Asserting Frame Properties

Yoonsik Cheon[1], Bozhen Liu[2] and Carlos Rubio-Medrano[2]
[1]*Department of Computer Science, The University of Texas at El Paso, El Paso, Texas, U.S.A.*
[2]*Department of Computer Science, Texas A&M University – Corpus Christi, Corpus Christi, Texas, U.S.A.*
ycheon@utep.edu, {bozhen.liu, carlos.rubiomedrano}@tamucc.edu

Keywords: Abstraction function, assertion, frame axiom, runtime check, side effect.

Abstract: Frame axioms and properties are crucial for ensuring the correctness of operations by defining which parts of a program's state may change during operation execution. Despite their significance, there has been no known method for asserting frame properties of operations for runtime checks. This paper introduces a practical approach that utilizes abstract models and executable assertions to effectively check frame properties at runtime. By defining abstract models that capture relevant state variables and their relationships, programmers can specify abstractly the parts of an object's state that may change during operation execution. These frame properties, specified in terms of abstract models and embedded as executable assertions within the code, enforce behavioral constraints and improve the readability, maintainability, and reusability of the assertion code. Additionally, the approach supports the concept of observable side effects.

1 INTRODUCTION

In programming, assertions serve as practical tools for ensuring the correctness and reliability of code. These assertions allow programmers to express conditions that must hold true at specific points in the code, serving as checkpoints to detect and diagnose errors during debugging and testing phases (Matuszek, 1976; Rosenblum, 1995). By embedding assertions, programmers can establish a set of criteria that the program's state must adhere to, providing a means to catch and address unexpected behavior or deviations from the expected program flow. Their adoption, commonly in the form of *assert* statements, has been widespread in programming languages, with empirical studies showing that code containing assertions has fewer defects (Casalnuovo et al., 2015; Counsell et al., 2017; Kochhar & Lo, 2017).

Frame axioms constitute fundamental elements in software specification and verification (Borgida et al., 1995). They specify which parts of a system's state are affected by an operation and which parts remain unchanged, often referred to as "and nothing else changes." By formalizing the rules governing state changes, frame axioms contribute significantly to the correctness and reliability of software systems, enabling developers to reason about the behavior of their implementations. However, research into the runtime checking of frame axioms is relatively uncommon, as these properties are usually verified statically during program analysis or verification (Marché et al., 2004). A runtime-based technique is particularly needed for dynamically or gradually typed languages such as Dart, where static methods may not be sufficient.

In this paper, we present a simple and practical approach to asserting frame properties, alongside preconditions and postconditions, in code for runtime checks. Our approach uses abstract models and executable assertions. Abstract models provide a structured representation of a program's state, abstracting away intricate implementation details and focusing solely on the essential aspects relevant to asserting frame properties. By capturing only the relevant state elements and their interrelationships, abstract models allow us to specify frame axioms in a representation-independent way. Embedded directly into the code, executable assertions enforce constraints on the program's behavior, ensuring adherence to specified frame properties throughout program execution. One contribution of our work is the formulation of a notion of observable side effects at different abstraction levels. We systematically define abstract models and associate them with executable assertions to detect observable changes to the program's state. Additionally, the use of abstract models enhances the readability, maintainability, and reusability of assertion code. Overall, our approach provides a practical method for asserting frame properties to ensure code correctness and reliability.

The paper is structured as follows. Section 2 describes the frame problem briefly. Section 3 explains our approach, detailing the use of abstract models and executable. Section 4 provides examples of applying our approach to a mobile app written in Dart/Flutter, followed by discussions in Section 5. Finally, Section 6 discusses related work, and Section 7 concludes the paper.

2 THE FRAME PROBLEM

Assertions, such as *assert* statements, represent a straightforward yet potent means to check the code logic during runtime. The following code snippet demonstrates the typical use of assertions, illustrating their role in a board game operation where a player can place a game piece at a specific location on the game board. The code is written in Dart/Flutter (Flutter, 2024) for a mobile app.

```
int playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x, y));
  …
  assert(playerAt(x, y) == player);
}
```

The first two assertions serve as prerequisites for the method, validating the assumptions made about its input parameters and the initial state and the last assertion scrutinizes the method's behavior, serving as its postcondition. This postcondition assertion encapsulates the operation's key aspect, which is placing a game piece at the specified location. However, it is somewhat limited and fails to comprehensively assess the method's behavior. Consider a scenario where the code inadvertently removes an existing piece from another location. Implicitly, it assumes that the code refrains from altering any other places on the board. These implicit or unspecified assumptions can often lead to subtle and elusive faults.

This issue is commonly referred to as *frame axioms* or *properties*, that specify which properties are not changed during the execution of an operation, essentially stating that "and nothing else changes" (Borgida et al., 1995).

Assertions are rarely used to check frame axioms or properties. For example, the *dart*:*core* package of the Dart Software Development Kit (SDK) is automatically imported into every Dart program to provide built-in types, collections, and other core functionality. In the Dart SDK version 3.1.2, we discovered 30 assertions in the core package within 16,275 lines of source code (SLOC), resulting in an assertion density of 1.84 assertions per 1000 SLOC. Among these assertions, 57% are preconditions that validate input parameters or initial object states, 33% are for checking internal code logic, and 10% are postconditions that verify return values or final object states. We did not find any assertions specifically designed to check frame properties.

3 OUR APPROACH

In this section, we describe several coding styles and techniques for asserting frame properties, which, in practice, can be used in combination. We believe that these approaches, although demonstrated in Dart, are adaptable to and applicable in other object-oriented programming languages.

3.1 Direct Embedding

The simplest approach is to embed assertion code directly within the operation itself. Specifically, we preserve the state of an object before it undergoes mutation, enabling comparison with its new state to detect any unexpected side effects resulting from the operation execution. To achieve this, we clone the object in the initial state and store it in a local variable. This local variable, introduced solely for assertion purposes, is commonly referred to as an "assertion-only variable" (Cheon, 2022). If no inherent clone operation is available for the object, we can construct an abstract model of the object using its observer operations. For example, let us examine the code snippet below, where the embedded frame assertions are highlighted in grey boxes. It creates a model of a board object represented as a map from pairs of *x* and *y* indices of locations of the board to a player and stores it in an assertion-only variable named *model*.

```
void playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x,y));
  // create and store an initial model.
  var model = <(int, int), Player?>{};
  for (var i = 0; i < size; i++) {
    for (var j = 0; j < size; j++) {
      model[(i, j)] = playerAt(i, j);
    }
  }
  …
  assert(playerAt(x, y) == player);
  // check state changes against the initial model.
  for (var i = 0; i < size; i++) {
      for (var j = 0; j < size; j++) {
        if (!(i == x && j == y)) {
          assert(model[(i, j)] == playerAt(i, j);
        }
      }
    }
  }
```

The assertion code in the final state ensures adherence to frame properties, guaranteeing that only the player at location (*x*, *y*) may undergo mutation. This is achieved by referencing the initial value of the object stored in an assertion-only local variable.

The use of an abstract model and observer methods enhances the maintainability of assertion code significantly, eliminating the need for updates when the board's representation changes. However, embedding assertion code directly into operations can lead to code clutter and diminish the readability and reusability of the assertion code. In particular, as the frame properties are coded imperatively, it becomes less evident which parts of the object may undergo changes and which parts must remain unchanged.

3.2 Assertion Method

The previous approach often leads to redundant code when asserting frame properties across multiple methods. To mitigate this issue, we can consolidate duplicated logic into helper methods, simplifying the encoding of frame axioms. These assertion-only methods typically consist of a model getter, a frame checker, and a model comparator (refer to the example code below). We use a custom annotation such as *@assertOnly* to indicate that these methods are solely for writing assertions, not for the operational logic of the code.

```
void playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x,y));
  comparator = (preModel) { // model comparator
    var postModel = model; // invoke the model getter
    for (var key in preModel.keys) {
      if (key != (x, y)) {
        assert(preModel[key] == postModel[key]);
      } } };
  var checker = frame(model, comparator);
  …
  assert(playerAt(x, y) == player);
  assert(checker()); // call the enclosed comparator
}

@assertOnly
Map<(int, int), Player?> get model { // model getter
    var result = <(int, int), Player?>{};
    for (var i = 0; i < size; i++) {
      for (var j = 0; j < size; j++) {
        result[(i, j)] = playerAt(i, j);
      }
    }
    return result;
}

@assertOnly
Function frame(model, comparator) { // frame checker
    return () => comparator(model); // return a closure
}
```

The *comparator* local function serves as a model comparator, encoding the frame properties. Intended for invocation in the post state, it takes a pre-state model and compares it with an internally generated post-state model obtained by calling the *model* getter. A model of a board object is obtained by invoking a *model* getter method, essentially functioning an *abstraction function* that maps a concrete program state to an abstract assertion state for assertion purposes (Cheon, 2022). The assertion-only method, *frame*(), facilitates the checking of the frame properties by accepting a pre-state model and a model comparator as arguments. It uses a closure to retain the pre-state model and invokes the comparator upon its invocation in the post-state. Although the example defines the model comparator as a local function for clarity, it can alternatively be directly coded as an argument to the *frame*() method or can be generalized into a helper method for use across multiple methods (see Section 3.3).

While there are some complexities involved due to the use of closures and lambda notation, the resulting code is cleaner, more maintainable, and more reusable compared to the previous approach. With the potential to transform the *model* getter and the *frame*() method into framework methods, the process becomes more streamlined. For each method, one simply needs to define a model comparator, effectively encoding and encapsulating the frame axiom of the method. This modular design not only simplifies the coding of frame assertions but also fosters code consistency and maintainability across multiple methods within a class.

3.3 Assertion Class

Building upon the previous approach, we can make further improvements by consolidating most of the assertion code into a dedicated assertion-only helper class. This class serves as a container for an abstract model of the asserted class, alongside operations for defining and validating frame properties.

```
void playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x,y));
  var preModel = BoardModel(this)..frame([(x, y)]);
  …
  assert(playerAt(x, y) == player);
  assert(preModel == BoardModel(this));
}
```

```
@assertOnly
class BoardModel {
  late final places;
  var _coords = const [];
  BoardModel(Board board) {
    for (var i = 0; i < board.size; i++) {
      for (var j = 0; j < board.size; j++) {
        places[(i, j)] = board.playerAt(i, j);
      }
    }
  }

  void frame([coords = const []]) => _coords = coords;

  @override
  bool operator ==(other) {
    if (other is _BoardModel) {
      for (var key in places.keys) {
        if (!_coords.contains(key)) {
          places[key] == other.places[key];
        }
      }
      return true;
    }
    return false;
  }
}
```

In this approach, assertions become concise and clear as frame logic is encapsulated within the helper class named BoardModel and the frame properties are written as expressions to have a declarative flavor. As before, a pre-state model is created in the initial state, and frame properties are subsequently defined. Leveraging the Dart cascade notation (..) proves useful for combining the creation of an abstract model and the specification of frame properties into a single expression. Frame properties are specified by providing a list of place coordinates through the *frame*() method. In the post state, the pre-state model is compared with the post-state model to ensure the specified frame properties. The comparison is done by invoking the overridden equality operator (==), which compares only those places on the board that must remain unchanged.

3.3.1 Wildcard Object

The equality operator (==) of the model class plays a crucial role by identifying the parts of the object that may change and ignoring them during comparison. To enhance reusability and versatility of the approach, we can introduce a special wildcard object, say *BoardModel.any*, which equates to any object and can be directly assigned to the parts of the object where changes are allowed. Its equality operator always returns true regardless of the argument. Frame properties can be directly coded by assigning this special object to the parts of the object allowed to change in a model. For example, *preModel.places*[(*x*, *y*)] = *BoardModel.any* indicates that the position (*x*, *y*) can be modified, as its value will match any object. Additionally, a *frame*() helper method can be defined to take a set of place coordinates and mutate the object accordingly by assigning the wildcard object. Of course, the equality operator of a model class is overridden to compare two model objects, part by part. In a sense, a pre-state model serves as a pattern that a post-state model must conform to. This approach offers a more flexible and scalable solution for specifying frame properties, while also providing the potential for creating a supporting library or framework.

3.3.2 Pattern, Path, and Declarative Style

Instead of manually coding wildcard assignments to denote parts of an object allowed to change, a more preferred approach would be to declare these parts explicitly. That is, we can support a declarative style to specify mutable parts of an object. For example, if we want to specify a specific column of a board, we can call the *frame*() method like *preModel.frame*([(*x*, '*')]). Here, the *frame*() method would assign the wildcard object to every place in the *x* column of the board model. To generalize this approach and create a framework method, we believe we can employ patterns, path expressions, or regular expressions to specify sets of object parts declaratively. The *frame*() method can then be implemented using the reflection facility to parse the specification of frame properties and update the model accordingly with the wildcard object. This method not only enhances readability but also allows for a more flexible and concise specification of frame properties.

3.4 Annotation

Annotations can offer a clear and concise means of specifying frame axioms. One way to utilize annotations for specifying frame axioms is by defining a custom annotation. This custom annotation can be applied to methods or classes to indicate which parts of the object may be changed or should remain unchanged during the execution of the methods. Below is an example:

```
@frame("[(x,y)]")
 void playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x,y));
  …
  assert(playerAt(x, y) == player);
 }
```

```
@assertOnly
Map<(int, int), Player?> get model { … }
```

The custom *@frame* annotation is applied to the *playStone*() method, specifying that the *model*[(*x*,*y*)] may be changed during the execution of the method but all other parts of the model should remain unchanged.

The annotation can be processed during compile time or at runtime to ensure that the specified frame properties are respected during method execution. This can involve reflection to inspect the annotations and validate the frame properties accordingly. For example, the above annotation can be translated into the following code by following the assertion method approach described in Section 3.2:

```
 void playStone(int x, int y, Player player) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert(isEmpty(x,y));
  var axiom = frame(model, (pre) {
   var post = this.model;
   for (var key in pre.keys) {
    if (key != (x, y)) {
      assert(pre[key] == post[key]);
    }
   }
  });
  …
  assert(playerAt(x, y) == player);
  assert(axiom());
 }
```

4 MORE EXAMPLES

Another core operation of the Board class is to determine whether the board has a winning configuration for a player. The *isWonBy*() method performs this check with the assistance of a helper method, and the specifications of their frame properties shown below are interesting.

```
 (bool, List<(int, int)>) isWonBy(Player player) {
   var preModel = model;
   …
   assert(preModel == model);
   return …;
 }
```

```
 bool _isWonBy(Player player, int x, int y, int dx, int dy) {
  assert(0 <= x && x < size && 0 <= y && y < size);
  assert({-1, 0, 1}.containsAll({dx, dy}));
   var preModel = (model, BoardModel.any);
   …
   assert(preModel == (model, _winSeq);
   return …;
 }
```

The *isWonBy*() method identifies an unbroken horizontal, vertical, or diagonal row of five stones belonging to the given player and returns it as a winning sequence. The assertion of its frame properties is typical in that it shouldn't produce any side effects. However, its implementation does entail side effects by invoking a series of calls to the *_isWonBy*() helper method. This helper method verifies if there is a winning sequence containing the specified place (*x* and *y*) in the given direction (*dx* and *dy*). It may modify the *_winSeq* private field to remember the winning sequence found.

Abstract models facilitate the decision-making process and coding of whether this state change should be considered part of the frame properties of methods. For instance, it isn't part of the frame properties of the *isWonBy*() method, as the side effect isn't observable by the method's client. This decision results in more maintainable and reusable frame assertions. Conversely, it is part of the frame properties of the helper method, as its client can observe and rely on this side effect. Our decision is grounded in the perspective of viewing frame properties as contracts between the implementor and the client. In summary, the use of abstract models enables us to codify the notion of "observable" side effects when formulating frame properties.

As hinted above, the use of abstract models reveals a compositional nature, wherein the model of a composite object can be constructed by composing the models of its component objects. Consider the Game class, whose instances comprise a board and two players.

```
 class Game {
   final Board board;
   final List<Player> _players;
   Player _current;
   …
   GameModel get model = GameModel(board,
     (_current, _players.firstWhere((p) => p != _current));
 }
```

We adopt a tuple view of an object's states. A game object's abstract model is represented as a tuple containing the board and its two players. It also abstracts from the representation of the current player. An intriguing design consideration arises when determining whether to utilize a part object or its abstract model in defining the model of the composite object (refer to Section 5).

From a frame perspective, methods within the Game class can be classified into four categories based on their potential side effects: observers and three kinds of mutators that may alter (a) only the board, (b) only the players, and (c) both the board and players. For instance, one responsibility of the Game class is to manage the turns of the players.

```
Player changeTurn() {
  var preModel = model..frame({'players'});
  …
  assert(preModel == model);
  return _current;
}
```

The *changeTurn*() method is expected to modify only the players, leaving the board unaffected. The *frame*() method of the model class serves to designate the parts of the object that may undergo changes, replacing them with a wildcard object that is equivalent to any object. Consequently, the overridden equality operator (==) of the model class ensures the equality of parts not marked as changeable. The frame properties of other kinds of mutation methods can be asserted similarly.

5 DISCUSSION

We performed a preliminary evaluation of our approach through a small case study involving the board game Omok, also known as Gomoku or Gobang. This strategic two-player game, “five pieces” in meaning, is traditionally played on a 15x15 grid where players take turns placing stones to form unbroken rows of five, either vertically, horizontally, or diagonally. We developed a cross-platform mobile app using Dart/Flutter (Flutter, 2024) comprising widget classes, UI-dependent model classes, and pure model classes, with our primary focus on the latter, including Board, Player, and Game classes.

Our implementation of the three classes consists of 256 lines of source code (SLOC) including comments, containing 21 methods. We ensured the assertion of frame properties for each method, employing a combination of approaches described in Section 3, excluding annotations. Although it is uncommon to assert frame properties for every method in practice, doing so serves as an instructive exercise for comparison and evaluation purposes.

We followed a straightforward process: (a) identifying the parts of the object susceptible to change by a method and observable by clients, (b) defining an abstract model consisting of only the observable parts, (c) implementing a model getter and, if needed, a model class to define a frame method and override the equality. This process was iteratively applied to refine the model getter and the model class for several representative mutation methods. Once the abstract model was formulated, asserting frame properties for methods became straightforward.

The process of adding frame assertions to all methods significantly increased the size complexity, expanding the source code from 256 to 408 SLOC, representing 59% overhead in SLOC. For example, a concise one-line method like *playerAt*() in lambda notation (see below) expanded to six lines in block notation to integrate frame assertions. Among the 21 methods, 12 (57%) are observers that do not alter the object state. Among the remaining methods, 6 (29%) modify only a single element of the composite model, 2 (10%) alter two elements, and 1 (5%) affects three elements. In summary, a significant number of methods exhibit either no side effects or side effects only on a single element. This underscores the necessity for some level of automation to facilitate widespread and practical adoption. As proposed earlier, an annotation-driven approach presents a promising avenue. Given that 57% of methods serve as observers, many of which are succinctly expressed in lambda notation, introducing a custom annotation like *@observer* could prove invaluable. This annotation could then automatically trigger the insertion of assertion code, as annotations are good to partially replace the time-consuming and error-prone process of implementing code (Yu et al., 2019).

```
@observer
Player? playerAt(int x, int y) => _places[x * size + y];

class Game {
  @model
  final Board board;
  @model("self.firstWhere(p) => p != _current)")
  final List<Player> _players;
  …
}
```

We also believe that a model getter and a model class can also be created using custom annotations, as shown above. The *@model* annotation can specify that a field or group of fields should be abstracted into an abstract model, with an optionally specified abstraction function. Once all elements or parts of a model are known, the annotation processor can generate the frame method and override the equality operator accordingly.

As outlined in the previous section, our approach to defining abstract models for classes and utilizing them in asserting frame properties follows a compositional strategy. Abstract models of classes, such as the Game class, can be constructed by composing their component or part objects. An intriguing question arises when composing objects to define an abstract model: should we compose the objects themselves or their abstract models? This question also pertains to comparing the initial and final states of an object to detect state changes and violation of frame properties.

Most object-oriented programming languages, including Dart, support two different notions of equality: reference equality and value equality. The coexistence of these two notions of object equality can present challenges when formulating and asserting frame properties. There is no strict rule governing their usage; instead, it depends on several factors such as object sharing and ownership. As a general guideline, value composition and equality are typically employed when the parts are encapsulated and not directly visible to the client, while reference composition and equality are preferred for parts that are shared and directly visible to the client. In the Game class, both the board and two players are exposed and visible to clients. Therefore, we opted for reference composition and equality for most methods. However, we found one method where value composition works better.

```
Outcome makeMove(int x, int y) {
  var preModel = (_board.model..frame([(x, y)]), any);
  _board.placeStone(x, y, _currentPlayer);
  …
  assert(preModel == (_board.model, modelPlayers));
  return …;
}

@assertOnly
get modelPlayers => (_currentPlayer, /* opponent */);
```

The *makeMove*() method implements the logic for the current player's next move by placing a stone on the specified place. It also updates the turn to the next player if the move does not result in a gaming-ending scenario, such as a winning move. Hence, it has the potential to modify both the board and the players' aspects of the game. However, since it might only change a specific place of the board, it would be advantageous to create a game model as a composition of the board's model, rather than the board object itself, as demonstrated above.

In our examples, we considered only the state changes of the receiver object. However, a method can access other objects, including parameters. Therefore, the state can be defined as a combination of the receiver and operation-specific state, representing a universe of objects accessible during method execution. In other words, the state accessible to a method includes not only the state of the receiver object but also additional state components accessed via parameters and global references. While the receiver remains constant across all methods within a class, other components may vary depending on the specific method in question. A limitation of our approach is that the programmer must specify this state information for each method as part of the frame axioms. It may be necessary to define multiple model getters or abstraction functions, including those tailored for various parts of the object.

Another caveat of our approach is that it allows intermediate changes, as only the final state is checked. While this has no impact in a sequential environment, it may be significant in a concurrent environment where intermediate changes can be visible to the client.

6 RELATED WORK

Frame axioms are foundational elements in formal specification languages, precisely specifying which parts of a system's state are impacted by an operation and which remain unaffected (Borgida et al., 1995). In behavioral interface specification languages like Larch (Guttag & Horning, 1993), JML (Leavens et al, 2005), and Spec# (Leino & Müller, 2007), frame axioms define the interface contracts of software components, including preconditions, postconditions, class invariants, and frame properties. Special constructs such as *modifies* and *assignable* clauses specify which variables or fields may be modified by methods, serving as frame axioms that specify the parts of an object's state subject to change during method execution (Chalin et al., 2005). Fields can also be grouped for writing frame properties (Leino, 1998). However, research into runtime checking of frame axioms is relatively uncommon, as these properties are usually verified statically during program analysis or verification (Marché et al., 2004). Consequently, our work stands out by offering a quick, straightforward, and practical solution to ensure frame properties with executable assertions.

The use of abstract functions, pioneered by Hoare in formal program verification (Hoare, 1972), appears in behavioral interface specification languages such as JML and Spec# in the form of model variables – specification-only variables whose values are defined and calculated in terms of concrete program variables and states (Cheon et al., 2005). It was also suggested to write executable assertions abstractly by mapping a program state into an abstract assertion state or model through the provision of an abstract function (Cheon, 2022). This approach proves particularly beneficial for writing design assertions – constraints and decisions translated from design that must be held regardless of concrete representation choices, such as data structures. The resulting assertions become not only more understandable but also more maintainable and reusable (Cheon, 2022; Cheon, 2024). Our approach aligns with this strategy and facilitates the translation of frame axioms and properties into executable assertions, in addition to pre and postconditions. The use of assertion-only immutable collection classes to create abstract models should equally apply to our approach (Cheon, 2023). Moreover, the utilization of abstract models in our approach allows for formulating the notion of observable side effects at different abstraction levels.

7 CONCLUSION

Our work has shown the potential of frame axioms and properties as practical tools for programmers. By employing abstract models and executable assertions, we provided a systematic and effective coding style and technique for asserting frame properties, along with pre and postconditions, in software development. This approach not only enhances the readability, maintainability, and reusability of assertion code but also enables the formulation of observable side effects at different abstraction levels. While research into the runtime checking of frame axioms remains relatively uncommon, our work stands out by offering a straightforward and practical solution for ensuring frame properties through executable assertions. Further exploration and refinement of our approach, especially through annotation-based automation, could pave the way for more robust and reliable software development practices.

REFERENCES

Borgida, A., Mylopoulos, J., & Reiter, R. (1995). On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering*, *21*(10), 785-798.

Casalnuovo, C., Devanbu, P., Oliveira, A., Filkov, V., & Ray, B. (2015). Assert use in GitHub projects. *IEEE/ACM 37th International Conference on Software Engineering (ICSE),* Florence, Italy, 755-766.

Chalin, P., Kiniry, J. R., Leavens, G. T., & Poll, E. (2006). Beyond assertions: Advanced specification and verification with JML and ESC/Java2. *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4 (pp. 342-363). Springer Berlin Heidelberg.

Cheon, Y. (2022). Design assertions: executable assertions for design constraints. *14th International Symposium on Software Engineering Processes and Applications (SEPA),* July 4-7, Malaga, Spain. Published as ICCSA 2022 Workshops, *Lecture Notes in Computer Science*, 13381, 617-631, Springer.

Cheon, Y. (2024). Constructive assertion with abstract models. *12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, Rome, Italy, February 21-23 (pp. 211-218).

Cheon, Y., Leavens, G. T., Sitaraman, M., & Edwards, S. (2005). Model variables: cleanly supporting abstraction in design by contract. *Software: Practice and Experience,* 35(6), 583-599, Wiley.

Cheon, Y., Lozano, R., & Prabhu, R. S. (2023). A library-based approach for writing design assertions. *IEEE/ACIS 21st International Conference on Software Engineering Research, Management, and Applications (SERA)*, Orlando, FL, USA, 22-27.

Counsell, S., Hall, T., Shippey, T., Bowes, T., Tahir, A., & MacDonell, S. (2017). Assert use and defectiveness in industrial code. *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW),* Toulouse, France, 20-23.

Flutter. (2024). Flutter – Build for any screen. Flutter.dev. https://flutter.dev/.

Guttag, J. V., & Horning, J. J. (1993). *Larch: Languages and Tools for Formal Specification*. Springer.

Hoare, C. A. R. (1972). October. Proof of correctness of data representations, *Acta Informatica*, 1(1), 271–281.

Kochhar, P.S. & Lo, D. (2017). Revisiting assert use in GitHub projects. *21st International Conference on Evaluation and Assessment in Software Engineering (EASE),* June, 298-307.

Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., & Cok, D. (2005). How the design of JML accommodates both runtime assertion checking and formal verification. *Science of Computer Programming*, 55(1-3), 185-208.

Leino, K. R. M. (1998, October). Data groups: Specifying the modification of extended state. *13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 144-153).

Leino, K. R. M., & Müller, P. (2007). Using the Spec# language, methodology, and tools to write bug-free programs. *LASER Summer School on Software Engineering* (pp. 91-139). Berlin, Heidelberg: Springer Berlin Heidelberg.

Matuszek, D. (1976). The case for assert statement. *ACM SIGPLAN Notices*, 36-37, August.

Marché, C., Paulin-Mohring, C., & Urbain, X. (2004). The Krakatoa tool for certification of Java/JavaCard programs annotated in JML. *The Journal of Logic and Algebraic Programming*, 58(1-2), 89-106.

Rosenblum, D. S. (1995). A practical approach to programming with assertions. *IEEE Transactions on Software Engineering,* 21(1), 19-31, January.

Yu, Z., Bai, C., Seinturier, L., & Monperrus, M. (2019). Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, *47*(5), 969-986.