehensively checked. This section introduces the problems of uninterpreted logic that motivate this work.

1.1.1 UNINTERPRETED LOGIC

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, i.e., the logic is *uninterpreted*. 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.

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. 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 lost.

As an example, Fig. 1(a) shows a block of Java text taken from an open-source library. The meanings of the values used by the parameters of the various functions are documented in part by the names of the parameters and in part by the comments. The block of Java text with the identifiers replaced with random strings and the comments removed is shown in Fig. 1(b). This version of the Java text compiles correctly, and execution of the software is unaffected by the changes. Although the logic function is unaffected, human understanding of what the logic does is almost destroyed. In the function originally named `getmonth`(), the comment explaining that the encoding of the months of the year used by the function is 0 – January, 1 – February, 2 – March, etc. is essential. That particular encoding is unusual and impossible to discern from the code.

```
/* Returns the year after 1900. */
 public int getYear() {
 if (!expanded)
     expand();
 return tm_year;}

/* Sets the year. */
 public void setYear(int v) {
 tm_year = v;
 valueValid = false;}

/* Returns the month. ie: 0-11 */
 public int getMonth() {
 if (!expanded)
     expand();
 return tm_mon;}
```

(a)

```
public int lnmgtyu() {
 if (!wpou88kj)
     xcvbbhu71();
 return tm_lnmgtyu;}

 public void tyugfds(int v) {
 tm_ugfdsrew = v;
 ascboi9jjk  = false;}

 public int sdgtyu() {
 if (!wpou88kj)
     xcvbbhu71();
 return tm_sdgtyu;}
```

(b)

Fig. 1. Source code with and without an ad hoc interpretation

1.1.2 CONSEQUENCES OF UNINTERPRETED LOGIC

For many software systems, especially safety-critical systems, the assurance of their correct operation depends on the interactions between real-world entities and software being complete and correct. Unless the interactions are precisely that which is intended and known to be so, doubt in the correctness of the effects of such systems is inevitable. The logic in a software system should respect invariants that derive from the real world that the system senses, models, and affects. For example, software that computes physical quantities should respect measurement units and physical dimensions, such as those defined by the ISO/IEC 80000 standards.

The failure of software to observe real-world invariants has been a causal factor in various accidents and incidents. In 1999, the Mars Climate Orbiter was lost because different parts of the software system used different units of measurement [53]. One piece of the ground software produced outputs using imperial units (pound-seconds), while a second system expected results to be in metric units (newton-seconds). More recently, in 2013 a delay in berthing Orbital's Cygnus spacecraft with the International Space Station (ISS) occurred because Cygnus and the ISS used different forms of GPS time data; one based on the original 1980 ephemeris, and the other based on an ephemeris designed in 1999 [6]. The two forms have a difference of exactly 1024 weeks. In both of these examples, real-world entities were affected by operations defined in software in ways that made no real-world sense, i.e., the systems failed to observe real-world invariants because the essential interaction was improperly defined.

Logic with no interpretation or casual interpretation is unsatisfactory to assure such correctness. Interpretations described by casual techniques are unstructured, incomplete, and unorganized. Such interpretations are usually incomplete or insufficient because:

- Important characteristics of real-world entities are undocumented or are documented incompletely, informally, and implicitly.
- The approximations inherent in finite-precision representations in the logic and in the sensing of values from the real world remain under-specified.
- Values used in the logic are provided by noisy, mis-calibrated sensors.
- Sensor and related failures could provide erroneous values for the logic.

As a result, uncertainties and assumptions are introduced by the software logic, and crucial, non-trivial relationships between software elements and real-world entities remain under-specified. Even more frequently, the completeness and correctness of the interaction are not checked comprehensively. Invariants derived from the real world are stated and enforced either in ad-hoc ways or not at all.

1.1.3 IMPLICATIONS FOR RESEARCH

Accidents and incidents such as cited those above suggest that substantial reductions in safety-critical defects might be achieved if developers could systematically enforce real-world invariants in software logic. Thus, the implication is that if we could thoroughly understand and trace real-world entities in software logic, we could have the opportunity to improve software reliability by reducing the potential violations of real-world invariants.

To be able to understand and trace real-world entities, elements in software logic need to be properly interpreted. An explicit and comprehensive interpretation should fulfill several requirements:

- The precise semantic information of the real world including all relevant characteristics and invariants are clear.
- The constraints and invariants of real-world entities are explicit and can be used to check the software for violations.
- The differences between real-world entities and their machine-world realizations are documented and analyzable.
- Specific analysis techniques derived from the real-world context can be developed and employed.

1.2 THE TARGET CHALLENGE: AN EXPLICIT INTERPRETATION

Various challenges arise when designing an interpretation. To meet the requirements listed above, in this section, these challenges are discussed.

1.2.1 REAL-WORLD SEMANTIC INFORMATION

Real-world semantic information is vital to software systems. The Mars Climate Orbiter was lost due to a misuse of units of measurement, and berthing Cygnus with ISS was delayed due to inconsistent forms of GPS time data, which were based on different ephemerides. The semantic attributes of units of measurement and ephemeris were the causal factors in the accident and incident.

Developers frequently treat the real-world semantic attributes of entities that computers manipulate informally. Software systems typically represent real-world entities as variables, values, or instances. Such program elements themselves have no inherent real-world meaning, so the real-world meanings must be added. In many cases, programmers are admonished to use “meaningful” identifiers and to comment code liberally. However, relying on identifiers and comments is an unsatisfactory approach. Identifiers are limited by their formats and styles; comments are unstructured and disorganized. There is no existing structured approach to determine what comments should be added, or whether comments are sufficient. Important real-world semantic information could be ignored or misunderstood.

In order to deal with this challenge, the goal of the work is to describe the real-world context, e.g., the semantics of all real-world entities explicitly. An interpretation should be able to convey real-world semantic information from one individual to another unambiguously. An interpretation needs to provide complete and comprehensive real-world semantic information in a consistent and accurate manner.

1.2.2 Relationships between Real-World Entities and Logic

The relationships between elements in software and real-world entities are complex but important. Such a relationship has two parts: (a) mapping links between real-world entities and program elements (e.g., variables, values), and (b) possible differences introduced by the hardware.

A software system relies upon sensors to acquire information about the real world within which the system operates. Of necessity, the sensed values are approximations of real-world values. However, the fact that sensed values are approximate is often neglected. Variable names in programs are often the names of sensed values, and calculations are performed assuming the variable *is* the real-world entity. In effect, the machine representations of real-world entities provided by computer systems are intertwined ambiguously with concepts in reality; such situations miss the fact that what is available to the software is not always what is present in the real world.

In order to deal with this challenge, the interpretation should (1) clearly separate real-world entities from their machine representations in software logic, and (2) explicitly document the differences. Real-world entities are what software programs intend to manipulate. Machine representations are virtual versions of real-world entities provided by hardware. Separation of the two reminds the programmers of the differences and urges them to pay attentions to the differences.

1.2.3 REAL-WORLD CONSTRAINTS

Real-world constraints and invariants are those inherited from real-world properties (e.g., the laws of physics), and these constraints and invariants should be observed in programs. Current research efforts focus on specific kinds of real-world constraints, yet lack systematic approaches to check constraints derive from various sources.

A common type of real-world constraints is unit consistency. Research efforts have been made to check unit inconsistency. For example, assigning a variable measured in feet to a variable measured in meters is erroneous. However, these efforts are limited to basic rules derived from dimensions or combinations of entities with different units.

Existing approaches for checking unit consistency cannot be readily extended to checking other real-world constraints. They often refine the built-in type systems with type qualifiers or other similar annotations that denote units of measurement. These approaches can be applied to check other machine-world context properties, such as nullness and interning. However, they are not suitable to express different kinds of real-world constraint, nor capable of conveying real-world semantic information.

In addition to unit consistency, other kinds of real-world constraints that should be observed. For example, the following constraints should be obeyed in a geographic software system:

- Adding latitude to longitude should probably not be allowed.
- Adding geodetic latitude to geocentric latitude should probably not be allowed.
- A point in one coordinate system should probably not be used in an expression with a point in another coordinate system.

In order to deal with this challenge, the interpretation should permit an approach that systematically allows the definition of properties derived from these real-world constraints and invariants. Properties derived from real-world constraints and invariants probably come from various domains or sources. Automated checking of these properties requires a systematic approach that documents and enforces different kinds of properties.

1.3 SOLUTION AND PREVIEW OF CONTRIBUTIONS

As noted above, for software systems that are safety critical, assurance of their correct operation depends on the interpretation of the logic being complete and correct. Dealing with this dependence begins with the interpretation being documented precisely and comprehensively. Without such documentation, doubt in the correctness of the effects of such systems will arise inevitably.

The strategy employed by the solution presented in this work is to document the interpretation of software explicitly and comprehensively, and leverage the interpretation pragmatically for software fault detection.

The explicit and systematic documentation of the interpretation of software logic provides three major advantages:

- The interpretation informs the design of the software of the actual entities that the software will affect, and thus allows better design choices.
- The interpretation documents essential reference material in a centralized and well-defined form allowing rigorous examination for correctness and completeness by human inspection.
- The real-world constraints and invariants that the interpretation exposes can be checked providing a new mechanism for detecting software faults.

All three of these advantages are valuable, but the provision of a significant new capability for detecting software faults is of particular importance. Static analysis of a system’s software where the analysis derives from the interpretation allows the detection of faults that result from misuse of real-world entities or violate real-world constraints.

The necessity of explicitly defining an interpretation for software indicates the need for a new artifact that includes both software logic and its interpretation. To satisfy this demand, this dissertation advocates a new structure, the *interpreted formalism*, to incorporate an explicit interpretation into the engineering artifacts that are needed for software systems.

1.3.1 GOALS AND APPROACH OF INTERPRETED FORMALISM

The main goal of the interpreted formalism is to improve software quality via reasoning and automated checking of real-world constraints in programs. This goal can be divided into three sub-goals:

- Explicitly specify the real-world context. The real-world context provides a broad basis for defining properties in analyses. Any limitations and constraints in the real-world context can be used in analyses of the software. This specification should also describe the characteristics of real-world entities of interests.
- Clearly separate real-world entities from their machine representations in software logic. Real-world entities are the entities that software programs intend to manipulate. Machine representations are virtual versions of real-world entities in the software logic. Separation of the two reminds the engineers of the differences and urges them to think from the perspective of the real-world context.

- Systematically establish properties derived from the real-world context in software programs. Properties derived from the real-world context probably come from various domains. Automated checking of these properties requires a systematic approach that documents and enforces different kinds of properties.

In order to achieve these goals, an interpreted formalism needs to accomplish the following tasks:

1. For the first goal, real-world entities and the relationships between these entities must be well documented. For each entity, its real-world semantic information should be completely documented so that the details of the entity can be accurately conveyed from one person to another. The constraints and invariants between real-world entities should be clearly documented; they are constraints that need to be observed in software programs by analyses.
2. For the second goal, the connections and differences between real-world entities and their machine representations should be explicitly documented. Such explicit documentation demonstrates the differences between real-world entities and their machine representations; it also enables possible techniques that analyze the discrepancy caused by different pieces of the logic.
3. For the third goal, an interpreted formalism should be able to link real-world entities with their corresponding program elements, such as variables, instances, and functions. With such links, programmers are able to locate all program elements in the process of establishing properties. In addition to the links, documentations for

the first and second goals should be machine readable, so that automated analyses can be employed.

1.3.2 INTERPRETED FORMALISM

In order to accomplish the tasks stated above, an interpreted formalism is structured with three components: (a) software logic, (b) specifications of relevant real-world entities that affected by the software logic, and (c) an interpretation that specifies the relationship between software entities and real-world entities.

The structure of an interpreted formalism is shown in Fig. 2. The real-world specification is a machine-readable specification that characterizes the real-world entities affecting and being affected by the system of interest. The interpretation is added explicitly to document the relationships between logic and real-world entities.

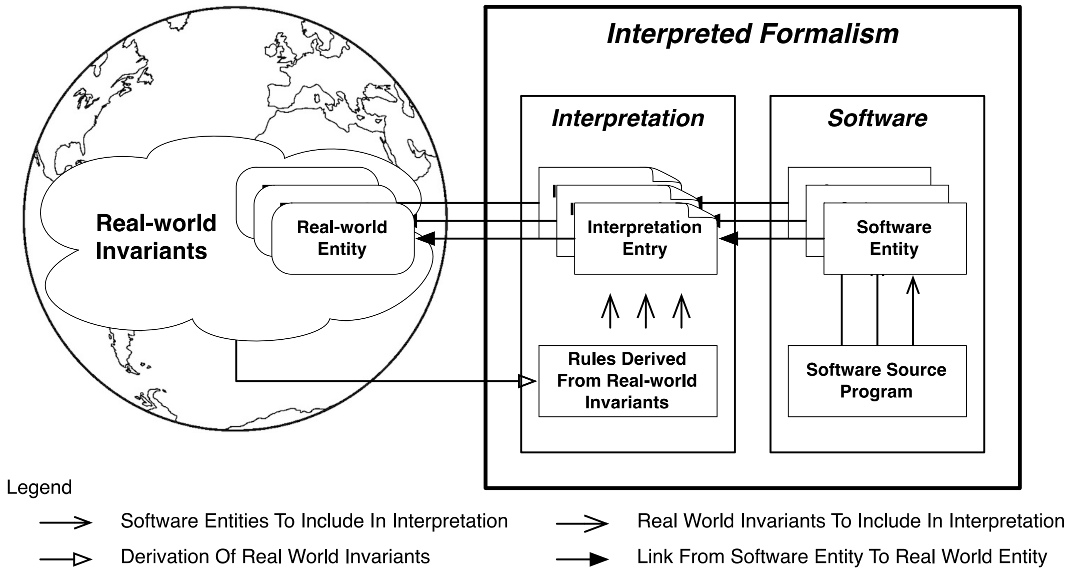

Interpreted Formalism
Interpretation
Software
Real-world Invariants
Real-world Entity
Interpretation Entry
Software Entity
Rules Derived From Real-world Invariants
Software Source Program
Legend
Software Entities To Include In Interpretation
Real World Invariants To Include In Interpretation
Derivation Of Real World Invariants
Link From Software Entity To Real World Entity

Fig. 2. Overview of interpreted formalism

In the development of a particular computer system of interest, the task is no longer to develop software. The task is, in fact, to develop an interpreted formalism. Without the explicit interpretation, whatever would be developed as "software" runs the risk of failing to define the desired interaction with the real world correctly, where the implementation of that interaction is the entire purpose of the software system.

The switch from developing software to developing an interpreted formalism is a *paradigm shift*. Such a change should not be undertaken lightly, but with the number, criticality, and consequences of the failure of safety-critical system increasing, the shift needs to be considered seriously.

1.3.3 THESIS STATEMENT

As originally observed by Jackson [40], the development of computer systems should begin with the definition of the problem to be solved as framed in the real world. The solution has to be implemented in the machine world. This idea was expanded by Gunter et.al in the reference model [31]. This dissertation research follows this paradigm. An initial practical mechanism to effect this was the four-variable model introduced by Parnas and Madey [65]. Although not stated as such, the four-variable model is a very preliminary form of an interpretation. The concept of interpretation links the real world to the machine world.

The thesis statement for this work is:

> *Explicit interpretation of software logic in the form of real-world type systems can practically promote greater confidence in software systems that manipulate real-world entities, by enabling automated checking of real-world constraints and invariants.*

1.3.4 PREVIEW OF CONTRIBUTIONS

The idea of situated formalism makes four main contributions:

- It introduces a new paradigm for software development. This dissertation advocates that the interpreted formalism rather than isolated software is the right artifact for the development of safety-critical systems.
- It introduces a preliminary explicit content and structure for defining an interpretation.
- It provides a framework for systematic detection of software faults that violate real-world invariants.
- It introduces several mechanisms that facilitate developing interpreted formalisms, which has the potential to be reused in other projects.

1.4 ORGANIZATION OF THE WORK

The remainder of the document is organized as follows:

Chapter 2 elaborates on the idea of logic interpretation. It explains the role and necessity of logic interpretation in details.

Chapter 3 introduces the concept of interpreted formalism. A new form of software systems based on an interpreted formalism is described. The definition of the interpreted formalism is introduced, and an example is presented.

Chapter 4 introduces real-world type. It is an implementation of the interpreted formalism concept. The concept and details of real-world types are introduced, and an example is given.

Chapter 5 describes the techniques by which the interpreted formalism is leveraged to established properties in software logic. Several analysis techniques based on interpreted formalisms are introduced. The analysis techniques include both compile-time and runtime assurance.

Chapter 6 addresses the processes and tool support for developing interpreted formalisms. Especially, it introduces a synthesis framework that extracts candidate interpreted formalisms from the source code of the targeted application.

Chapter 7 introduces the prototype implementation based on real-world types and the interpreted formalism. The prototype is implemented with Java language.

Chapter 8 describes the overview of evaluation undertaken to assess the feasibility, efficiency, and other properties of the interpreted formalism.

Chapter 9 and Chapter 10 present details of the two case studies conducted on open-source software for the purposes of evaluation.

Chapter 11 illustrates the practical application of the interpreted formalism and real-world types by a case study.

Chapter 12 illustrates how to use synthesis framework of the interpreted formalism by a case study on an open-source project.

Chapter 13 reviews the related literature concerning every aspect of the interpreted formalism.

Chapter 14 summarizes the idea, findings, and contributions of the work.

CHAPTER 2

2. LOGIC INTERPRETATION

This chapter describes the concept of interpretation, various forms of insufficient interpretation, and advantages of an explicit and comprehensive interpretation.

2.1 THE CONCEPT OF LOGIC INTERPRETATION

The role of interpretation can be thought of as an enhanced version of an abstraction function in logic that maps concrete representations such as variables to abstract representations such as abstract data types. A stack, for example, is an abstract data type that has a concrete implementation as an array and an integer index into the array.

In a similar way, an interpretation maps an element of logic to its real-world meaning. 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.

Fig. 3 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 abstract (real world) details of a concrete entity (logic).

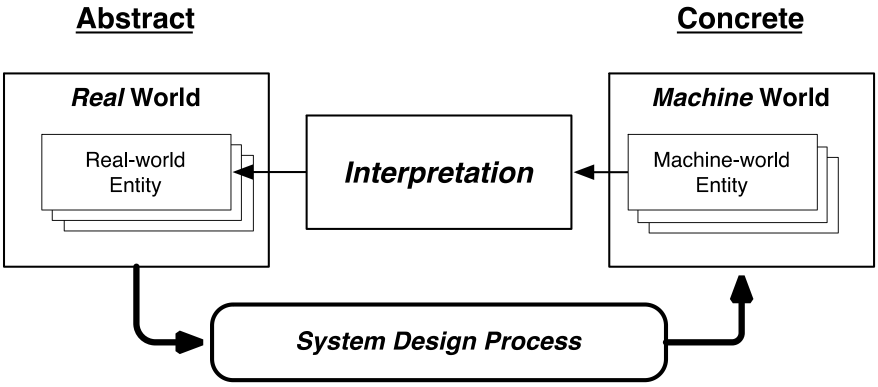


Fig. 3. Interpretation as an abstraction from concrete to abstract

2.2 IMPLICIT INTERPRETATION

Interpretation of software is not a new idea. As stated in the first chapter, the interpretation of a software system is always present in practice, but usually documented in an ad hoc, informal and sometimes implicit manner. The figures below show examples of implicit interpretation that are documented using casual techniques.

```
final double a =
    alt - to.getElevation();
final double t =
    a / plan.getAircraft()
        .getSinkSpeed() / 60;
```

(a)

```
final double altToBOD  =
    altitude - to.getElevation();
final double timeToBOD =
    altToBOD / plan.getAircraft()
        .getSinkSpeed() / 60;
```

(b)

```
/* Calc the time to get to Bottom Of Decrease (BOD).
altitude is the altitude of the BOD;
getSinkSpeed() returns the horizontal speed of the plane;
60 is used to convert the rate */
final double altToBOD  = altitude - to.getElevation();
final double timeToBOD = altToBOD /
      plan.getAircraft().getSinkSpeed() / 60;
```

(c)

Fig. 4. Examples of insufficient interpretation

All three examples serve the same function: calculating the time needed for the aircraft to decline to the point of BOD (Bottom Of Decrease).

Each of them documents its interpretation differently:

- In Fig. 4 (a), only pieces of the real-world meanings are implied. The identifiers of the method invocations, e.g., getElevation(), imply possible meanings of the statements. However, a great deal of information about the real-world meanings is incomplete. For example, the real-world entities represented by variables a and t are not clear.

- In Fig. 4 (b), the interpretation has been improved a little comparing with (a), since the identifiers are named in a more “meaningful” way. The identifier altToBOD implies that the variable is representing the altitude of the point of BOD; the identifier time implies that the variable is representing time. However, such information is still partial and leaves relevant information in doubt, for example, what does BOD mean? What are the units of measurement for altitude and time?

- In Fig. 4 (c), the interpretation is presented with descriptive comments. More details about the variables and statements are documented. The comments can be further expanded to include more information such as units of measurement of time and reference level of the altitude. However, interpretation is still incomplete, since the interpretation should document the details of the connection between variables and their corresponding real-world entities. For example, the interpretation should explain the differences between the value of variable *altToBOD*, and the real value of the altitude. Such differences are caused by the hardware system and are crucial when understanding the effect on the real world made by the program statements.

All of the forms of interpretation above document or imply some details about the real-world entities being affected, but such details are prone to three kinds of problems:

- *Real-world meanings are incomplete*. Casual techniques lack a clear structure to document and convey real-world meanings. Characteristics of the affected real-world entities are frequently unspecified or under-specified. For example, what are the units for the distance?

- *The connections between elements in logic and their real-world entities are under-specified*. Such connections are not simply one-to-one correspondences. Details about the connections, such as the differences, should be documented as well.

- *Real-world constraints are possibly violated*. Program statements such as those shown in Fig. 4 introduce faults into the system due to the misunderstanding about the effect of real-world entities. The function *getSinkSpeed()*, for example, returns the horizontal speed of the aircraft, while the speed in the vertical direction is needed for a correct calculation.

2.3 EXPLICIT INTERPRETATION

In order to deal with the issues that arise with implicit interpretations. This dissertation advocates building *explicit interpretations* for software logic.

An explicit interpretation documents the real-world meanings of logic elements in a precise, clear and explicit manner. With an explicit interpretation, important characteristics of real-world entities, such as units and dimensions, and associated real-world constraints, such as not mixing units, can be stated and enforced. In addition, crucial relationships between logic representations and real-world entities, such as accuracy of sensed values, can be fully specified.

As an example, consider the altitude of an aircraft and the representation of altitude in avionics software. Aircraft altitude is not just a number even though it might be represented as such in software. Altitude has many important attributes that impact the way that a software system, such as an aircraft’s autopilot, computes using altitude, etc. A partial list of those attributes is:

- *Measurement units*. The altitude value will be measured in prescribed units (feet, meters, etc.).

- *Physical dimensions*. Altitude has the fundamental physical dimension of length.
- *Frame of reference*. Altitude is defined based on an origin and the direction, i.e., a frame of reference.
- *Reference level*. Altitude is measured vertically between a point or object and a reference level. The reference level could be mean sea level or local ground terrain.
- *Sensor performance*. A sensor, i.e., a transducer, will have determined the value to be supplied to the software and so that value will be of limited precision and accuracy because sensors are imperfect transducers.
- *Sensing schedule*. Transducers supply values according to a discrete-time schedule. The value supplied to the software is the value obtained when the sensor sample was taken, not the “current” value, i.e., the altitude of the aircraft “now”.

With an explicit interpretation that documents details of a quantity such as altitude, a wide variety of checks of the software in a software system such as an autopilot are possible, such as:

- *Mixed measurement units*. Expressions which mix units of measurement are probably erroneous unless an appropriate conversion is provided. For example, adding an altitude measured in feet to a displacement measured in meters is probably an error if no conversion factor is included in the computation.
- *Mixed physical dimensions*. Dimensional analysis is a standard error detection mechanism in physics. Thus, for example, assigning the result of dividing altitude

with physical dimension length by a time to a variable that is not a speed (speed has the dimensions of length/time) is probably an error.

- *Mixed frames of reference*. Altitude is measured in a frame of reference with an origin and an orientation. A distance calculation between two points is probably erroneous if the two points are from different frames of reference.
- *Mixed reference levels*. Altitude is a vertical measurement between a point and a reference level. A calculation between two points is probably erroneous if the two points are measured to different reference levels.
- *Inaccuracy caused by sensors.* Altitude is measured by sensors and other hardware devices. Sensed values are finite, imperfect and inaccurate. These inaccurate values could have been involved in different calculations.

2.4 ADVANTAGES OF AN EXPLICIT INTERPRETATION

The explicit and systematic documentation of the interpretation of software logic provides three major advantages:

- The interpretation informs the design of the software of the actual entities that the software will affect, and thus allows better design choices.
- The interpretation documents essential reference material in a centralized and well-defined form allowing rigorous examination for correctness and completeness by human inspection.

- The real-world constraints and invariants that the interpretation exposes can be checked providing a new mechanism for detecting software faults.

All three of these advantages are valuable, but the provision of a significant new capability for detecting software faults is especially important. Static analysis of a system’s software where the analysis derives from the interpretation allows the detection of faults that result from misuse of real-world entities or violate real-world constraints. In the case studies, such analyses revealed both unrecognized faults and faults that had been reported as bugs in real systems after deployment [83]. Details about this new fault detection capability are given in Chapter 5.

CHAPTER 3

3. INTERPRETED FORMALISM

Having established the role and value of an interpretation of the logic in a software system, we turn to the structure needed to incorporate an explicit interpretation into the engineering artifacts that are needed for software systems. The structure introduced is called an *interpreted formalism*.

This chapter introduces the concept and structure of the interpreted formalism. The next chapter will describe a pragmatic implementation of interpreted formalisms.

3.1 OVERVIEW OF INTERPRETED FORMALISM

The basic structure of the interpreted formalism concept is shown in Fig. 5.

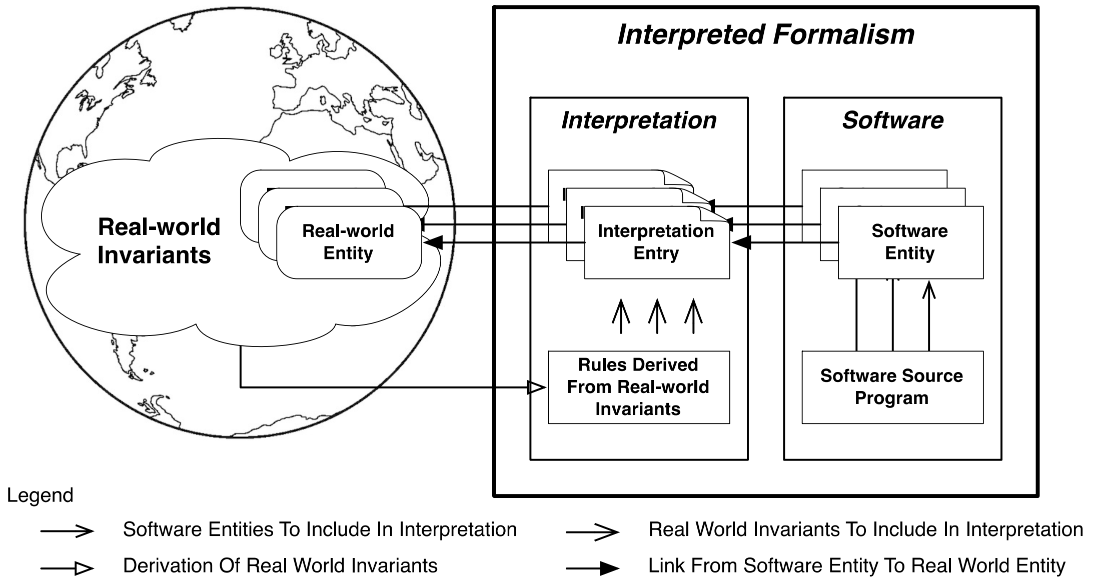

Interpreted Formalism
Interpretation
Software
Real-world Invariants
Real-world Entity
Interpretation Entry
Software Entity
Rules Derived From Real-world Invariants
Software Source Program
Legend
Software Entities To Include In Interpretation
Real World Invariants To Include In Interpretation
Derivation Of Real World Invariants
Link From Software Entity To Real World Entity

Fig. 5. An interpreted formalism combines logic with an explicit interpretation

The logic in an interpreted formalism is defined in whatever manner is appropriate for the system of interest, i.e., the choice of programming language, programming standards, compiler, and so on, are unaffected by the interpreted formalism structure. The key difference, of course, is the addition of the explicit interpretation.

As discussed in Chapter 1, an interpretation is always present for software systems that interact with the real world. The interpreted formalism combines an interpretation and the software in a manner that makes the interpretation a first-class entity.

In the development of a particular software system, the task is no longer to develop software. The task is, in fact, to develop an interpreted formalism for the system of interest. Without the explicit interpretation, whatever would be developed as “software” runs the risk of failing to define the desired interaction with the real world correctly, where the implementation of that interaction is the entire purpose of the software system.

As noted in Chapter 1, the switch from developing software to developing an interpreted formalism is a *paradigm shift*. The shift needs to be considered seriously due to the importance of software systems, especially safety-critical ones.

3.2 A NEW DEFINITION OF A SOFTWARE SYSTEM

This section introduces a general form of software systems and a novel form based on the interpreted formalism and interpretation.

3.2.1 A GENERAL FORM OF SOFTWARE SYSTEMS

In practice, software systems have the general form shown in Fig. 6. An important aspect of a software system is the distinction between continuous functions and discrete functions. Simple two-state (“on” and “off”) discrete functions arise in the real world with switches, but many signals in the real world that are intended to change the state of the logic function are continuous. Signals to which the logic function might react and the signals that it generates are discrete. Transducers process continuous signals in the real world to produce discrete signals that are input to the logic function. By making various state transitions, the logic function effects a “computation” that produces digital signals that are designed to affect real-world entities. Transducers process these digital signals to produce continuous signals that are output to the real world.

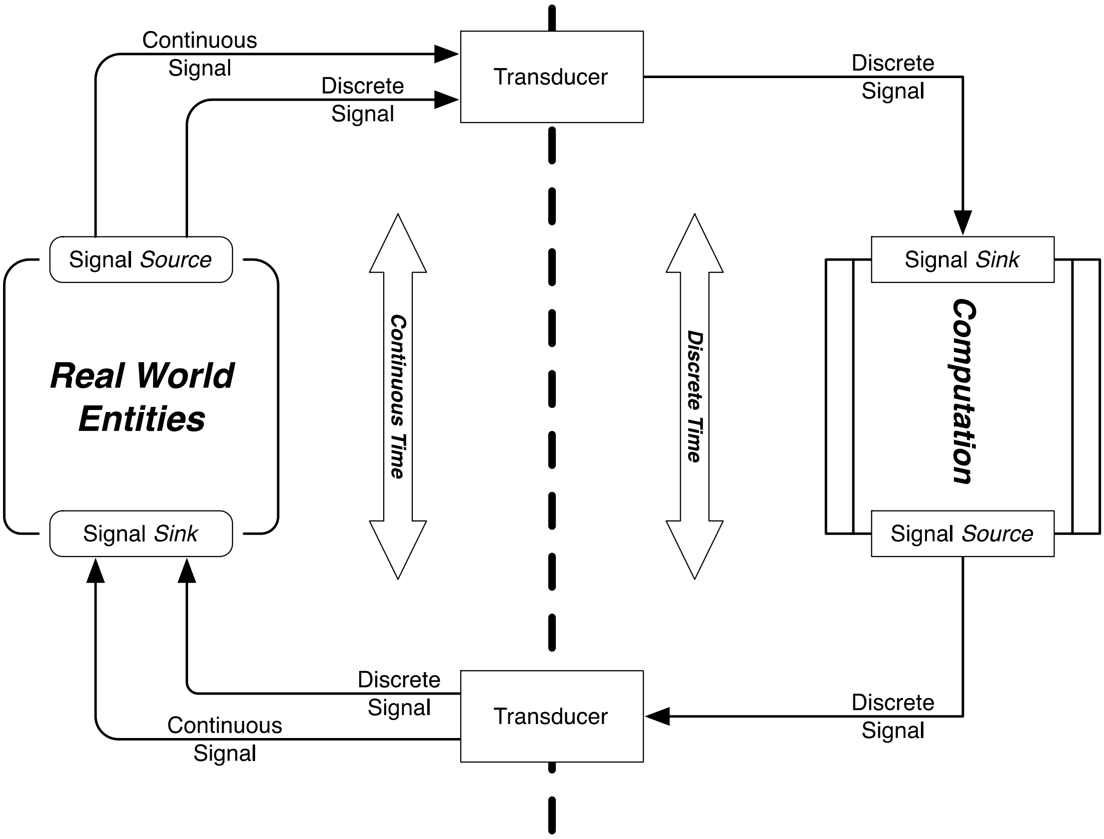


Fig. 6. The basic elements of a software system

An important element of the distinction between continuous and discrete functions is *time*. Time is continuous for both the real world entities and logic, but logic models the progress of time as a discrete function; and the logic changes state only at regular discrete-time intervals.

3.2.2 A New Form of Software Systems

With the concept, role, and importance of the interpretation defined and with the interpreted formalism structure in place, we have the elements needed to define a software system in a new and more comprehensive way.

A software system that interacts with the real world, e.g., cyber-physical systems and embedded systems, is a triple {RW, HW, IF} where:

- RW: A set of entities in the real world with which the system interacts.
- HW: A hardware platform that: (a) executes the system's software, and (b) provides the physical connections between the set of real-world entities and the system's software.
- IF: An interpreted formalism that is composed of software logic and an interpretation. The interpreted formalism links the two parts above.

For the most part, RW and HW are familiar. The set RW can be enumerated based on the problem to be solved by the software system. For example, in the case of a drug-infusion pump, the pump has to interact with (1) the patient's physiology, (2) the patient and medical professionals as "users" through a human-machine interface, and (3) the environment including gases, radiation fields and physical entities that might cause damage.

HW for a drug-infusion pump is a collection of specialized hardware items including a pump, a drug reservoir, multiple valves, switches, a keyboard, one or more computers, a network interface, and so on.

IF is the new concept introduced in this dissertation. It defines the logic of the computer system and links elements in computations to real-world entities.

Fig. 7 shows this new form of software systems. It includes an explicit interpretation and an explicit indication that the design of the computation derives fundamentally from the real-world context within which the system will operate.

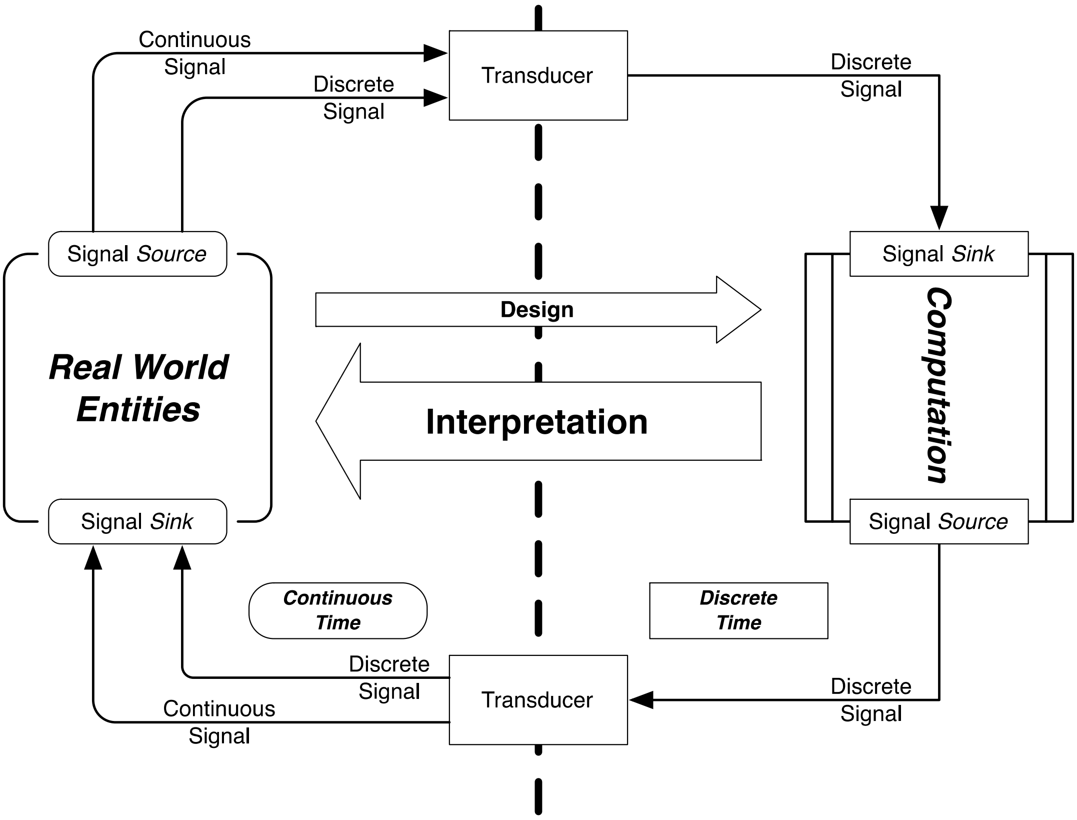


Fig. 7. The basic elements of a software system with an interpretation

3.3 THE STRUCTURE OF INTERPRETED FORMALISM

This section introduces the structure of an interpreted formalism, i.e., the software logic of the system coupled with the interpretation of that software.

As stated above, the logic is developed unaffected by the interpreted formalism structure. Therefore, the focus of this section is on the structure of interpretations.

An interpretation consists of two components: (1) *a real-world specification* that explicates all real-world entities accessed in elements of logic, and (2) *a relationship specification* that explains the relationships between entities in logic and their corresponding real-world entities. These two components are described in the following subsections.

3.3.1 A REAL-WORLD SPECIFICATION

Defining the content and structure of an effective and complete specification of the real world for *practical* use is a significant challenge. The real-world specification in an interpreted formalism has two goals: (a) convey the meanings of real-world entities by revealing important characteristics of the entities, and (b) expose real-world constraints and invariants so analysis techniques can be developed to check these constraints in logic.

To satisfy the two goals, the real-world specification is designed as a composition of (a) a set of specifications for real-world entities, and (b) a set of real-world constraints and invariants. Such design is based on previous work on real-world types [83]. The details are explored in Chapter 4.

SPECIFICATIONS OF REAL-WORLD ENTITIES

The first part of a real-world specification is a set of specifications for real-world entities. Each of these specifications explicates defines one type of real-world entity. For example, in a geographic software system, one specification could be defined for latitude, and another specification could be defined for longitude.

A specification for a real-world entity contains two components:

- **The real-world semantic attributes associated with the entity**. These attributes are properties possessed by the real-world entity. For example, for the real-world altitude of an aircraft, the semantic attributes include measurement of units, physical dimensions, the frame of reference, and so on.

- **An explication**. An explication is a careful, detailed, and precise statement in natural language that provides the reader with a detailed explanation of the various terms and concepts in use. For example, the explication for the real-world altitude is a vertical distance measurement between a reference level and a point or object. The reference level varies according to the context. Altitude is often used to mean the height above sea level of a point.

Real-World Constraints and Invariants

The second part of the real-world specification documents the real-world constraints and invariants. These constraints and invariants define allowable operations on real-world entities. They should be enforced on entities in the logic to avoid misuse of real-world entities. For example, values of altitude involved in the same calculation should have the identical unit of measurement and reference ground.

3.3.2 A Relationship Specification

Defining the contents and structure of the relationships between elements in logic and their corresponding real-world entities is also a non-trivial task. The relationship specification has two goals: (a) exhibit the mapping connections between elements in logic and their corresponding real-world entities, and (b) reveal the details of the relationship, such as the differences between the two, so that automatic or manual analysis can estimate the discrepancy in the real world.

To satisfy the two goals, the relationship specification is designed as a composition of a set of specifications for relationships. Each specification characterizes one kind of relationship between elements in logic and their corresponding real-world entities. For example, a local variable named `alt` is used to represent the values of altitude. The altitude values are measured by conventional GPS devices. The discrepancy involved could be as far as 400 feet. The delay of data could be as late as 5 seconds.

A specification of relationship contains two components:

- **A set of mapping links**. A mapping link is a pair that combines an element in logic with its corresponding piece in the real-world specification. A relationship specification contains a set of mapping links.

- **An explanation of the relationship**. This explanation specifies details about the relationship. It includes information such as details of differences between real-world values and values in logic that arise inevitably in sensor systems.

A SET OF MAPPING LINKS

A mapping link connects an element in logic to its corresponding real-world entity. Frequently, a relationship specification has more than one mapping links, since one real-world entity could have been accessed by logic elements from different places. These mapping links are all connected to one real-world entity, but connected from different program elements. Various kinds of elements can be connected to real-world entities through these links, such as program variables, constants, and method signatures. Mapping links essentially expose real-world constraints and invariants to programs. Analysis techniques rely on these links to conduct error checking.

An Explanation of the Relationship

The relationship between a logic elements and a real-world entity is more than just a simple mapping. Differences arise due to hardware failures and inaccuracy. To document such differences explicitly, the second part of a relationship specification is a detailed explanation of the relationship.

Errors and inaccuracy exist in different dimensions. For example, sensors provide values of real-world entities to the software, and the values have limited precision and accuracy. Also, these values are sensed according to a discrete-time schedule. The values provided to the software are not the current values of the real-world entities.

In order to specify the relationship, this explanation is defined as a proposition. It typically has the form of a conjunction of predicates where each predicate indicates one dimension of imprecision. For example, the explication for an altitude might take the form:

```
error < delta AND delay < tau
```

The first predicate documents the maximum difference between the actual altitude and the value supplied by the sensing system, and the second documents the maximum delay between sensing the altitude and the associated value being available in the software system.

With the details of an interpretation as outlined above, a new spectrum of analyses become possible. The explicit links to the real-world specification it documents help the programmers track real-world meanings of software entities, and thus reduce misunderstandings. The details of the relationship it exposes, e.g. the discrepancies caused by sensors and timing differences, can be inspected to assure the accuracy of the software system. The analyses based on interpretations will be further described in Chapter 5.

3.4 AN INTERPRETED FORMALISM EXAMPLE

This section illustrates the interpreted formalism concept with an example of altitude. The logic part of the example comes from the software Kelpie flight planner [43]. The code fragment is shown below.

3.4.1 LOGIC COMPONENT

```
public void setElevations(…){
    double alt = from.getElevation();
    …
    if (…)
       …
       alt = maxAlt;
       …
    } else if (…) {
       alt+=legTime*plan.getAircraft().getClimbRate() /60;
       …
    } else if (…) {
       …
```

```
          wp.setElevation(alt);
          …
        } else (…) {
          …
          alt-=legTime*plan.getAircraft().getSinkSpeed()/60;
          …
    }
```

In this logic part, a program variable named *alt* is used to represent altitude of airplanes. The value of altitude is accessed in several statements and involved in several computations.

3.4.2 INTERPRETATION

The piece of interpretation defined for *alt* contains a real-world specification and a relationship specification.

REAL-WORLD SPECIFICATION

- An explication

  Altitude is a vertical distance measurement between a reference level and a point or object. The reference level varies according to the context. Altitude is often used to mean the height above sea level of a point. In geography domain, the term elevation is often preferred rather than the term altitude.

- Real-world semantic attributes

  Altitude has a list of semantic attributes. The attributes are:

(a) *Units of measurement*: the unit for altitude used in this software is feet.

(b) *Dimension*: the dimension for latitude is length.

(c) *Technology*: the technique used to measure the altitude value is radar.

(d) *Geometry plane*: altitude is a value measured vertically.

(e) *Reference level*: the reference level of the altitude value is mean sea level.

(f) *Coordinate system*: the coordinate system is Cartesian.

(g) *Feasible range*: the reasonable range of altitude in this software is between 0 and 30000 feet.

- Real-world constraint and invariants

(a) *A constraint for units*: the units of altitude values must match if they are added/subtracted. The result is of altitude measured in the same units.

(b) *A constraint for dimension*: the dimensions of altitude values must match if they are added/subtracted. The result is of altitude measured in the same dimension.

(c) *A constraint for geometry plane*: values must be measured vertically if they are calculated with values of altitude

(d) *A constraint for reference level*: the reference datum of altitude values must be the same if they are added/subtracted.

(e) *A constraint for coordinate system*: the coordinate system of altitude values must be the same when they are used in one computation.

(f) *An invariant for reasonable range*: values of altitude should stay in the reasonable range.

(g) *An invariant for altitude*: a value of altitude subtracted by a value of altitude yields a value of height.

A RELATIONSHIP SPECIFICATION

- A set of mapping links

  (a) *Variable* `alt` ⇎ *altitude*: this local variable in the function `setElevation()` represents values of altitude.

  (b) *Return values of function* `getElevation()` ⇎ *altitude*: the function returns values of altitude.

- An explanation of the relationship

  This proposition is a conjunction of two predicates:

  $|altitude - alt| < 50\ feet \qquad AND \qquad delay\ (altitude, alt) < 1\ second$

  (a) *Inaccuracy*: variables of altitude, e.g. `alt`, used in the software are different from the values of altitude in the real world. The maximum difference is 50 feet.

  (b) *Delay:* variables of altitude, e.g. `alt`, used in the software are delayed values of the current values. The maximum delay is 1 second.

CHAPTER 4

4. REAL-WORLD TYPE: AN IMPLEMENTATION OF THE INTERPRETED FORMALISM

The concept of logic interpretation is well established, but defining the content and structure of an effective and complete interpretation for *practical* use is a significant challenge. In this chapter, an implementation of the interpreted formalism concept is introduced.

The implementation is based on the concept of *real-world types* [83]. The design of the interpretation is a set of real-world types and a set of real-world type rules defined within the framework of a *real-world type system*.

This chapter first introduces the concept of real-world types and real-world type rules, then presents the overall structure of a real-world type system. In addition, some issues relevant to building a real-world type system are discussed.

4.1 REAL-WORLD TYPES

4.1.1 THE CONCEPT OF REAL-WORLD TYPES

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. For entities of a given real-world type, the type definition documents the real-world specification, the machine representation, and the relation between the two. The specification of a real-world type defines relevant, observable properties of real-world entities of that type. The machine representation defines how a real-world entity is represented in the machine and thus becomes accessible by software. The relationship defines the connection between real-world entities and associated machine elements.

Real-world types are structured 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 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 can include.

4.1.2 THE STRUCTURE OF REAL-WORLD TYPES

A real-world type has a name and a structure that consist of three parts:

- The specification of the type derived from the associated real world entities.
- The machine representation of instances of the type.
- The relationship between the specification and the representation.

Details of these three parts are presented in the remainder of this section.

SPECIFICATION

The specification inside a real-world type contains two parts:

1. An explication.
2. The real-world attributes associated with real-world types.

The first part of the specification is an explication of the type intended for humans. An explication is a careful, detailed and precise explanation provides the reader with a detailed explanation of the various terms and concepts in use. Explications are required for real-world types so as to ensure that a single source of meaning is provided for all the entities with which the computing system interacts. The explication could be in natural language or a combination of natural and formal languages. The explication is the means by which the interpretation is given to the real-world type.

Continuing the example of altitude in an avionics system, given that altitude could mean height above local ground level or height above mean sea level, and could be determined by radar, barometric pressure, or GPS, all these terms need to be defined. Exactly how each semantic concept is used for this type needs to be explained in the explication for altitude.

The second part of the specification is a set of real-world semantics. A real-world semantic can be any real-world property of interest. Every semantic is defined through (a) an explication of the semantic, (b) the set of values that the semantic can take, and (c) reference sources such as online resources or a dictionary.

Returning to the example of an aircraft's altitude, a semantic is the reference level used in the measurement. This semantic could be either sea level or local ground level. Documentation for the specification of such a semantic is shown in Table. 1

Table. 1. Example real-world semantic

Semantic Name	**Taken Value**	**Possible values**	**Explication**

Reference level	local ground	Mean sea level;
		local ground;
			• Reference datum from which altitude value is measured
			• Basic Geo Vocabulary
			• DAML location ontology
			……

A complete set of semantics for altitude would include reference level, frame of reference (surface location, Earth center, etc.), units of measurement, etc.

Units and physical dimensions are examples of real-world semantics, and their introduction into programming languages along with analysis techniques to perform type correctness checks have been explored previously [32, 37, 44]. In our theory of real-world types, units, and dimensions are just special case semantics and are predefined because of their widespread use and importance in real-world properties.

Units can be enumerated as needed by an application. The dimensions semantic consists of the seven basic dimensions of physics (mass, length, time, electric current, temperature, luminosity, and amount of substance) [36]. The existence of this semantic allows the standard dimensional analysis of physics to be applied. For simplicity, in our own use of dimensional analysis, we added *angle* to the set for a vector length of eight. Thus, a semantic value of dimensions is an eight-element vector of integers defining the real-world dimensions of the associated variable. Some example dimensions are:

```
Speed          :    (0,1,-1,0,0,0,0,0)
Acceleration   :    (0,1,-2,0,0,0,0,0)
Energy         :    (1,2,-2,0,0,0,0,0)
```

REPRESENTATION

The machine representation of a type is characterized by a set of semantics that describe the properties derived from the machine context. Machine-world semantics in the representation use a similar format to that used for the real-world semantics in the specification.

Some examples of machine-world semantics and the associated values that they can take are:

```
Encoding       :    integer, floating point, double
Mutability     :    mutable, non-mutable
```

The mutability semantic indicates whether objects of the type are constant, thereby allowing for detection of unintended assignments.

Relationship

The relationship that connects the specification to the machine representation of a real-world type is defined as a proposition. The function typically has the form of a conjunction of predicates where each predicate indicates one dimension of imprecision. In the altitude example, the mapping might take the form:

```
value: error < delta and delay < tau
```

The first predicate documents the maximum difference between the actual altitude and the value supplied by the sensing system, and the second documents the maximum delay between sensing the altitude and the associated value being available in the machine world.

4.1.3 Real-World Type Example

An example of a real-world type is a point in three-dimensional space, `3Dloc`. Measurements designed to locate a point are only relevant if the associated coordinate system is defined completely. If multiple coordinate systems are in use in a program, they must be distinguished. Thus, the real-world type information associated with an instance of the class needs to document the different aspects of the coordinate system.

A possible definition for a point in 3D space, including all of the coordinate system information is shown in Fig. 8. The field names in this definition are the attributes of interest. Many other formats and sets of attributes are possible. In this definition, the explications are omitted for simplicity. Note that this type definition is created just to distinguish coordinate systems. Separately, we need the types of the three fields that will be used for a point in the coordinate system.

```
geographic_cartesian_coord_sys:
  Specification
    explication                 : <text>
    real_world_semantics
      coordinate_sys_type       : cartesian
      target_space              : Earth
      origin                    : center of mass of Earth
      dimensionality            : 3
      earth_model               : spheroid
      x_axis_orientn            : positive toward 0 degrees longitude
      y_axis_orientn            : positive toward 90 degrees east longitude
      z_axis_orientn            : positive northward
  Representation
    machine_semantics
      representation            : record structure – (x, y, z)
  Relationship                  : <null>
```

Fig. 8. A real-world type definition for a coordinate system

For the three fields of the record, one, two, or three different type definitions might be needed. For this example, we assume that the `x` and `y` variables can share a type definition and a second definition is used for `z`. For x and y, we define the type shown in Fig. 9.

```
horizontal_cartesian_axis:
  Specification
    explication            : <text>
    real_world_semantics
      linear_units         : mile
      dimension            : length
      technology           : GPS
      geometry_plane       : horizontal
  Representation
    machine_semantics
      representation       : float
      mutable              : no
  Relationship
      value_error < delta1 and delay < tau1
```

Fig. 9. A real-world type definition for x and y axis

In this example, variable z is part of a complete reference frame with an origin at the center of mass of the Earth but with a presumed offset to mean sea level. The appropriate type definition is shown in Fig. 10. Such a type might be used to hold data in any space of interest. For example, the type could be used to hold location information for aircraft, climbers, balloons, etc.

```
vertical_cartesian_axis:
  Specification
    explication            : <text>
    real_world_semantics
      linear_units         : feet
      dimension            : length
      technology           : radar
      geometry_plane       : vertical
      offset_origin        : mean sea level
  Representation
    machine_semantics
      representation       : float
      mutable              : no
  Relationship
      value_error< delta2 and delay < tau2
```

Fig. 10. A real-world type definition for z axis

4.2 REAL-WORLD TYPE RULES

Within an interpretation defined using real-world types is a set of real-world type rules. The type rules derive from what constitutes meaningful operations in the real world. For example, rules about units conversion, valid and invalid expressions using variables with units, and the types resulting from expressions have to be defined in their entirety to enable suitable checking. The examples given earlier for aircraft altitude (measurement unit compatibility, etc.) will all be included. Rules that derive from application-specific details will also be included. For example, an aircraft's altitude might be measured by radar and by barometric pressure, but for managing altitude in an autopilot, the system developers might require that the radar measurement is used. Type rules can be established to identify incorrect uses of altitude derived from barometric pressure.

Developers define type rules based on the semantics of types and the desired effect on the semantics of operations by programs. In arithmetic expressions, for example, units must match, the dimensionality rules of physics must be observed, arithmetic operations can only be applied to types for which they are defined, and the results of arithmetic operations must have the correct real-world type.

Example type rules include:

- The units of an angle and a latitude must match if they are added. The result is of type latitude measured in the same units.
- A velocity, dimensions (0,1,-1,0,0,0,0,0), cannot be added to a distance, dimensions (0,1,0,0,0,0,0,0).
- A latitude or a longitude cannot be added to a latitude or a longitude.
- An x coordinate in one frame of reference cannot be used in any arithmetic operation with a coordinate from a different frame of reference.
- A variable of type *magnetic* heading cannot be used in an expression expecting a variable of type *true* heading, even if both are represented as integers and are commensurable.

- A variable of type *geodetic* latitude cannot be used in an expression expecting a variable of type *geocentric* latitude, even if both are represented as floating point numbers and are commensurable.

As an example of type-rule definition, consider the semantics of the result of subtracting two operands of type `vertical_cartesian_axis`, e.g., for calculating the altitude difference between two points in the same Cartesian coordinate system. The definition is illustrated in Fig. 11.

$$\dfrac{e1 : \left\{\begin{array}{c} unit: feet \\ Dimension: length \\ coordinate\_sys : cartesian \\ technology: radar \\ offset\_origin: mean\ sea\ level \\ geometry\_plane: vertical \\ axis: z \end{array}\right\} \quad e2 : \left\{\begin{array}{c} unit: feet \\ Dimension: length \\ coordinate\_sys : cartesian \\ technology: radar \\ offset\_origin: mean\ sea\ level \\ geometry\_plane: vertical \\ axis: z \end{array}\right\}}{e1 - e2 : \left\{\begin{array}{c} unit: feet \\ Dimension: length \\ axis: z \\ geometry\_plane: vertical \\ coordinate\_sys : cartesian \end{array}\right\}}$$

Fig. 11. Example type rule definition.

The notation e:T denotes a type judgment (e is of type T), and the overall construct defines an inference rule defining the type of the result of applying a specific operator, here subtraction, to operands, e1 and e2, of the specified types.

4.3 REAL-WORLD TYPE SYSTEM

Our preliminary structure for an explicit interpretation is a set of real-world types and a set of real-world type rules, together with a set of *bindings* between entities in the real world and entities in the logic. The whole structure is referred to as a *real-world type system*.

4.3.1 REAL-WORLD TYPE BINDING

The goal of real-world types is to enable automated checking of the logic based on the real-world entities with which the logic has to interact. To exploit real-world types, we introduce the concept of connecting them with entities in programs such as variables, constants, and functions so as to extend the programmer designated type. This concept is referred to as r*eal-world type bindings*.

Real-world types support analysis and enforcement of real-world constraints on programs in a systematic way, thereby enabling new classes of software fault detection. Real-world type bindings link real-world types *to* machine-level values in ways that such fault detection is enabled without requiring programmers to adopt new machine-level programming languages.

In general, one real-world type binding links one program entity to one real-world type. One program entity is only linked to one real-world type. Multiply program entities can be linked to one real-world type. All program entities that access real-world entities needed to be linked to their real-world types through real-world type bindings. The number of real-world type bindings in a software system depends on the size of software and the number of program entities that access the real world.

4.3.2 REAL-WORLD TYPE SYSTEM

The interpretation is implemented as real-world types and real-world type rules in the context of a *real-world type system*. A real-world type system and its connections to an application system of interest are shown in Fig. 12. In the figure, real-world entities are sensed and affected by sensors and actuators. In the system software, there are software entities that model the associated real-world entities. The relationship between a real-world entity and a software entity documents information such as sensing precision and sensing delay. The details of both and their relationships are documented in the real-world types shown on the left of the figure.

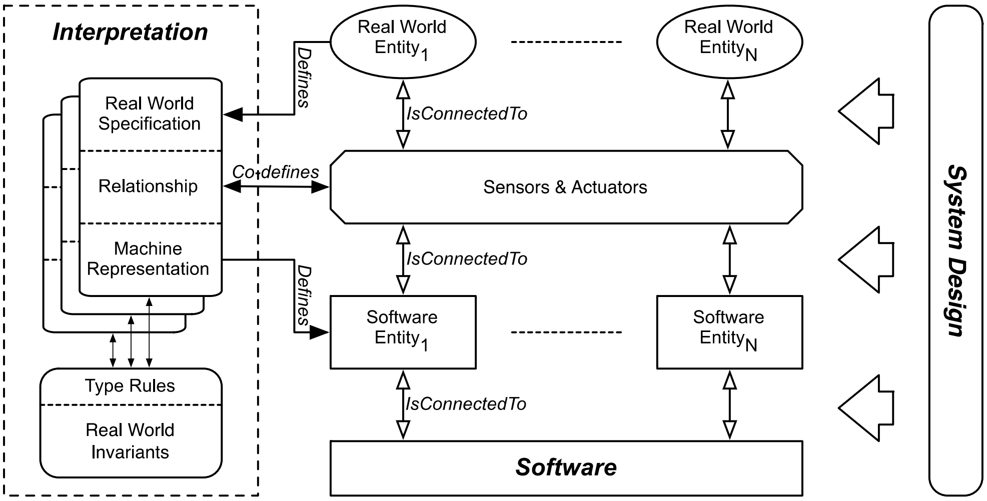


Fig. 12. Real-world type system

In general, a real-world type system documents *three* major items:

1. **A set of real-world types.** These real-world types characterize the properties possessed by entities of the associated types.

2. **A set of real-world type rules.** The type rules define allowable operations on entities of the various types.
3. **A set of real-world type bindings**. The type bindings link program entities to their real-world types.

4.4 REAL-WORLD TYPES AND PROGRAM STRUCTURES.

The notion of type in programming languages includes structures such as arrays and records. An array of integers and a record with multiple fields are types with various associated usage and equivalence rules.

Real-world types do not have any structure beyond what has been discussed. The reason is that the inclusion of all of the structures that arise in programming languages is neither possible nor necessary. Defining real-world types as presented above and using them as building blocks in language structures provides the necessary structure. For a user-defined class, for example, a real-world type would be defined for the class as a whole and separate real-world types could be defined for each field. The type for a given field is then the *union* of the type information for the record and the type information for that field. Nested records accumulate type in the union as each level is progressed.

This approach raises an issue with structured types, because different instances of a user-defined type might have different real-world types. A record structure might be instantiated more than once to hold information with different real-world semantics but identical structures. This issue is dealt with in our present theory of real-world types by associating different type information with different instances of a structure but requiring that the components of a structure have the same type information in all instances. This restriction might be relaxed in future.

CHAPTER 5

5. ESTABLISHING PROPERTIES USING INTERPRETED FORMALISM

The high-level goal of the interpreted formalism concept is to provide a mechanism to improve software quality. Real-world type systems bring the opportunity to realize this goal by establishing new properties in software programs. This chapter first introduces the properties being concerned, then describes several analysis techniques developed to establish these properties.

5.1 PROPERTIES BEING ESTABLISHED

The properties associated with real-world type systems are properties derived from real-world contexts. These properties should be established in software systems that manipulate real-world entities. The properties includes:

- **Property #1**: Program statements conform to static real-world constraints

  Real-world constraints and invariants are derived from real-world context. A large number of real-world constraints describe static relationships between real-world entities, for example, speed multiplied by time yields distance. Such relationships or constraints should be observed in programs.

  Establishing this property reduces violations of real-world constraints at compile time. Errors such as unit inconsistency can be detected before execution of the programs.

- **Property #2**: References from program elements to real-world entities are precise and consistent

  References to a real-world entity normally are scattered in programs. These references could be incorrect; for example, a variable that is expected to represent latitude is used as longitude. The references could also be inconsistent; for example, two variables representing the same altitude use different reference ground: one uses mean sea level, the other uses local terrain. Such references to real-world entities should be precise and consistent.

  Establishing this property reduces misuse of real-world entities by program elements at compile time. Errors such as variables representing incorrect real-world entities can be detected before execution of the programs.

- **Property #3:** Approximations caused by hardware are accessible by users

  The differences or approximations introduced by hardware are crucial to software systems, especially safety-critical ones. Many of these software systems have requirements on the accuracy of data. For such systems, approximation information should be accessible by system experts for necessary inspection.

Establishing this property allows analysis of the differences caused by hardware. The information can also be used to estimate the error in a variety of computations throughout an entire software system.

- **Property #4:** Runtime values of program variables conform to real-world constraints

  In the real world, entities consistently observe real-world constraints, e.g., laws of physics. The values of these real-world entities stay in a reasonable range. For example, the value of latitude ranges from -90 to 90 degrees. In programs, however, the runtime values of real-world entities could be random and do not obey the constraints. Therefore, it is necessary to ensure these values stay in reasonable ranges at runtime.

  Establishing this property prevents programs from unwanted behaviors, thereby reducing improper manipulations of entities in the real world.

5.2 ESTABLISH PROPERTIES

In order to establish these properties, *four* analysis techniques were developed.

- Real-world constraint checking
- Reasonable range analysis
- Targeted inspection

- Runtime assertion generator

Table. 2 summarizes the analysis techniques with their intended properties and sources in real-world types that enable the analyses. The details of these techniques are described in the follow sections.

Table. 2. Analysis techniques provided by real-world type system

Property	Analysis Technique	Source of analysis
Program elements conform to static real-world constraint	Real-world constraint checking	Real-world semantic attributes Real-world type rules
	Reasonable range analysis	Real-world semantic attributes
References to real-world entities are precise and consistent	Real-world constraint checking	Real-world semantic attributes Real-world type rules
	Targeted inspection	Real-world semantic attributes
Approximations caused by hardware are accessible by users	Targeted inspection	Real-world semantic attributes Relationships between real-world entities and elements in logic
Runtime values of variables conform to real-world constraints	Assertion generator	Real-world semantic attributes

5.2.1 REAL-WORLD CONSTRAINT CHECKING

The first kind of analyses assures that logic statements conform to static real-world constraints. This analysis technique is referred to as *real-world constraint checking*.

With an interpreted formalism, static real-world constraints and invariants of concern are defined in the real-world specification. These constraints and invariants specify the sets of semantic attributes that result from operations involving sets of semantic attributes. These constraints and invariants are to be checked throughout the logic statements, and diagnostics displayed for the user identifying violations. Subsets of the constraints can be checked separately if desired. For example, separate checking of just the constraints for units might be useful.

The checking of static real-world constraints operates by assessing compliance of the subject logic with the constraints. For example, adding a floating-point value representing latitude (an angle) to a floating-point value representing longitude (also an angle) is an error, despite the fact that both are floating-point and commensurable. More importantly, even adding two variables of latitude is almost certain to be meaningless. Adding an angle to a variable of latitude might be meaningful because the addition might represent an update to latitude resulting from motion. Nevertheless, that addition would also be meaningless if the latitude was measured in degrees and the angle measured in radians.

In the constraint checking process, various difficulties arise and various choices have to be made about how to deal with issues such as real-world type conversion. In Chapter 0 summarizes the issues and discusses solutions in the context of Java and the prototype implementation of real-world types.

5.2.2 RANGE AND REASONABLENESS ANALYSIS

Variables that manipulate real-world entities frequently have their boundaries or restrictions from the point of natural science and common sense. For example, values of latitude in degrees should be in the range of [-90,90]. The restriction should not be violated at any time. A real-world type system provides a capability to statically detect violations of such restriction. In this dissertation, the range is referred to as *reasonable range*. The detection mechanism is referred to as *reasonable range analysis* or *range analysis*.

The underlying mechanism relies on the technique of interval analysis [23, 24, 50, 55, 80]. Reasonable range is a default semantic attribute in all real-world entity specifications. Interval arithmetic uses these ranges to compute ranges for all variables involved in scientific computations. If in any piece of the computation, the computed range exceeds the reasonable range of the corresponding real-world entity, a warning message will be issued, and an inspection suggested.

For example, for the program statement below:

```
final double roughLonSep = range / ( 60 *
            Math.cos(Math.toRadians(point.getLatitude())));
```

The method `point.getLatitude()` has reasonable range of `[-90, 90]`, `Math.toRadians(point.getLatitude())` has range of `[-pi/2, pi/2]`; so that `Math.cos(…)` has range of `[0,1]`. Then, for the right side of the statement, interval arithmetic produces a range with a high bound of +∞, but the left side is a central angle of Earth with the range of `[-90, 90]`.

5.2.3 ASSERTION GENERATION FOR RUN-TIME ASSURANCE

Some real-world constraints and invariants cannot be checked statically and have to be deferred to execution time. These real-world invariants document relationships between real world entities of constraints on values. For example, an aircraft’s altitude should lie between approval flight limits, and an autopilot should not command a pitch up angle beyond a safe limit. Such restrictions are common, and assertions are sometimes included in software logic to check them at runtime.

The richness of an interpreted formalism, e.g., the real-world specifications and interpretations, leads to extensive execution-time checking. Where an entity is implemented in logic with a machine type, its real-world specification and interpretation specify details that can be turned into constraints for semantic attributes that have accessible values such as numeric variables.

As an example, consider the use of the real-world entities latitude and longitude measured in radians. Variables of these entities might use the floating-point type in software logic, but the real-world specification documents the ranges. The real-world specification could also document properties such as the maximum rate of change of variables of these entities. Thus execution-time range checks can be generated for assignments, and reasonableness checks can be generated for replacement values.

5.2.4 TARGETED INSPECTION

Careful inspection of software artifacts has proven to be a general and effective mechanism for detecting defects. The availability of details in real-world specifications permits inspections of how the software system interacts with the real world in a systematic and comprehensive way. For example, variables used to hold values read from sensors and sent to actuators can be identified and selected for inspection and analysis of details such as precision and delay.

The inspection based on an interpreted formalism is referred to as *targeted inspection*. The concept is to use the information in the interpreted formalism to identify locations in the program where human checking is needed. In general, all of the variables of a given real-world entity can be identified and selected for inspection to check for any details of interest. Since each real-world specification encapsulates the semantics of real-world entities, detailed inspections of semantic issues can be conducted. The natural language explication for a specification of a real-world entity can be examined to ensure that all of the uses of variables of that entity are consistent with the semantics.

As an example of targeted inspection, consider the notion of bearing in an avionics system. Bearing is the angle to the east of west from the reference north. The reference north could mean the true north or the magnetic north. A targeted inspection can determine whether all the uses of bearing in the logic are consistent with the intended use.

The second area of concern with altitude is the interpretation details between true altitude and the available values in the logic. A targeted inspection can be used to examine the approximation and ascertain by inspection and appropriate numerical analysis that the logic values are adequate for the associated computation.

CHAPTER 6

6. DEVELOPING INTERPRETED FORMALISMS

Real-world type systems provide analysis techniques to establish useful properties in software programs. The benefits provided by real-world type systems can be substantial. However, the effort involved in creating and using a real-world type system could be significant. Using a real-world type system requires developing its three components: (1) real-world type definitions, (2) real-world type bindings, and (3) real-world type rules. Each component requires a certain amount of effort.

This research developed a comprehensive framework to facilitate the development of real-world type systems by synthesis. Fig. 13 shows an overview of this framework. In this framework, development of a real-world type system starts with three sources: (a) existing context documentation, (b) existing libraries of real-world type systems, and (c) existing software application materials. All three sources can be used to produce candidate artifacts of a real-world type system. Existing system documentation has domain and linguistic models; existing real-world type systems can be reused as real-world type libraries; existing software contains application materials such as source code and documents. Candidate real-world type system elements produced from these sources are reviewed by domain experts. Those candidates considered valid by experts are then refined to construct a real-world type system.

In this chapter, all three sources of development materials are introduced and the way they can be used to support creation of real-world type systems is discussed. A collection of mechanisms is used to synthesize candidate real-world type systems from applications materials.

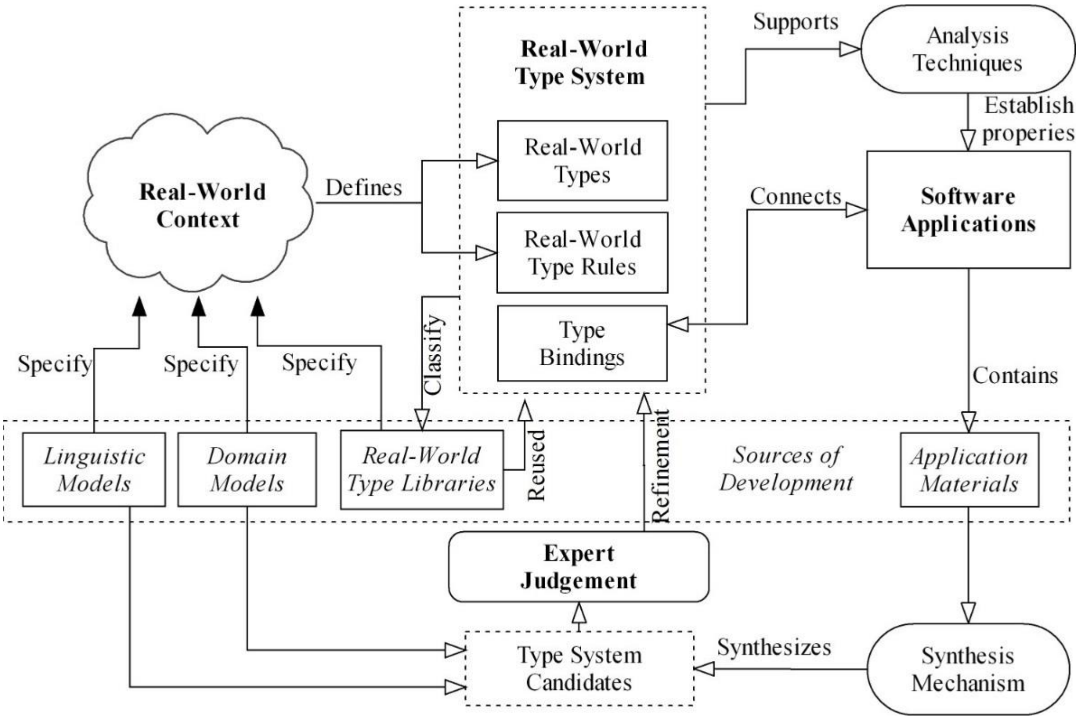


Fig. 13. Development of interpreted formalisms

6.1 SOURCES OF DEVELOPMENT

6.1.1 EXISTING CONTEXT DOCUMENTATION

Domain models and linguistic models can be used to characterize the domain affected by software systems. They are important sources for producing candidate real-world types and type rules.

- Domain models

  Domain models identify key entities in an application domain and capture the relationships among the entities. These models provide candidate real-world type systems. For example, in order to develop a real-world type system for a flight control application, a geographical or geospatial ontology is a useful source. Classes and instances in the ontology are candidate real-world types, and properties and restrictions in the ontology are candidate real-world type rules. In addition to ontologies, other kinds of domain models such as UML models, requirements models in KAOS [47], and i* [14], can also be used in creating real-world types and type rules.

- Linguistic model

  Linguistic models are another important source. The theories and results provided by linguistics describe the abstract entities and processes manipulated by humans in the process of achieving communications. Examining linguistic results allows recognition of which entities and processes contribute to the outcome of communications and in what ways. These entities and processes can significantly improve the understanding of real-world context. The entities are ideal input candidate for explications and semantic attributes inside real-world type definitions. Connections between these entities are candidates for real-world type rules.

6.1.2 EXISTING REAL-WORLD TYPE SYSTEMS

Artifacts in real-world type systems are highly reusable. Real-world types and type rules define characteristics of real-world entities, and those characteristics are unlikely to change. Therefore, real-world types and type rules are ideal candidates for reuse. The notion of *real-world type library* is introduced to be the major reusable unit. Each library contains a set of real-world types and type rules, and each library serves a particular purpose. For example, a real-world type library usually specifies a particular domain, e.g., the geographic domain; a separate library might be specifically created for unit consistency.

Real-world type libraries are involved both before and after developing a real-world type system for a given software application. Developing its real-world type system, real-world type libraries pertinent to the application can be referenced and reused. Real-world types and type rules in the libraries can be directly plugged into a new real-world type system. After a real-world type system is developed, newly created real-world types and type rules are classified. The results are used to update existing real-world type libraries or create new ones.

6.1.3 EXISTING SOFTWARE APPLICATION MATERIALS

A software application itself provides sources as well. A software application contains various kinds of materials, such as source code, requirement documents, design documents, bug history, and maintenance records. These materials support creating a real-world type system. Concepts in these materials are candidates for real-world type definitions. Relationships between these concepts are candidates for real-world type rules. In the next section, mechanisms that extract candidates for real-world type systems from application materials, e.g. source code, are discussed.

6.2 DEVELOPMENT OF REAL-WORLD TYPE SYSTEMS FROM APPLICATION MATERIALS

Domain models, linguistic models, and real-world type libraries, demand effort likely from other parties before the development of a new system can begin. Thus, in many cases, when starting development of a real-world type system, domain models, linguistic models, and real-world type libraries will be unavailable or incomplete; therefore existing application materials are the only sources that can be relied upon. Therefore, the real-world type system development framework focuses on developing real-world type systems from application materials.

The framework:

- Synthesizes candidate real-world types and their semantic attributes.
- Infers real-world type bindings for program variables.

- Synthesizes candidate type rules from verified or trusted programs.

The mechanism starts with a software application's source code. It collects identifiers and programming patterns automatically and analyzes program statements in both automatic and semi-automatic ways. With this information and a variety of reference sources including ontologies, dictionaries, and other online resources, the synthesis mechanism produces candidates of a real-world type system.

6.2.1 SYNTHESIS OF INTERPRETATIONS

An overview of the synthesis framework is shown in Fig. 14. Starting with a variety of assets including the target application's sources, other application documents, and references (including ontologies, dictionaries, and other natural-language information), the synthesis mechanism:

- Extracts candidate real-world type definitions from the subject software.
- Infers candidate type bindings from program statements for which developers have high confidence, and from defined type inference rules.
- 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.

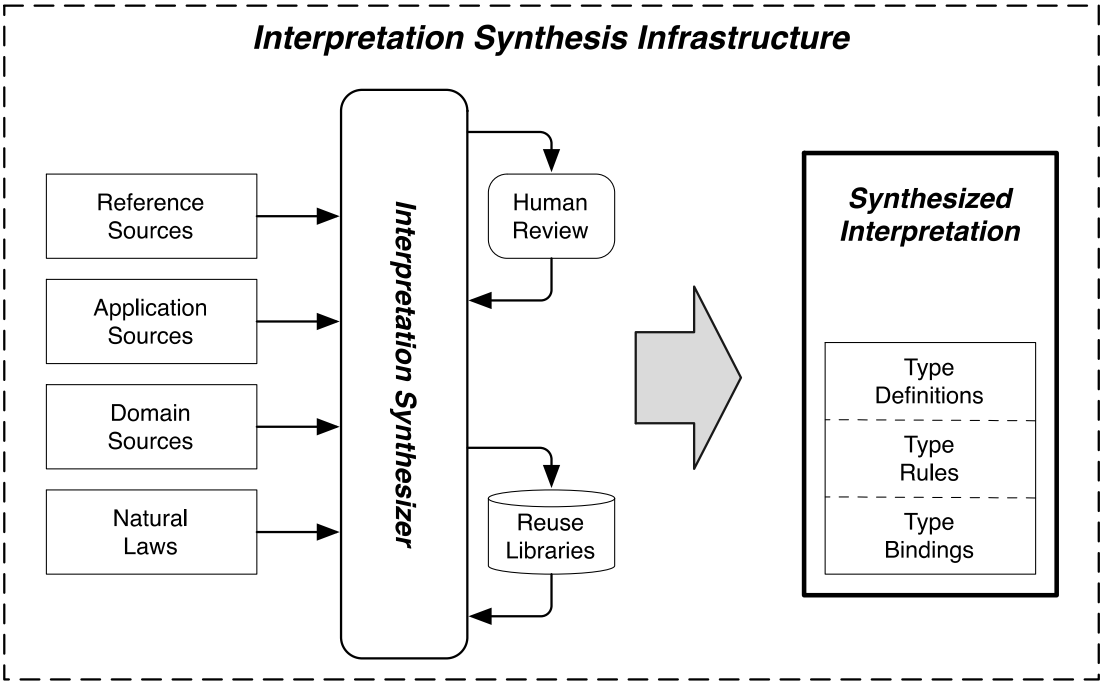


Fig. 14. Overview of real-world type system synthesis framework

6.2.2 SYNTHESIS OF REAL-WORLD TYPES

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 real-world specification contains attributes that describe the associated real-world characteristics, (2) a machine representation that is relevant to the computation, and (3) a specification of the relationship between the two. 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 frequently prepare. Programmers usually follow some 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 classes are frequently named as 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 Fig. 15. The individual steps operate as follows:

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

- **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*.

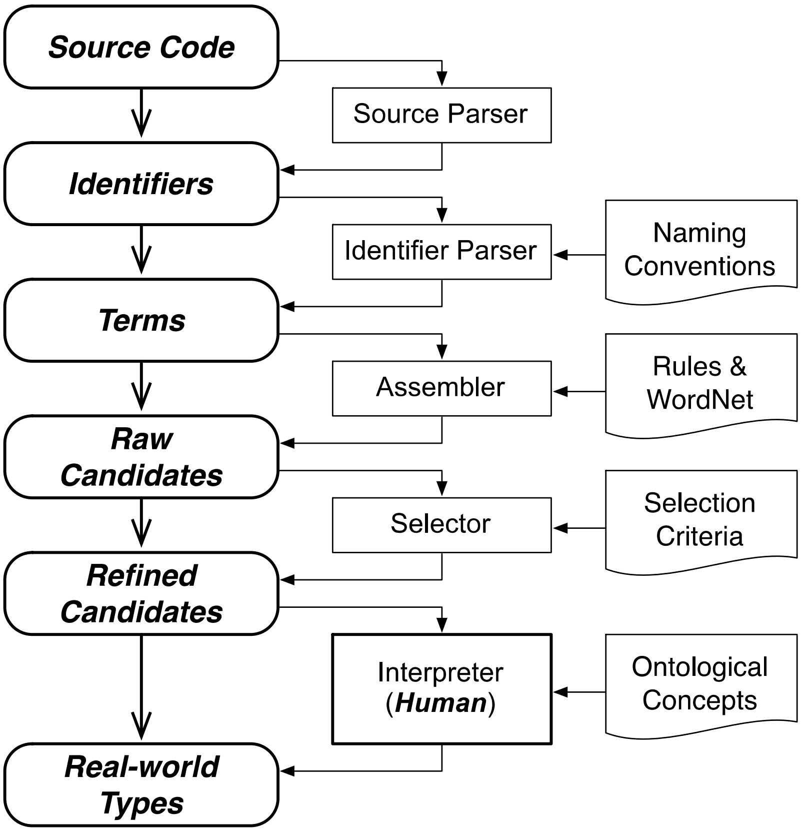


Fig. 15. Synthesis of candidates for real-world types

- **Assembler**. For each term located in an identifier, the WordNet lexical database is consulted to determine whether the term is a noun [78]. 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 abbreviated 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.

TYPE SYNTHESIS EXAMPLE

The following code snippet comes from the software application used in the case study in Chapter 9:

```
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.

6.2.3 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.

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.

Fig. 16 illustrates the parameter inference approach. The individual steps are as follows:

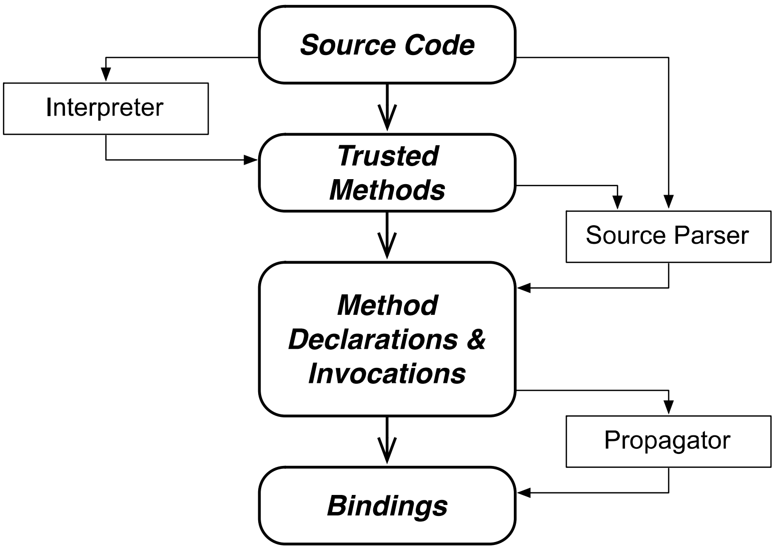


Fig. 16. Type binding inference from parameters to arguments

- **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.

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.

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 the 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.

6.2.4 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. This phase is based on the hypothesis that, for an existing program that 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.

The mechanism in the framework proceeds in three steps shown in Fig. 17:

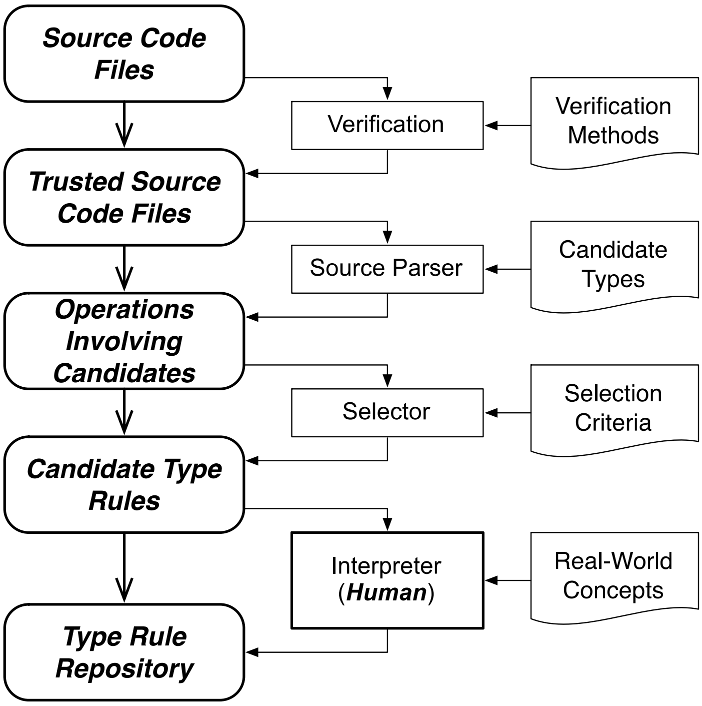


Fig. 17. Synthesizing real-world type rules

- **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.

CHAPTER 7

7. PROTOTYPE IMPLEMENTATION

If interpreted formalisms are to be used in the development of realistic software systems, an approach to integrating them into widely-used languages and development methods is needed. This necessity demands tools that support the approach. This chapter introduces a prototype designed and developed to support interpreted formalism in the form of real-world type systems for Java.

In order to develop and apply the idea of the interpreted formalism in software practice, a pragmatic design of systems for interpreted formalisms that developers can follow is needed. For pragmatic purposes, a design choice was made that interpreted formalisms should be developed, inspected, and analyzed *without* requiring changes to the subject logic, i.e., the software of intent. This choice provides three major advantages:

- Interpreted formalisms do not obscure the basic structure of the logic.
- Interpreted formalisms can be added to existing logic without having to modify (and possibly break) the original logic.

- Interpreted formalisms can be added to logic without impeding the development of the logic itself.

Motivated by this design choice, systems of interpreted formalisms should hide the internal representation of interpreted formalisms and hide the internal structure of the system from the users. Users are provided with a straightforward viewpoint in which they can focus on developing and analyzing interpreted formalisms.

From the users' viewpoint, development of real-world specifications and interpretations are independent to the development of logic. In this manner, the development process of each piece of interpreted formalism can be highly parallel and incremental without impeding each other.

In the following two sections, the design of the Java prototype is introduced first, and then the detailed user interfaces of the prototype are presented.

7.1 DESIGN OF THE JAVA PROTOTYPE

Based on the design choice stated above, a Java prototype is designed and implemented for interpreted formalisms in the form of real-world type systems.

In general, this prototype allows users to conduct two activities:

- **Develop real-world type systems for software applications of interests**. This activity includes developing real-world type definitions, real-world type rules, and real-world type bindings. Since development of real-world type systems has multiple starting points, different development methods are provided to support different starting points.

- **Use analysis techniques provided by real-world type systems**. Various kinds of analysis techniques are supported. Results of the analyses are presented with diagnostics and causes.

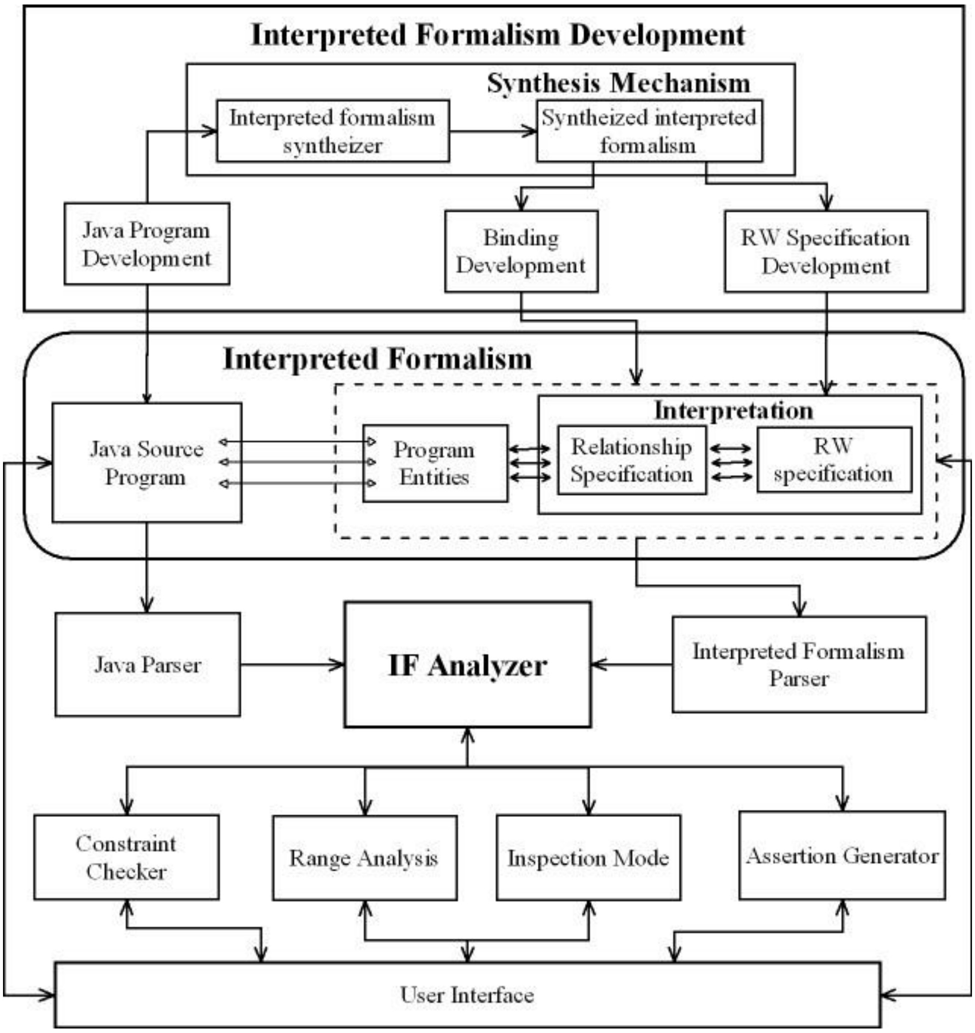


Fig. 18. Design of the Java prototype

The structure of the prototype is illustrated in Fig. 18. The development of the interpreted formalism is shown at the top of the figure. Interpretations and subject Java source programs are shown in the middle of the figure. The analyzer of interpreted formalisms is shown at the lower part of the figure.

7.1.1 USE OF THE PROTOTYPE

In the prototype implementation, the interpretation is accessed via the user interface enabling:

- The establishment and display of bindings between items in the Java program and real-world type definitions in the interpretation.

  Selecting an entity in the Java program that is to have a real-world type (clicking on the text) and selecting the particular real-world type to be used (clicking on the type name) establishes a binding. This binding corresponds to concept of real-world type binding introduced in section 4.3.1. In this prototype, the bindings are often referred to as *annotations*. These annotations can be displayed as comments in JavaDoc of Java programs.

- Reference to the details of the interpretation.

  All definitional aspects of the real-world types and all bindings to Java entities can be displayed. The set of bindings can be displayed in various ways, e.g., all bindings, binding of a given Java entity, all Java entities bound to a particular real-world type, etc.

ANALYSIS TECHNIQUES

To support analysis of the system, a custom parser produces a representation of the subject Java program as an abstract syntax tree, and the implementation of the interpretation produces a database that documents all of the details of the interpretation. The abstract syntax tree and the details of the interpretation are processed by an analyzer shown in the center of the figure that supports four types of analysis:

- *Real-world constraint checking*. A constraint checker was implemented for this analysis. It loads the real-world constraints, examines the parsed subject Java programs, and then statically checks for violations of real-world constraints. Diagnostics are displayed for user to confirm.

- *Reasonable range analysis.* A range analyzer was implemented to conduct interval analysis on the source programs. Warning messages are issued when calculated intervals of program elements exceed their reasonable ranges.

- *Assertion generation.* The assertion generator synthesizes assertions as Java fragments that can be inserted into the subject program to implement runtime checking of real-world invariants that cannot be checked statically.

- *Targeted inspection.* The inspection mode provides a display allowing all interpreted Java entities to be traced to their interpretations. Definitional aspects of the real-world specifications and all interpretations to Java entities can be displayed. It also synthesizes a checklist of locations in the subject program at which human inspection is required to check real-world constraints or invariants that cannot be checked statically or dynamically.

DEVELOPING AN INTERPRETED FORMALISM

As indicated by the top part, Java programs are separately developed and parsed without being affected by the development of interpreted formalisms. In this way, the two artifacts can be developed in parallel without impeding each other. Engineers can manually create interpreted formalisms through user interface. In addition, the prototype implements two features that facilitate developing interpreted formalisms:

- *Synthesis of interpreted formalisms*. The concept of synthesis has been introduced in Chapter 6. The prototype has mechanisms that implement the synthesis framework. Specifically, three mechanisms were developed: (1) synthesis of candidate real-world types, (2) synthesis of real-world type rules, and (3) synthesis of real-world type bindings.
- *Reuse interpreted formalism*. Existing real-world type systems can be reused instantly for developing new real-world type systems. In this prototype, real-world types and type rules are stored as text files and can be readily reused in other real-world type systems.

REPRESENTATION OF AN INTERPRETED FORMALISM

The middle part of Fig. 18 shows that an interpreted formalism is composed of Java source programs and an interpretation. The interpretation comprises (1) a real-world specification and (2) a relationship specification that document the relationships between real-world entities and relevant program entities.

7.1.2 TYPED PROGRAM ELEMENTS

Software entities that have real-world meanings should be interpreted with their real-world specification. In the context of a real-world type system, these software entities are bound with real-world types. The Java prototype covers most of these software entities. In the prototype, the Java entities being bound with real-world types are: (a) local variables, (b) fields in classes, (c) method parameters, (d) method return value, and (e) class instances. In order to make the development of the prototype tractable, the current version imposes some restrictions on the use of interpretations in Java, specifically:

- *Fields*. Fields in classes are assumed to be monomorphic, i.e., a field in a class is assumed to have the same corresponding real-world entity in all class instances. Fields are interpreted with real-world specifications inside the class declaration body.
- *Class instances*. Different instances of a class might have different real-world meanings and so the interpretation is of the instance, not the class. For example, suppose a class Point has three fields x, y, z. Further, suppose that pt1 and pt2 are both instances of Point but are from different coordinate systems. Writing a statement that involves both `pt1.x` and `pt2.x` such as `pt1.x + pt2.x` might be an error and so the two instances need to be distinguished.
- *Method return value*. Each function with a return value is interpreted with a real-world specification. If a particular method is not interpreted with a real-world specification, the analysis treats the method as polymorphic. For a polymorphic

method, at each invocation site, all the expressions in the method declaration body are examined to determine the real-world type of the return statement. That ultimately will be the real-world type of the method invocation. If the method contains multiple return statements, the interpretation for the return value will be the one with no errors. Also, if interpretations for return statements are inconsistent, a warning message is issued.

- *Arrays*. Since individual array elements cannot be interpreted separately, all objects inside an array are treated as having the same interpretation.
- *Constants*. Variables are interpreted when declared, but constants are used as needed. Constants are dealt with simply by associating each one with a hidden variable and associating an interpretation with the variable.
- *Compound objects*. Class instances introduce the possibility of nesting of interpreted real-world entities because the class might have an interpretation and the fields within the class might have interpretations. In that case, the real-world specification of a qualified name is the union of the specifications of all the elements in the path to a specific item of interest in an expression. This same rule applies to method invocation where fields are retrieved such as `cs2.get_x();`

7.1.3 TYPE CONVERSION

An important issue in the prototype is the conversion between real-world types. For example, a variable whose real-world type indicates that the measurement unit is “feet” could be switched to a different measurement unit, say “meters”; by calling a function that effects the switch, multiplying by a constant, or multiplying by a variable. Each of these mechanisms could be implemented as standalone assignment statements; within other expressions, as expressions stated as actual parameters, as return values, and so on.

The prototype analyzer deals with explicit interpretation conversion simply by including conversion rules associated with whatever special operator or function is used. For example, a conversion function is documented as taking one real-world type as its input parameter and a second real-world type as its return value.

Implicit type conversion is more difficult. Conversions between real-world types can be syntactically simple. For example, a conversion from feet to inches requires multiplying a variable storing a value in feet by 12, and the constant might not be identified specifically to support analysis. The difficulty lies in locating such conversions automatically without generating false negatives.

Implicit type conversion is dealt with in the prototype by requiring that the programmer investigates each diagnosed error and mark implicit type conversions as such. Thus, diagnostics will be generated for type conversion of which the prototype was unaware, because the mismatch appears to be a violation of the real-world constraints. In those cases, the programmer suppresses the diagnostic by indicating that there is an expected implicit conversion. By doing so, the programmer indicates that the diagnostic has been investigated and the code found to be as desired.

7.1.4 POSSIBLE ERRONEOUS STATEMENTS

Real-world constraint checking and range analysis are the two primary analysis techniques implemented in the prototype. They analyze the abstract syntax tree of the source code, inspect every node in the tree, and discover possible violations of real-world type rules. Violations are reported in the following syntactic structures:

- *Assignment and VariableDeclarationStatement*. In these two kinds of expressions, the real-world types for the left-hand side and right-hand side could be inconsistent. For example, assigning a variable of latitude to a variable representing longitude is inconsistent. Analysis techniques issue error reports when the two sides are inconsistent.
- *InfixExpression*. The infix expression involves calculations of different variables. The computation could be wrong. The prototype assumes a calculation is disallowed if no real-world type rule permits it. For example, the expression `lat - lon`, intends to compute the difference between two latitude values, but mistakenly refers to the variable of longitude. This error will be detected by the analysis of constraint checking.
- *MethodInvocation*. Typically, a method or a function has real-world types bound with its function parameter(s). When the function is called with a set of arguments, analysis techniques check if real-world types of the argument(s) are consistent with the real-world types of the parameter(s). For example, the function `distanceTo(float lat, float lon)` expects latitude and longitude values of

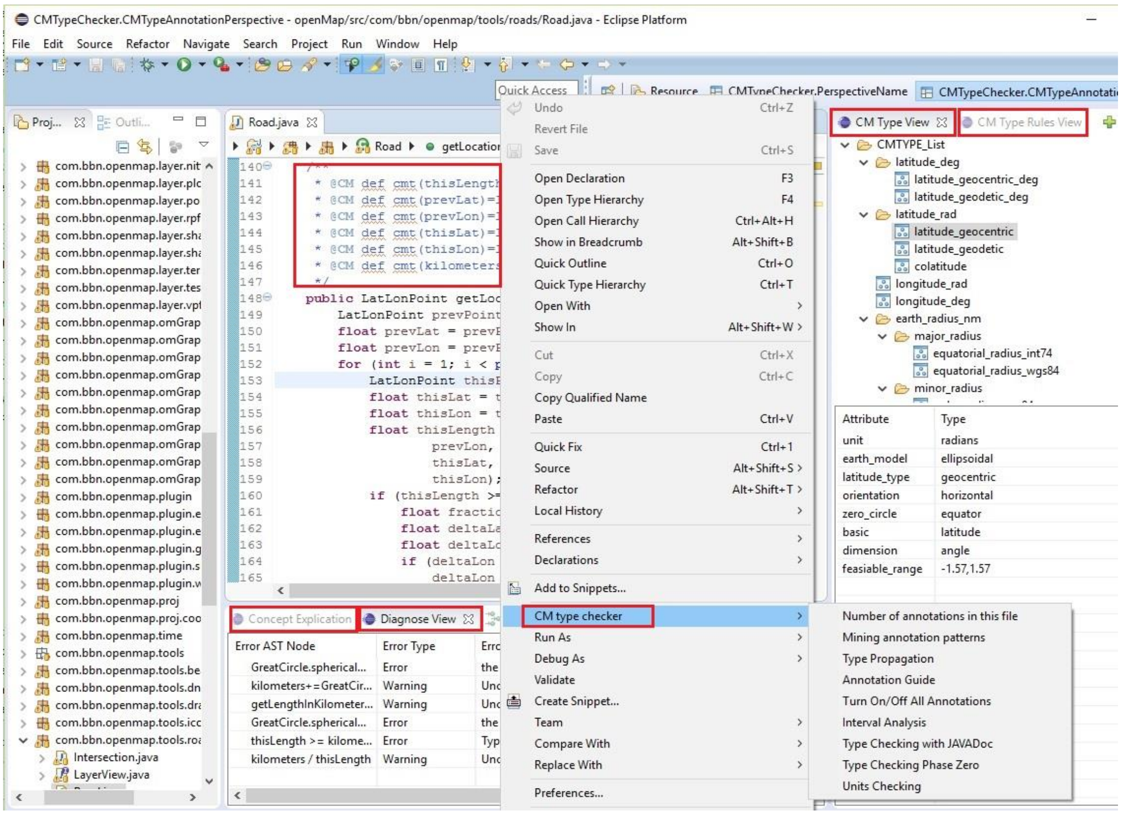
units radians. If a method invocation `distance(lat1, lon2)` has arguments `lat1` and `lon2` of units degree; an error is issued.

- *ReturnStatement*. Frequently, a method or function has its return value bound with a real-world type. The constraint checking analyzes the body of the function, check if one (or all) of the return statements is bound with the real-world type declared for the return value. For instance, if a method returns void while it declares returning a valid real-world type, an error message is issued.

7.2 JAVA PROTOTYPE USER INTERFACES

This section describes details of the Java prototype. The prototype is named *CMTypeChecker*, and is implemented as an Eclipse Rich Client Platform (Eclipse RCP). The prototype is about 25,000 line of code, is composed of 136 source files, and is organized in 20 packages. The figure below shows a snapshot of startup page of the prototype.

The term *CMType* is just an old name for the real-world type. The two names of *CMType* and *real-world type* can be used interchangeably. The red boxes in Fig. 19 mark a few important components implemented in the prototype.



```
    /**
     * @CM def cmt(thisLengt
     * @CM def cmt(prevLat)=
     * @CM def cmt(prevLon)=
     * @CM def cmt(thisLat)=
     * @CM def cmt(thisLon)=
     * @CM def cmt(kilometers
     */
    public LatLonPoint getLoc
        LatLonPoint prevPoint
        float prevLat = prevP
        float prevLon = prevP
        for (int i = 1; i < p
            LatLonPoint thisP
            float thisLat = t
            float thisLon = t
            float thisLength
                    prevLon,
                    thisLat,
                    thisLon);
            if (thisLength >=
                float fractio
                float deltaLa
                float deltaLo
                if (deltaLon
                    deltaLon
```

Error AST Node	Error Type	Errc
GreatCircle.spherical...	Error	the
kilometers+=GreatCir...	Warning	Unc
getLengthInKilometer...	Warning	Unc
GreatCircle.spherical...	Error	the
thisLength >= kilome...	Error	Typ
kilometers / thisLength	Warning	Unc

Attribute	Type
unit	radians
earth_model	ellipsoidal
latitude_type	geocentric
orientation	horizontal
zero_circle	equator
basic	latitude
dimension	angle
feasiable_range	-1.57,1.57

Fig. 19. The Java prototype

CMTypeChecker is organized as one Eclipse perspective extension, two popup menu extensions, two wizard extensions, and four view extensions [87]. Table. 3 below summarizes their extensions and their purposes.

Table. 3. Prototype as Eclipse RCP

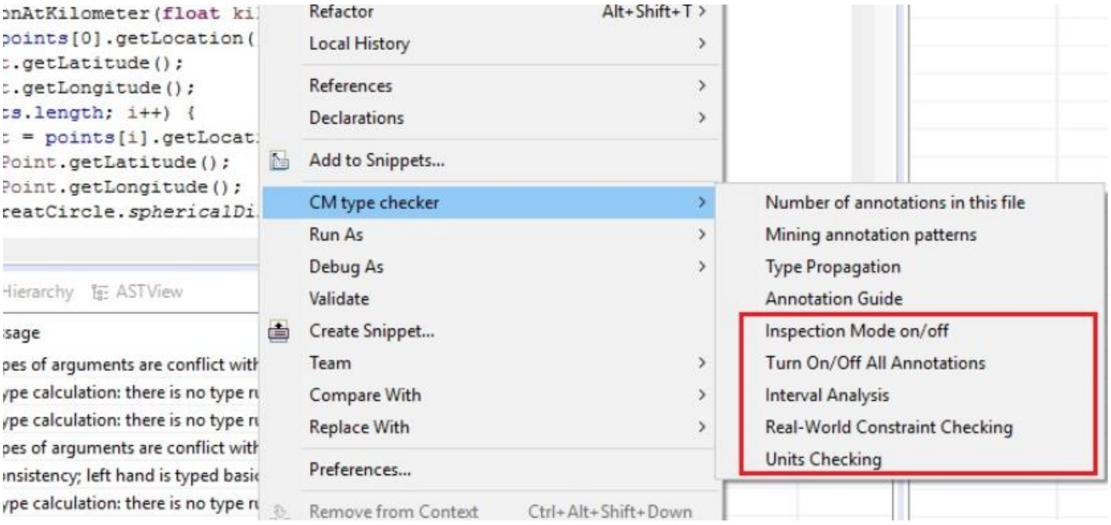
Eclipse Extension	Component Name	Purposes
Eclipse Perspective	CMTypePerspective	The main interface when the prototype starts
Eclipse Popup Menu in Java Editor	CM Type Checker	All checking capabilities are triggered from this menu: (1) Real-world constraint checking (2) Units checking

		(3) Reasonable range analysis (4) Inspection mode (5) Switch displays for real-world type bindings
Eclipse Popup Menu in Resource Navigator	CM Type Facilities	All synthesis framework capbilities are triggered from this menu: (1) Synthesis mechanism (2) Analysis on a package or a project (3) Config the real-world type system
Eclipse View	Concept Detail View	(1) Display the details of specification for a real-world entity; (2) Display the explication of a real-world entity
Eclipse View	CM Type View	(1) Trigger the wizard of creating new real-world types; (2) Display the tree of all real-world types; (3) Display semantic attributes of a specific real-world type; (4) Bound real-world types to elements in a Java program; (5) Find all occurrences of a real-world type
Eclipse View	CM Type Rule View	(1) Trigger the wizard of creating new real-world type rules; (2) Display the defined type rules in current Java project
Eclipse View	Diagnose View	(1) Display the diagnostics produced by analysis techniques; (2) Trace the sources of the diagnostics
Eclipse Wizard	CM Type Wizard	Wizard that allows users to create, manage, and delete real-world types.

Eclipse Wizard	CM Type Rule Wizard	Wizard that allows users to create, manage, and delete real-world type rules.

7.2.1 POPUP MENU: CM TYPE CHECKER

The popup menu *CM type checker* is an Eclipse extension on the default Java editor. The popup menu contains menu items that trigger all analysis techniques, including real-world constraint checking, units checking, reasonable range analysis, and inspection mode. It also contains a menu item that can switch on and off the display of real-world type bindings. The figure below shows the popup menu and its items:



```
onAtKilometer(float ki
points[0].getLocation(
t.getLatitude();
t.getLongitude();
ts.length; i++) {
t = points[i].getLocat
Point.getLatitude();
Point.getLongitude();
reatCircle.sphericalDi
```

```
sage
pes of arguments are conflict with
ype calculation: there is no type ru
ype calculation: there is no type ru
pes of arguments are conflict with
nsistency; left hand is typed basic
ype calculation: there is no type ru
```

Fig. 20. Popup menu: analysis techniques

In the figure, the menu contains five menu items. Their functions are:

- *Inspection Mode on/off*. The menu item triggers inspection mode. When the inspection mode is on, users can readily trace real-world types that correspond to program elements. The details of real-world types are displayed in a tooltip when a mouse click is pressed to a program element, e.g., variable, function.

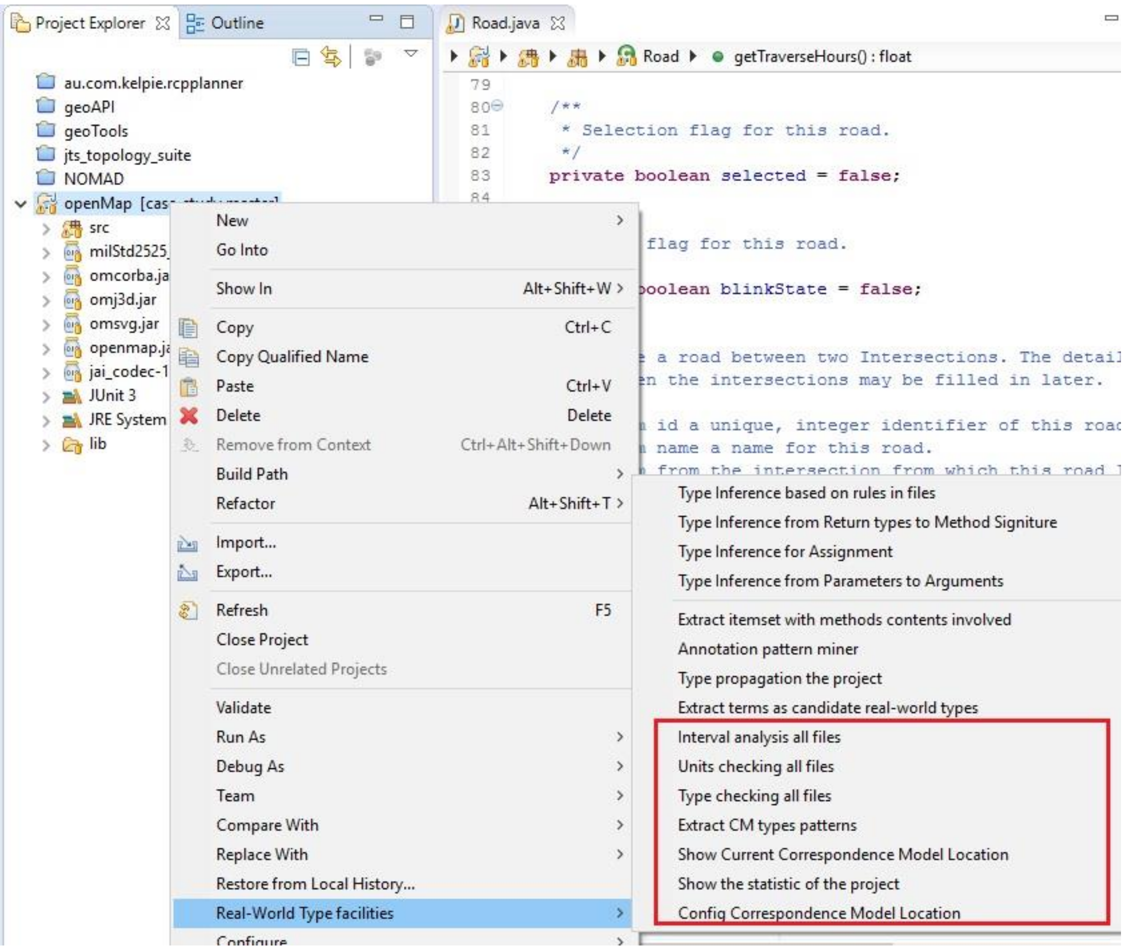
- *Turn on/off All Annotations*. This menu item is used to switch display of real-world type bindings. As indicated in Fig. 20, some comments started with tag name *@CM* are displayed in the Javadoc. These comments can be switched off by this menu item if engineers prefer not to see these comments.
- *Interval Analysis*. This menu item triggers reasonable range analysis on the source programs. Candidate errors are displayed in the *diagnose view*.
- *Real-World Constraint Checking*. This menu item triggers real-world constraint checking on the source programs. Candidate errors are displayed in the *diagnose view*.
- *Units Checking*. Units checking has been studied extensively by other researchers. Therefore, in this prototype, units checking is implemented as an independent function that can be triggered by this menu item.

7.2.2 POPUP MENU: CM TYPE FACILITIES

The popup menu *CM type facilities* is an Eclipse extension on Project Explorer of Eclipse. The popup menu contains menu items that trigger all synthesis framwork techniques. Several menu items are created to analyze all source files in the current project.

Fig. 21 shows the popup menu and its items. In the figure, the rectangle contains 7 menu items. Their functions are:

- *Interval Analysis all files*. The menu item conducts reasonable range analysis on all source files in the project. The results of analysis are recorded in a .csv file. The user then inspects the files which have errors reported.
- *Units checking all files*. Similar to the menu item above, this menu item triggers units checking analysis on all source files in the project.
- *Type checking all files*. This menu item triggers real-world constraint checking analysis on all source files in the project.
- *Extract CM Type patterns*. This menu item collects the programming patterns in the whole project and attempts to discover programming patterns. At this stage, the pattern is limited to only one kind: two or more real-world types appear in the same function. If a pattern exists, program statements that violate such a pattern might be erroneous.
- *Setup real-world type system location*. The other menu items are used to display and setup the location of real-world type system for the current project. Real-world types and type rules are stored as files by the prototype. Setting the location of real-world type system is in fact choosing a folder in the local file system.



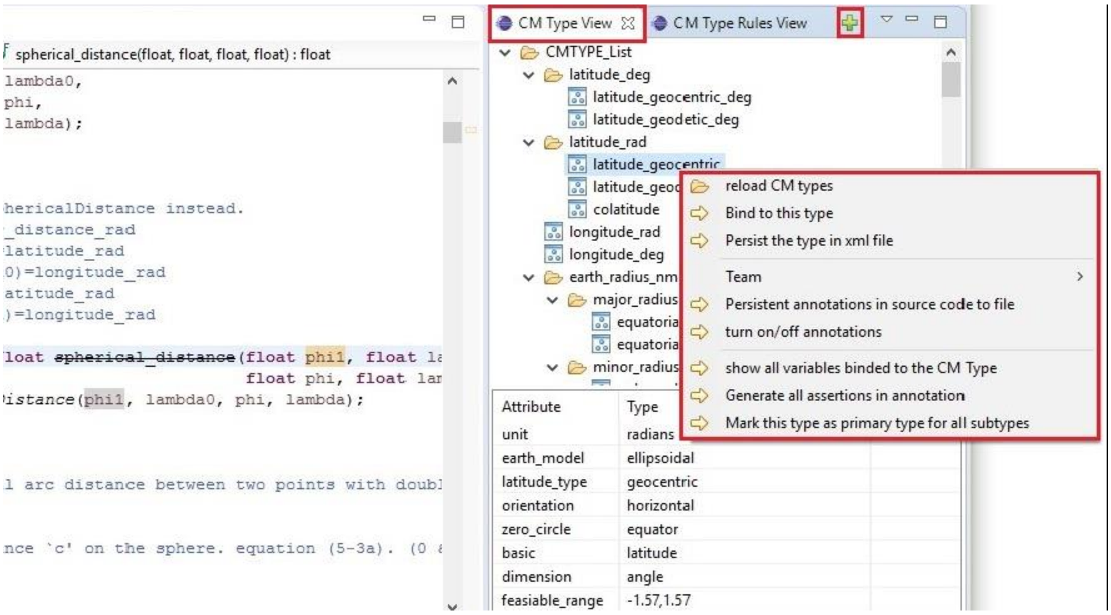
```
    /**
     * Selection flag for this road.
     */
    private boolean selected = false;

                   flag for this road.

                  boolean blinkState = false;

                  a road between two Intersections. The detail
                  n the intersections may be filled in later.

                  id a unique, integer identifier of this road
                  name a name for this road.
                  from the intersection from which this road
```

Project Explorer
Outline
au.com.kelpie.rcpplanner
geoAPI
geoTools
jts_topology_suite
NOMAD
openMap [cas
src
milStd2525
omcorba.ja
omj3d.jar
omsvg.jar
openmap.ja
jai_codec-1
JUnit 3
JRE System
lib
Road.java
Road
getTraverseHours() : float
New
Go Into
Show In Alt+Shift+W
Copy Ctrl+C
Copy Qualified Name
Paste Ctrl+V
Delete Delete
Remove from Context Ctrl+Alt+Shift+Down
Build Path
Refactor Alt+Shift+T
Import...
Export...
Refresh F5
Close Project
Close Unrelated Projects
Validate
Run As
Debug As
Team
Compare With
Replace With
Restore from Local History...
Real-World Type facilities
Configure
Type Inference based on rules in files
Type Inference from Return types to Method Signiture
Type Inference for Assignment
Type Inference from Parameters to Arguments
Extract itemset with methods contents involved
Annotation pattern miner
Type propagation the project
Extract terms as candidate real-world types
Interval analysis all files
Units checking all files
Type checking all files
Extract CM types patterns
Show Current Correspondence Model Location
Show the statistic of the project
Config Correspondence Model Location

Fig. 21. Popup menu: synthesis framework mechanisms

7.2.3 ECLIPSE VIEW: CM TYPE VIEW

CM Type View is a view extension to Eclipse created to manage real-world types. The view has a popup menu that supports a list of functionalities. The figure below shows the popup menu and its items.



```
lambda0,
phi,
lambda);


hericalDistance instead.
_distance_rad
latitude_rad
0)=longitude_rad
atitude_rad
)=longitude_rad

loat spherical_distance(float phi1, float la
                        float phi, float lar
istance(phi1, lambda0, phi, lambda);


l arc distance between two points with doubl


nce `c' on the sphere. equation (5-3a). (0 &
```

Attribute	Type
unit	radians
earth_model	ellipsoidal
latitude_type	geocentric
orientation	horizontal
zero_circle	equator
basic	latitude
dimension	angle
feasiable_range	-1.57,1.57

Fig. 22. View: CM type view

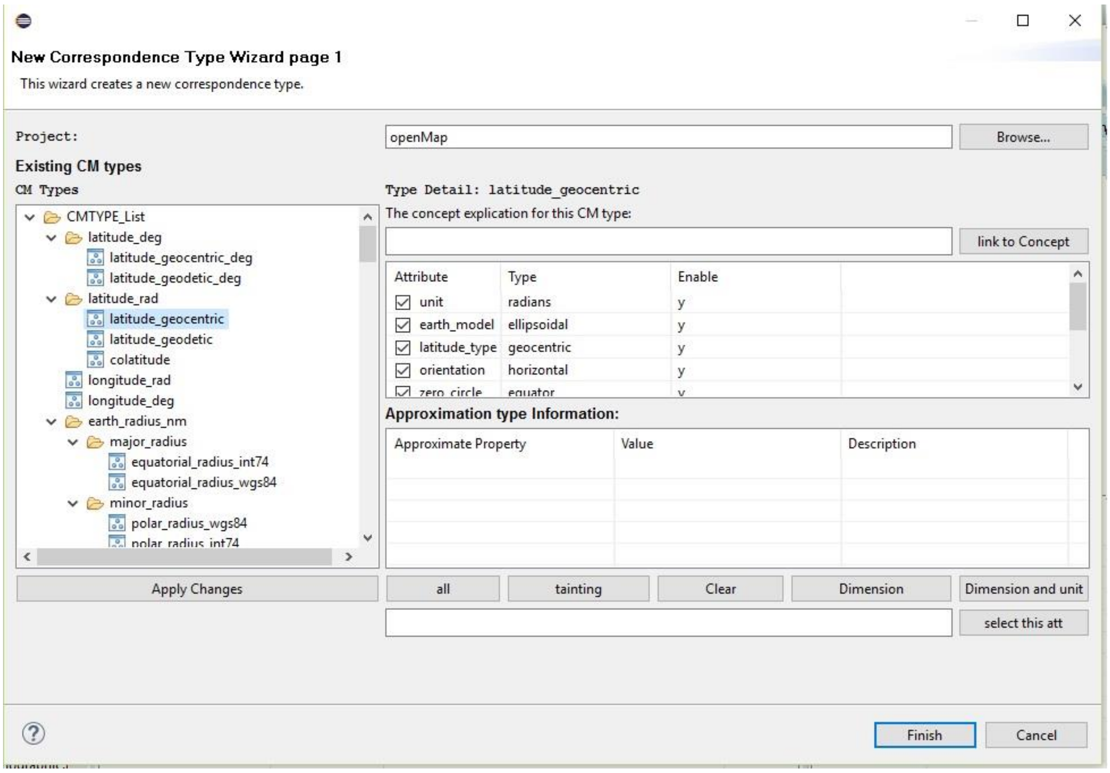
There are three rectangle boxes in the figure. One rectangle marks the name of CM Type View; one rectangle contains the popup menu, and one small rectangle triggers the wizard for managing real-world types. The major functionalities supported are:

- *Display real-world types.* All real-world types are displayed in this view. They are organized as a tree structure. When a real-world type in the tree is selected, its semantic attributes are displayed in the table below the tree.
- *Reload CM types*. Right click on a real-world type pops a menu. The first item in the menu is *reload CM types*. This menu item can refresh the tree of real-world types and reload the contents of real-world types.
- *Bind to this type*. This menu item allows users to bind the selected real-world type to the program element clicked in the Java editor. This is the main approach of

creating real-world type bindings. The bindings can be displayed as comments in JavaDoc.

- *Persist the type in XML file*. This menu item saves the clicked real-world type to an external XML file for easy read and transfer.
- *Persistent annotations in source files*. Sometimes, users may prefer that real-world type bindings are displayed in the comments which reside within the source code. In this way, users can easily manage real-world type bindings. This menu item allows users to save all bindings they created in comments.
- *Turn on/off annotations*. This menu item can change the display of real-world type bindings in source code, as introduced in the popup menu for Java editor.
- *Persist the type in XML file*. This menu item saves the clicked real-world type to an external XML file for easy read and transfer.
- *Show all variables bound to a real-world type*. This menu item shows all program elements that are bound to the clicked real-world type.

The small rectangle in the right top of Fig. 22 triggers the wizard of real-world types. Fig. 23 shows the wizard. Users can create, modify, organize and delete real-world types. In the left pane, real-world types for the current project are organized as a tree. In this right pane, real-world semantic attributes can be added, modified, and deleted. Also, a real-world type can be linked to its explication.



New Correspondence Type Wizard page 1

This wizard creates a new correspondence type.

Project: openMap

Existing CM types

CM Types

```
CMTYPE_List
  latitude_deg
    latitude_geocentric_deg
    latitude_geodetic_deg
  latitude_rad
    latitude_geocentric
    latitude_geodetic
    colatitude
  longitude_rad
  longitude_deg
  earth_radius_nm
    major_radius
      equatorial_radius_int74
      equatorial_radius_wgs84
    minor_radius
      polar_radius_wgs84
      polar_radius_int74
```

Type Detail: latitude_geocentric

The concept explication for this CM type:

Attribute	Type	Enable
☑ unit	radians	y
☑ earth_model	ellipsoidal	y
☑ latitude_type	geocentric	y
☑ orientation	horizontal	y
☑ zero_circle	equator	y

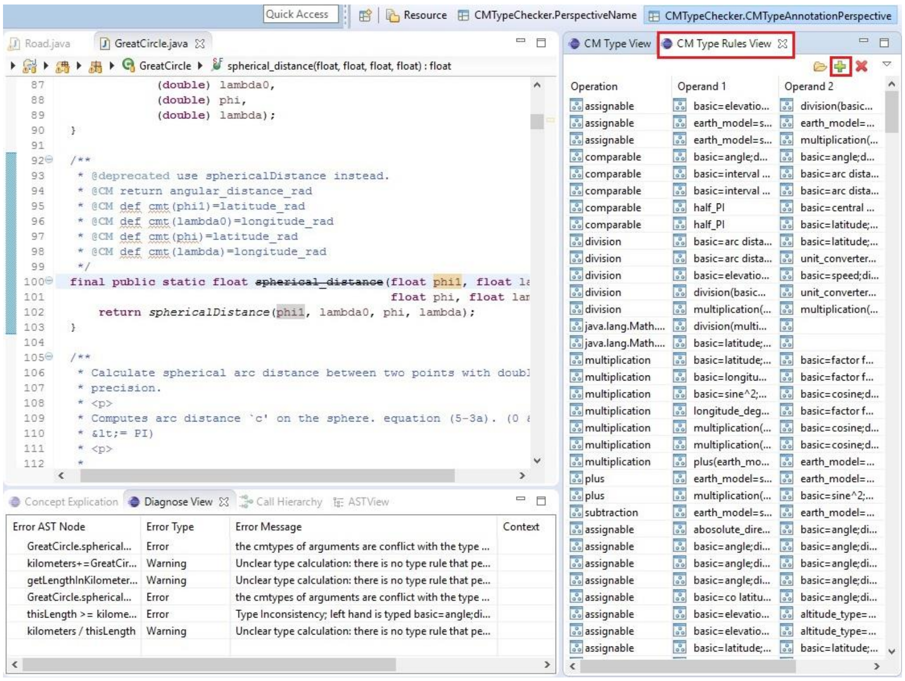
Approximation type Information:

Approximate Property	Value	Description

Fig. 23. Wizard: CM type wizard

7.2.4 ECLIPSE VIEW: CM TYPE RULES VIEW

CM Type Rule View is a view extension to Eclipse created to manage real-world type rules. Fig. 24 shows the view. There are two rectangles in the figure. One box marks the name of *CM Type Rule View*, and the other triggers the wizard of *CM Type Rules Wizard*. The view is in a table format that displays all the type rules defined for the current project. Fig. 25 shows a screenshot for the wizard. Real-world types are organized as a tree in the left pane. In the right pane, users can select an operator from a list of operators; then choose operands and return type from the tree of real-world types in the left.

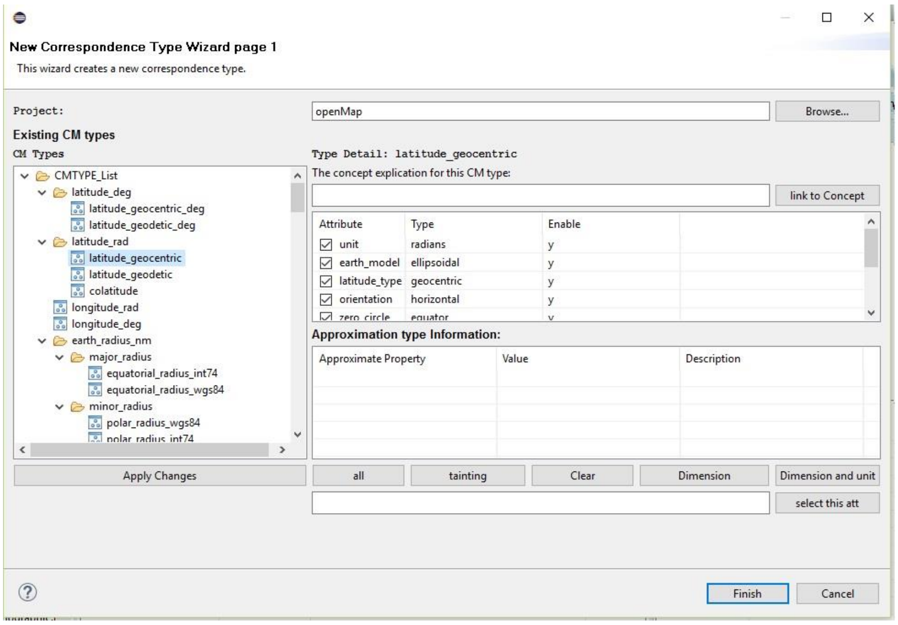


```java
                (double) lambda0,
                (double) phi,
                (double) lambda);
    }

    /**
     * @deprecated use sphericalDistance instead.
     * @CM return angular_distance_rad
     * @CM def cmt(phi1)=latitude_rad
     * @CM def cmt(lambda0)=longitude_rad
     * @CM def cmt(phi)=latitude_rad
     * @CM def cmt(lambda)=longitude_rad
     */
    final public static float ~~spherical_distance~~(float phi1, float la
                                              float phi, float lar
        return sphericalDistance(phi1, lambda0, phi, lambda);
    }

    /**
     * Calculate spherical arc distance between two points with doubl
     * precision.
     * <p>
     * Computes arc distance `c' on the sphere. equation (5-3a). (0 &
     * &lt;= PI)
     * <p>
     *
```

Error AST Node	Error Type	Error Message	Context
GreatCircle.spherical...	Error	the cmtypes of arguments are conflict with the type ...	
kilometers+=GreatCir...	Warning	Unclear type calculation: there is no type rule that pe...	
getLengthInKilometer...	Warning	Unclear type calculation: there is no type rule that pe...	
GreatCircle.spherical...	Error	the cmtypes of arguments are conflict with the type ...	
thisLength >= kilome...	Error	Type Inconsistency; left hand is typed basic=angle;di...	
kilometers / thisLength	Warning	Unclear type calculation: there is no type rule that pe...	

Operation	Operand 1	Operand 2
assignable	basic=elevatio...	division(basic...
assignable	earth_model=s...	earth_model=...
assignable	earth_model=s...	multiplication(...
comparable	basic=angle;d...	basic=angle;d...
comparable	basic=interval ...	basic=arc dista...
comparable	basic=interval ...	basic=arc dista...
comparable	half_PI	basic=central ...
comparable	half_PI	basic=latitude;...
division	basic=arc dista...	basic=latitude;...
division	basic=arc dista...	unit_converter...
division	basic=elevatio...	basic=speed;di...
division	division(basic...	unit_converter...
division	multiplication(...	multiplication(...
java.lang.Math....	division(multi...	
java.lang.Math....	basic=latitude;...	
multiplication	basic=latitude;...	basic=factor f...
multiplication	basic=longitu...	basic=factor f...
multiplication	basic=sine^2;...	basic=cosine;d...
multiplication	longitude_deg...	basic=factor f...
multiplication	multiplication(...	basic=cosine;d...
multiplication	multiplication(...	basic=cosine;d...
multiplication	plus(earth_mo...	earth_model=...
plus	earth_model=s...	earth_model=...
plus	multiplication(...	basic=sine^2;...
subtraction	earth_model=s...	earth_model=...
assignable	abosolute_dire...	basic=angle;di...
assignable	basic=angle;di...	basic=angle;di...
assignable	basic=angle;di...	basic=angle;di...
assignable	basic=angle;di...	basic=angle;di...
assignable	basic=co latitu...	basic=angle;di...
assignable	basic=elevatio...	altitude_type=...
assignable	basic=elevatio...	altitude_type=...
assignable	basic=latitude;...	basic=latitude;...

Fig. 24. View: CM type rule view



New Correspondence Type Wizard page 1

This wizard creates a new correspondence type.

Project: openMap

Existing CM types

CM Types

- CMTYPE_List
  - latitude_deg
    - latitude_geocentric_deg
    - latitude_geodetic_deg
  - latitude_rad
    - latitude_geocentric
    - latitude_geodetic
    - colatitude
  - longitude_rad
  - longitude_deg
  - earth_radius_nm
    - major_radius
      - equatorial_radius_int74
      - equatorial_radius_wgs84
    - minor_radius
      - polar_radius_wgs84
      - polar_radius_int74

Type Detail: latitude_geocentric

The concept explication for this CM type:

Attribute	Type	Enable
☑ unit	radians	y
☑ earth_model	ellipsoidal	y
☑ latitude_type	geocentric	y
☑ orientation	horizontal	y
☑ zero_circle	equator	y

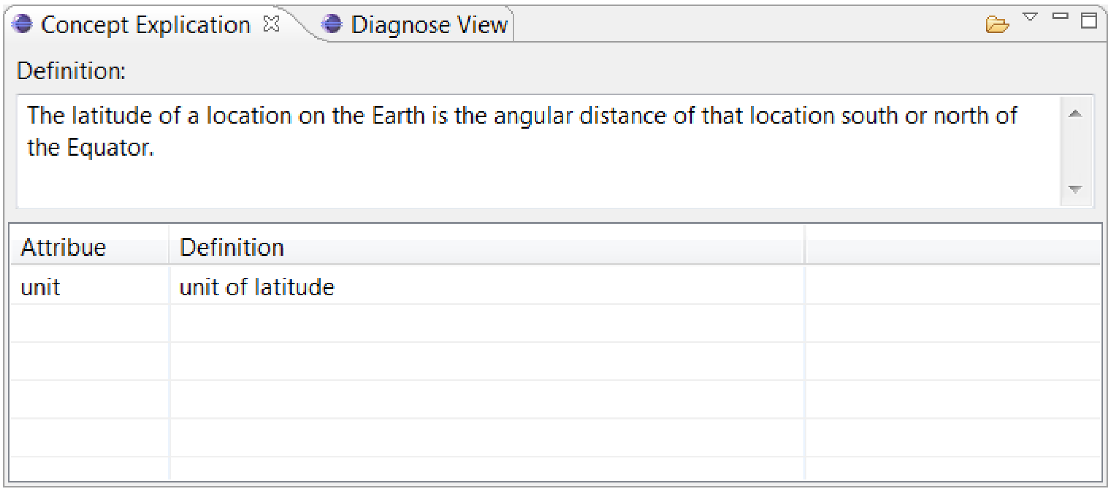
Approximation type Information:

Approximate Property	Value	Description

Fig. 25. Wizard: CM type rules wizard

7.2.5 Concept Explication View

Concept Explication View presents explications for a real-world concept. When a real-world type is selected, explications associated with the type are displayed here.



Definition:

The latitude of a location on the Earth is the angular distance of that location south or north of the Equator.

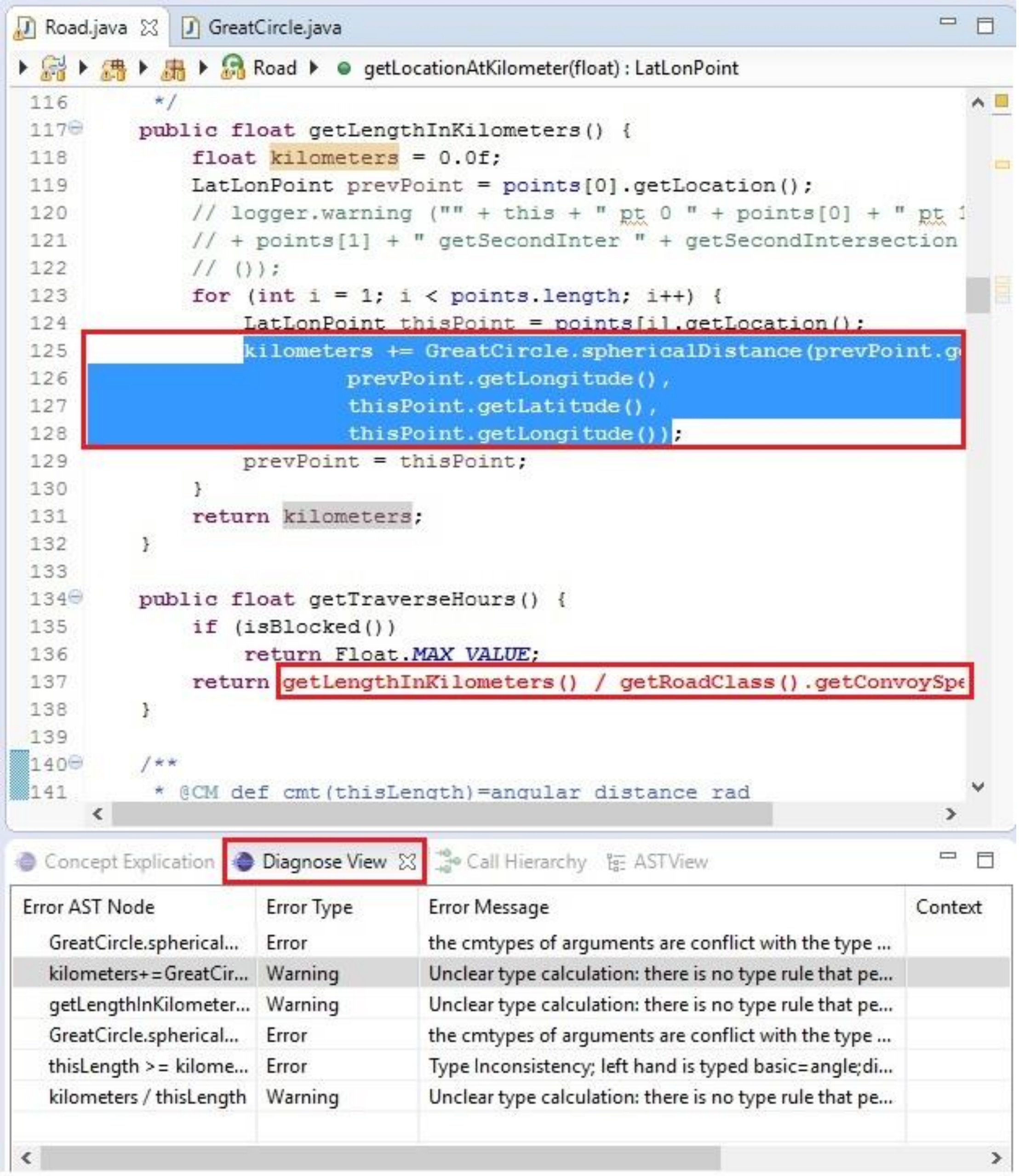
Attribue	Definition	
unit	unit of latitude	

Fig. 26. View: concept explication

7.2.6 Diagnose View

Diagnose view displays the error messages generated by analysis techniques. Fig. 27 shows a snapshot of the diagnose view. The rectangle in the bottom part marks the *diagnose view*. The first column, *Error AST Node*, shows expressions that cause the errors. The second column of *Error type* defines the type of diagnosis, whether it is a warning or error. The third column describes the contents of the error in detail. And the column of *Permission* allows users to suppress this error.

The diagnose view can be used to trace sources of errors. Two rectangle boxes in the figure show the traces. When an error is pressed in the diagnose view, the statement that causes the error is covered with shadow, as indicated by the top red box. All errors found by the analysis are highlighted with color, as indicated by the middle rectangle box.



```
     */
    public float getLengthInKilometers() {
        float kilometers = 0.0f;
        LatLonPoint prevPoint = points[0].getLocation();
        // logger.warning ("" + this + " pt 0 " + points[0] + " pt 1
        // + points[1] + " getSecondInter " + getSecondIntersection
        // ());
        for (int i = 1; i < points.length; i++) {
            LatLonPoint thisPoint = points[i].getLocation();
            kilometers += GreatCircle.sphericalDistance(prevPoint.g
                    prevPoint.getLongitude(),
                    thisPoint.getLatitude(),
                    thisPoint.getLongitude());
            prevPoint = thisPoint;
        }
        return kilometers;
    }

    public float getTraverseHours() {
        if (isBlocked())
            return Float.MAX_VALUE;
        return getLengthInKilometers() / getRoadClass().getConvoySpe
    }

    /**
     * @CM def cmt(thisLength)=angular distance rad
```

Error AST Node	Error Type	Error Message	Context
GreatCircle.spherical...	Error	the cmtypes of arguments are conflict with the type ...	
kilometers+=GreatCir...	Warning	Unclear type calculation: there is no type rule that pe...	
getLengthInKilometer...	Warning	Unclear type calculation: there is no type rule that pe...	
GreatCircle.spherical...	Error	the cmtypes of arguments are conflict with the type ...	
thisLength >= kilome...	Error	Type Inconsistency; left hand is typed basic=angle;di...	
kilometers / thisLength	Warning	Unclear type calculation: there is no type rule that pe...	

Fig. 27. View: diagnose view

CHAPTER 8

8. EVALUATION OVERVIEW

The interpreted formalism and real-world type system concepts were evaluated by conducting case studies on two open-source software projects [86]. This chapter first introduces the setup of the evaluation; and then summarizes the purposes and processes of the case studies. The details of the case studies are presented in Chapter 9, 10, 11, and 12.

8.1 INTRODUCTION

Comprehensive sets of statistical experiments are the best way to evaluate a complex concept like the interpreted formalism. However, it is not feasible to conduct comprehensive sets of experiments and statistically analyze the results with the resources available for this research. Instead, pilot case studies were conducted to get initial observations about the utility and performance of the interpreted formalism concept and real-world type systems. These initial observations can be used to motivate and inform more extensive experiments.

The evaluation is organized as two major parts. The two parts serve different purposes:

- *Evaluating properties*. The first part of the evaluation aims at assessing several properties of the interpreted formalism. In this part, complete real-world type systems were created for the two software projects Kelpie flight planner and OpenMap. Various elements of the projects were bound with real-world types, a set of type rules were defined, and analysis techniques were performed. The data collected in this part were used to evaluate various properties of the interpreted formalism. The case studies of the two projects are presented in Chapter 9 and 10.
- *Demonstrating pragmatics*. The main purpose of the second part was to demonstrate the pragmatics of the interpreted formalism. Applying and using the interpreted formalism might not be an easy task for new users. Chapter 11 and 12 demonstrate the pragmatic mechanisms of interpreted formalisms with two case studies. Chapter 11 demonstrates the detailed process of applying the interpreted formalism on a software project; and Chapter 12 illustrates the synthesis framework that was implemented to facilitate developing interpreted formalisms.

The remainder of this section first describes the two software projects, and then focuses on the setup of the first part of evaluation. The properties being evaluated are introduced. In addition, the processes of evaluating the properties are presented.

8.2 THE CASE STUDY SUBJECTS

The interpreted formalism has been evaluated on two open-source software projects. Both projects are from the *geography* domain. The two projects are:

- The Kelpie flight planner [43]. This is an open-source Java project based on FlightGear [25]. The Planner project 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 moderate-sized project is 13,884 lines long in total.
- OpenMap [58]. OpenMap is a JavaBean-based toolkit for building applications and applets needing geographic information. Using OpenMap components, users can access data from legacy applications. The core components of OpenMap are a set of Swing components that understand geographic coordinates. These components allow users to show map data and manipulate that data. The large-sized project has 157,858 lines of code.

8.3 EVALUATED PROPERTIES

The main research question that needs to be answered in the first part of the evaluation is:

Is interpreted formalism feasible and effective?

Specifically, four questions need to be answered:

- Is it *feasible* to apply the interpreted formalism in modern software projects of different sizes?
- How effective are the *analysis techniques* in modern software projects?
- How much *effort* is required to apply the interpreted formalism on software projects of different sizes?
- Is interpreted formalism *scalable*? Does the size of real-world type system increase linearly when the size of software projects increases?

In order to answer these questions, the pilot case studies were conducted to assess *feasibility*, *effort level*, *error detection capability*, and *scalability*. Two major case studies were conducted on the two open-source software projects. Data items pertinent to these properties were recorded. Analyzing these data helps assessing the performance of the real-world type system, thereby making reasonable predictions about the overall performance in more comprehensive experiments.

This section presents these properties, and then introduces the approaches by which these properties were evaluated.

8.3.1 FEASIBILITY

The first purpose of case studies was to determine if applying the interpreted formalism is feasible in modern software projects.

APPROACH

In order to assess feasibility, complete real-world type systems were created for the Kelpie flight planner project and the OpenMap project. Real-world types were created for all real-world entities accessed by the software applications, and variables and methods that access real-world entities were bound to their real-world types. A set of type rules were defined so that relevant relationships between real-world entities could be established. After setting up the real-world type system, analyses were conducted on the two software projects. Real-world constraint checking was used to detect violations of real-world constraints. Reasonable range analysis was conducted on the projects to detect error-prone computations.

ASSESSMENT

Feasibility was assessed by answering a list of questions:

- Is it fit to apply the interpreted formalism to a moderate-sized software system?
    - Can different real-world types and type rules be defined and used?
    - Can different program elements be bound with real-world types?
    - Can the interpreted formalism be applied to different source files in the software?
    - Can the interpreted formalism be applied to all source files that access real-world entities?
    - Can analysis techniques be applied to the software?

- Is it fit to apply the interpreted formalism to a large-sized software system?
    - Can the interpreted formalism be applied to all of the files in the system?
    - Can all of relevant variables be interpreted?
    - Can analysis techniques be used in the software system?

DATA COLLECTED

In order to answer these questions, the case studies collected the following items:

- Size of the software:
    - Number of variables.
    - Number of source code files.
    - Number of packages.
- Size of real-world type systems:
    - Number of real-world types defined.
    - Number of real-world type rules defined.
    - Number of real-world type bindings.
- Coverage of interpreted variables:
    - Number of variables requiring interpretations.
- Numbers relevant to error checking:
    - Number of source files that were checked.

  - Number of source files with error reported.

These data were used to determine the feasibility of real-world type systems and discover possible difficulties in the use of real-world type systems.

8.3.2 ERROR DETECTION CAPABILITY

This part of the evaluation was conducted to gain insights into the interpreted formalism's capability of detecting real errors, and how effective it is.

APPROACH

After setting up interpreted formalisms, analysis techniques were conducted on the Kelpie flight planner and OpenMap projects. Real-world constraint checking and reasonable range analysis were the primary analysis techniques.

ASSESSMENT

The error checking capability was assessed by answering a list of questions:

- Are the error checking techniques useful?
  - Can they detect real errors?
  - Can they detect more than one kind of error?
  - Can they detect errors from different source files?
- Are the error checking techniques effective?
  - How many errors were reported?

    - How many real errors were reported?
    - What was the ratio of real errors to error reported?
- Are the error checking techniques versatile?
    - Can they detect real errors in a moderate-sized project?
    - Can they detect real errors in a large-sized project?

Data Collected

A list of data were collected to assess the capability of error checking, including:

- Numbers of errors reported:
    - Number of errors reported in total.
    - Number of source files with error reported.
- Numbers of real errors:
    - Number of real errors in total.
    - Number of source files with real errors.
- Numbers of false warnings:
    - Number of false warnings in total.
    - Number of source files with false warnings.
    - Number of source files with only false warnings.

These data was used to determine the error detection capability of real-world type systems, and discover possible difficulties in the use of real-world type systems.

8.3.3 EFFORT LEVEL

In Chapter 6, a synthesis framework was introduced to alleviate the burden required from engineers. This part of the evaluation was conducted to measure the effort required from engineers when they receive help from the synthesis framework.

APPROACH

In the process of setting up the interpreted formalism for the Kelpie flight planner and OpenMap, data relevant to the effort level required were collected.

ASSESSMENT

The effect level is assessed by answering a list of questions:

- Does developing real-world type system require excessive effort from the users?
    - Can real-world type rules be reused?
    - Can real-world type specifications be reused?
- Can synthesizers provide support for reducing users' effort?
    - Can synthesizers provide engineers with candidate real-world types?
    - Can synthesizers provide engineers with candidate real-world type rules?
    - Can synthesizers provide engineers with candidate real-world type bindings?

- Are the synthesizers effective in reducing users' effort?
    - What's the percentage of real-world type bindings can be provided by the synthesizers?
    - Can synthesizers create a useful set of real-world type bindings?

DATA COLLECTED

For a software system, a list of data were collected to assess the effort, including:

- Size of the real-world type system:
    - Number of real-world types in total.
    - Number of semantic attributes for real-world types in total.
    - Number of real-world type rules in total.
    - Number of real-world type bindings in total.
- Size of the real-world type system created by engineers:
    - Number of real-world types created by engineers.
    - Number of semantic attributes created by engineers for real-world types.
    - Number of real-world type rules created by engineers.
    - Number of real-world type bindings created by engineers.
- Size of the real-world type system provided by reusing artifacts:
    - Number of real-world types provided by reusing existing real-world type libraries.

  - Number of real-world type rules provided by reusing existing real-world type libraries.

8.3.4 SCALABILITY

This part of the evaluation was conducted to assess how real-world type systems scale with larger software systems.

APPROACH

OpenMap is about 11 times the size of the Kelpie flight planner. In the process of setting up the interpreted formalism for the Kelpie flight planner and OpenMap, the data for the two projects were compared in different aspects. The results of the comparison show the potential of using real-world type systems in larger systems.

ASSESSMENT

The scalability was assessed by answering three questions:

- Does the size of real-world type system scale linearly?
- Does effort level required from user scales linearly?
- Do error checking techniques perform similarly in software of different sizes?

DATA COLLECTED

For both software systems, a set of data were compared:

- Size of software:
    - Number of files.
    - Number of packages.
    - Lines of code.
- Size of the real-world type system:
    - Number of real-world types.
    - Number of real-world type rules.
    - Number of real-world type bindings.
- Effort required from engineers:
    - Number of real-world types created by engineers.
    - Number of real-world type rules created by engineers.
    - Number of real-world type bindings created by engineers.
- Error checking capability:
    - Number of errors reported.
    - Number of real errors found.

CHAPTER 9

9. CASE STUDY: KELPIE FLIGHT PLANNER

This chapter presents the case study on the Kelpie flight planner. The case study aims at evaluating the performance of interpreted formalism in moderate-sized software by applying the real-world type system on the software. Results of the case study show that the interpreted formalism is feasible in moderate-sized software, and the error checking techniques are effective in detecting real errors.

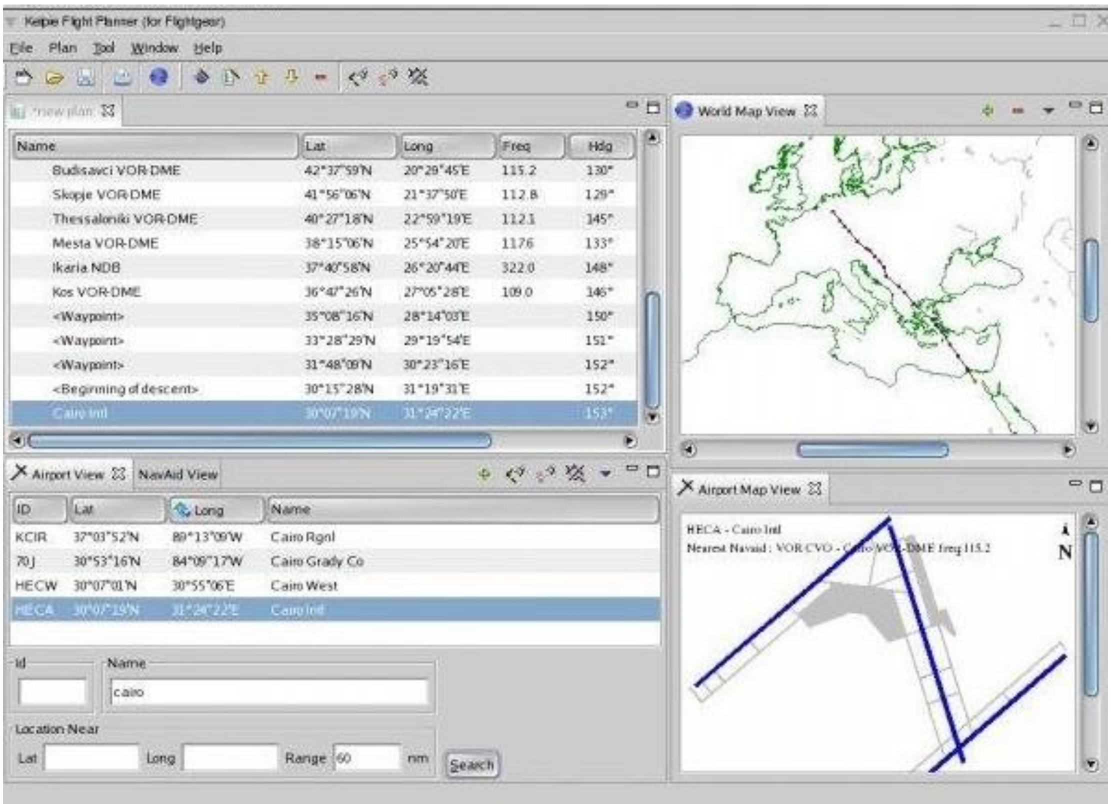
The planner software is briefly introduced and then the purposes and process of the case study are described. Finally, the results of the error checking are presented.

9.1 SYSTEM OF CASE STUDY

9.1.1 BASIC INFORMATION

The Kelpie flight planner is an open-source Java software system based on FlightGear [25]. The software project is hosted at the website of sourceforge.net [43].

The software is highly reviewed by the users, and is being actively downloaded for around 50 times per week. The figure below shows a screenshot of the software.



Name	Lat	Long	Freq	Hdg
Budisavci VOR-DME	42°37'59"N	20°29'45"E	115.2	130°
Skopje VOR-DME	41°56'06"N	21°37'50"E	112.8	129°
Thessaloniki VOR-DME	40°27'18"N	22°59'19"E	112.1	145°
Mesta VOR-DME	38°15'06"N	25°54'20"E	117.6	133°
Ikaria NDB	37°40'58"N	26°20'44"E	322.0	148°
Kos VOR-DME	36°47'26"N	27°05'28"E	109.0	146°
<Waypoint>	35°08'16"N	28°14'03"E		150°
<Waypoint>	33°28'29"N	29°19'54"E		151°
<Waypoint>	31°48'09"N	30°23'16"E		152°
<Beginning of descent>	30°15'28"N	31°19'31"E		152°
Cairo Intl	30°07'19"N	31°24'22"E		153°

ID	Lat	Long	Name
KCIR	37°03'52"N	89°13'09"W	Cairo Rgnl
70J	30°53'16"N	84°09'17"W	Cairo Grady Co
HECW	30°07'01"N	30°55'06"E	Cairo West
HECA	30°07'19"N	31°24'22"E	Cairo Intl

Id:

Name: cairo

Location Near

Lat: Long: Range: 60 nm

Search

HECA - Cairo Intl
Nearest Navaid : VOR CVO - Cairo VOR-DME freq 115.2

Fig. 28. Screenshot of Kelpie flight planner

9.1.2 IMPORTANT REAL-WORLD SEMANTICS

Some real-world semantic attributes are important in understanding this software. These attributes are introduced first.

DIMENSIONAL AND UNITS ATTRIBUTES

The Kelpie flight planner software makes calculations involving distances, velocities, speeds, accelerations, angles, time and so on, and it does so using a variety of units. Clearly, the software is of the type for which dimensional and units analysis has the potential to discover faults.

The dimensions and units are all real-world concepts that are defined in the real-world type system by default, and the type rules follow immediately from elementary physics.

Velocity Surface Attribute

A critical element of the data used by the Kelpie flight planner in modeling aircraft movement is a two-element vector consisting of the horizontal velocity (motion across the Earth's surface) and the vertical velocity (climb or sink rate) of the aircraft. The vector as the *velocity surface*. The details of the velocity surface are shown in Fig. 29.

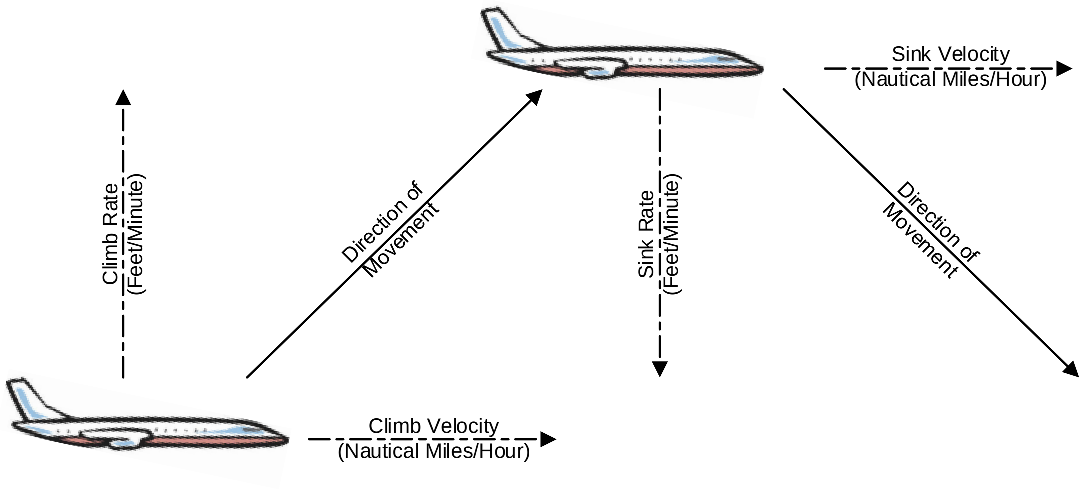


Fig. 29. The velocity surface

EARTH MODEL ATTRIBUTE

The Earth is neither a sphere nor any other simple shape. In order to undertake useful calculations, programs like the Kelpie flight planner have to operate with a model of the Earth's geometry.

Various models of the Earth's geometry have been created, including a sphere and an ellipsoid. For the ellipsoid, different models have been developed for special purposes. For example, the International Geomagnetic Reference Field IGRF [35] and the World Magnetic Model WMM models [79] are popular for calculations involving the Earth's magnetic field.

The model or models in use for a particular piece of software have to be used carefully so that the "right" model is used in any given calculation. Misplacement of variables and values in different models could introduce mistakes.

9.2 DATA COLLECTED AND ASSESSMENT

The data that need to be collected for the evaluation are outlined in Chapter 8. This section presents and analyzes the data collected in this case study.

9.2.1 FEASIBILITY

DATA COLLECTED

The data collected in this case study were:

- Size of the software:

  As introduced before, the Kelpie flight planner software is 13,884 lines long, is organized as 10 packages, and is contained in 126 source files. The total number of identifiers in the software is 28,754.

- Size of real-world type systems:

  In this case study, the real-world type system for the project contains 35 real-world types, 97 real-world type rules, and 255 real-world typing bindings.

- Coverage of interpreted variables:

  Variables in 32 source files have been interpreted with real-world types. Real-world types have been accessed by program elements inside 50 source files. The other source files do not interact with real-world entities. They do not have real-world type bindings.

- Numbers relevant to error checking:

  With real-world type system deployed, the real-world constraint checking and reasonable range analysis were conducted to detect errors. In the case study, all 126 source files were checked for errors. Errors have been reported in 4 files.

ASSESSMENT

- Can different real-world types and type rules be defined and used?

Yes. The real-world type system for the software contains 35 different real-world types and 97 different real-world type rules.

- Can different program elements be bound with real-world types?

  The real-world type system contains 255 real-world type bindings in total. The bindings are associated with different kinds of program elements, including local variables, global variables, function signatures, arrays, and class instances.

- Can the interpreted formalism be applied to different source files?

  Interpreted formalism has been applied to all the files in the Kelpie flight planner. The software has 126 source files, 32 files contain real-world type bindings, and 50 files access real-world types through bindings or calling external functions. The other 76 files do not interact with real-world entities.

- Can the interpreted formalism be applied to all source files that access real-world entities?

  Among the 126 source files, 50 files access real-world entities through variables and function calls. The interpreted formalism has been applied to these files.

- Can analysis techniques be applied to the software?

  Real-world type checking and reasonableness range analysis have been used in error checking. All 126 source files have been checked. Real errors were reported. Detailed results of error checking are presented in the next section.

9.2.2 ERROR DETECTION CAPABILITY

The analyses performed on the Kelpie flight planner software showed promising results. Both real-world constraint checking and reasonable range analysis found real errors that have not been reported before. Real-world constraint checking found 6 real errors, and range analysis found 12 statements with error-prone computations.

DATA COLLECTED

In total, real-world constraint checking reported 12 errors. For real-world constraint checking, 6 of the 12 reported errors are real. Real errors were found in 4 source files. Reasonable range analysis found 12 statements with error-prone computations. For reasonable range analysis, all 12 statements could in fact produce outbound values of the feasible ranges, so all of them are worth notice.

Six reported errors were considered as false warnings. They come from the same source file. The reason is that one array is holding values of different real-world types.

In addition to errors, analysis revealed several locations in the software that were not errors but which I classify as “improper usage”. The term of *improper usage* means either (a) a variable took on different real-world entities (but the same programming datatype) in different parts of the program, or (b) the elements of an array were not all of the same real-world entities (but were of the same programming type). This usage of variables is probably not a good practice.

ANALYSIS OF REAL ERRORS

A summary of errors found by real-world constraint checking is shown in the table below.

Table. 4 . Real errors found by real-world constraint checking

Program File	#Faults	Semantic attribute involved
PlanHelper.java	2	velocity surface
PlanHelper.java	3	units
FGPlanWriter.java	1	units
Plan.java	1	Earth model
PlanEditor.java	1	Earth model

These errors come from six statements residing in 4 different source files. We list these files and the details of the errors below.

- Plan.java:

```
heading -= Magfield.getMag(loc.getLat(), loc.getLong());
```

The first parameter of the function `getMag()` expects a geodetic latitude when the shape of Earth is modeled as an ellipsoid. However, the argument loc.getLat() is a latitude when the Earth's shape is modeled as a sphere.

- PlanEditor.java:

```
bearing -= Magfield.getMag(locator.getNavaid().getLat(),
             locator.getNavaid().getLong());
```

The error found in this statement is caused by the same reason as the error we stated above. The argument and parameter of the function `getMag()`refer to different shapes of the Earth.

- FGPlanWriter.java

The file contains a statement below:

```
double roughLonSep = range / ( 60 * Math.cos(point.getLatitude()));
```

The function `point.getLatitude()` returns a latitude value with unit of `degree` when unit of `radians` is needed.

- PlanHelper.java

The file contains three erroneous statements. The first statement is:

```
alt -= legTime * plan.getAircraft().getSinkSpeed()/60;
```

The expression references the wrong data. `getSinkSpeed`() returns a quantity measured horizontally and `alt` is measured vertically.

The second statement is:

```
alt += legTime * plan.getAircraft().getClimbRate()/60;
```

`plan.getAircraft().getClimbRate()` returns the climb rate in `feet/minute`, the variable legTime is time in `hours`, and `alt` is altitude in `feet`. The conversion factor is 60, but the conversion requires multiplication by 60, not division.

The third statement is:

```
alt -= legTime * plan.getAircraft().getSinkSpeed()/60;
```

The expression references the wrong data. As in the first statement, `getSinkSpeed`() returns a quantity measured horizontally and alt is measured vertically. Correcting this fault yields code with the same units issue as arose in the second fault requiring an additional fix.

The reasonable range analysis detected 12 error-prone statements. The table below summarizes the error-prone statements found in this case study.

Table. 5 . Possible errors found by reasonable range analysis

Program File	# of Warnings	Possible errors
FGplanwriter.java	1	Division of zero
Mercator.java	2	Infinite bounds
Taxiway.java	4	Division of zero
AutoPlanner.java	1	Division of zero
PlanHelper.java	3	Out of reasonable range
Coodinate.java	1	Division of zero

The table shows the possible errors that can be caused by the detected statements. There are 2 common sources:

- Possible outbound of reasonable range. All values of real-world entities have their reasonable ranges. If a statement involves values that go beyond real-world reasonable ranges, a warning is issued.
- Infinity or Not A Number error (NAN). If a statement produces values with a bound of infinity or NAN, a warning is issued. A frequent example is a division of zero.

The details of the errors are described below.

- PlanHelper.java

  The source file contains three statements that could lead to values beyond their reasonable ranges. The first statement is:

  ```
  timeToBOD = altToBOD / plan.getAircraft().getSinkSpeed() / 60;
  ```

  The right side invokes the wrong function `getSinkSpeed()`. The correct function should be `getSinkRate()`. The two function calls return values whose ranges are more than ten times different. As a result, the range computed for the right side is largely different from the reasonable range of `timeToBOD`.

  The other two statements in this file have the similar situation. One of them also refers the wrong function `getSinkSpeed()`. And the other statement converts a value of time measured in hours to time measured in minutes by division instead of multiplication.

- Coordinate.java

  A warning was issued by range analysis on the following statement:

  ```
  double heading = Math.acos(
        (Math.sin(lat2)-Math.sin(lat1)*Math.cos(d))
                    / (Math.sin(d) * Math.cos(lat1)));
  ```

  This is caused by the range of argument for `Math.acos()`. The range computed for the long expression is `[-∞,+∞]` due to the fact that the range of the divisor is `[0,1]`.

- FGplanwriter.java

  The file contains a statement:

```
double roughLonSep = range / ( 60 * Math.cos(point.getLatitude()));
```

The `point.getLatitude()` returns latitude values in degree, ranging from -180 to 180 degrees. Therefore, the right side of the statement could lead to a division of zero. Such kind of statements needs to be carefully handled with assertions or if statements.

- Taxiway.java

  Four statements were found:

```
ALong = longitude + (feetEastL - feetEastW) /
          (FEET_PER_DEGREE *   Math.cos(Math.toRadians(latitude)));
BLong = longitude + (feetEastL + feetEastW) /
          (FEET_PER_DEGREE * Math.cos(Math.toRadians(latitude)));
CLong = longitude + (-feetEastL + feetEastW) /
          (FEET_PER_DEGREE * Math.cos(Math.toRadians(latitude)));
DLong = longitude + (-feetEastL - feetEastW) /
          (FEET_PER_DEGREE * Math.cos(Math.toRadians(latitude)));
```

  All these four statements could possibly lead to division of zero since the value of `latitude` could be 90 degree.

- Autoplanner.java

  One erroneous statement was found:

```
double roughLonSep = range /
       ( nm_deg * Math.cos(Math.toRadians(point.getLatitude())));
```

  And one was found in `Coordinate.java`:

```
heading = Math.acos((Math.sin(lat2) - Math.sin(lat1) * Math.cos(d))
                              / (Math.sin(d) * Math.cos(lat1)));
```

- Mercator.java

  In addition to division of zero, some statements lead to infinite bounds in other ways. For example, this source file contains two statements:

```
return rad2Deg * Math.log(Math.tan(pi4 + deg2Rad * 0.5 * lat));
return -rad2Deg * Math.log(Math.tan(pi4 - deg2Rad * 0.5 * lat));
```

  The arguments of `Math.tan()`would be $\pi/2$ when value of `lat` equals $\pi/2$, which results in an infinite upper bound.

ASSESSMENT

The error checking capability is evaluated by answering the questions posed in section 8.3.2:

- Can the error checking techniques detect real errors?

  In total, 6 real errors have been detected by real-world constraint checking, and 12 error-prone statements have been found by reasonable range analysis.

- Can the error checking techniques detect more than one kind of error?

  Real-world constraint checking detected real errors that violate constraints on different real-world semantics, e.g., Earth model and units. Reasonable range analysis found statements that could lead to runtime errors.

- Can the error checking techniques detect errors from different source files?

Real-world constraint checking detected errors in 4 different source files, and reasonable range analysis issued reports in 6 different source files.

- Is the error checking effective?

  The real-world constraint checking reported 12 faults, and 50% of them are real errors. Reasonable range analysis reported 12 statements, and all of them possibly lead to runtime errors. In total, the ratio of the number of real errors to the number of reported errors is 18/24 = 75%. Such ratio suggests that the error checking techniques are very effective. The reported errors and warnings deserve engineers' attention.

9.2.3 EFFORT LEVEL

The effort required from users include effort for creating real-world types, real-world constraints, and real-world type bindings. The effort required from users is largely reduced by employing the synthesis framework and reusing existing artifacts. The evaluation of effort level is part of the case study of synthesis framework. The details are presented in Chapter 12.

CHAPTER 10

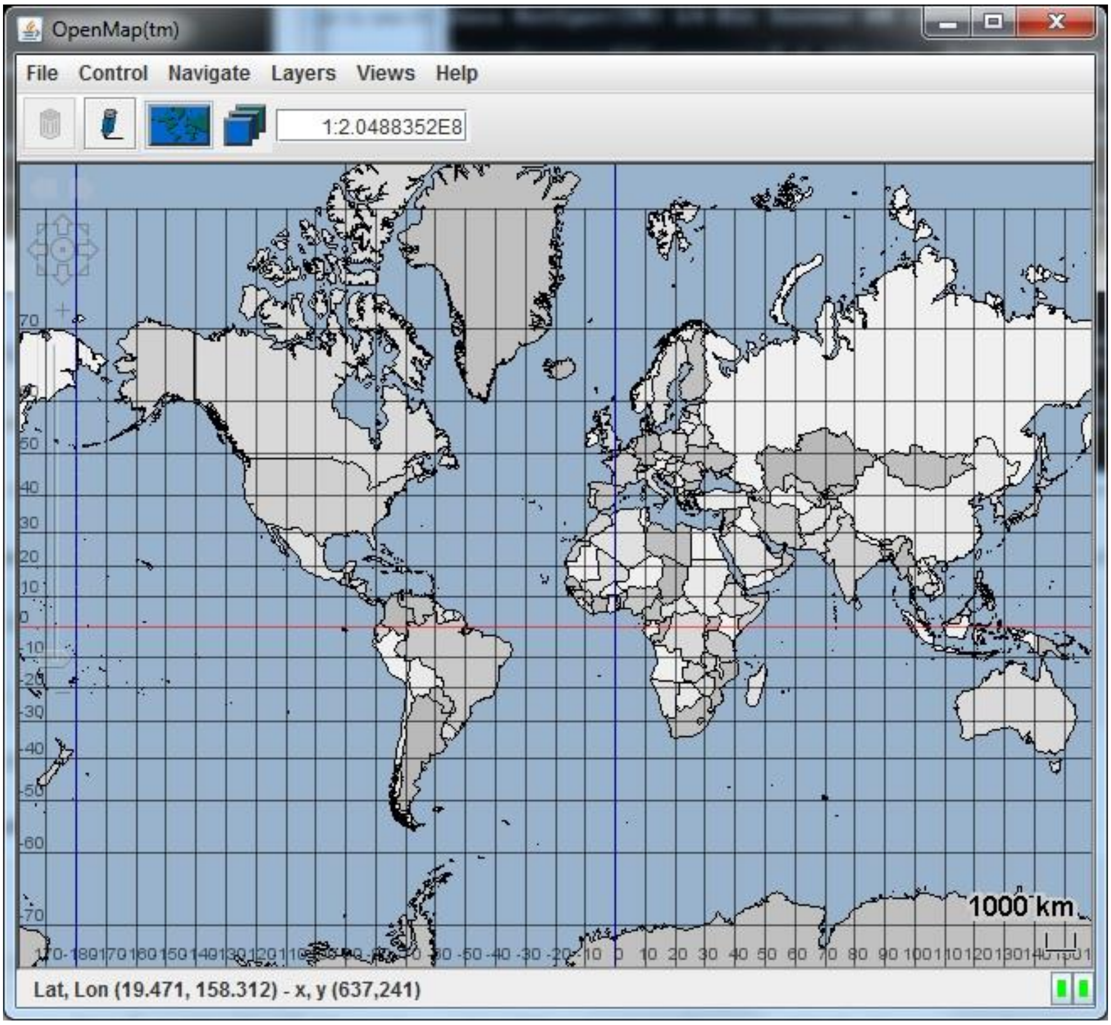
10. CASE STUDY: OPENMAP

This chapter presents the case study of OpenMap [58]. OpenMap is about 11 times the size of Kelpie flight planner. This case study evaluates the performance of the interpreted formalism concept in large-sized software. In addition to feasibility, error detection, and effort level, this case study assesses scalability. The results of this case study suggest that (1) the interpreted formalism is easily applied to a large-sized software system, (2) error checking techniques are still effective, and (3) users' effort is greatly reduced by reusing existing artifacts.

10.1 SYSTEM OF CASE STUDY

10.1.1 BASIC INFORMATION

OpenMap is a JavaBean-based toolkit for building applications and applets needing geographic information. OpenMap allows users to show map data and manipulate geographic data. The OpenMap software has been studied and used by many developers [59, 60, 61, 62]. The figure below is a screenshot of the user interface.



1:2.0488352E8

Lat, Lon (19.471, 158.312) - x, y (637,241)

Fig. 30 . Snapshot of OpenMap software

10.1.2 IMPORTANT REAL-WORLD SEMANTICS

Some real-world semantic attributes are important in understanding OpenMap. These attributes are introduced here.

DIMENSIONAL AND UNITS

Similar to the Kelpie flight planner, the OpenMap software makes calculations involving distances, heights, speeds, angles, time and so on, and it does so using a variety of units. Clearly, the software is of the type for which real-world constraint checking has the potential to discover units related errors.

The dimensions and units are all real-world concepts that are defined in the real-world type system by default.

GEOGRAPHIC AND GEOCENTRIC LATITUDE

The real-world entity *latitude* has been widely used in the OpenMap software. The software uses two types of latitude: geographic (geodetic) latitude and geocentric latitude. The two types of latitude are different, and the difference is shown in Fig. 31. Such difference is crucial when the shape of Earth is modeled as an ellipsoid.

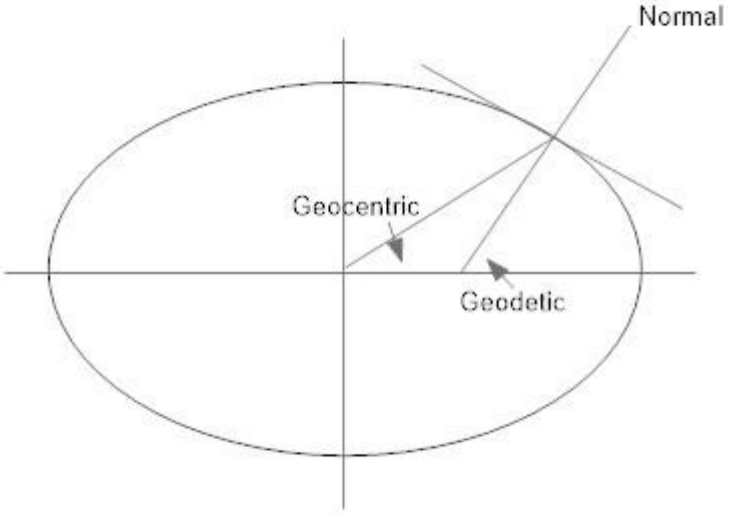


Fig. 31 . Two different types of latitude

REFERENCE LEVEL OF ELEVATION AND ALTITUDE

In OpenMap, the computation of the distance between two objects on Earth's surface frequently involves objects' elevations. The elevations have different reference levels. Two important reference levels are local ground and mean sea level. The difference between the two reference levels should be carefully handled when the computation expects high accuracy.

10.2 DATA COLLECTED AND ASSESSMENT

10.2.1 FEASIBILITY

Section 8.3.1 describes the metrics to evaluate feasibility. The data collected in this case study are presented and analyzed below.

DATA COLLECTED

The data collected in this case study were:

- Size of the software

  The OpenMap software used in this case study comes from the GitHub repository of OpenMap of the latest version 5. The software has 157,858 lines of code. It contains 1193 files, and the files are organized in 92 packages.

- Size of the real-world type system

  The real-world type system created for Kelpie flight planner was reused in this case study. The 35 real-world types and 97 real-world type rules were reused. Real-world type bindings cannot be reused. In order to fully interpret OpenMap, 1932 real-world type bindings were created.

- Coverage of real-world type bindings

  Variables in 196 source files have been bound with real-world types. Real-world types have been accessed by program elements inside 232 source files. The other source files are not interacting with real-world entities. They do not have real-world type bindings.

- Numbers relevant to error checking

  With real-world type system deployed, real-world constraint checking and reasonable range analysis were conducted to detect errors. All 1193 source files were checked for errors. Real errors were found in different files.

ASSESSMENT

Section 8.3.1 poses several questions for evaluating feasibility. These questions are answered here based on the data collected.

- Can different real-world types and type rules be defined and used?

  This case study for OpenMap reused the real-world type system for the Kelpie flight planner. The real-world type system for the Kelpie flight planner contains 35 different real-world types and 97 different real-world type rules. In addition to these existing artifacts, *one* real-world type was newly created. Thus, the real-world type system for OpenMap has 36 real-world types, 97 real-world type rules, and 1932 real-world type bindings.

- Can different program elements be bound with real-world types?

  The real-world type system contains 1932 real-world type bindings in total. The bindings were associated with different kinds of program elements, including local variables, global variables, function signatures, arrays, and class instances.

- Can the interpreted formalism be applied to different source files in the software?

  The interpreted formalism and real-world type system were applied to OpenMap. The software has 1193 source files, 193 files contain real-world type bindings, and all other files access real-world types through bindings or calling external functions.

- Can analysis techniques be applied to the software?

  Real-world constraint checking and reasonable range analysis were both used for error detection. All 1193 source files have been checked. Detailed results of error checking are presented in the next section.

- Is it feasible to apply the interpreted formalism to a large-sized software system?

  The interpreted formalism was successfully applied to OpenMap. Real-world types were bound with various program elements, and analysis techniques successfully detected real errors.

10.2.2 ERROR DETECTION CAPABILITY

The analyses performed on OpenMap showed useful results. Both real-world constraint checking and reasonable range analysis were used for error detection. Real-world constraint checking found real errors that have not been reported before. These errors were reported to the author(s) of OpenMap through the GitHub website and the author(s) confirmed them. Reasonable range analysis found statements that could lead to runtime errors.

DATA COLLECTED

The data collected in this case study on error checking are shown below:

- Number of errors reported

Table. 6 summarizes the results of error reported.

Table. 6. Errors reported by analyses in OpenMap

Analysis technique	**# of errors reported**	**# of files involved**
Units checking	49	15
Real-world constraint checking	52	18
Reasonable range analysis	29	18

Overall, real-world constraint checking reported 52 errors from 18 source files. Reasonable range analysis reported 29 warnings from 18 source files.

- Number of real errors

Among the errors reported, some errors are genuine, and some others are false warnings. Table. 7 presents the relevant data:

Table. 7. Real errors found in OpenMap

Analysis technique	**# of errors reported**	**# of files with reported errors**	**# of real errors/warnings**	**# of files with real errors**
Units checking	49	15	8	4
Real-world constraint checking	52	18	23	10
Reasonable range analysis	29	18	12	6

As stated before, units checking is a special form of real-world constraint checking. Overall, units checking found 8 real errors. Real-world constraint checking found 24 real errors in total from 11 files. Table. 8 summarizes the real errors found and source files that contain these errors:

Table. 8. Real errors found by real-world constraint checking

Program File	**# of faults**	**Semantic attributes involved**
RoadFinder.java	1	Latitude and longitude
Route.java	4	Units
Road.java	4	Units
Gonomic.java	1	Latitude and longitude
OMDistance.java	2	Units
TX7.java	1	Earth radius
LOSGenerator.java in (com/bbn/openmap/tools/terrain/)	3	Reference level of elevation
LOSGenerator.java in (com/bbn/openmap/layer/terrain/)	3	Reference level of elevation
GeoTestLayer.java	1	Geodetic and geocentric latitudes
GeoCrossDemoLayer.java	3	Geodetic and geocentric latitudes
QuadTreeNode.java	1	Units

- Number of false warnings

In this case study, analysis techniques reported more errors than the number of real errors. Real-world constraint checking reported 52 errors, while 23 of them are real errors. The other reported errors are not real errors, but they could be useful. These unreal errors are categorized into two kinds: improper usage and false warning.

The table below summarizes the improper usage and false warnings found by all analysis techniques.

Table. 9. False warnings and improper usage

Analysis techniques	**# of improper usage**	**# of false warning**
Units checking	24	17
Real-world constraint checking	25	4
Reasonable range analysis	4	12

The definition of *improper usage* was introduced in section 9.2.4. The improper usage refers to either (a) a variable took on different real-world entities (but the same programming datatype) in different parts of the program, or (b) the elements of an array were not all of the same real-world entities (but were of the same programming type).

Most improper usages come from statements that are similar to the statements below:

```
        lat = Math.toRadians(lat);
        lon = Math.toRadians(lon);
```

Variables `lat` and `lon` on the left side represent values of latitude and longitude values in units of `radians`, but the two variables represent values in units of `degree` on the right side. The variables take different real-world entities in the same statements. The statements are flagged as improper usage.

Other non-real errors that were reported are false warnings. These false warnings frequently involve conversion between different real-world types. For example, false warnings were reported in statements below:

```
double lambda = lon * Degree;
double phi = Math.abs(lat * Degree);
```

In the first statement, variable `lon` which represents longitude of `radians` is converted to variable `lambda` which represents longitude measured in `degrees`. The second statement is similar.

The statements reported as improper usages and false warnings indicate some error-prone operations. Programmers should double check these statements to make certain that the entities referenced are being used correctly.

ANALYSIS OF REAL ERRORS

The details of these real errors are presented in this section. The source files where the errors were found are listed, and then the details of the errors are presented.

- RoadFinder.java

An incorrect statement was found. It adds values of `latitude` to values of `longitude`.

```
distance = (Math.abs(foundLoc.getLatitude()-loc.getLatitude())
         + Math.abs(foundLoc.getLongitude()-loc.getLongitude()));
```

- Route.java

Four errors have been reported in source file `Route.java`. All of them are related to misuses of units. Two of them exist in the statement below:

```
float timeLimitBase = GreatCircle.sphericalDistance(toLat,
             toLon, fromLat, fromLon)
             / worstConvoySpeed;
```

For the first error, the function `GreatCircle.sphericalDistance()` expects parameters in units of `radians`, while the arguments `toLat`, `toLon`, `fromLat`, and `fromLon` all are in units of `degree`. Another error in this statement is caused by the division. This function `sphericalDistance()` returns a value in units of `radians`, and `worseConvoySpeed` is in units of `kilometer per hour`. Such division is an inconsistent use of units.

Two other units related errors were found in this file. They are:

```
crowsPathDistance = GreatCircle.sphericalDistance(toLat,
                             toLon,
                             nextLoc.getLatitude(),
                             nextLoc.getLongitude());
```

and

```
   float crowsPathHours = crowsPathDistance
                                / bestConvoySpeed;
```

These two errors were caught for the same reasons of the two errors mentioned above. For the first error, the function `GreatCircle.sphericalDistance()` expects arguments in units of `radians`, but the arguments are all in units of `degrees`. For the second error, `crowsPathDistance` is a variable in units of `radians`, and `bestConvoySpeed` is in units of `kilometer per hour`.

- Road.java

Four real errors were found in this source file. They are all misuse of units. The statement below contains two real errors:

```
        kilometers += GreatCircle.sphericalDistance(
                          prevPoint.getLatitude(),
                          prevPoint.getLongitude(),
                          thisPoint.getLatitude(),
                          thisPoint.getLongitude());
```

The reasons of the errors are the same as those stated above for `route.java`. For the first error, the function `GreatCircle.sphericalDistance()` expects the unit of measurement to be `radians` for the parameters, but the arguments in this statement are all of units `degree`. For the second real error, the return value of the function is of units radians, which is not commensurable with variable `kilometers`. The other two real errors were found in the two statements below:

The problems seen in the function `GreatCircle.sphericalDistance()` occur again. Arguments of the function should use the unit of measurement `radians`, but variables in units of `degrees` are provided; variable `thisLength` is not commensurable with variable `kilometers`.

```
                thisLength = GreatCircle.sphericalDistance(
                                        prevLat,
                                        prevLon,
                                        thisLat,
                                        thisLon);
                if (thisLength >= kilometers) {
```

- Gonomic.java

One real error was found in the statement below:

```
        double lambdaMinusCtrLon = lambda - centerY;
```

Variable `lambda` represents values of `longitude`, and variable `centerY` represents values of `latitude`. The right side of the statement expects a difference between two `longitude` values. The correct variable here should be `centerX`, instead of `centerY`.

- OMDistance.java

Two real errors were found in this source file. The first one is an error related to units, in the statement:

```
 float lonDist = ProjMath.lonDistance((float) g2.getLongitude(),
                                      (float) g1.getLongitude());
```

The function `lonDistance()` defines its parameters in units of `radians`, but the arguments, two calls of `getLongitude()`, are all in units of `degree`.

Another real error was found in this statement:

```
new OMText((float) mid.getLatitude(), (float) mid.getLongitude(),
                                text, OMText.JUSTIFY_LEFT);
```

The constructor `OMText()` defines its parameters with a hypothesis that Earth is modeled as a sphere. And the first argument of the constructor expects a value of `geocentric` latitude. However, in this statement, the value returned by function `mid.getLatitude()` is a value of `geodetic` latitude and the function models Earth as an ellipsoid.

- TX7.java

One statement in this source file contains an inaccurate computation. The statement is:

```
distance = GreatCircle.sphericalDistance(lt1, ln1, lt2, ln2)
             * Planet.wgs84_earthEquatorialRadiusMeters;
```

This statement computes the distance between two points on the Earth's surface. According to basic geometry, angular distance (or angle) multiplied by radius yields distance on a great circle of a sphere. In this statement, function `GreatCircle.sphericalDistance()` computes the angular distance between the two points on Earth surface, with the assumption that Earth is a sphere. However, variable `wgs84_earthEquatorialRadiusMeters` represents Earth's equatorial radius with Earth modeled as an ellipsoid. The computation of distance in this statement is not accurate.

- com.bbn.openMap.tools.terrain.LOSGenerator.java

  Three statements in this source file contain inaccurate computations. The three statements are below:

```
double P = Math.sin(arc_dist) *
    (endTotalHeight + Planet.wgs84_earthEquatorialRadiusMeters);
double xPrime = Math.cos(arc_dist) *
    (endTotalHeight + Planet.wgs84_earthEquatorialRadiusMeters);
double cutoff =
    startTotalHeight + Planet.wgs84_earthEquatorialRadiusMeters;
```

  All three statements intend to compute the distance between an object and Earth's center by adding Earth's radius and the object's height above Earth's `surface ground`. Earth's radius is the distance between Earth's center and Earth's surface. However, in the three statements above, the variable `wgs84_earthEquatorialRadiusMeters` represents Earth's equatorial radius; variables `endTotalHeight` and `startTotalHeight` represent values of objects' heights measured above `mean sea level`. The reference levels are different.

- com.bbn.openMap.layer.terrain.LOSGenerator.java

  This source file has the same filename as the previous one. They both serve the same purpose. The errors found in this file are similar to the ones stated above. The statements are:

```
double P = Math.sin(arc_dist) *
          (xyheight + Planet.wgs84_earthEquatorialRadiusMeters);
```

```
double xPrime = Math.cos(arc_dist) *
            (xyheight + Planet.wgs84_earthEquatorialRadiusMeters);
  double cutoff =
       LOScenterHeight + planet.wgs84_earthEquatorialRadiusMeters;
```

The reasons for the errors are the same as the ones described for the previous one.

- GeoTestLayer.java

  One real error is found in the statement:

```
OMPoint center = new OMPoint(
              centerGeo.getLatitude(), centerGeo.getLongitude());
```

  The constructor `OMPoint()` expects values of geocentric latitude for the first argument, but `centerGeo.getLatitude()` returns values of geodetic latitude. The two latitude values are different. Also, the class `OMPoint` uses latitude values with the assumption that Earth is a sphere, but `centerGeo` uses latitude values with Earth modeled as ellipsoid.

- GeoIntersectionLayer.java

  This source file contains a real error in the statement below:

```
OMPoint pgeo = new OMPoint(
            (float) geo.getLatitude(), (float) geo.getLongitude());
```

  Function `geo.getLatitude()` returns values of geodetic latitude, while values of geocentric latitude are expected.

- GeoCrossDemoLayer.java

  Three real errors were found in this source file. They are similar to the errors described above. The three statements are:

```
OMLine line2 = new OMLine(
             (float) ogc.getLatitude(), (float) ogc.getLongitude(),
                (float) gc.getLatitude(), (float) gc.getLongitude(),
                                     OMGraphic.LINETYPE_GREATCIRCLE);
new OMPoint((float) gc.getLatitude(),
                  (float) gc.getLongitude(), 3);
p = new OMPoint((float) i.getLatitude(),
                 (float) i.getLongitude(), 3);
```

  Values of geodetic latitude are used when values of geographic latitude are needed.

- QuadTreeNode.java

  One units related error was found in this source file. The error is located in the statements:

```
double distanceSqr = dx * dx + dy * dy;

if (distanceSqr < bestDistance.value) {
   bestDistance.value = distanceSqr;
   closest = qtl.object;
}
```

The variable `distanceSqr` and variable `bestDistance.value` are different in units, so they are not commensurable.

The errors reported above were detected by real-world constraint checking. In addition, reasonable range analysis was used to detection possible erroneous computations. The reasonable range analysis found 12 statements in 6 files. Table. 10 summarizes them:

Table. 10. Statements found by reasonable range analysis

Program File	**# of faults**	**Possible errors**
CADRG.java	1	Division of zero
Road.java	2	Out of reasonable range
Route.java	2	Out of reasonable range
OMDistance.java	1	Out of reasonable range
OMRasterObject.java	2	Division of zero
MercatorUVGCT.java	4	Infinite bound

- CADRG.java

  A warning was issued in the code segment below:

```
        if (lon1 > lon2) {
            dlon = (180 - lon1) + (180 + lon2);
        } else {
            dlon = lon2 - lon1;
        }
        deltaDegrees = dlon;
        deltaPix = getWidth();
    }
    ret = pixPerDegree / (deltaPix / deltaDegrees);
```

The variable `deltaDegrees` represents the difference between two longitude values. Such difference could be zero, which would lead to a division of zero in the last statement.

- Road.java

Range analysis reported two warnings in this source file. The two warnings come from statements below:

```
GreatCircle.sphericalDistance(prevPoint.getLatitude(),
                 prevPoint.getLongitude(),
                 thisPoint.getLatitude(),
                 thisPoint.getLongitude());
GreatCircle.sphericalDistance(prevLat,
                 prevLon,
                 thisLat,
                 thisLon);
```

The arguments of `GreatCircle.sphericalDistance()` expect units of `radians`, while the statements provide values in `degree`. These two errors were reported by real-world constraint checking as introduced above. Reasonable range analysis reported these two errors for different reasons. The ranges / bounds for the arguments in these two statements are different from the reasonable ranges expected by the function. For example, variable `thisLat` has reasonable range of [-90,90], and the expected range is $[-\pi/2,\ \pi/2]$.

- Route.java

  Reasonable range analysis found two errors in this source file. The two errors are almost the same as the two found in Road.java. The two errors reside in statements:

```
float crowsPathDistance = GreatCircle.sphericalDistance(toLat,
                                        toLon,
                                        nextLoc.getLatitude(),
                                        nextLoc.getLongitude());
float timeLimitBase = GreatCircle.sphericalDistance(toLat,
                                            toLon,
                                           fromLat,
                                            fromLon)
```

  These two statements were detected because the ranges for arguments are inconsistent with the ranges expected from the arguments.

- OMDistance.java

  One warning was issued in the statement below:

```
float lonDist = ProjMath.lonDistance((float) g2.getLongitude(),
                                     (float) g1.getLongitude());
```

  This error was also reported by real-world constraint checking. The arguments of function `lonDistance()` refer to wrong values in units of `degree`.

- RpfFrameCacheHandler.java

  One warning was issued on the last statement of the segment below:

```
int numHFrames = (int) Math.ceil((lrlon - ullon) /
                                  frameLonInterval);
int numVFrames = (int) Math.ceil((ullat - lrlat) /
                                  frameLatInterval);
……
return count / (float) (numHFrames * numVFrames);
```

Variables `numHFrames` and `numVFrames` represent the difference between latitude values and longitude values. The differences could be zero. Therefore, the last statement might lead to a division of zero.

- MercatorUVGCT.java

Four warnings were reported in this source file. These warnings were generated by the four statements below:

```
Math.tan(lat * Math.PI / 180.0
1.0 / Math.cos(lat * Math.PI / 180.0)
Math.tan(lat * Math.PI / 180.0)
(1.0 / Math.cos(lat * Math.PI / 180.0))
```

These computations lead to values of infinity when variable `lat` is $\pi/2$ or $-\pi/2$.

ASSESSMENT

The error checking capability is evaluated by answering the questions raised in section 8.3.2:

- Can the error checking techniques detect real errors?

  Units checking found 8 real errors. Real-world constraint checking was able to find 23 real errors. In total, 24 real errors were found.

  In addition, reasonable range analysis found 12 error-prone statements.

- Can the error checking techniques detect different kinds of real errors?

  The real errors found by real-world constraint checking involve different real-world semantic attributes, e.g., units of measurement, Earth model, and reference level. Reasonable range analysis found statements that are prone to errors. These statements could lead to different kinds of errors: possible division of zero, infinite values, or outbound of reasonable ranges.

- Can it detect errors from different source files?

  Yes. Units checking reported errors from 15 different source files. Real units errors were detected in 4 files. Real-world constraint checking reported errors from 18 source files. Real errors were found in 11 different files. Reasonable range analysis reported warnings in 6 source files.

- Is the error checking effective?

  Units checking reported 49 errors, and 8 of them are real errors, the ratio of real errors is 16%. Units checking is a lightweight version of real-world constraint checking, it gave a very promising starting point regarding error detection.

  Real-world constraint checking reported 52 errors, and 23 of them are real errors, the ratio of real errors is 44%. The ratio is arguably good.

Reasonable range analysis reported 29 errors, and 12 of them are real errors, the ratio of real errors is 41%.

Overall, the ratio of the number of real errors to the number of reported errors is more than 40%. Such ratio suggests that the error checking is very effective. About half of the reported errors are real errors.

10.2.3 EFFORT LEVEL

OpenMap software is 11 times the size of Kelpie flight planner. If real-world type system scales well, the effort required for OpenMap should be about 11 times the effort required for Kelpie flight planner. This section introduces and analyzes the data collected relevant to effort level required from users.

The results of this case study suggest that (1) the real-world type system is readily reusable, (2) the synthesis framework effectively reduces the effort required from users, and (3) the real-world type system works well for larger sized software system.

This part presents and analyzes the data collected relevant to user effort in this case study for OpenMap software.

DATA COLLECTED

The data collected in this case study relevant to effort level are list below:

- Size of the real-world type system in total.

As stated before, real-world type system for the Kelpie flight planner was reused in this OpenMap software. The real-world type system for Kelpie flight planner contains 35 real-world types and 97 real-world type rules. Only one real-world type was newly created for OpenMap. Therefore, real-world type system for OpenMap software contains 36 real-world types in total and 97 real-world type rules.

Since most real-world types and type rules were reused from existing artifacts, the major effort required in this case study is creating real-world type bindings. After fully interpreting the OpenMap software, 1932 real-world type bindings were created. A large part of the bindings were created by the synthesis mechanisms provided.

- Size of the real-world type system created by users.

  Reusing existing real-world type system saves users from creating all the real-world types and type rules. Most effort required from users are on creating real-world type bindings. Among the 1932 real-world type bindings created for OpenMap, a part of them were created by user, and another part of them were synthesized.

In this case study, the process of adding type bindings was carefully organized. It was composed of a *sequence* of two binding operations: *binding seeding* and *binding propagation*.

Binding seeding was done by the user. Users read source files and seeded type bindings to program elements manually. At the beginning of this case study, bindings were seeded to a few JavaBean files that directly interact with real-world entities. For example, `LatLonPoint.java` was the first source file into which type bindings were seeded. The file contains global variables representing latitude, longitude, and altitude. The file also has a collection of utility functions, e.g., a function that calculates distance between two coordinates; a function that computes the heading between two coordinates. During later parts of the case studies, bindings were seeded to different source files that access real-world entities.

Binding propagation was done by the type binding synthesizer. After type bindings were seeded to source files, the binding synthesizer propagated these bindings to other source files in OpenMap. From user’s perspective, binding propagation can be done on one or several files in one operation. For example, binding propagation was frequently applied to a package of source files. Type bindings in all files inside the package were propagated sequentially.

Table. 11 summarizes the sequence of binding operations in this case study.

Table. 11. Sequence of binding operations for OpenMap

Binding operation details	**# of bindings before operation**	**# of bindings after operation**
Seeded 80 bindings in LatLonPoint.java	0	80
Propagated bindings in LatLonPoint.java	80	203
Propagated bindings in package: com.bbn.openmap.proj	203	297
Seeded 119 bindings in GreatCircle.java	297	416
Propagated bindings in GreatCircle.java	416	507
Seeded 138 bindings in package: com.bbn.openmap.proj	507	645
Propagated bindings in package: com.bbn.openmap.proj	645	681
Seeded 61 bindings in Geo.java	681	742
Propagated bindings in Geo.java	741	835
Propagated bindings in package	835	896

com.bbn.openmap.geo		
Seeded 90 bindings in package: com.bbn.openmap.dataAccess.dted	896	986
Propagated bindings in package: com.bbn.openmap.dataAccess.dted	986	1024
Seeded 13 bindings in package: com.bbn.openmap.tools.road	1024	1037
Seeded 11 bindings in package: com.bbn.openmap.dataAccess.shape	1037	1048
Seeded 19 bindings in package: com.bbn.openmap.event	1048	1067
Seeded 6 bindings in package: com.bbn.openmap.graphicLoader	1067	1073
Seeded 10 bindings in package: com.bbn.openmap.gui	1073	1083
Seeded 28 bindings in package: com.bbn.openmap.layer.beanbox	1083	1111
Seeded 16 bindings in package: com.bbn.openmap.layer.dted	1111	1127
Seeded 30 bindings in package: com.bbn.openmap.layer.link	1127	1157
Propagated bindings in package: com.bbn.openmap.layer.link	1157	1189
Seeded 83 bindings in package: com.bbn.openmap.layer.link	1189	1272
Seeded 64 bindings in package: com.bbn.openmap.layer.location	1272	1336
Seeded 17 bindings in package: com.bbn.openmap.layer.policy	1336	1353
Seeded 94 bindings in package: com.bbn.openmap.layer.rpf	1353	1447
Seeded 16 bindings in package:	1447	1463

com.bbn.openmap.layer.shape		
Seeded 5 bindings in package: com.bbn.openmap.layer.terrain	1463	1468
Seeded 37 bindings in package: com.bbn.openmap.layer.vpf com.bbn.openmap.layer.test	1468	1505
Seeded 52 bindings in package: com.bbn.openmap.omGraphics	1505	1557
Propagated bindings in package: com.bbn.openmap.omGraphics	1557	1596
Seeded 20 bindings in package: com.bbn.openmap.omGraphics.geom	1596	1616
Seeded 120 bindings in all other packages	1616	1736
Propagated bindings in all files	1736	1932

In summary, the total number of real-world type bindings for OpenMap is 1932. 1129 type bindings were seeded in different source files, and 803 type bindings were synthesized. The binding synthesizer demonstrated better efficacy at early stages of developing type bindings. For the first 507 type bindings, 199 bindings were seeded and 298 bindings were synthesized. For the first 1024 type bindings, 488 type bindings were seeded and 536 bindings were synthesized.

ASSESSMENT

The effort level required from users is evaluated by answering the questions raised in section 8.3.3:

- Does the use of a real-world type system require excessive effort from the users? The data collected in this case study suggest that necessary effort are needed, but for a project with 1193 files, and 150,000 lines of code, the effort is not excessive.

Reusing existing artifacts made the effort even less. The major effort required was developing real-world type bindings. And the binding synthesizer reduced such effort by about 50%. Without any prior knowledge about the OpenMap software, approximately 35 man-hours were expended to develop the real-world type system for it. Analysis techniques then were able to reveal candidate errors in a few minutes.

- Can synthesizers provide support for reducing user's effort?

  The effort was largely reduced by two facilities: (1) reusing existing artifacts from the case study of Kelpie flight planner, and (2) using synthesizer of real-world type bindings.

  Most real-world types and type rules came from reuse. The number of real-world types needed in this case study was 36, and 35 of them were created in the case study of Kelpie flight planner. All real-world type rules used in this case study were created in the case study for Kelpie flight planner.

  The effort for developing real-world type bindings was reduced by synthesizers. The total number of real-world type bindings for OpenMap is 1932, 803 of them was synthesized; the ratio is 41.6%.

- Are the synthesizers effective in reducing users' effort?

  As described above, 35 of 36 real-world types came from reuse; the percentage is 97%; all real-world type rules came from reuse. Reusing existing artifacts alleviated user from creating real-world types and type rules.

  The binding synthesizer successfully reduced the effort required to develop real-world type bindings. The synthesis is very effective when the bindings are sparse in the software. For the first 507 type bindings, 298 bindings were synthesized, the ratio is 59%; for the first 1024 type bindings, 536 type bindings was synthesized, the ratio is 52.3%. When 1024 type bindings were developed, source files that frequently access real-world entities were associated with real-world types. After that, source files that rarely access real-world entities were inspected one-by-one, and type bindings were seeded accordingly. In total, the number of type bindings was 1932, and 803 type bindings were synthesized. The ratio is 41.6%. The synthesis mechanisms are very effective in reducing effort.

10.2.4 SCALABILITY

In order to get insights about the scalability of interpreted formalism, a comparison was performed between results of case studies on Kelpie flight planner and OpenMap. The comparison was on (1) sizes of the software, (2) sizes of real-world type systems, (3) results of error detection, and (4) effort made by the user. The results of the comparison suggest that interpreted formalism scales well as the sizes of software systems increase.

DATA COLLECTED

The data collected in both case studies are presented and compared in the tables below:

- Size of software projects

Table. 12. Software size comparison

Data collected	**Kelpie flight planner**	**OpenMap**	**Scale**
Number files	126	1193	9.5
Number of packages	10	92	9.2
Lines of code	13,884	157,858	11.4

- Size of real-world type system

Table. 13. Real-world type system comparison

Data collected	**Kelpie flight planner**	**OpenMap**	**Scale**
Number of real-world types	35	36	1.03
Numer of real-world type rules	97	97	1
Number of type bindings	255	1932	7.6

- Effort required from engineers

Table. 14. Effort level comparison

Data collected	**Kelpie flight planner**	**OpenMap**	**Scale**
Number of real-world types created by user	35	1	0.03
Number of real-world type rules created by user	97	0	0
Number of real-world type bindings created by user	122	1129	9.25

- Results of error detection

Table. 15. Results of error detection comparison

Data collected	Kelpie flight planner	OpenMap	Scale
Number of errors reported by real-world constraint checking	12	52	4.3
Number of real errors detected by real-world constraint checking	6	23	3.8
Number of errors reported by Reasonable range analysis	12	29	2.4
Number of real errors reported by Reasonable range analysis	12	12	1

ASSESSMENT

The scalability is assessed by answering several questions:

- Does the size of real-world type system scale linearly?

In fact, the size of real-world type system scales super-linearly. The size of OpenMap is 11.4 times the size of Kelpie flight planner. If the size of real-world type system scales linearly with the size of software, the size of the real-world type system for OpenMap would be about 11 times the size of the real-world type system for Kelpie flight planner.

However, the comparison in Table. 13 suggests that the size of real-world type system increases super-linearly to the size of software. The numbers of real-world types and type rules for OpenMap are almost equal to the numbers for Kelpie flight planner. The number of real-world type bindings in OpenMap is 7.6 times of the number of Kelpie flight planner.

- Does effort level required from user scales linearly?

  The comparison in Table. 14 suggests that effort required from user increases super-linearly to the size of software. The number of user-created bindings for OpenMap is 9.1 times of the number for Kelpie flight planner.

- Does error checking capability scales well?

  The comparison of error checking results suggests that the error checking techniques perform equally well in moderate and large sized software. The ratio of the number of real error to the number of reported errors for Kelpie flight planner is 6/12 = 50%, the same ratio for OpenMap is 23/52 = 44%. Such ratio is almost equal for the two software systems. The difference is that real-world constraint checking reported 52 errors, which is about 4 times the number of errors reported in Kelpie flight planner.

CHAPTER 11

11. CASE STUDY: PRAGMATIC APPLICATION

This research advocates a *new* engineering paradigm: building interpreted formalisms instead of building software logic. This paradigm switch introduces various analysis opportunities and confers substantial benefits to the programmers. In order to help programmers understand, and thereby adapt to this new technique, this chapter illustrates using a case study how a programmer will assess this new technology.

In essence, the question for this case study is:

> *How difficult is this technology to be applied to a programmer' working software project?*

To answer this question, this case study proceeds in an attempt to follow programmers' thinking process. Specifically, we hypothesize that programmers want to address four questions when they encounter this new technique:

- How to try the technique with minimum effort?
- What are the features or functionality of this technique?

- Is this technique useful to my software?
- What effort is required to maximize the benefits?

To address these questions in order, the demonstration is organized in four phases: *familiarity*, *exploration*, *relevance*, and *full utilization*. The remainder of this chapter introduces the details of these four phases.

This case study is on the software Kelpie flight planner. The details of the software have been introduced in Chapter 8.

11.1 PHASE #1: FAMILIARITY.

GOAL OF THE PHASE

When a new technique is presented to programmers, they frequently start with trials of a few features. They may prefer not to spend great deal of effort at the very beginning, but still want to see the potential. So the goals of the first phase are (a) using the technique with minimum effort, and (b) demonstrating possible benefits.

In order to address the goal of achieving familiarity, it's necessary to determine three quantities in applying interpreted formalisms:

- The minimum number of variables that need to be interpreted.
- The minimum number of real-world entities that need to be specified.
- The minimum amount of detail that need to be provided in interpretations.

STRATEGY AND RESULTS

In this demonstration, the function `distanceTo()` was chosen at random as the starting point. The function computes the distance from one coordinate to another coordinate. The source code is displayed below:

```
public double distanceTo(final Coordinate l) {
  final double lat1 = Math.toRadians(this._latitude);
  final double lat2 = Math.toRadians(l._latitude);
  final double lon1 = Math.toRadians(this._longitude);
  final double lon2 = Math.toRadians(l._longitude);
  final double dlon = lon1 - lon2;
  final double dlat = lat1 - lat2;
  final double a =
    Math.pow(Math.sin(dlat / 2), 2d)+ Math.cos(lat1)
    *Math.cos(lat2) * Math.pow(Math.sin(dlon / 2), 2d);
    final double d =
      2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));
    return Math.abs(earth_radius * d);
  }
```

Initially, the following three quantities were chosen for this first phase:

- The minimum number of variables to be interpreted is *two*. They are two global variables used in the function: `_latitude` and `_longitude`.
- The minimum number of real-world entities to be specified is *two*. The two real-world entities are latitude in degrees and longitude in degrees.
- The minimum amount of detail needed in interpretations is described below:

Interpretation contains specification for real-world entities, and specification for the relationship between real-world entities and their program elements.

- In the specification of real-world entities, two real-world semantic attributes `unit of measurement` and `reasonable range` are specified. Real-world entity latitude has the unit of degree, and its reasonable range is `[-90, 90]`. Real-world entity longitude has the unit of degree, and its reasonable range is `[-180, 180]`.
- In the specification of the relationship, possible sources of differences and errors are specified. In the Planner project, the values of real-world latitude and longitude come from the FlightGear simulator. No sensors are involved. The major source of difference is floating-point error. Therefore, no additional details are necessary for this specification.

With the variables interpreted, programmers might want to try error analysis on the function. The easiest way to see the error-checking capability is by finding out if the analyses can detect seeded errors. Two errors were seeded in the `distanceTo()` function then real-world constraint checking and range analysis on the function were conducted. One error was seeded in assignments and the other was seeded in computations.

The first error was seeded in line 2 where the code was changed to:

```
final double lat1 = this._latitude;
```

With real-world constraint checking, an error was reported on line 7, since `lat1` is of unit `degree`, `lat2` is of unit `radians`, and their units are inconsistent.

The second error was seeded in line 9 where the operator + was changed to -:

```
Math.pow(Math.sin(dlat / 2), 2d)- Math.cos(lat1)
```

A warning was reported in line 12 by range analysis. The range computed for variable a is `[-1,1]`, which caused `sqrt(a)` to be invalid.

In addition to the above analysis techniques, targeted inspection is available to programmers at all times. They can access the interpretations and real-world meanings of the variables through the GUI. More external resources can be found through the *explications* embedded in real-world specifications. Further, using the synthesis technique for real-world type bindings introduced in Chapter 6, the user can access the interpretations from all calling sites of the two variables among other source files.

CONCLUSION

In this phase, the interpreted formalism was applied with minimum amount of effort. The basic idea of using interpreted formalisms is demonstrated. Interpretations have been added to two program variables in one function, and the analysis techniques found the seeded errors. After this phase, users would have some familiarity with using interpreted formalisms, and would be ready to proceed to the next phase.

11.2 PHASE #2: EXPLORATION

GOAL OF THE PHASE

After the first phase, programmers are ready to discover more features about the new technique. They want to know what it is capable of and how it performs under different circumstances. So the goal of this second phase is to explore the capabilities of the interpreted formalism in different dimensions.

STRATEGY AND RESULTS

In this phase, the performance of the interpreted formalism is explored in two dimensions:

- *Scalability*: Exploration in this dimension intends to allow the programmer to discover if the interpreted formalism concept can be applied on program units of different sizes.
- *Error detections*: Exploration in this dimension intends to allow the programmer to discover how the error checking behaves in different program elements.

In order to explore the scalability, the interpreted formalism was applied to three programs of different sizes.

- `Coordinate.java`: This class represents coordinates in coordinate systems.
- `MagField.java`: This class computes Earth's magnetic field at a given coordinate using various models of Earth.
- `PlanEditor.java`: This class constructs the graphic interface for adding flying routes between coordinates.

Table. 16 shows relevant statistics of the three programs with interpreted formalism defined:

Table. 16. Interpreted formalism for programs of different sizes

Program	**# of interpreted variables**	**Lines of code**	**# of real-world entities specified**
Coordinate.java	33	89	9
MagField.java	14	640	11
PlanEditor.java	4	1138	2

In this phase, real-world specifications are defined with more details. For example, the specification for real-world latitude contains seven real-world semantic attributes: `unit of measurement`, `dimension`, `type of latitude`, `model of Earth`, `zero circle`, `orientation` and `feasible range`. Every attribute is associated with an explication which links to more external resources.

As indicated by the table, interpreted formalisms can be created for programs with different sizes. The effort involved depends on the contents of the subject programs. As the largest program among the three, `PlanEditor` has only four variables that require interpretations. Other variables in the program do not interact with the real world. On the contrary, the program `Coordinate` has the most interpretations, though its size is the smallest.

To explore the error checking capability, real-world constraint checking and range analysis were the focuses. The exploration was in two parts. The first part was conducting the two analyses on the three programs directly. The second part was detecting seeded errors.

For the first part, Table. 17 summarizes the results:

Table. 17. Results of analyzing the three programs directly

Function	**# of errors reported by constraint checking**	**# of warning issued by range analysis**
Coordinate.java	0	1
MagField.java	1	0
PlanEditor.java	0	0

For the `Coordinate.java` program, a warning message was issued by range analysis on the following statement:

```
double heading = Math.acos(
      (Math.sin(lat2)-Math.sin(lat1)*Math.cos(d))
               / (Math.sin(d) * Math.cos(lat1)));
```

This is caused by the range of argument for `Math.acos()`. The range computed for the long expression is `[-∞,+∞]` due to the fact that the range of the divisor is `[0,1]`. A division of zero would break the computation.

In the `MagField` program, the error reported by constraint checking is caused by a misuse of latitude values of different types (geocentric latitude versus geodetic latitude).

For the second part, three errors were seeded to different program elements in the same function used in Phase #1. One error is seeded in assignments, one error is seeded in an infix expression, and one error is seeded in a return statement.

The first error was seeded in line 2 as in Phase #1, but now the code is changed to:

```
final double lat1 = Math.toRadians(this._longitude);
```

If we were in Phase #1, real-world constraint checking cannot detect this error due to limited definition of real-world semantics. But now, because the semantics of latitude and longitude are comprehensive, the error can be detected.

The second error was seeded in the computation in line 7 by changing `lat2` into `lon2`. The statement after error seeding was:

```
final double dlat = lat1 - lon2;
```

This error was detected by constraint checking since such calculation between latitude and longitude is disallowed.

The third error was seeded in the return statement by changing line 13 to:

```
return Math.abs(earth_radius*d*2);
```

Real-world constraint checking cannot detect this error, but range analysis issued a warning message. The real-world entity corresponding to the return value of this function is surface distance on Earth in the unit of nautical mile. Its reasonable range is `[0,10820]`. The range of `earth_radius*d*2` exceeds this reasonable range.

CONCLUSION

In this phase, programmers explore the interpreted formalism to discover its capabilities. In this demonstration, scalability and error checking capability of the interpreted formalism were explored. The interpreted formalism was applied to three files of different sizes. The error checking techniques detected both real errors and seeded errors in these files. After this phase, users know certain features of interpreted formalisms. They are ready to apply interpreted formalisms for their benefits.

11.3 PHASE #3: RELEVANCE

GOAL OF THE PHASE

Phase #1 and Phase #2 are limited to small functions and a few programs. The effort required from programmers is quite small. Programmers gain a decent amount of experiences with the interpreted formalism in those two phases. Now they may start thinking about how to make this technique useful in their software projects. Therefore the goal of this phase is to *relate* the capabilities provided by the interpreted formalism to the projects of programmers.

Programmers may prefer to establish specific properties of their projects. For example, some programmers may want to assure that consistency of units of measurement are not broken in all program statements. Some programmers may want to assure that all program variables stay within their reasonable ranges. In this phase, the interpreted formalism is applied to assure the properties of most interest to programmers.

STRATEGY AND RESULTS

In this demonstration, we hypothesized that programmers mostly want to assure the consistency of units of measurement in the software. The interpreted formalism was applied to the whole Kelpie flight planner project for that purpose. Every suitable program variable was interpreted and thereby linked with their real-world specifications. The specification for each real-world entity has only one semantic: unit of measurement. The real-world constraint checking was used to check the errors.

Table. 18 summarizes the statistics about the project with the interpreted formalism added in this phase:

Table. 18. Stats of interpreted formalism for analyzing units of measurement

# of programs	**# of interpreted variables**	**Line of code**	**# of real-world concepts accessed**
126	255	13,884	35

The project has 13,884 lines of code in total. A total number of 35 real-world specifications were defined, and 255 variables were interpreted.

Constraint checking found ***four*** unit related errors in this phase. Two errors are caused by incorrect conversion between units of `hour` and units of `minute`. Another error is caused by misuse of variables in units of `feet per minute` and variables in units of `nautical mile per hour`. The fourth error is caused by misuse of the units `radians` and `degree`.

CONCLUSION

In this phase, the interpreted formalism is related to users' software by applying it to the whole software project. The analysis techniques provided by interpreted formalism are utilized to assure a specific real-world semantic. In this demonstration, real-world constraint checking was able to find several real errors pertinent to unit consistency. After this phase, users are familiar with the usage of interpreted formalisms. The next phase is full utilization of the interpreted formalism.

11.4 PHASE #4: FULL UTILIZATION

GOAL OF THE PHASE

Phase #3 shows an application of the interpreted formalism to assure a specific project-wide property. With that experience, programmers may want to fully utilize the interpreted formalism in their software projects. The goal of this phase is to completely apply the interpreted formalism in programmers' projects. In this phase, the goal is to discover how much benefits programmers will receive, and how much effort is needed for full utilization.

STRATEGY AND RESULTS

The setup of the interpreted formalism in this phase is based on the setup of Phase #3 where program variables have already been interpreted and real-world specifications have been created. However, compared to the last phase, the details of the interpreted formalism in this phase are much more comprehensive. Real-world specifications document complete semantic attributes, and real-world constraints are defined for the whole project. Both real-world constraint checking and range analysis are conducted on the project.

In this case study of the Planner project, 97 real-world constraints were defined. An average of 5 semantic attributes were created for real-world specifications. The semantic attribute of `reasonable range` was specified for every real-world entity.

A number of real errors were found by constraint checking and range analysis. A summary of the numbers is shown in Table. 19:

Table. 19. Real errors found with full utilization

# of errors reported by constraint checking		**# of warnings reported by range analysis**
violation of constraints	***improper usage***	
6	6	12

Constraint checking found 6 violations of constraints. Four of them came from inconsistency of units we described in Phase #3, the other two are misuses of latitudes. Details about these six errors have been introduced in the case study of Kelpie flight planner.

Range analysis found 12 statements in which computations are error-prone. Some of the computations have possible division of zero. For example, the computation below could be broken when `headingRadians` is zero:

```
feetEastL = length / 2 * Math.sin(headingRadians);
```

Some of the computations involve inconsistent ranges. For example,

```
final double timeToBOD =
         altToBOD /   plan.getAircraft().getSinkSpeed() / 60;
```

The right side invokes the wrong function `getSinkSpeed()`. The correct function should be `getSinkRate()`. The two function calls return values whose ranges are ten times different. Thus, for the whole statement, the range for the left side is largely different from the range for the right side. Such difference raised a warning.

CONCLUSION

In this phase, the interpreted formalism is fully utilized in the Kelpie flight planner software to gain maximum benefits. Real-world constraint checking and range analysis both have found several real errors in the software. The benefits introduced by the interpreted formalism is substantial.

11.5 OBSERVATIONS

This demonstration lets us gain some insights about the utility of the interpreted formalism. Observations from this demonstration are summarized below.

- Versatility

Interpreted formalism supports various analysis opportunities. Programmers can expect different kinds of benefits. As shown in all phases, real-world semantics can be inspected to clarify misunderstandings, constraint checking was used to detect violations, and range analysis was used to find error-prone calculations. The power of the interpreted formalism can be considerable. Programmers can choose their preferred way of utilizing the interpreted formalism.

- Effectiveness

Application of the interpreted formalism requires effort from programmers. Fortunately, such effort yields benefits early. As illustrated in Phase #2, with only three programs and limited effort involved, a real error was found and a warning was issued.

- Usability

Application of the interpreted formalism emphasizes integrity of the original source code. Artifacts of interpreted formalism are manipulated separately from the source code; analyses and error messages are shown in standalone interfaces. Programmers can develop source code and utilize interpreted formalism in parallel.

CHAPTER 12

12. CASE STUDY: THE SYNTHESIS FRAMEWORK

12.1 INTRODUCTION

Chapter 6 introduces the concept of synthesis framework to develop interpreted formalisms. As indicated by the results in the evaluation, the synthesis mechanisms can reduce the effort required from users. In order to get more insight into the utility and performance of real-world type synthesis, the mechanisms were applied to the software project *Kelpie flight planner* for which a complete real-world type system has already been developed in Chapter 9 with no automated support. Having an existing complete example real-world type system, we were able to compare the results of the synthesis mechanisms with the human-generated system. Details of Kelpie flight planner has already been introduced in Chapter 9.

The following of this chapter presents data related to synthesizing candidate real-world types, real-world type bindings, and real-world type rules in turn.

12.2 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. In total, 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.

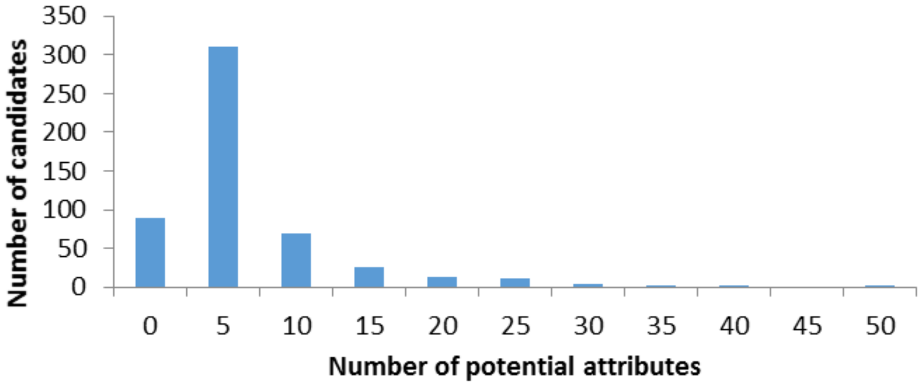


Fig. 32. Number of potential attributes for candidates

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. Fig. 32 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. 20 shows the results of applying the first criterion with different thresholds. The first column is the threshold values, and the second column lists 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. 20. Selection based on frequency of major terms

Threshold values	**# of terms**	**Average frequency**	**Average # 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. 21 shows the results of applying the second criterion with different thresholds.

Table. 21. Selection based on number of potential attributes

Threshold values	**# of terms**	**Average # 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

12.2.1 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. 22 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. 22. Connections between candidates and known real-world types based on criterion 1

Threshold values	**# of terms**	**# of Important major terms**	**# of Real-world types 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 the second column are repeated from Table. 20. 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. 23 shows the assessment of the candidate set based on selector criterion 2 (number of potential attributes of the major term) together with our assessment of the connections between the set of candidates and the real-world types that we identified.

The first column and the second column are repeated from Table. 21. With a threshold value of 5, we could construct 20 real-world types and 23 when the threshold was set to 3.

Table. 23. Connections between candidates and known real-world types based on criterion 2

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

The results shown in Table. 22 and Table. 23 suggest that reviewing approximately 80 terms from the candidate set can reveal a substantial fraction the useful set of actual real-world types.

12.3 SYNTHESIS OF REAL-WORLD TYPE BINDINGS

Section 6.2.3 introduces the synthesis mechanism for real-world type bindings. The synthesis proceeds in 3 stages: field, method, and local variable. Table. 24 describes data collected at each stage.

Table. 24. Performance of real-world type binding synthesis

Process stage	**# of bindings seeded by developers**	**# of bindings synthesized**	**Total # of bindings**
Field	28	64	92
Method	64	65	221
Local variable	30	4	255

In the table, the first column shows the stage of the type binding 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 demonstration, 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.

12.4 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.

12.5 PRAGMATIC APPLICATION WITH SYNTHESIS

Chapter 11 illustrates the practical application of the interpreted formalism. Such application is composed of four phases: familiarity, exploration, reference, and full utilization. In practice, the application of the interpreted formalism is facilitated by the synthesis mechanisms introduced in this chapter. Synthesis mechanisms assisted creating the real-world type system; existing artifacts were reused. The figure below shows the process of applying the interpreted formalism with the support of synthesis mechanisms:

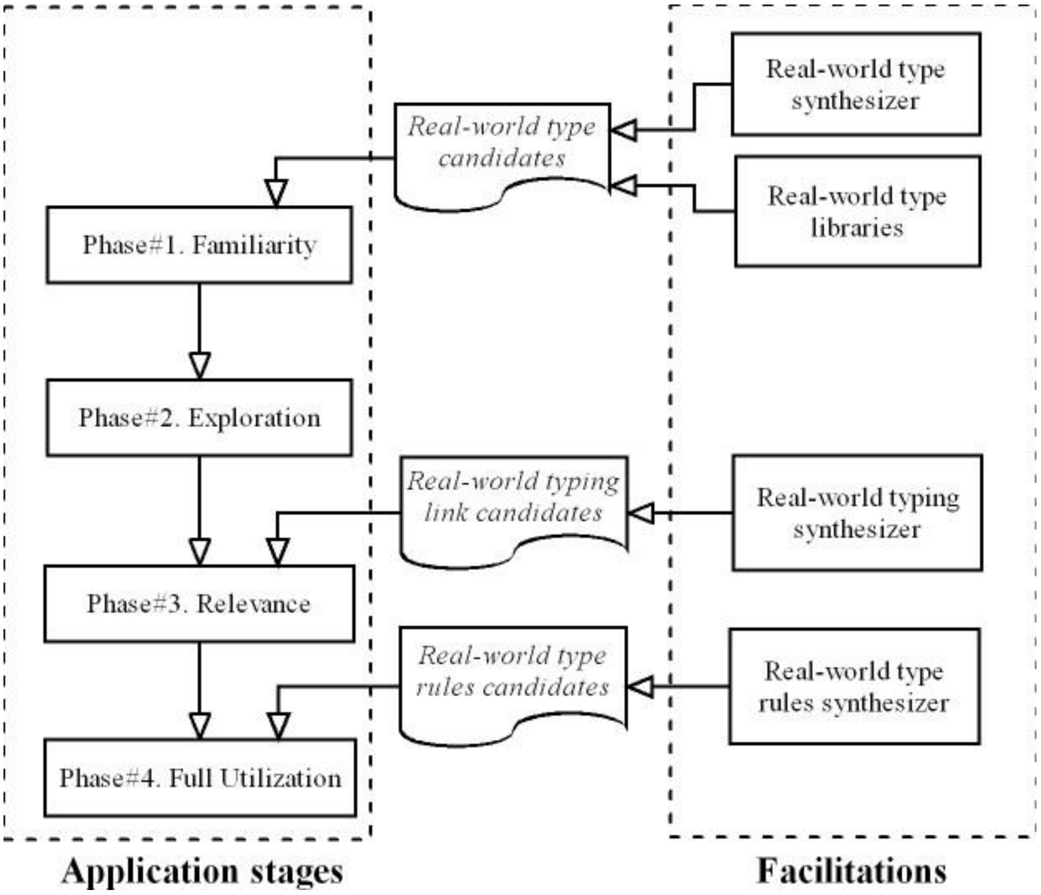

Real-world type synthesizer
Real-world type libraries
Real-world type candidates
Phase#1. Familiarity
Phase#2. Exploration
Real-world typing link candidates
Real-world typing synthesizer
Phase#3. Relevance
Real-world type rules candidates
Real-world type rules synthesizer
Phase#4. Full Utilization
Application stages
Facilitations

Fig. 33 . Pragmatic application with synthesis framework

In this process, synthesis mechanisms were leveraged in three phases:

- Familiarity Phase. In this phase, real-world type candidates are provided by synthesizer and existing libraries. Engineers can review these candidates, and create real-world types as needed.

- Relevant Phase. In this phase, engineers attempt to add real-world type bindings to all elements in the software to gain maximum benefits. A large number of the bindings were automatically created by the binding synthesizer.
- Full Utilization Phase. In this phase, real-world constraint checking is used to check all real-world type rules. A large portion of these rules are synthesized.

12.6 EFFORT LEVEL ASSESSMENT

The data presented in this chapter can be used to assess the effort level as part of the evaluation. The effort level required from users is evaluated by answering the questions raised section 8.3.3.

The effect level is assessed by answering a list of questions:

- Does real-world type system require excessive effort from the users?

  The data collected in this case study suggest that necessary effort is needed, but not excessive effort. As described in Chapter 6, users can develop real-world type systems incrementally. Also, with the help of synthesis mechanism, the effort required from users are largely reduced. Without any prior knowledge about the software system, the author spent approximately 40 man-hours to fully utilize the capability of the real-world type system.

- Can synthesizers provide support for reducing users' effort?

  Yes. Efforts are required in developing real-world types, real-world type rules, and real-world type bindings. As introduced in Chapter 6, the synthesis framework provides support for development. In this case study, the framework clearly demonstrates its capability. For the whole software system, 21 of the 35 real-world types can be formed from the candidate real-world types, 133 of the 255 real-world type bindings are synthesized, and all 97 real-world type rules can be chosen from the synthesized 260 candidate rules.

- Are the synthesizers efficient in reducing users' effort?

  Yes. As described above, 21 of 35 real-world types can be formed from candidates; the percentage is 60%; 133 of 255 real-world type bindings were generated; the percentage is 52%; all 97 real-world type rules can be formed from the 260 candidate rules; the percentage is 100%. On average, all kinds of effort were reduced by more than 50 percent.

CHAPTER 13

13. RELATED WORK

This dissertation advocates that software engineers should build not just traditional software, but *interpreted formalisms* that combine software logic with rigorously documented interpretations. A variety of previous research results are relevant to the ideas presented in this dissertation. This chapter summarizes the results in various related areas and provides references for more detailed information.

The notion of uninterpreted logic and the associated need for an interpretation is clear and well understood in the field of mathematic logic. Unfortunately, these notions are far less well understood in the field of software engineering.

As noted in the Chapter 1, section 1.3.3, the four-variable model [65] is very preliminary form of an interpretation for the logic defined by software. Despite this work, no significant progress has been made in the development of interpretation since the work by Miller et.al [57].

This dissertation presents the first comprehensive approach to the concept of interpretation for software. As such it builds on several related fields, especially the field of type theory. However, it doesn't build on an existing literature on interpretation beyond the work of Parnas and Miller et.al.

13.1 Model the Relationships between the Real World and the Machine World

Interpreted formalism is a new concept that models the relationship between the real world and the machine world. Research effort on requirements and specification have also modeled the connections between the real world and the machine world [31, 38, 39, 40, 57, 65].

13.1.1 Four-Variable Model

The four-variable model proposed by Parnas and Madey defines the connections in the IN and OUT relations [65]. The IN and OUT relations define the connections between the mathematical variables as available to the software and the environmental variables in the real world. The relationship between real world entities and machine world is described mathematically. The hardware and software requirements are intertwined in the REQ relation.

Parnas and Madey proposed a four-variable model that implied the connections in the IN and OUT relations. The IN and OUT relations in the four-variable model define the connections between the mathematical variables as available to the software and the environmental variables in the real world. As Parnas and Madey state, "IN describes the behavior of the input devices. IN is a relation rather than a function as a result of imprecision in the measurement and transducer devices. The Out relation describes the behavior of the output devices. It is a relation rather than a function because of unavoidable device imperfections.

Four-variable model is different from the interpreted formalism in several ways. First, the four-variable model focuses mostly on variables and attempts to describe the relationship between real world entities and machine world as purely mathematically relations. We argue that the semantic information for a real-world entity is critical as well. The interpretation for a real-world entity embodies all details that will convey the entity accurately. Second, the four-variable model uses absolute real time, which is implicitly assumed to be the same throughout requirements and specification. Interpreted formalism states there are substantial distinctions between the physical real time and the time implemented by a machine. Environment variables refer real time, whereas machine variables refer the time used in the machine. Third, system hardware and software requirements are intertwined in the REQ relation in the four-variable model. This makes the tracing of both hardware and software requirements hard. Interpreted formalism explicitly separates the effect made by software and hardware.

13.1.2 EXTENDED FOUR-VARIABLE MODEL

Miller and Tribble's extended four-variable model had noticed that in the original four-variable model, system and software requirements are inextricably intertwined. They introduced an extension to the four-variable model that isolates the virtual versions of the monitored and controlled values in subsystems [57]. The relationships emerge as relations between virtual and real versions of variables.

Several new concepts were introduced. MON’ and CON’ are the virtual versions of the monitored variables and controlled variables defined in the subsystem specification in the software. REQ’ contains the software requirements. IN’ and OUT’ map to the hardware specification. The extended model also states that the MON’ and CON’ are different from MON and CON. The differences in timing are introduced when sensing and setting the input and output variables. With interpreted formalism, such differences should be documented explicitly.

Interpreted formalism not only documents the connections and relationships between variables in software systems and elements in the real world, but also leverages such connections to enforce real-world constraints on software systems.

13.1.3 PROBLEM FRAME AND REFERENCE MODEL

The work of Zave and Jackson characterizes phenomena of interest to the system and separates world phenomena from machine phenomena [38, 39, 40]. The reference model of Gunter et al. gives a detailed explanation of different classes of phenomena and the relationship between environment and system [31].

These results model the picture of the connection between machines and the real world. In contrast with these results, real-world types provide a comprehensive set of real-world semantic attributes, and emphasize imposing constraints inherited from the real world on programs.

13.1.4 CYBER-PHYSICAL SYSTEM

Researchers attempted to model the relationship in the context of cyber-physical systems. Johnson et al developed an approach to detection of undocumented assumptions in Simulink/Stateflow models in which traces are used to develop system invariants [41]. Since the source for the approach is a high-level system model, the approach can detect undocumented assumptions at the level of real-world entities. The approach does not include mechanisms to define real-world properties over and above those available in Simulink/Stateflow.

Representing multiple domains, including both physical and cyber, and determining inconsistencies between those domains has been addressed by Bhave et al using a set of architectural views [7]. The views are derived from various models and a base architecture for the subject cyber-physical system. The approach is based on typed graph models of the subject system, and consistency is defined by morphisms from views to the base architecture.

13.2 TYPE SYSTEM

This dissertation introduces a new type system, real-world type system, as an implementation of interpreted formalism. The analysis of real-world constraint checking is also a form of type checking. This section compares real-world types with a few other types and type systems.

13.2.1 CONVENTIONAL TYPE SYSTEMS

A real-world type system is an enhanced and extended version of the concept underlying conventional type systems with the goal of supporting checking constraints inherited from the real world in addition to default type rules.

Conventional types are closely coupled with the machine context. As a result, they cannot comprehensively describe real-world information. For example, most real-world semantic attributes are associated with compile-time values; these attributes and values should not be represented as variables. Some real-world semantic attributes might be represented as variables, fields, or other structures. However, these program elements can only convey limited real-world meanings through identifier names or unstructured comments. In addition to real-world semantic attributes, the relationships between real-world entities and their machine representations are also ignored or vaguely expressed in conventional types. The discrepancies caused by sensors and timing differences are frequently neglected. As a result of these problems, real-world constraints are insufficiently documented, thereby enforced in ad hoc ways or absent in conventional type systems.

Real-world types are designed to document the missing information and use the information to enforce real-world constraints.

13.2.2 ENHANCED TYPE CHECKER

Powerful extensions to the basic notion of type have been developed, in particular in the form of pluggable type systems [21, 52, 56, 66]. Pluggable type systems [66] enhance the built-in type systems in applicable formal languages and provide support for additional checking capabilities. The Checker framework [21, 66] and JAVACOP [52] implement the idea of pluggable type system for Java. These frameworks refine built-in type systems to allow users to define additional types and check additional type rules.

Dependent types [12] are another powerful type system concept that allows programmers to specify and enforce rich data invariants and guarantee that unwanted program behaviors are detectable by analysis. They are important in computing environments where users must certify and check properties of un-trusted programs [63]. Dependent type systems, such as Coq [17] and Agda [3] provide formal languages to write mathematical definitions, executable algorithms, and theorems, and then support development of proofs of these theorems.

Pluggable type systems and dependent type systems are designed to provide greater flexibility in type mechanisms. Increased flexibility is valuable in supporting language expressivity. 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. They provide limited support on documenting real-world semantic attributes and checking real-world constraints. Relationships between real-world entities and their machine representations are frequently undocumented.

13.3 CHECK REAL-WORLD CONSTRAINTS

An interpretation based on real-world types support analyzing real-world constraints. As special kinds of real-world constraints, dimensional analysis and unit checking have been explored in many programming languages [15, 30, 82]. Previous research focused on extending programming languages to allow checking these constraints on dimensions of equations are not broken. Extensions to support dimensional and unit analysis have been developed for several programming languages. For the most part, previous research focused on checking dimensions of equations and validating unit correctness [5, 19, 30, 37, 44, 68]. Nevertheless, these efforts are limited to basic rules derived from dimensions or combinations of entities with different units.

The Autocert system develops source-code constraints based on analysis of real-world information contained in Simulink models [20]. 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.

Jung and Saglietti defined a language for describing interfaces between system components to support fault detection in component interfaces [42]. The language facilitates the definition of system details including those that we refer to as real-world attributes such as units. From interface descriptions, software wrappers can be derived to align interfaces thereby enabling the reuse of reusable components. The approach does not address the issue of real-world information and analysis within software source code.

By basing the analysis on a type system, the real-world type system permits a general approach that can be tailored to a specific domain or to a specific application. Type rules in a real-world type system can be derived from dimensions, units, or any other real-world sources.

13.4 IMPROVE LOGIC UNDERSTANDING AND MAINTENANCE

One of the benefits provided by interpreted formalism is increasing understanding of the formal logic. Real-world semantic attributes in real-world types can improve the understanding of logic. Various research efforts have been made to improve human understanding by linking structured semantic information in real-world context to logic.

13.4.1 INTEGRATION OF SEMI-FORMAL AND FORMAL NOTATIONS

Some research work attempted to improve the understanding of formal specification by integrating semi-formal and formal notations. Such integration may make formal specification more approachable. In a survey of industry, Craigen, Gerhart and Ralston [18] found that "better integration of formal methods with existing software assurance techniques and design processes was commonly seen as a major goal". They concluded, "Successful integration is important to the long-term success of formal methods." Fraser, Kumar and Vaishnavi [26] described a framework for classifying formal specification processes.

13.4.2 VISUALIZATION OF FORMALISM

Several research groups have developed frameworks to build visualization of formal logic. Visualization of formalism is often realized by mapping from formalism to graphical notations. An early approach is the Z visualization [45], which makes use of constraint diagrams. The notation is able to express predicate logic, but there is no integration into existing frameworks.

The approach of Fekih et.al maps B specifications to UML [75]. It takes the state space of the specification and creates a UML class for every abstract set that is element in the domain of relations. Idani and Ledru improve the approach by mapping occurring relations to UML associations [34]. Other than B method, the work of Bollin [10] discusses ways in transforming formal Z specifications to UML in order to open the documents to a wider range of stakeholders. A few other researchers have done some similar works. M. B. Özcan [64] described an approach to visualizing executable formal specifications based on Z notation. Razali. etc [70] proved the efficiency of using graphical notation in understanding formal specifications by doing experimental comparison. Ait-Ameur et.al [4] proposed to bring more information in the context of system into the development of software systems.

The results of those works provide an indication that the integration of both semi-formal and formal notation is useful in promoting specification or model comprehensibility as compared to the formal notation alone. Real-world type systems use real-world semantic attributes to describe the real world concepts. Natural languages are used to give explications for real-world concepts.

13.4.3 ONTOLOGY

Ontology is a widely used structure for documenting domain concepts and relationships among the concepts. Ratiu et.al developed techniques to improve the understanding of program elements by making explicit mappings between ontology classes and program elements [71, 72]. This paper [71] presents a formal framework for describing the mappings between domain concepts and the program elements, real-world relations and program relations. This framework allows describing typical classes of diffusion of the domain knowledge in code. Based on this formal framework, they describe an algorithm to recover the mappings between entities from an ontology and program elements.

Another paper from Ratiu [72] presents a formal framework which can be used to evaluate the implementation of the real-world concepts within a library. This framework is based on a common representation of ontologies and programs. Using this framework they characterize general classes of mismatch between the concepts implementation and the real world. They also present relevant mismatches together with examples that they (semi-)automatically identified within the Java library.

13.4.4 INTENT SPECIFICATION

One important propose of interpretation is to describe the desired effect of formal statements. The idea of explaining why things are to be done the way they are specified had been proposed in Leveson’s work on intent specifications [48, 49, 88]. Intent specifications explicitly state the relationship between means and ends in the intent dimension so that developers can use the information not only during software construction but also during other life cycle phases such as validation and maintenance.

Leveson's position also makes several essential points, but it is incomplete regarding theoretical arguments motivating why, on a more basic level, formal languages are insufficient to define software function. These arguments focus on supporting developers' cognitive processes, but do not address the foundational elements to enable developers to understand what a specification says. If a specification lacks the natural language necessary to enable a developer to understand it precisely, even knowing why particular statements were refined the way they were could leave the developer missing important details necessary to specify a system correctly.

13.5 SYNTHESIS MECHANISMS

Real-world type system contains synthesizers that extract real-world information from software logic. The information is used to construct candidate real-world types. Other researchers have made efforts that are relevant.

13.5.1 TYPE PROVIDER

Research effort had been made on synthesizing types in other type systems. Type providers in F# extend the language so that the compiler can generate both new types and new code that leverage these types with particular emphasis on the schemas of external data [76]. Type providers in Idris extend this idea to languages with dependent types [13, 16]. Neither F# nor Idris addresses the general issue of defining logic interpretations.

13.5.2 TYPING SYNTHESIS

The real-world type system contains a synthesizer of real-world type binding. The synthesizer produces candidate type bindings for program elements in the real-world type system. Efforts have been made to infer types for other type systems. For pluggable type system [21], researchers have implemented type inference algorithms to add types for non-annotated code [22, 56].

13.5.3 CONCEPT LOCATION

Concept location is the idea of identifying parts of a software system that implements some aspect of the problem. Concept location is related to the mapping of real-world entities to software [1, 2, 28, 29, 33, 51, 69] but with the goal of improving program understanding.

Information retrieval based approaches have been developed to reduce the effort required to understand and to locate source code that requires change. Poshyvanyk attempted to further reduce the effort by producing a concept lattice using the most relevant attributes (terms) selected from the top ranked documents (methods) [69]. Grant et al proposed approaches that identify statistically independent signals that could lead to concepts [28].

The use of parts of speech of terms in identifiers has been investigated as a means to extract information from the source code. Binkley improved identifier name tagging using templates and defined rules to improve the structure of field names [4]. Hill et al generated noun, verb and prepositional phrases from the signatures of program elements [15].

13.5.4 PARTS OF SPEECH

Synthesizer for real-world types leverages parts-of-speech (Pos) of the terms composing identifiers. Other researchers have also used PoS of the terms in identifiers to extract information from the source code [1, 2, 8, 85]. Binkley [8] improved identifier name PoS tagging using templates and defined rules to improve the structure of field names. Hill et al. [33] have generated noun, verb and preposition phrases from the signatures of program elements.

Parts of speech have also been used to extract domain models, such as ontologies. Abebe and Tonella [1, 2] have used the parts of speech of terms and the natural language dependencies to extract ontologies from source code. Raitu et al have proposed an approach to extract domain-specific ontologies from APIs [71, 72].

WordNet [78] is used in the process of synthesizing real-world types. Other researchers [74] also used it to automatically extract semantics and relationships between the semantics.

13.6 CONTEXT REPRESENTATION AND REASONING

The notion of context is important in many areas of computing. In ubiquitous computing, for example, context is needed to enable suitable processing of inputs received [46, 67]. This notion of context is related to the basic functionality of the system and is closely linked to machine learning and other aspects of artificial intelligence. Context representation and reasoning are related to the rigorous definition developed in this chapter only to the extent that it helps to identify the real-world entities with which the system of interest interacts.

13.7 UNCERTAINTY OF HARDWARE AND SOFTWARE

An interpretation documents the approximation between the values in the real world and the values in the machine world of the same real-world entity. Such approximation is usually caused by sensors and hardware. It is an important source of uncertainty. Research effort have been made on quantification, communication, and interpretation of such uncertainty [11, 81]. The uncertainty is modeled as an abstraction which can be added to different programming languages. The pragmatics of the approach has been demonstrated for different applications and hardware systems.

CHAPTER 14

14. CONCLUSION

This section concludes the work and summarizes the contributions, limitations, and future work of this dissertation research.

14.1 OVERVIEW

This dissertation research introduces a new artifact, interpreted formalism, to define software systems. The interpreted formalism is based on the idea that a computing system is composed of a computing platform, a set of physical entities, and a relationship between the two. The emphasis of interpreted formalism is an explicit definition of the relationship between the physical entities and the computing platform.

The relationship has been defined using the notion of logic interpretation. The interpretation defines explicitly the meaning of items in the logic regarding physical entities and their associated properties. Many important invariants can be derived from physical, i.e., real world, entities in a comprehensive and systematic way. Defining the relationship between physical entities and the computing platform in this way enables a new class of fault detection mechanism for the logic.

This dissertation presents a pragmatic approach to the development and application of the interpreted formalism concept. A preliminary implementation of the interpreted formalism, the real-world type system, is introduced. For a specific system, the real-world type system can be developed without impeding the development of software logic. The development of real-world type systems is facilitated by an automated synthesis framework. The synthesis framework can effectively guide creating components of real-world type systems. The effort required from users is greatly reduced.

The pragmatics of developing and applying the interpreted formalism was illustrated with case studies. In these case studies, the interpreted formalism was successfully applied to open-source software systems in the form of real-world type systems. The new analysis techniques provided by the interpreted formalism detected real errors that had not been reported before.

The interpreted formalism and real-world type systems were evaluated by applications on several case studies. The evaluation was on several properties: feasibility, error detection capability, effort level, and scalability. The results suggest that (1) the interpreted formalism is feasible for medium and large software systems, (2) error detection mechanisms have detected a substantial number of real errors from different software systems, (3) the synthesis framework significantly reduces the effort required from users.

14.2 CONTRIBUTIONS

This dissertation research supports its thesis statement by the following contributions:

- It introduces a new paradigm for software development. This work advocates that the interpreted formalism rather than isolated software is the right artifact for the development of safety-critical systems. An interpreted formalism combines logic with an explicit interpretation of the logic.
- It introduces an explicit structure for defining interpretations. In this work, the structure of the interpretation is a set of real-world types and a set of real-world type rules defined within the framework of a real-world type system. Real-world types convey meanings of real-world entities by documenting their real-world semantic attributes. Real-world constraints document constraints and invariants derived from the real-world context so analysis techniques can be developed to check these constraints.
- It provides a framework for systematic detection of software faults that violate real-world invariants. Several analysis techniques that leverage the contents in an interpretation were developed to detect candidate errors that violate real-world constraints. Real-world constraint checking detects violations of real-world type rules statically. Reasonable range analysis looks for possible values outside of reasonable ranges by conducting interval analysis on the programs. Targeted

inspection allows users to inspect the programs for possible violations. Assertion generation produces assertions for runtime checking around program entities.

- It introduces a synthesis framework that facilitates developing real-world type systems for software systems. Three synthesizers were developed: (a) a synthesizer that produces candidate real-world types by processing application materials, (b) a synthesizer that automatically infers real-world type bindings for various kinds of program entities, (c) a synthesizer that extracts candidate real-world type rules from program expressions. The synthesis framework significantly reduces the effort from users when developing real-world type systems.
- It provides a practical approach to conduct units checking on software systems. The error analysis of units checking has been provided by the real-world type system as a standalone functionality. Prior research work on units checking lack evidence and results on large-scale software system mostly due to the amount of effort required for using them. Our approach combines synthesizers of bindings with error detection so that it can be pragmatically applied to large-scale software systems. The case study of OpenMap provides evidence that the approach is feasible, practical, and effective.
- It conducted several case studies that assessed and evaluated the feasibility, error detection capability, effort level, and scalability of the interpreted formalism by applying interpreted formalisms to two geographic software systems. The results provided by case studies suggest that the interpreted formalism is applicable, practicable, and useful in modern software systems.

- It developed a prototype that can be used on modern software systems for error detections. The prototype has been used in the case studies and has found a substantial number of errors. It can be practically used in different open-source software projects.

14.3 LIMITATIONS

Although the interpreted formalism provides significant benefits to the engineers, it has limitations in several aspects.

The development of interpreted formalisms requires developing interpretations in addition to software logic. Development of interpretations requires extra effort from users. Engineers might be reluctant to develop the interpretations.

Also, the contents in an interpreted formalism, e.g., real-world semantics and real-world invariants, introduce a source of detects over and above those that might be present in the programs.

The current implementation of interpreted formalisms, real-world type systems, has not established the theory of soundness and completeness. The results of real-world constraint checking greatly depend on real-world type rules created by the users. Also, the analysis techniques provided by interpreted formalisms target static real-world invariants. Real-time related properties are not considered. For example, the rate of change of aircraft's speed obeys constraints in the physics. It not practical to enforce such constraints by current structure of interpreted formalism.

14.4 FUTURE WORK

The time and resources available for this dissertation research were limited. Improvement of the interpreted formalism is left to future work. The future work contains but is not limited to:

- Expand the structure of interpretations to systematically document and check complex real-world constraints and invariants. For example, temporal properties of real-world entities, e.g. rate of changes, could be documented and enforced.
- Improve the synthesis framework to provide comprehensive candidate real-world types and type rules.
- Expand the analysis techniques to leverage the differences between real-world entities and entities in programs documented for error detection.
- Expand the interpreted formalism so it can be easily integrated with different programming languages.

This research mainly concerns the feasibility and practicality of the interpreted formalism, which was evaluated through several case studies for application on two open-source software systems. The case studies were conducted by the author of the research and were restricted by time and resources available to a single engineer. More evaluations of the interpreted formalism, e.g. efficacy and utility of interpreted formalisms when applied on huge-scale software systems, application of interpreted formalisms involving different engineers, and application of interpreted formalisms on different systems and domains, are left to future work.

BIBLIOGRAPHY

1. Abebe, S., and P. Tonella. 2010. "Natural language parsing of program element Names for Concept Extraction". In Proceedings of the 18th International Conference on Program Comprehension (ICPC), Braga, 2010, 156-159. IEEE Computer Society, 2010.
2. Abebe, S., and P. Tonella. 2011. "Towards the extraction of domain concepts from the identifiers". In Proceedings of the 18th Working Conference on Reverse Engineering (WCRE), Limerick, 2011, 77-86. IEEE Computer Society, 2011.
3. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php.
4. Ait-Ameur, Y., J. P. Gibson, and D. Méry. 2014. "On implicit and explicit semantics: Integration issues in proof-based development of systems." In Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications, edited by Tiziana Margaria and Bernhard Steffen, 604–618. Springer, 2014.
5. Antoniu, T., P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. 2004. “Validating the Unit Correctness of Spreadsheet Programs”. In Proceedings of the 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, 2004, 439-448. IEEE Computer Society, 2004.
6. Bergin, C., and P. Harding. 2013. “Cygnus delays ISS berthing following GPS discrepancy.” http://www.nasaspaceflight.com/2013/09/cygnus-cots-graduation-iss-berthing/

7. Bhave, B., B. H. Krogh, D. Garlan and B. Schmerl. "View Consistency in Architectures for Cyber-Physical Systems." In Proceedings of the 2011 IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS), Chicago, 2011, 151-160. IEEE Computer Society, 2011.
8. Binkley, D., M. Hearn, and D. Lawrie. 2011. “Improving identifier informativeness using part of speech information.” In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR), Waikiki, 2011, 203-206. ACM Press, 2011.
9. Bogdan, P. and R. Marculescu. 2011. "Towards a Science of Cyber-Physical Systems Design." In Proceedings of the 2011 IEEE/ACM International Conference on Cyber-Physical Systems (ICCPS), Chicago, 2011, 99-108. IEEE Computer Society, 2011.
10. Bollin, A.. Crossing the Borderline - From Formal to Semi-Formal Specifications. SET 2006: 73-84
11. Bornholt, J., T. Mytkowicz, and K. S. McKinley. 2015. “Uncertain<T>: Abstractions for Uncertain Hardware and Software.” IEEE MICRO Top Picks, 35(3):132-143, May-June, 2015.
12. Bove, A. and P. Dybjer. 2009. “Dependent types at work”. In Language Engineering and Rigorous Software Development, edited by Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto. 57-99. Springer, 2009.

13. Brady, E. 2011. “Idris: systems programming meets full dependent types.” In Proceedings of the 5th ACM workshop on Programming languages meets program verification (PLPV), Austin, 2011, 43-54. ACM Press, 2011.

14. Castro, J., M. Kolp, and J. Mylopoulos. 2001. “A requirements-driven development methodology”. In Advanced Information Systems Engineering, edited by K.R. Dittrich, A. Geppert, M. Norrie, CAiSE 2001. LNCS, vol. 2068, 108–123. Springer, 2001.

15. Chen, F., G. Rosu, and R. P. Venkatesan. 2003. “Rule-based analysis of dimensional safety.” In Proceedings of the 14th international conference on Rewriting techniques and applications (RTA), edited by Robert Nieuwenhuis. 197-207. Springer, 2003.

16. Christiansen, D. 2013. “Dependent type providers”. In Proceedings of the 9th ACM workshop on generic programming, 23-34. ACM Press, 2013.

17. Coq. https://coq.inria.fr/

18. Craigen, D., Gerhart, S. L., and Ralston, T. 1992. An International Survey of Industrial Applications of Formal Methods. In Proceedings of the Z User Workshop, pp. 1-5. Springer, London, 1992.

19. Delft, V. 1999. “A Java extension with support for dimensions”. In Softw. Pract. Exper. 605-616. John Wiley & Sons, Inc., 1999

20. Denney, E. 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.

21. Dietl, W., S. Dietzel, M. Ernst, K. Muşlu, and T. Schiller. 2011. “Building and using pluggable type-checkers.” In Proceedings of the 33rd International Conference on Software Engineering (ICSE). Waikiki, Honolulu, 681-690. ACM Press, 2011.
22. Ekman, T. and G. Hedin. Pluggable checking and inferencing of non-null types for Java. J. Object Tech., 6(9):455--475, Oct. 2007.
23. Fang, C. F., Rob A. Rutenbar, Markus P üschel, and Tsuhan Chen. 2003. Toward efficient static analysis of finite-precision effects in DSP applications via affine arithmetic modeling. In Proceedings of the 40th annual Design Automation Conference (DAC '03). ACM, New York, NY, USA, 496-501.
24. Figueiredo, D. LH., and J. Stol. Self-validated numerical methods and applications. Brazilian Mathematics Colloquium monograph, IMPA, Rio de Janeiro, Brazil, July 1997
25. FlightGear. http://www.flightgear.org/.
26. Fraser, M. D., Kumar, K., and Vaishnavi, V. K. 1994. Strategies for incorporating formal specifications in software development. ACM 37, 10 (Oct. 1994), 74-86.
27. Geoconvertor. https://code.google.com/p/geoconvertor/.
28. Gay, G., S. Haiduc, A. Marcus, and T. Menzies. 2009. “On the use of relevance feedback in IR-based concept location.” In Proceedings of the 25th International Conference on Software Maintenance (ICSM). Edmonton, 2009, 351-360. IEEE Computer Society, 2009.

29. Grant, S., J. R. Cordy, and D. Skillicorn. 2008. “Automated concept location using independent component analysis.” In Proceedings of the 15th Working Conference on Reverse Engineering (WCRE), Antwerp, 2008, 138–142. IEEE Computer Society, 2008.

30. Grein, C., D. Kazakov, and F. Wilson. 2003. “A survey of physical unit handling techniques in ada”. In Proceedings of the 8th Ada-Europe international conference on Reliable software technologies (Ada-Europe), edited by Jean-Pierre Rosen and Alfred Strohmeier. 258-270. Springer, 2003.

31. Gunter, C. A., E. L. Gunter, M. Jackson, and P. Zave. 2000. “A Reference Model for Requirements and Specifications.” IEEE Softw. 17, 3, 37-43. IEEE, 2000.

32. Hangal, S., and M. S. Lam. 2009. “Automatic dimension inference and checking for object-oriented programs.” In Proceedings of the 31st International Conference on Software Engineering (ICSE). 155-165. IEEE Computer Society, 2009.

33. Hill, E., L. Pollock, and K. Vijay-Shanker. 2009. “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, 232–242. IEEE Computer Society, 2009.

34. Idani, A., Ledru, Y.: Object oriented concepts identification from formal B specifications. In: Formal Methods in Industrial Critical Applications, FMICS'04. (2004)

35. International Geomagnetic Reference Field.
http://www.ngdc.noaa.gov/IAGA/vmod/igrf.html

36. International System of Units, National Institute of Standards Technology, Washington, DC.

37. Jiang, L. and Z. Su. 2006. “Osprey: a practical type system for validating dimensional unit correctness of C programs.” In Proceedings of the 28th international conference on Software engineering (ICSE). Shanghai, 262-271. ACM Press, 2006.

38. Jackson, M. and P. Zave. 1993. "Domain Descriptions," In Proceedings of the Second IEEE International Symposium on Requirements Engineering. Los Alamitos, CA, 56-64. IEEE, 1993.

39. Jackson, M. and P. Zave. 1995. “Deriving specifications from requirements: an example.” In Proceedings of the 17th international conference on Software engineering (ICSE). 15-24. ACM Press, 1995.

40. Jackson, M. 2000. “Problem Frames: Analyzing and Structuring Software Development Problems.” Boston, Addison-Wesley Longman Publishing Co., Inc., 2000.

41. Johnson, T., B. Stanley and D. Steven. 2015. “Cyber-physical specification mismatch identification with dynamic analysis”. In Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS). 208-217. ACM press, 2015.

42. Jung, M., and F. Saglietti. 2005. “Supporting component and architectural re-usage by detection and tolerance of integration faults.” In Proceedings of the 9th IEEE International Symposium on High-Assurance Systems Engineering (HASE). 47-55. IEEE Computer Society, 2005

43. Kelpie flight planner for FlightGear. http://sourceforge.net/projects/fgflightplanner/

44. Kennedy, A. 1999. “Dimension types.” In Proceedings of the 5th European Symposium on Programming (ESOP): Lecture Notes in Computer Science volume 788. Springer, 1999.

45. Kim, S.K., Carrington, D.: Visualization of formal specifications. In: In Proceedings Sixth Asia Pacific Software Engineering Conference (ASPEC'99), IEEE Computer. Society Press, Los Alamitos, CA, USA(1999) 102-109

46. Kofod-Petersen, A., and M. Mikalsen. 2005. “Context: Representation and Reasoning: Representing and Reasoning about Context in a Mobile Environment”. Special issue of the Revue d’Intelligence Artificielle on "Applying Context-Management". 2005.

47. Lamsweerde, V. 2001. “A Goal-Oriented Requirements Engineering: A Guided Tour.” In Proceedings of the 5th IEEE International Symposium on Requirements Engineering. Toronto, 2001, 249–263. IEEE Computer Society, 2001.

48. Leveson, N. G. “Intent Specifications: An Approach to Building Human-Centered Specifications.” IEEE Transactions on Software Engineering, 26(1):15-35, January 2000.

49. Leveson, N.G. "Completeness in Formal Specification Language Design for Process Control Systems", Formal Methods in Software Practice, Portland, 2000.

50. Linderman, M. D., Matthew Ho, David L. Dill, Teresa H. Meng, and Garry P. Nolan. 2010. Towards program optimization through automated analysis of numerical precision. In Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization (CGO '10). ACM, New York, NY, USA, 230-237.

51. Marcus, A., V. Rajlich, J. Buchta, M. Petrenko, and A. Sergeyev. 2005. "Static Techniques for Concept Location in Object-Oriented Code." In Proceedings of the 13th International Workshop on Program Comprehension (IWPC). 33-42. IEEE Computer Society, 2005.

52. Markstrum, S., D. Marino, M. Esquivel, T. Millstein, C. Andreae, and J. Noble. 2010. "JavaCOP: Declarative pluggable types for java." In ACM Trans. Program. Lang. Syst. 1-37. ACM press, 2010.

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

54. McKinna, J. 2006. "Why dependent types matter". In Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). 1-1. ACM Press, 2006.

55. Meng, N.J., Diane Kelly, and Thomas R. Dean. 2011. Towards the profiling of scientific software for accuracy. In Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '11), Marin Litoiu, Eleni Stroulia, and Stephen MacKay (Eds.). IBM Corp., Riverton, NJ, USA, 257-271.

56. Milanova, A. and W. Huang. 2012. "Inference and checking of context-sensitive pluggable types." In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE). Article 26, 4 pages. ACM press, 2012

57. Miller, S. P., and A. C. Tribble. 2001. “Extending the four-variable model to bridge the system-software gap.” In Proceedings of the 20th digital Avionics System Conferences. Daytona Beach, 14-18. IEEE Computer Society, 2001.

58. OpenMap. http://openmap-java.org/

59. OpenMap developer guide. http://openmap.bbn.com/developer_hints.html

60. OpenMap in Github. https://github.com/openmap-java/openmap

61. OpenMap in Twitter. https://twitter.com/openmap

62. OpenMap tutorial. https://www.javacodegeeks.com/2015/10/openmap-tutorial-part-1.html

63. Ou, X., G. Tan, Y. Mandelbaum, and D. Walker. 2004. “Dynamic typing with dependent types.” In Exploring new frontiers of theoretical informatics. 437-450. Springer, 2004.

64. Özcan, M. B., P. W. Parry, I. C. Morrey, and Jawed I. A. Siddiqi. 1998. Requirements Validation based on the Visualization of Executable Formal Specifications. In Proceedings of the 22nd International Computer Software and Applications Conference (COMPSAC '98). IEEE Computer Society, Washington, DC, USA, 381-386.

65. Parnas, D. L. and L. Madey. 1995. “Functional documents for computer systems.” In Sci. Comput. Program. 41-61. Amsterdam: Elsevier North-Holland, Inc., 1995.

66. Papi, M., M. Ali, C. T. Luis, J. H. Perkins, and M. D. Ernst. 2008. “Practical pluggable types for Java.” In Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), edited by B. G. Ryder and A. Zeller. 201–212. ACM Press, 2008.

67. Perttunen, M., J. Riekki, and O. Lassila. 2009. “Context representation and reasoning in pervasive computing: a review”. In International Journal of Multimedia and Ubiquitous Engineering. Vol. 4, No. 4, October, 1-28. 2009.

68. Petty, G. 2001. “Automated computation and consistency checking of physical dimensions and units in scientific programs.” In Softw. Pract. Exper. 1067-1076. New York: John Wiley & Sons, Inc., 2001.

69. Poshyvanyk, D. and A. Marcus. 2007. “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), Banff, Alberta, 2007, 37-48. IEEE Computer Society, 2007.

70. Rajlich, V., and N. Wilde. 2002. “The Role of Concepts in Program Comprehension.” In Proceedings of the 10th International Workshop on Program Comprehension (IWPC). 271-278. IEEE Computer Society, 2002.

71. Ratiu, D. and F. Deissenboeck. 2007. “From Reality to Programs and (Not Quite) Back Again.” In Proceedings of the 15th IEEE International Conference on Program Comprehension (ICPC). Banff, Alberta, 91-102. IEEE Computer Society, 2007.

72. Ratiu, D., M. Feilkas, and J. Jurjens. 2008. “Extracting Domain Ontologies from Domain Specific APIs.” In Proceedings of the 2008 12th European Conference on Software Maintenance and Reengineering (CSMR). Athens, 203-212. IEEE Computer Society, 2008.

73. Razali, R., Snook, C. F., Poppleton, M. R., Garratt, P. W. and Walters, R. J. Experimental Comparison of the Comprehensibility of a UML-based Formal Specification versus a Textual One. In: 11th International Conference on Evaluation and Assessment in Software Engineering (EASE'07), 2-3 April 2007, Keele, Staffordshire, UK. pp. 1-11.

74. Ruiz-Casado, M., E. Alfonseca, and P. Castells. 2005. "Automatic extraction of semantic relationships for wordnet by means of pattern learning from Wikipedia." In Proceedings of the 10th international conference on Natural Language Processing and Information Systems (NLDB), edited by A. Montoyo, R. Muńoz, and E. Métais. 67-79. Springer, 2005.

75. Snook, C., M. Butler. 2006. UML-B: Formal modeling and design aided by UML, ACM Transactions on Software Engineering and Methodology (TOSEM), v.15 n.1, p.92-122, January 2006.

76. Syme. D., 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. 2012. "Strongly-typed language support for internet-scale information sources." Technical Report MSR-TR-2012-101, Microsoft Research, September 2012.

77. T. Taylor, B. Stanley and D. Steven. 2015. "Cyber-physical specification mismatch identification with dynamic analysis". In Proceedings of the ACM/IEEE Sixth International Conference on Cyber-Physical Systems (ICCPS). 208-217. ACM press, 2015.

78. Wordnet. http://wordnet.princeton.edu

79. World Magnetic Model. http://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml

80. Wu, B., Jianwen Zhu, and F. N. Najm. 2004. Dynamic range estimation for nonlinear systems. InProceedings of the 2004 IEEE/ACM International conference on Computer-aided design (ICCAD '04). IEEE Computer Society, Washington, DC, USA, 660-667.

81. Whitaker, R., W. Thompson, J. Berger, B. Fischhof, M. Goodchild, M. Hegarty, C. Jermaine, K. McKinley, A. Pang, J. Wendelberger. 2015. “Quantification, Communication, and Interpretation of Uncertainty in Simulation and Data Science”. CRA Computing Computing Community Consortuim (CCC), pp. 1-22, September, 2015.

82. Wyk, E. and Y. Mali. 2007. “Adding Dimension Analysis to Java as a Composable Language Extension.” In Generative and Transformational Techniques in Software Engineering II. Lecture Notes In Computer Science, Vol. 5235. 442-456. Springer, 2007.

83. Xiang, J., J. Knight, and K. Sullivan. 2015. “Real-world Types and Their Application”. In Proceedings of the 34th International Conference on Computer Safety, Reliability and Security (SAFECOMP). Delft, 2015, 471-484. Springer, 2015.

84. Xiang, J., J. Knight and K. Sullivan, 2016. "Synthesis of Logic Interpretations," In Proceedings of the 17th International Symposium on High Assurance Systems Engineering (HASE), Orlando, FL, 2016, pp. 114-121.

85. Yang, L., H. Yang, and W. Chu. 2000. “Generating linkage between source code and evolvable domain knowledge for the ease of software evolution.” In Proceedings International Symposium on Principles of Software Evolution, Kanazawa, 2000. 196-205. IEEE Computer Society, 2000

86. Yin, R. K. 2009. Case study research: Design and methods. Thousand Oaks, Calif: Sage Publications.

87. Eclipse Plug-in Development. http://www.vogella.com/tutorials/EclipsePlugin/article.html

88. Zimmerman, M.K., K. Lundqvist, N.G. Leveson, “Investigating the Readability of Formal Requirements Specification languages”, International Conference on Software Engineering, May 2002.