CHAPTER 3

A Rigorous Definition of Cyber-Physical Systems

John Knight, Jian Xiang, and Kevin Sullivan

CONTENTS

3.1 INTRODUCTION

In this chapter, we present a rigorous definition of Cyber-Physical Systems (CPS) based on the three essential components of such a system: (1) a computing platform, (2) a set of physical entities with which the computing platform interacts, and (3) a connection between the first two components. These components seem familiar, and the third probably seems trivial. But the third component 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 (for example, whether an integer input of "73" that is meant to be a heading is measured relative to true North or magnetic North and whether an output with logic value of "1" is meant to cause a pump to start or stop).

Thus, correctly and precisely defining the connection between the computing platform and the physical entities is crucial, because defects in the definition can lead to undesired and possibly dangerous behavior of the CPS.

We characterize the details of the three components precisely, placing particular emphasis on the connection between the computing platform and the physical entities. We use these characterizations to construct a rigorous definition of CPS. From this definition, we derive various validation, verification, and specification properties that can serve to support the design, analysis, and assurance of CPS.

3.1.1 Cyber and Physical

The fundamental concept underlying CPS is that a logic function defined by a digital computer (cyber) interacts with the real world (physical) in order to sense and affect entities in the real world. The details of the implementation of CPS, such as the use of particular sensors and/or actuators, do not affect this underlying concept. Similarly, the functionality of a system, whether the system is in the health-care, transportation, or some other domain, does not affect the underlying concept. In other words, the role of CPS is to implement an explicit *interaction* between the two parts of the cyber-physical combination.

With these informal notions in mind, an intuitive definition of CPS might be:

> *Cyber-physical system*: A system that includes a discrete logic function in its design such that either: (a) the state of the logic function could be changed by a change in the state of the real world, or (b) the state of the real world could be changed by a change in the state of the logic function, or (c) both.

Based on this intuitive definition, most systems that involve a computer are CPS. And, in the most general sense, this is the case. The only computer systems that are not CPS are those that do not affect and are not affected by the real world, that is, software that is pure logic—a compiler, for example. Such software is unusual.

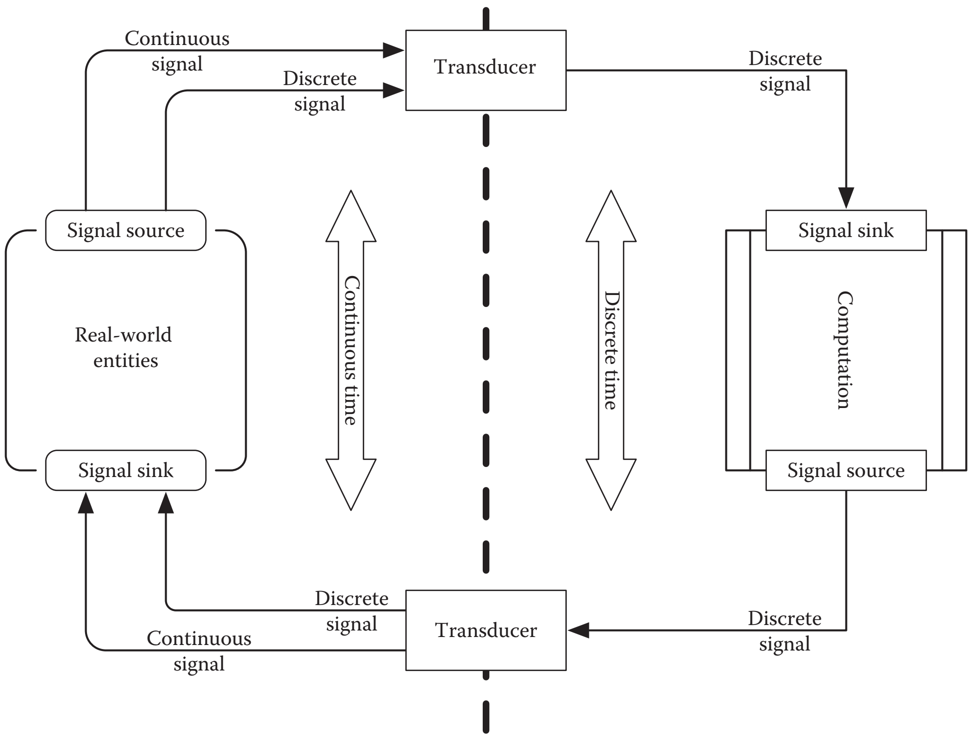

FIGURE 3.1 The basic elements of CPS.

For practical purposes, a rigorous definition and the associated ability to undertake system fault detection is more important for certain types of CPS than others. Systems for which a rigorous definition is likely to be important are those that have serious consequences of failure, such as medical systems that operate a control loop aiding a patient, automobile engine controllers that execute complex algorithms to manage pollution control, and nuclear shutdown systems that include complex decision-making algorithms. In such systems, failure could lead to equipment damage, human injury or death, or environmental pollution, so required levels of assurance of correct operation are necessarily high.

In practice, CPS have the general form shown in Figure 3.1. An important aspect of CPS 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 affects a "computation" that produces digital signals designed to affect real-world entities. Transducers process these digital signals to produce continuous signals that are inputs to the real world.

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

Since the implementation of CPS logic functions are almost exclusively in software, we will use the term “software” in the remainder of this chapter. Nevertheless, we note that the definition presented in this chapter also applies to CPS in which the logic functions are implemented in hardware or in FPGAs.

The importance of the notion of interaction between real-world entities and software can be seen when one notes that many CPS are safety critical, and assurance of their correct operation depends on the interaction between real-world entities and software being complete and correct. Unless the interaction is precisely that which is intended *and* known to be so, doubt in the effects of such systems is inevitable.

Software failing to interact with real-world entities with sufficient fidelity 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 [1]. 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 [2]. In both of these examples, real-world entities were affected by operations defined in software in ways that made no real-world sense, that is, the systems failed because the essential interaction was improperly defined.

The software in CPS should respect invariants that derive from the real world the system senses, models, and affects. For example, software that computes physical quantities should respect measurement units and physical dimensions, for example, as defined by the ISO/IEC 80000 standards. Frequently, however, the interaction between real-world entities and the system’s software is undocumented or is documented incompletely, informally, or implicitly.

Even more frequently, the completeness and correctness of the interaction is not checked comprehensively. Invariants derived from the real world are stated and enforced either in ad hoc ways or not at all. Crucial, non-trivial relationships between software representations and real-world entities remain under-specified, and programs treat machine-world representations 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 derived from the real world. The results are software and system failuresas well as adverse downstream consequences.

3.1.2 Logic Interpretation

The need to define the interaction between the software in CPS and entities in the real world explicitly arises because software is a logic function with no *interpretation*. The notations that are used for defining software are formal languages. High-level languages, assembly languages, and machine languages are *all* formal and, as formal languages, have no inherent real-world meaning, that is, the logic is *uninterpreted*.

For a 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

(a)

```
/* 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;}
```

(b)

```
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;}
```

FIGURE 3.2 Source code with (a) and without (b) an ad hoc interpretation.

to constraints and invariants that derive from the real world, such as the laws of physics. The logic must conform to these constraints and invariants to provide a rich opportunity for error checking of the software.

Surprisingly, the interpretation of a software system is *always* present in practice but usually documented in an ad hoc, informal, and sometimes implicit manner using casual techniques, such as "descriptive" comments, "meaningful" identifiers, and design documents. Figure 3.2a 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 Figure 3.2b. 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 completely 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.

As noted previously, for CPS 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 precisely and comprehensively documented. Without such documentation, doubt in the correctness of the effects of such systems is inevitable.

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, allowing 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 case studies, such analyses revealed both unrecognized faults and faults that had been reported as bugs in real systems after deployment [3]. Examples of this new fault detection capability are given in Section 3.3.7. Q1

3.2 CONCEPT OF INTERPRETATION

The role of interpretation can be thought of as an enhanced version of *abstraction functions* in logic that map 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 defines the real-world meaning of an element of logic. For example, an integer variable in an avionics program might be used to represent the altitude of an aircraft. Within the logic of the software, the variable is merely an integer. The role of the interpretation is to reveal everything about the *actual* altitude of the aircraft.

Figure 3.3 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). 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. The introduction of faults due to unrecognized discrepancies resulting from violating rules inherited from the real world can be significantly reduced or even eliminated.

As an example, consider again 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 CPS, such as an aircraft's autopilot, compute using altitude. A partial list of those attributes include:

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

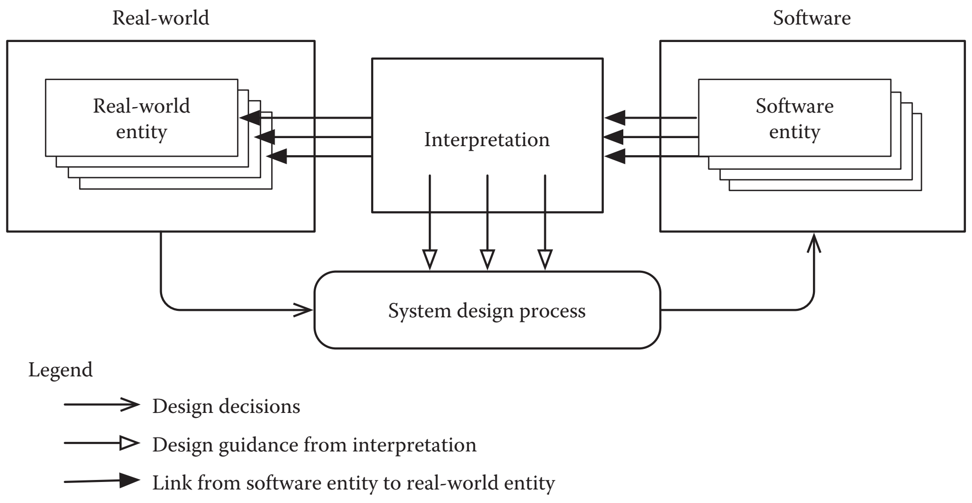

FIGURE 3.3 The role of interpretation in CPS.

- *Physical dimensions*. Altitude has the fundamental physical dimension of length.
- *Frame of reference*. Altitude is defined based on an origin and the direction, that is, a frame of reference.
- *Sensor performance*. A sensor, that is, 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, that is, the altitude of the aircraft "now."

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

- *Mixed measurement units*. Expressions that 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 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 dimensions length/time) is 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.

The necessity of defining an interpretation for software explicitly indicates the need for a composite entity that includes both software and its interpretation. A candidate entity is the *interpreted formalism*.

3.3 INTERPRETED FORMALISMS

Having established the role and value of an interpretation of the logic in CPS, we turn to the structure needed to incorporate an explicit interpretation into the engineering artifacts that are needed for CPS. The structure that we introduce is called an *interpreted formalism* and has the form shown in Figure 3.4.

The logic in an interpreted formalism is defined in whatever manner is appropriate for the system of interest, that is, 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.

Q1 As discussed in Section 3.3.1, in practice an interpretation is always present for any CPS. The interpreted formalism combines the interpretation and the software in a manner that makes the interpretation a first-class entity.

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

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,

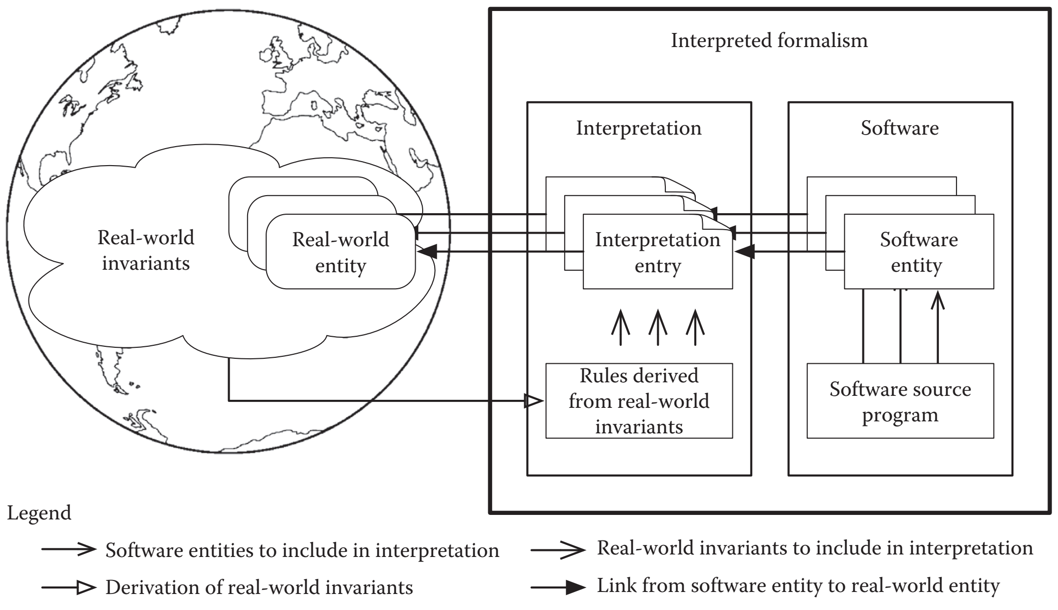
Interpreted formalism
Interpretation
Software
Real-world invariants
Real-world entity
Interpretation entry
Software entity
Rules derived from real-world invariants
Software source program
Legend
Software entities to include in interpretation
Real-world invariants to include in interpretation
Derivation of real-world invariants
Link from software entity to real-world entity

FIGURE 3.4 An interpreted formalism combines logic with an explicit interpretation.

criticality, and consequences of failure of CPS increasing, the shift needs to be seriously considered.

3.4 RIGOROUS DEFINITION OF CYBER-PHYSICAL SYSTEMS

With the concept, role, and importance of interpretation defined and with the interpreted formalism structure in place, we have the elements needed to define CPS.

CPS are triple {Rw, Hw, If}, where:

- Rw is a set of entities in the real world with which the system interacts.
- Hw is 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 is an interpreted formalism that combines the definitions of: (a) the software of the system, and (b) the interpretation of that software.

For the most part, Rw and Hw are familiar. The set Rw can be enumerated based on the problem to be solved by theCPS. In the case of a drug-infusion pump, for example, the pump has to interact with the patient's physiology; the patient and medical professionals as "users" through a human-machine interface; and 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 a new concept, and so we focus on the details of the interpreted formalism in the remainder of this chapter.

With this definition, we need to modify the basic elements of CPS shown in Figure 3.1. Figure 3.5 shows the original elements but 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.

3.5 STRUCTURE OF AN 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 section, we present a preliminary design. In later sections we present an implementation of the design and the results of experiments undertaken to assess the feasibility and performance of the concept.

The design we present is based on the concept of a *real-world type* system [3]. The design of the interpretation is a set of real-world types defined within the framework of a real-world type system.

3.5.1 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 an object in the real world of that type can have and the operations

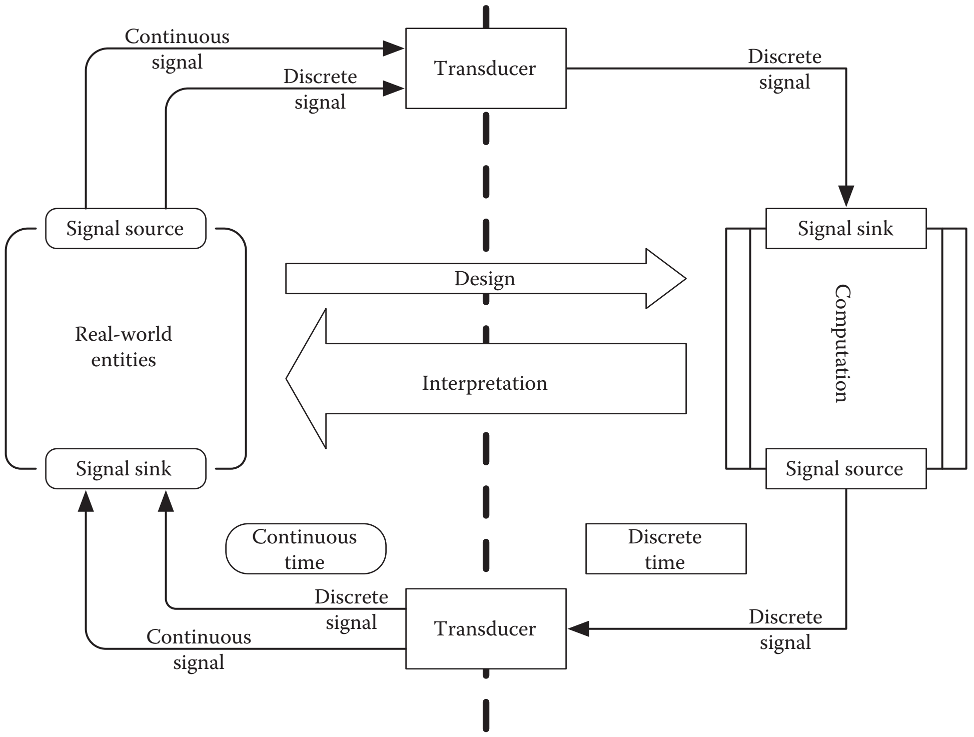

FIGURE 3.5 The basic elements of CPS based on the rigorous definition.

Q1 in which the object can engage. The real-world type for the physical concept of an aircraft's altitude would include all the details noted in Section 3.3.2.

More generally, a real-world type *system* documents:

- The real-world attributes associated with the type. These attributes are properties possessed by an entity of the associated type. For the aircraft altitude example discussed earlier, the attributes include measurement units, physical dimensions, sensor precision, sensed value delay, frame of reference, and so on.
- The type rules that define allowable operations on entities of the various types. The type rules derive from what constitutes meaningful operations in the real world. The examples given earlier for aircraft altitude (measurement unit compatibility, etc.) will all be included. Rules that derive from application-specific details will be included also. 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 be used. Type rules can be established to identify incorrect uses of altitude derived from barometric pressure.
- The machine representations for entities of the real-world types. This allows analysis of the implication of limitations within that representation. For example, a particular precision might be defined, but analysis of the computations in which that item is used

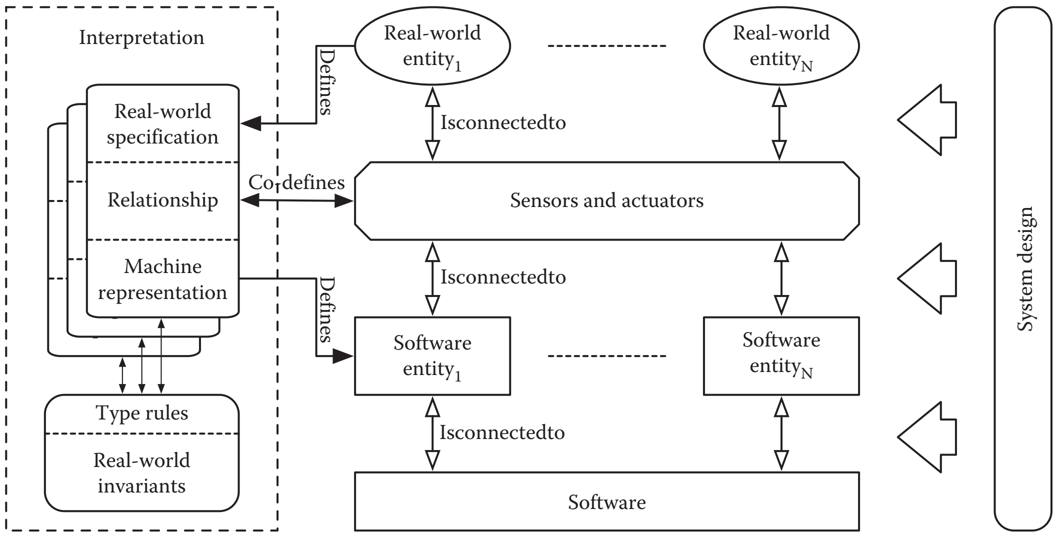

FIGURE 3.6 Real-world type system.

or expected range values derived from the associated real-world entity might reveal the possibility of overflow.

- The relations between real-world entities and their machine representation. The relation defines the mapping of values to meaning in the real world. In the aircraft altitude example, the machine representation might be feet, meters, or flight level (altitude in hundreds of feet). In addition, for engineering reasons, values might be offset by a fixed increment.
- An explication. All of the above provide reference information. 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.

Our preliminary structure for an explicit interpretation is a set of real-world types together with a set of bindings between entities in the real world and entities in the logic.

A real-world type system and its connections to an application system of interest are shown in Figure 3.6. 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.

Real-world types are intended to facilitate design in computer 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. Within an interpretation defined using real-world types is a set of type rules (see Figure 3.6) that are derived from real-world invariants. For example, rules about unit conversion, valid and invalid expressions using variables with units, and types resulting from expressions have to be defined in their entirety to enable suitable checking.

3.5.2 Example Real-World Type

An example of a real-world type is a point in 3D space. 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 needs to document all of 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 Figure 3.7. 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.

For the types of the three fields, 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`. All three dimensions are actually distances within the coordinate system. As such, the real-world physical dimensions are length. In this example, the encoding of physical dimensions is based on the standard set of seven dimensions from physics: length, mass, time, electric current, temperature, amount of substance, and luminosity [52]. For convenience in many CPS, we have added angle as a basic dimension, bringing the total to eight. In the real-world type definitions, the actual physical dimensions of an entity are specified as a vector of powers of these dimensions. Thus, for example, velocity, which is length per unit time and has physical dimensions LT^{-1}, is encoded as `(0,1,-1,0,0,0,0,0)`.

```
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 E longitude
      z_axis_orientn        : positive northward
  Representation
    machine_semantics
      representation        : record structure - (float, float, float)
  Relationship              : <null>
```

FIGURE 3.7 A real-world type definition for horizontal axis (a) and for vertical axis (b) for a coordinate in a frame of reference.

(a)
```
horizontal_cartesian_axis:
  Specification
    explication               : <text>
    real_world_semantics
      linear_units            : mile
      physical dimensions     : (1,0,0,0,0,0,0,0)
      technology              : GPS
      geometry_plane          : horizontal
  Representation
    machine_semantics
      representation          : float
      mutable                 : no
  Relationship
      value_error < delta1 and delay < tau1
```

(b)
```
vertical_cartesian_axis:
  Specification
    explication               : <text>
    real_world_semantics
      linear_units            : feet
      physical dimensions     : (1,0,0,0,0,0,0,0)
      technology              : radar
      geometry_plane          : vertical
      offset_origin           : mean sea level
  Representation
    machine_semantics
      representation          : float
      mutable                 : no
  Relationship
      value_error< delta2 and delay < tau2
```

FIGURE 3.8 Real-world type definitions for lengths in a coordinate system.

For x and y, we define the type shown in Figure 3.8a. In this example, altitude 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 Figure 3.8b. 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.

3.5.3 Real-World Type Rules

Given the real-world type definition for a coordinate, example type rules that could be stated, checked, and indicate operations that are probably erroneous if violated include:

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

FIGURE 3.9 Example type rule definition.

- 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 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`, for example, for calculating the altitude difference between two points in the same Cartesian coordinate system. The definition is illustrated in Figure 3.9.

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.

3.6 AN IMPLEMENTATION OF INTERPRETATIONS FOR JAVA

3.6.1 Design of The Implementation

If explicit interpretations are to be used in the development of realistic CPS, an approach to integrating them into widely used languages and development methods is needed. We have developed a prototype that implements interpretations as sets of real-world types for Java. The implementation supports:

- The definition of a set of real-world types for a Java program of interest.
- The definition of set real-world type rules by system experts based on real-world and application invariants.
- Creation of bindings between the real-world type definitions and entities in the Java source program.
- Static type checking of the Java program based on the set of real-world type rules.
- Synthesis of assertions as Java fragments that can be inserted into the subject program to implement runtime checking of type rules that cannot be checked statically.
- Synthesis of a checklist of locations in the subject program at which human inspection is required to check type rules that cannot be checked statically or dynamically.

To illustrate the analysis capabilities that are possible with an explicit interpretation, we present case studies in Section 3.3.7 in which real-world types were applied to a set of pertinent projects, including a project with 14,000 LOC. The real-world type checking revealed both unreported faults and faults that had previously been reported as bugs. Q1

An important design choice made for the implementation was that the system should operate without requiring changes to the subject Java program. This choice provides four major advantages:

- The interpretation does not obscure the basic structure of the Java program.
- An interpretation can be added to an existing Java program without having to modify (and possibly break) the original source text.
- An interpretation can be added to a Java program asynchronously, thereby not impeding development of the Java program itself and permitting an interpretation to be added to an existing legacy program.
- An interpretation can be added incrementally so that developers can experiment with the technology to determine its efficacy for their application.

Motivated by this design choice, the implementation operates separately from the compiler via its own user interface.

The structure of the Java prototype system is shown in Figure 3.10. The subject Java source program is shown on the left toward the top of the figure, and the interpretation is shown on the right. The development of the interpreted formalism, the combination of the source of the Java program and the interpretation, is shown at the top of the figure.

3.6.2 Use of The Implementation

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

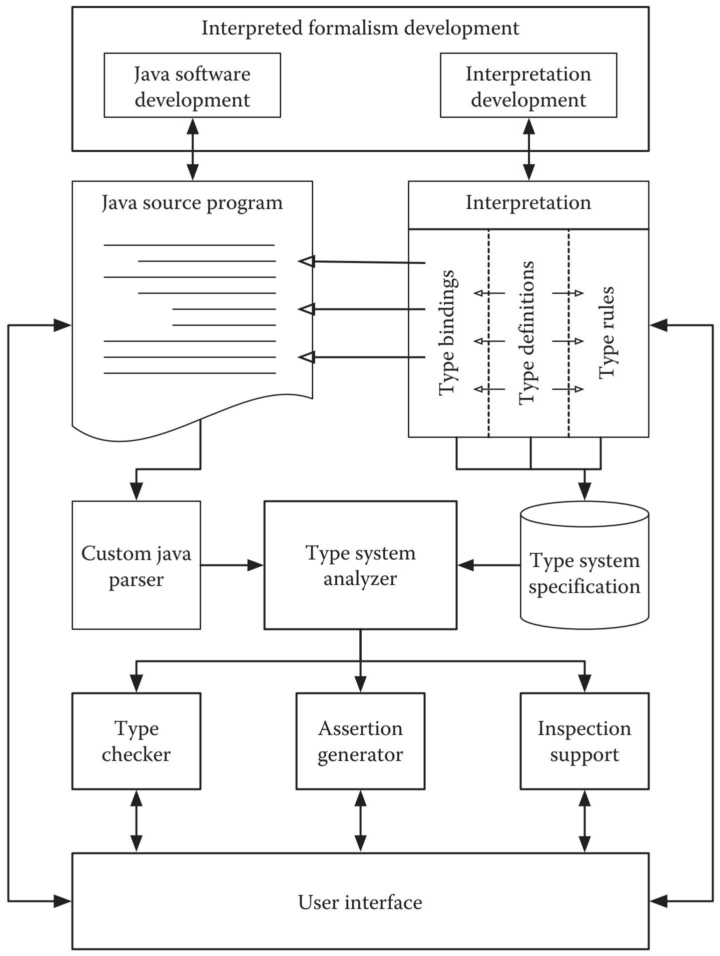

FIGURE 3.10 Structure of Java prototype.

- 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.

- 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, for example, all bindings, binding of a given Java entity, all Java entities bound to a particular real-world type, etc.

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, which supports three types of analyses:

- *Type checking*. The type rules define invariants that are derived from the real world and from the application. All of the type rules are checked throughout the Java program and diagnostics displayed for the user identifying violations.
- *Assertion generation*. Some type rules cannot be checked statically. Real-world invariants often 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 to check them at runtime. The analysis system synthesizes assertions as Java code fragments and optionally will insert them (with explanatory comments) into the Java source program.
- *Inspection checklist generation*. Some aspects of real-world type checking cannot be performed either statically or dynamically. For these type rules, the analysis develops simple checklists that can be used to guide an inspection. An example of this type of rule is that the maximum difference between the value of a physical entity available to the system's software should be less within some prescribed tolerance of the actual value. An aircraft's altitude as represented in an autopilot might be required to be within some fraction of the aircraft's true altitude in order to establish stability in an autopilot. The difficulty that arises in checking derives from the sensor inaccuracy, sensor sampling delay, delay in processing of the sampled value prior to its being available to the autopilot, and so on. The only reasonable way to have confidence in the required type rule is to conduct human inspection. The analysis system synthesizes inspection checklists based on the use of identified relevant software entities so as to provide focus for human inspectors.

3.6.3 Type Conversion

An important issue in the type system is type *conversion*. For example, a length 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 stand-along assignment statements, within other expressions, as expressions stated as actual parameters, as return values, and so on.

The analysis system deals with explicit type conversion simply by including type 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 implementation by requiring that the programmer investigate each diagnosed type error and mark implicit type conversions as such. Thus, diagnostics of which the type system was unaware will be generated for type conversion because the mismatch appears to be a violation of the type rules. In those cases, the programmer suppresses the diagnostic by indicating that there is an expected implicit type conversion. By doing so, the programmer indicates that the diagnostic has been investigated and the code found to be as desired.

3.6.4 Typed Java Entities

The Java entities that require binding to real-world types are: (a) local variables, (b) fields in classes, (c) method parameters, (d) method return parameters, and (e) class instances. In order to make the development of the prototype tractable, the current version imposes some restrictions on the use of real-world types in Java, specifically:

- *Fields*. Fields in classes are assumed to be monomorphic, that is, a field in a class is assumed to have the same real-world type in all class instances. Fields are bound to real-world types inside the class declaration body.
- *Class instances*. Different instances of a class might have different real-world meanings and so the type definition 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, so the two instances need to be distinguished.
- *Method return parameters*. If a particular method is not bound to a real-world type, the analysis treats the method as polymorphic. For a polymorphic method, at each invocation site, the type checker examines all the expressions in the method declaration body and determines the real-world type for the return statement. That will be the real-world type for the method invocation ultimately. If the method contains multiple return statements, the real-world type for the return value will be the one with no type errors. Also, if real-world types for return statements are inconsistent, the type checker issues a warning message.
- *Arrays*. Since individual array elements cannot be typed with real-world types, all objects inside an array are treated as having the same real-world type.

Type checking can only rely on limited type inference because many of the result types of operations are defined by real-world circumstances. For example, adding a variable of type angle measured in degrees to a variable of type latitude also measured in degrees (commonly but not necessarily) yields a result of type latitude because the purpose of the addition is (probably but necessarily) to compute a new heading in some sort of navigation system. Without a comprehensive type inference system, checking type rules leads to three difficulties:

- *Determining the types of constants*. Variables are given a real-world type when declared, but constants are used as needed. Constants are dealt with simply by associating each one with a hidden variable and associating a real-world type with the hidden variable.
- *Function-return types*. Defining a function essentially introduces a new type rule. Function return types are dealt with by associating a real-world type with each function that returns a value with a real-world type.
- *Defining the types of compound objects*. Classes introduce the possibility of nesting real-world types because the class might have a real-world type and the fields within the class might have real-world types. In that case, the type of a qualified name is the union of the attributes 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();`.

3.7 PERFORMANCE/EXPERIMENTAL RESULTS

Informal arguments, as presented in this chapter, suggest that the rigorous definition of CPS and the associated analysis might have value, but that is insufficient reason to change current development methods. Recall from Section 3.3.3 that the change is of the magnitude of a paradigm shift. Empirical evidence to show feasibility and utility is essential. Q1

To gain some insight into the feasibility and utility of interpreted formalisms, we conducted a two-part study in which we developed interpretations structured as sets of real-world types for several open-source projects with which we have no association. In the first part, a complete set of real-world types were defined for a project called the Kelpie Flight Planner [54]. Various elements of the software were given real-world types, a set of type rules were defined, and type checking was performed.

In the second part, we reused real-world types and type rules created in part one on a set of projects that access the same real-world entities. For these projects, type checking has only been applied to pertinent packages and files to detect errors.

3.7.1 Kelpie Flight Planner

The Kelpie Flight Planner is an open-source Java project based on Flightgear [5]. The program uses the airport and navaid databases of Flightgear to determine routes between airports based on user inputs. The program is 13,884 lines long, is organized as 10 packages, and is contained in 126 source files.

A critical element of the data used by the Kelpie Flight Planner in modeling aircraft movement is the *velocity surface,* 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 details of the velocity surface are shown in Figure 3.11.

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

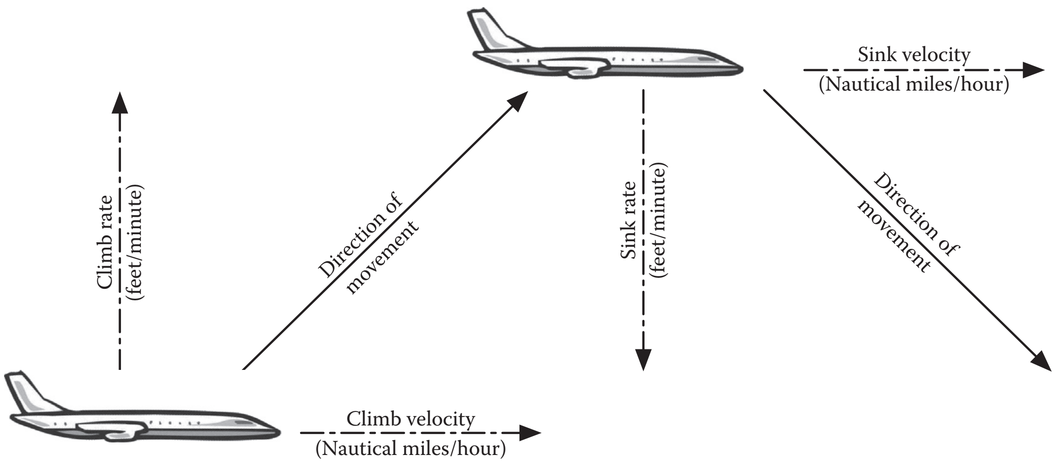

FIGURE 3.11 The velocity surface.

example, the International Geomagnetic Reference Field (IGRF) [6] and the World Magnetic Model (WMM) [7]. In order to undertake useful calculations, programs like the Kelpie Flight Planner have to operate with a model of the Earth's geometry, and details of the model need to be included in the real-world type system in order to allow appropriate checking.

For the Kelpie Flight Planner, 35 different real-world types were defined along with 97 type rules. The total number of type bindings for the project was 255. For illustration, we summarize three of six faults that were identified by the real-world type analysis. None of these faults had been reported in the project error log.

The program source code containing the first fault 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 source code containing the second fault 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 source code containing the third fault is:

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

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

3.7.2 Other Java Applications

We reused the real-world types and type rules created for the Kelpie Flight Planner project to check packages and source files in other applications with pertinent functions. We chose applications for which a log was available for defects reported from the field after release. We include examples here for illustration.

OpenMap is a Java Beans-based tool kit for building applications and applets that access geographic information [8]. The code for the first fault we detected is:

```
lon2 = ll2.getY();.
```

The variable `lon2` is a longitude, but the method `getY()` returns a latitude.

The code for the second fault is:

```
double[] llpoints = GreatCircle.greatCircle(startLLP.getY(),
   startLLP.getX(), endLLP.getY(), endLLP.getX(), numPoints, true);.
```

The arguments to `greatCircle()` should be in radians, but in this call the arguments are measured in degrees.

Geoconvertor is a Java API that converts latitude and longitude to points in the Universal Transverse Mercator coordinate system [9]. The faulty code detected is:

```
if (datum.name().equals("WGS84")) {
      e1=eccentricityOne(WGS84.MAJOR_AXIS, SAD69.MINOR_AXIS);
      e2=eccentricityTwo(WGS84.MAJOR_AXIS, SAD69.MINOR_AXIS)};}.
```

The constructors `eccentricityOne()` and `eccentricityTwo()` each expect two arguments of the same Earth model, SAD69 for the first and WGS84 for the second. The code has the argument Earth models confused.

3.8 RELATED WORK

A variety of previous research results are relevant to the ideas presented in this chapter. In this section, we summarize the results in various related areas and provide references for more detailed information.

3.8.1 Modeling The Relationships between The Real World And The Machine World

Interpretation of logic is about binding the real world to the machine world. Research efforts on requirements and specification have also modeled the connections between the real world and the machine world [10–14,47].

Jung and Saglietti defined a language for describing interfaces between system components to support fault detection in component interfaces [53]. 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.

Johnson et al. developed an approach to detection of undocumented assumptions in Simulink/Stateflow models in which traces are used to develop system invariants [4]. 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 [15]. The views are derived from various models and a base architecture for subject CPS. The approach is based on typed graph models of the subject system, and consistency is defined by morphisms from views to the base architecture.

The four-variable model proposed by Parnas and Madey introduces relationships that the authors labeled IN and OUT [14]. The IN and OUT relationships define the connections between mathematical variables available to the software and the environmental variables in the real world. The relationship between real-world entities and the machine world is described mathematically. The hardware and software requirements are intertwined in the REQ relation. Miller and Tribble introduced an extension to the four-variable model that isolates the virtual versions of the monitored and controlled values in subsystems [13]. The relationships emerge as relations between virtual and real versions of variables.

The work of Zave and Jackson characterized phenomena of interest to the system and separated world phenomena from machine phenomena [11,12,16]. The reference model of Gunter et al. gave a detailed explanation of different classes of phenomena and the relationship between environment and system [10]. These results model the picture of the connection between machines and the real world. In contrast to these results, real-world types provide a comprehensive set of real-world semantic attributes and emphasize imposing constraints on software inherited from the real world.

3.8.2 Conventional Types

Conventional types as used in formal languages are closely related to the machine context, and as a result they cannot comprehensively describe real-world information. For example, most real-world semantic attributes are associated with compile-time values; such attributes and associated 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, real-world constraints are incompletely documented and are thereby enforced in conventional type systems in ad hoc ways or not at all.

3.8.3 Enhanced 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.

Powerful extensions to the basic notion of type have been developed, in particular in the form of pluggable type systems [17–20]. Pluggable type systems [20] enhance the built-in type systems in applicable formal languages and provide support for additional checking capabilities. The Checker framework [17,20] and JAVACOP [18] 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 [21,48,50,56] 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 untrusted programs [22]. Dependent type systems, such as Coq [23] and Agda [24], 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.

3.8.4 Real-World Constraints

An interpretation based on real-world types supports analyzing real-world constraints. As special kinds of real-world constraints, dimensional analysis and unit checking have been explored in many programming languages [25–27,51]. Previous research focused on extending programming languages to allow checking that 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 [26,28–32]. Nevertheless, these efforts are limited to basic rules derived from dimensions or combinations of entities with different units.

3.8.5 Improved Program Understanding

Real-world semantic attributes in real-world types can improve the understanding of programs. Other researchers have attempted to improve program understanding by linking structured semantic information in real-world contexts to programs [33–35,49]. 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 [33,34].

3.8.6 Concept Location

Concept location, that is, identifying parts of a software system that implement some aspect of the problem, is related to the idea of mapping real-world entities to software and has been studied extensively [35–40,55,57]. 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 efforts by producing a concept lattice using the most relevant attributes (terms) selected from the top-ranked documents (methods) [41]. Grant et al. proposed approaches that identify statistically independent signals that could lead to concepts [39].

The use of parts of speech of terms in identifiers has been investigated as a means of extracting information from the source code. Binkley et al. improved identifier name tagging using templates and defined rules to improve the structure of field names [42]. Hill et al. generated noun, verb, and prepositional phrases from the signatures of program elements [40].

Parts of speech have also been used to extract domain models, such as ontologies. Abebe and Tonella [36,37] 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 [33,34]. Other researchers have used WordNet to automatically extract semantics and relationships between the semantics [43,44].

3.8.7 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 [45,46]. 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 is 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.

3.9 SUMMARY

In this chapter, we have presented a rigorous definition of CPS based on the idea that such a system is composed of a computing platform, a set of physical entities, and a connection between the two. The emphasis in the definition is on a precise and complete definition of the connection between the physical entities and the computing platform.

The connection has been defined using the notion of logical interpretation. The interpretation precisely defines the meaning of items in the logic in terms of physical entities and their associated properties. Many important invariants can be derived from physical, that is, real-world, entities in a comprehensive and systematic way. Defining the connection between physical entities and the computing platform in this way enables a new class of fault detection mechanisms for the logic, that is, for CPS software.

REFERENCES

1. Mars Climate Orbiter Mishap Investigation Board Phase I Report, 1999. National Aeronautics and Space Administration, Washington DC, November 10, 1999.
2. Bergin, C. and P. Harding. 2013. Cygnus delays ISS berthing following GPS discrepancy. http://www.nasaspaceflight.com/2013/09/cygnus-cots-graduation-iss-berthing/
3. 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.

4. 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. Q2
5. FlightGear. http://www.flightgear.org/
6. International Geomagnetic Reference Field. http://www.ngdc.noaa.gov/IAGA/vmod/igrf.html
7. World Magnetic Model. http://www.ngdc.noaa.gov/geomag/WMM/DoDWMM.shtml
8. OpenMap. https://code.google.com/p/openmap/
9. Geoconvertor. https://code.google.com/p/geoconvertor/
10. 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.
11. 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.
12. 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.
13. 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.
14. Parnas, D. L. and L. Madey. 1995. Functional documents for computer systems. In *Sci. Comput. Program*. 25:41–61. Amsterdam: Elsevier North-Holland, Inc., 1995.
15. 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.
16. Jackson, M. 2000. *Problem Frames: Analyzing and Structuring Software Development Problems*. Boston, Addison-Wesley Longman Publishing Co., Inc., 2000.
17. Dietl, W., S. Dietzel, M. Ernst, K. Mužlu, 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.
18. 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.* 32:1–37. ACM Press, 2010. Q3
19. 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. ACM Press, 2012. Q4
20. 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. Q5
21. 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.
22. Ou, X., G. Tan, Y. Mandelbaum, and D. Walker. 2004. Dynamic typing with dependent types. In *Exploring New Frontiers of Theoretical Informatics*. 155:437–450. Springer, 2004.
23. Coq. https://coq.inria.fr/
24. Agda. http://wiki.portal.chalmers.se/agda/pmwiki.php
25. 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.
26. 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.

27. 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.
28. 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.
29. Delft, V. 1999. A Java extension with support for dimensions. In *Softw. Pract. Exper*. 29:605–616. John Wiley & Sons, Inc., 1999.
30. 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.
31. Kennedy, A. 1999. Dimension types. In *Proceedings of the 5th European Symposium on Programming (ESOP)*: Lecture Notes in Computer Science volume 788. Springer, 1999.
32. Petty, G. 2001. Automated computation and consistency checking of physical dimensions and units in scientific programs. In *Softw. Pract. Exper*. 31:1067–1076. New York: John Wiley & Sons, Inc., 2001. Q6
33. 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.
34. 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.
35. 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.
36. 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.
37. 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.
38. 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.
39. 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.
40. 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.
41. 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.
42. 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.
43. 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. Muñoz, and E. Métais. 67–79. Springer, 2005.

44. Wordnet. http://wordnet.princeton.edu
45. 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.
46. Perttunen, M., J. Riekki, and O. Lassila. 2009. Context representation and reasoning in pervasive computing: A review. *International Journal of Multimedia and Ubiquitous Engineering*. 4(4): October, 1–28. 2009. Q7
47. 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.
48. 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.
49. 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.
50. Christiansen, D. 2013. Dependent type providers. In *Proceedings of the 9th ACM Workshop on Generic Programming*, 23–34. ACM Press, 2013. Q8
51. 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.
52. International System of Units. National Institute of Standards Technology, Washington, DC. Q9
53. 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.
54. Kelpie Flight Planner for Flightgear. http://sourceforge.net/projects/fgflightplanner/
55. 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.
56. 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.
57. 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.

Cat#: K26798 **Chapter: C003**

TO: CORRESPONDING AUTHOR

AUTHOR QUERIES - TO BE ANSWERED BY THE AUTHOR

The following queries have arisen during the typesetting of your manuscript. Please answer these queries by marking the required corrections at the appropriate point in the text.

Query No.	**Query**	**Response**
Q1	Please confirm if section number correct here.	
Q2	Please check venue of conference in ref. [4,12,25,40,51].	
Q3	Please provide editors name and place of publication in references [18,22,27,29].	
Q4	Please provide venue of *symposium* in references [19,31,53,57].	
Q5	Please provide for place of publication in references [20,21,25,26,43,47,49].	
Q6	Please provide for editors name in [32].	
Q7	Please provide for page number in [46].	
Q8	Please provide venue of *workshop* in references [50,55,57].	
Q9	Please provide for book title and year of publication in [52].	