Constructive Assertions with Abstract Models

Yoonsik Cheon

Department of Computer Science, The University of Texas at El Paso, El Paso, Texas, U.S.A.

Keywords: Abstraction Function, Assertion, Assertion Library, Assertion State, Constructive Assertion, Dart, Object Constraint Language.

Abstract: An assertion is a statement that specifies a condition that must be true at a particular point during program execution. It serves as a tool to ensure the program functions as intended, reducing the risk of introducing subtle errors. Usually expressed algebraically, an assertion utilizes Boolean expressions to specify permissible relationships among program variables. In complex scenarios, calculating the expected value of a program variable often proves more effective than specifying the constraints it must adhere to. In this paper, we present an approach to formulating assertions using abstract models in a constructive manner, which complements the traditional algebraic style. Constructive assertions empower programmers to articulate comprehensive assertions, including pre and postconditions, in a succinct, comprehensible, reusable, and maintainable manner.

1 INTRODUCTION

An assertion, as defined by Matuszek (1976), is a statement establishing a condition expected to be true at a specific point during program execution. It is essentially a Boolean expression inserted directly into the code. If the condition evaluates as true, the program proceeds without disruption; if false, the assertion fails, triggering an exception or displaying an error message. Assertions can validate code assumptions, verify code logic, and aid in identifying potential errors promptly. Their adoption, often 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).

While traditional assertions express conditions algebraically, involving relationships among program variables and state components, writing such assertions can become complex, especially dealing with multiple variables or various object parts.

This paper introduces a complementary approach called *constructive assertions*. In this approach, expected values for program variables are computed to produce more straightforward and understandable assertions. It leverages an abstraction of program states with an abstraction function (Hoare, 72; Sitaraman, Weide, & Ogden, 1997), providing an immutable model, to write assertions constructively and representation-independently. The paper also introduces an assertion library offering immutable collection classes specifically designed for writing assertions. Although demonstrated using Dart (Bracha, 2016) and Flutter (Flutter, 2023) for mobile applications, the concept of constructive assertions is applicable to other languages and platforms.

The integration of traditional algebraic and new constructive assertions broadens the range of assertion techniques. Algebraic assertions define constraints on program variables and states, whereas constructive assertions provide a more streamlined approach to express expected outcomes. Therefore, this paper empowers programmers to select the most appropriate technique depending on the assertion's unique context and complexity. This enhances the thoroughness and effectiveness of assertions, ultimately fostering improved code quality and more efficient debugging practices.

While existing literature extensively explores various applications of assertions, including debugging and testing tools utilizing pre and postconditions (Rosenblum, 1995; Chalin, 2014; Cheon, 2021; Cheon, 2022), as well as test oracles (Cheon & Leavens, 2005; Watson et al., 2020), a noticeable gap exists in publications studying distinct styles of writing assertions, particularly the comparative analysis of algebraic and constructive

Cheon, Y.
Constructive Assertions with Abstract Models.
DOI: 10.5220/0012418800003645
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 12th International Conference on Model-Based Software and Systems Engineering (MODELSWARD 2024)*, pages 211-218
ISBN: 978-989-758-682-8; ISSN: 2184-4348
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

methodologies. This specific aspect has received limited research attention. The use of an abstraction function to define an abstract assertion state can be considered a way to implement a model variable in a specification language (Cheon et al., 2005). The assertion library, designed to support constructive assertions, draws inspiration from the collections library of the Object Constraint Language (OCL) (Warmer & Kleppe, 2003).

This paper is organized as follows. Section 2 introduces assertions, covering algebraic and constructive writing styles. Section 3 presents an assertion library consisting of immutable collection classes. Section 4 demonstrates constructive assertions via examples, including app written in Flutter. Section 5 explores open issues and future research directions. Section 6 concludes the paper.

2 CONSTRUCTIVE ASSERTIONS

2.1 Assertions

In programming, assertions are straightforward statements that declare conditions expected to be true at specific points during program execution (Matuszek, 1976; Rosenblum, 1996). These Boolean expressions, embedded within the code, facilitate smooth program flow when conditions are met. If false, assertions trigger errors, highlighting unexpected situations. Operating as lightweight mechanisms, assertions enable programmers to validate code assumptions and logic, aiding in testing, debugging, and enhancing software quality.

To illustrate, consider a Dart function calculating the maximum value from a list of integers. The function assumes a non-empty input list, a critical point addressed by an assertion at the code's outset. If the function encounters an empty list, this assertion signals an error to inform the invalid input. Further assertions, strategically placed post-computation, verify that the returned value represents the maximum within the input list. Any failure triggers an assertion exception, signaling potential implementation errors.

```
int max(List<int> list) {
  assert(list.length > 0, 'Invalid argument');
  var result;
  ... // calculate the result.
  assert(list.contains(result), 'Wrong result');
  assert(list.every((n) => n <= result, 'Wrong result');
  return result;
}
```

By using assertions during development and testing, one can effectively verify both the preconditions (input assumptions) and postconditions (expected output) of the code, thereby enhancing its reliability. Note that assertions are typically disabled in production environments and automatically removed from the Dart/Flutter production code.

2.2 Assertion Styles

Traditionally, assertions are formed by examining program states to assert facts that these states must meet. This algebraic approach constrains program states by specifying permissible relationships among program variables and state components. For instance, in the earlier *max*() function, the first assert statement ensures the list's length is greater than 0, indicating a non-empty list. Similarly, other assert statements verify that the *result* is an element of the list and is greater than or equal to every other element in the list.

An alternative approach involves calculating the expected value of a state component and asserting the equivalence between the actual and expected values, termed the *constructive style*. For example, instead of explicitly listing properties of a maximum value in a list, we can find the maximum value using the *reduce*() method and assert that the *result* is equal to this calculated value:

```
assert(result == list.reduce((r,e) => math.max(r,e));
```

This constructive style focuses on calculated expected results and their equivalence with actual values. Both algebraic and constructive styles have merits, depending on specific code requirements.

In this paper, we informally use the term *constructive assertions*. These assertions take a Boolean expression, denoted as $P(x)$, where x represents program variables or state components. $P(x)$ outlines constraints on a program's state or establishes permissible relationships among its components. When $P(x)$ is structured as $x == E(y)$, we classify it as a constructive assertion, where $E(y)$ calculates or constructs the anticipated value or state for the specific variable or component x. The essence lies in computing a solution rather than enumerating solution properties.

For example, asserting the sorted order of a list, denoted as l, conventionally involves scrutinizing the sorted property of the list:

```
for (var i = 0; l < l.length - 1; i++) {
  assert(l[i] <= l[i + 1]);
}
```

Contrastingly, a constructive approach creates a sorted version of the list and asserts the equivalence between the sorted and original lists:

```
assert(listEquals(l, List<int>.from(l)..sort()));
```

Although the assertion involves Dart-specific constructs and syntax, its essence is summarized as *l* == *l.clone*()..*sort*(). The *listEquals*() function from the Flutter SDK compares two lists element-wise; the List class's == operator evaluating object identity. To create a fresh list and prevent inadvertent alterations to the original, we use the *List.from*() method. Dart's cascade notation (..), exemplified as *e*..*m*(), allows executing a series of operations on the same object without introducing temporary variables. Thus, the expression *List<int>.from*(*l*)..*sort*() efficiently clones and sorts the list, yielding its sorted version.

This constructive approach is particularly effective for asserting state changes and specifying the behavior of mutation operations. For example, asserting the insertion of a value *n* into a sorted list *l* while maintaining its sorted order is succinctly expressed using constructive assertions:

```
var expected = List<int>.from(l)..append(n)..sort();
// ... code to insert n to l.
assert(listEquals(l, expected));
```

The local variable *expected* is an assertion-only variable introduced to store the expected value of *l* calculated in the initial state. This calculated value is then compared to the actual value of *l* in the final state. As done previously, before the calculation, the list is cloned to prevent unintended modifications to the original list.

Now, let's formulate the same assertion using the conventional style of stating properties. The skeletal code is shown below:

```
var preL = List<int>.from(l); // initial value of l
// ... code to insert n into l.
assert(...);
```

In the final state, we need to formulate the properties that *l* is a version of *preL* with *n* appropriately inserted. This entails articulating the following properties (see Section 3):

- The list *l* maintains its sorted order. This implies that for each index *i* from 0 to *l.length* - 2, the element at index *i* is less than or equal to the element at index *i* + 1.
- The value *n* is present within the list *l* after the insertion operation.
- There are no other changes in *l*. That is, all elements are retained and no new elements other than *n* have been introduced. This principle, referred to as the *frame axiom* (Borgida et al., 1993), makes the assertion comprehensive and conclusive.

The complexity of stating certain properties, such as the last one above, can be notably intricate, especially when verifying not only the existence of an element within a list but also the exact count of its occurrences when duplicates are allowed.

Modern object-oriented programming languages, such as Dart (Bracha, 2016), offer a spectrum of language constructs and functionalities that streamline the composition of assertions through a constructive approach. These include lambda expressions, higher-order functions, cascade notation, operator overriding, and collections. Complementing these capabilities, we can enhance the efficacy of constructive assertions by introducing a dedicated assertion library.

3 ASSERTION LIBRARY

Constructive assertions are highly valuable for validating modifications in object state or side effects, enabling programmers to formulate complete postconditions. For example, let us consider the code snippet to insert an element *e* into a list *l* at a position *i*. Below, we present its pre and postconditions without utilizing constructive assertions:

```
// precondition
assert(i >= 0 && i <= l.length);
var preL = List<int>.from(l); // initial value of l
l[i] = e;
// postcondition
for (var j = 0; j < i – 1; j++) {
  assert(l[j] == preL[j]);
}
assert(l[i] == e);
for (var j = i; j < preL.length; j++) {
  assert(l[j+1] == preL[j]);
}
```

Surprisingly, for this simple one-line operation, the postcondition assertions are long and somewhat complicated, meticulously detailing the effects on each location within the list, including the newly added one. We can enhance the clarity of the postcondition by employing the constructive approach and incorporating more appropriate terminology, such as “insert,” as illustrated below:

```
assert(listEquals(l, preL..insert(i,e));
```

Assuming familiarity with the *insert*() operation, the resultant assertion offers a notably simpler and more elegant expression compared to the original version. It states that the final state of l is equivalent to the result state of "inserting" the element at the position i within its initial state. It is worth noting that the *insert*() operation used in the assertion carries a side effect. However, this side effect is confined to a duplicate copy (*preL*) that is created to write assertions, rendering it inconsequential for actual operations of the code. In a way, this duplicated copy represents an assertion state that remains imperceptible to the operational code.

Two immediate improvements can enhance the clarity of the above assertion further. Firstly, using the overridable == operator available in Dart instead of the *listEquals*() function improves readability. Secondly, a more significant refinement involves avoiding the manual cloning of objects in the initial state, such as *preL* in the example, and utilizing mutation operations like insertion. The reason is that it poses the risk of inadvertent mutations on the program state and thus potential reliability issues, as assertions should remain free from side effects. To address these concerns and enhance the effectiveness of constructive assertions, we propose introducing an assertion library.

The assertion library aims to offer a high-level vocabulary tailored for manipulating program states, with a specific emphasis on creating anticipated values without unintentionally impacting the actual program state. This is accomplished through the provision of immutable collection classes designed for abstracting concrete program states into assertion states. The underlying idea involves mapping a program state to an abstract state and formulating assertions in terms of this abstract state. Thus, assertions indirectly impose constraints on the code without explicitly referring to concrete program states. To illustrate this concept, the preceding assertion can be elegantly rewritten using the library:

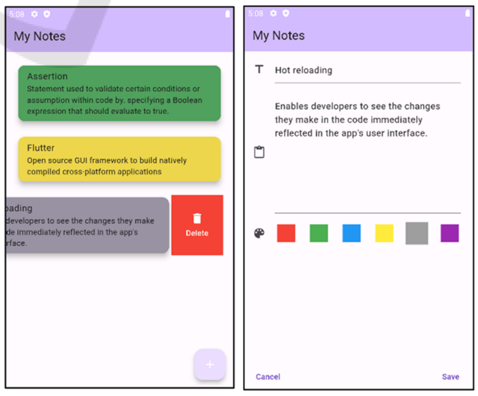
```
var expected = OCLSequence<int>(l).insertAt(e, i);
l[i] = e;
assert(expected == l);
```

The program state *l* is abstracted into an immutable sequence, and the *insertAt*() operation is an immutable insertion operation as it creates a new sequence incorporating the original data and the newly inserted element. Importantly, this operation leaves the initial sequence untouched. The thoughtful override of the == operator facilitates the comparison of sequences, allowing the conversion of a concrete collection, such as a list, into a sequence when necessary. This approach ensures the clarity of the assertion without compromising the integrity of the program state.

The assertion library operates similarly to mathematical toolkits commonly found in formal specification languages like VDM-SL and Z (Jones, 1986; Spivey, 1989). Realizing the assertion library can be achieved through various avenues. In this paper, we utilize the Dart implementation of OCL collection classes (Cheon, Lozano, & Prabhu, 2023). The implementation provides various collections, including sets, ordered sets, bags, and sequences, meticulously designed for immutability, with operations conforming to the OCL standard (Object Management Group, 2023). It also incorporates streamlined Dart language-specific adaptations, such as translating iteration operations into higher-order functions, enabling conversion between Dart and OCL collections, and introducing other Dart-inspired operations.

4 EXAMPLES

In this section, we apply constructive assertions to a small Flutter app for text-based notetaking adapted from Zammetti (2019). The app comprises two main screens: a note list display and a note editor (see Figure 1). Our focus is on the app's model and state management, crucial in the reactive programming of Flutter, where the user interface (UI) functions as a mapping from the state to widgets. Widgets are regenerated and adapted automatically in response to state modifications.



Assertion
Statement used to validate certain conditions or assumption within code by. specifying a Boolean expression that should evaluate to true.

Flutter
Open source GUI framework to build natively compiled cross-platform applications

oading
developers to see the changes they make de immediately reflected in the app's rface.

Hot reloading

Enables developers to see the changes they make in the code immediately reflected in the app's user interface.

Figure 1: Sample screens: list screen and note editor.

Figure shows the architecture of the app, consisting of three widget classes and several non-widget classes. The main widget class hosts two widgets—one for note display and the other for editing. This class dynamically selects and updates its view, driven by the screen index (*screenIndex*) stored within the model class.

The app's state, managed by the NotesModel class, comprises an ordered collection of notes, with an optional note for buffered editing. All notes persist in a local database, and the NotesModel class ensures synchronized updates between this database and in-memory notes. The state also includes a screen index (*screenIndex*), indicating the active screen widget—NotesList or NoteEntry. Changing this index triggers seamless UI updates facilitated by the state management framework in use.

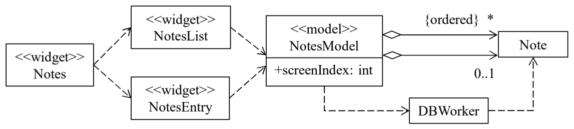


Figure 2: Class diagram.

Now, let's examine a portion of the NotesModel class implementation to explore its assertions. Below, we show the skeletal code that instantiates the previously discussed associations and attributes. This model class is defined as a subclass of a state management framework class, facilitating automatic UI reconstruction in response to state changes. The specific choice of the state management framework does not impact our discussion in this paper.

```
class NotesModel extends Model {
  final List<Note> _notes = [];
  Note? noteEdited;
  int screenIndex = 0;
  ...
  get __notes => OCLSequence<Note>(_notes);
  get __notesDB async => OCLSequence<Note>(
    await DBWorker.db.getAll());
}
```

The code features two getter methods (*__notes* and *__notesDB*) serving as abstraction functions to access abstract views of notes within the (in-memory) model and those stored in the database. These getters transform Dart lists into immutable sequences (OCLSequence). They are assertion-only methods designed solely for writing assertions; we may introduce an annotation like @assert to indicate this.

We are now ready to examine several key operations of the NotesModel class and formulate their assertions. Writing assertions is a common practice for verifying code assumptions and logic. Our aim is to construct complete pre and postconditions for these operations to evaluate the effectiveness of constructive assertions.

The *loadNotes*() method populates notes to be displayed by the notes list screen by fetching them from a local database (see below). This method should be invoked upon app launch and when the list screen widget is created.

```
Future<void> loadNotes() async {
  ...
  assert(__notes == await __notesDB);
}
```

Its constructive postcondition is expressed as *__notes == await __noteDB*. The abstraction functions simplify its formulation, resulting in a clear assertion. They return OCL sequences, and the overridden == operator evaluates the equality of these sequences. The use of abstraction functions improves the reusability and maintainability of assertions. If the implementation changes, only the abstraction functions need redefining, eliminating the need to rewrite assertions. This assertion also shows the advantage of abstraction, as the database is perceived as a sequence of notes, independent of specific storage intricacies.

The *createNote*() method, shown below, serves as a callback function to be invoked when the user taps the floating action button on the notes list screen (see Figure). This action facilitates the creation of a new note, which will later either be discarded or added to the list of notes using the *saveNote*() method to be introduced later in this section.

```
void createNote() {
  ...
  assert(noteEdited == Note.empty());
  assert(screenIndex == screenIndexNote);
}
```

The method creates an empty note and assigns it to the editing buffer *noteEdited* field, anticipating its later presentation by the note editor widget. It also modifies the *screenIndex* field to prompt a UI update, transitioning to the note editor screen. The overridden == operator in the Note class enables proper comparison, and the side effect of creating a new empty note in the first assertion occurs within the assertion state, remaining unobservable to the code.

The *deleteNote*() method removes a note from the list and the database (see below), triggered by a user tap on the delete slide action (see Figure). This method alters the list of notes, presenting an interesting case for assertions. The first two assertions confirm the presence of the note in the list and a non-null id, indicating its existence in the database. Considering the side effects, pre-state calculations anticipate final values for both the notes list and the database, stored in *notesExp*. The *excluding*() operation is an immutable removal operation defined in OCL collection classes. Post-state assertions then validate the equivalence between these final values and the expected sequences of notes. As expected, abstraction functions are used to retrieve the actual final values. The last assertion confirms that the method prompts an update of the notes list screen to reflect the note's removal.

```
void deleteNote(Note note) async {
  assert(__notes.includes(note));
  assert(note.id != null);
  var notesExp = __notes.excluding(note);
  ...
  assert(__notes == notesExp);
  assert(await __notesDB == notesExp);
  assert(screenIndex == screenIndexList);
}
```

The user can edit a note by tapping on it displayed in the notes list screen, triggering the *editNote*() method to transition to the editing screen (see below). The precondition requires the tapped note to exist in the list of notes. The postcondition ensures that the *noteEdited* field contains an equivalent note to the argument, emphasizing that it's not the same instance. This field serves as an editing buffer, signifying that changes are temporary until explicitly saved. The last assertion confirms that the method triggers a UI update to transition to the editing screen.

```
void editNote(Note note) {
  assert(__notes.includes(note));
  ...
  assert(noteEdited == note
    && !identical(noteEdited, note));
  assert(screenIndex == screenIndexNote);
}
```

The *saveNote*() method (see below), invoked when the user taps the save button on the note editing screen (see Figure), handles updates to both the notes list and the database, addressing the complexities of editing new and existing notes.

```
Future<void> saveNote() async {
  assert(noteEdited != null);
  assert(noteEdited!.id == null
    || _notes.any((e) => e.id == noteEdited!.id));
  var preNote = noteEdited!.clone();
  var preNotes = __notes;
  ...
  if (preNote.id != null) {
    var index = preNotes.indexWhere(
        (e) => e.id == preNote.id);
    assert(__notes ==
        preNotes.setAt(index, preNote));
    assert(await __notesDB) == __notes;
  } else {
    var lastNote = __notes.last;
    assert(lastNote.id != null);
    assert(preNotes.collect((e) =>
      e.id).excludes(lastNote.id));
    assert(__notes ==
      preNotes.append(preNote..id = lastNote.id));
    assert(await __notesDB  ==__notes);
  }
  assert(screenIndex == screenIndexList);
}
```

The first assertion ensures that the method is invoked when there is an active editing buffer. The second confirms that the edited note is either new or an existing one based on its ID. The subsequent pair of local variables, *preNote* and *preNotes*, store pre-state values for later reference in the post-state, necessitating cloning due to potential modifications by the code. For an existing note, the postcondition locates the index of the edited note, replaces it in the pre-state list, and ensures that the change is reflected in the database. In the case of a new note, the post condition asserts the uniqueness of the ID field, and using a constructive approach, appends the edited note to the list with an ID selected by the code.

5 DISCUSSIONS

Creating assertions covering complete pre and postconditions is uncommon, except in certain cases, such as aiming for formal code verification. However, when such needs arise, using constructive assertions with an assertion library enables programmers to express them in a concise, understandable, reusable, and maintainable manner. To substantiate this claim, we measured the size complexities of assertions by comparing constructive assertions to conventional algebraic assertions across 24 methods in the Notes and NotesModel model classes. For the comparison, we wrote equivalent assertions in conventional algebraic style. The results are summarized in Table I, where only assertions involving abstraction functions for non-primitive values were counted as constructive; that is, assertions in the form of $x = E$, with E being a primitive expression, were not considered constructive.

The first row compares the total source lines of code (SLOC) for both code and assertions, revealing a 13% reduction in SLOC for the constructive style (257 vs. 295). The second row compares only assertions, including supporting code, showing a notable difference. Constructive style includes 49 assert statements within 75 SLOC, while algebraic style incorporates 59 assertions within 115 SLOC. This translates to 35% less code for the constructive style. If we specifically examine only those assertions written constructively and their corresponding algebraic assertions (the last row), it becomes evident that constructive style requires fewer assertions (36% less) and significantly fewer SLOC (67% less). These findings underscore the efficiency of constructive assertions, providing a concise and understandable approach for expressing complete pre and postconditions.

Table 1: Size complexities of assertions.

	Con.	Alg.	C/A	1 – C/A
Total SLOC	257	295	0.87	0.13
No. all assert stmts	49	59	0.83	0.17
SLOC	75	115	0.65	0.35
No. con./alg. assert	18	28	0.64	0.36
SLOC	20	60	0.33	0.67

We observed in our assertions that most accessor methods, such as the *getNoteId*() method listed below, exhibit an interesting assertion pattern where postconditions, whether algebraic or constructive, often parallel the code logic for calculating the return value. Constructive assertions, structurally similar but referencing specification-only variables (e.g., __*notes*), highlight their advantage. In contrast to algebraic-style assertions, constructive assertions retain reusability despite changes in the underlying representation. When the representation changes, only the abstraction functions require redefinition, simplifying maintenance and preserving assertion integrity. This shows the robustness and adaptability of constructive assertions, contributing to more reusable and maintainable code.

```
Note? getNoteId(int id) {
var result = notes.any((e) => e.id == id) ?
  notes.firstWhere((e) => e.id == id) : null;
...
assert(result == (__notes.exists((e) => e.id == id) ?
 __notes.any((e) => e.id == id) : null));
 return result;
}
```

In the previous section, we briefly touched upon how we assert that a method initiates a UI update by indicating the expected *screenIndex* value. However, in our actual implementation code, we employed two assert statements: one in the pre-state to clear a flag and another in the post-state to verify if the flag is set to a specific value (refer to the skeletal code below). To facilitate this, we introduced an assertion-only method __*uiNotified*(), which serves the dual purpose of clearing the flag and checking it for a specific value.

```
assert(__uiNotified()); // clear the flag.
...
assert(__uiNotified(screenIndexNote); // check.
```

While effective, this technique has inherent challenges, especially when used to verify if a specific method was invoked during execution, such as state management framework methods. It can be error-prone and may require additional handling for nested cases. Investigating built-in mechanisms to assert the occurrence of method calls could simplify and streamline the formulation of such assertions (Cheon & Perumandla, 2007).

An assertion typically focuses on a single program state, representing either the initial state for a precondition or the final state for a postcondition. However, in scenarios where a postcondition requires referencing the initial value of a changing program variable, the introduction of a specification-only local variable becomes essential. Despite its utility, there are drawbacks. For objects, careful handling is crucial to ensure accurate cloning, particularly when the object's state might be altered by the code. While OCL collection classes offer automatic cloning for collections, manual intervention is needed for non-collection types. Caution is also warranted if the object undergoes mutation within assertions, as the order of assertions could impact correctness. The use of specification-only local variables may introduce namespace clutter and the risk of accidental references in the code, which could be addressed through a well-defined naming convention or annotation. Overcoming these challenges presents a promising avenue for future research and refinement.

In addition to assertion-only variables, assertion-only code—comprising code sections, functions, methods, and classes—can coexist within the asserted codebase. Unlike assertion-only variables, integrating assertion-only code within regular code does not pose issues, aside from its presence in the production code. It is advisable, however, to confine the use of assertion-only code to assertion-specific contexts. Clear indication of its purpose can be achieved through a well-defined naming convention or annotations. While most instances of assertion-only code are private within classes or libraries, certain scenarios may require public visibility. For instance, when introducing a method to clone a Note object for assertion purposes, the method needs to be public if the Note class and NotesModel class are in separate libraries. Abstraction functions serve as another example, enabling client assertions to manipulate values abstractly. Exploring the potential establishment of a specialized interface, such as an *assertion interface*, could be an intriguing avenue for future research, allowing client code to formulate assertions about objects with hidden states and enhancing the assertion framework's capabilities and flexibility.

6 CONCLUSIONS

This paper has introduced and explored the concept of constructive assertions as a valuable technique for enhancing code reliability and verification. By leveraging abstraction functions and assertion-only variables, we have demonstrated how constructive assertions offer a concise and comprehensible approach to specifying program behavior. The assertion library of immutable collection classes further underscores the potential of this approach. While challenges such as proper cloning and namespace pollution warrant consideration, the future holds promise for refining and expanding the utility of constructive assertions. As software development continues to evolve, the judicious application of this technique stands to contribute to more reusable, maintainable, and reliable software systems.

REFERENCES

Borgida, A., Mylopoulos, J., & Reiter, R. (1993). '. . . and nothing else changes': the frame problem in procedure specifications. *15th International Conference on Software Engineering (ICSE)*, IEEE, 303-314.

Bracha, G. (2016). *The Dart Programming Language*. Addison-Wesley.

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. (2014). Ensuring that your Dart will hit the mark: An introduction to Dart contracts. *International Conference on Information Reuse and Integration,* Redwood City, CA, August 13-15, IEEE, 369-377.

Cheon, Y. (2021). Toward more effective use of assertions for mobile app development. *International Conference on Progress in Informatics and Computing*, Shanghai, China, December 17-19, IEEE, 319-323.

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. & Leavens, G. T. (2002). A simple and practical approach to unit testing: The JML and JUnit way. In *16th European Conference on Object-Oriented Programming (ECOOP),* Malaga, Spain, June. *Lecture Notes in Computer Science*, 2374, 231–255, Springer.

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.

Cheon, Y. & A. Perumandla (2007). Specifying and checking method call sequences of Java programs. *Software Quality Journal*, 15(7), 7-25, Springer.

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. (2023). Retrieved from https://flutter.dev.

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

Jones, C. B. (1986). *Systematic Software Development Using VDM*. Prentice Hall.

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.

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

Object Management Group. (2023). *Object Constraint Language, version 2.4.* Retrieved November 13, 2023, from https://www.omg.org/spec/OCL/.

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

Spivey, J. (1989). *The Z Notation: A Reference Manual*. Prentice Hall.

Sitaraman, M., Weide, B. W., & Ogden, W. F. (1997). On the practical need for abstraction relations to verify abstract data type representations, *IEEE Transactions on Software Engineering*, 23(3), 157-170, March.

Warmer, J. & Kleppe, A. (2003). *The Object Constraint Language: Getting Your Models Ready for MDA* (2nd ed.). Addison-Wesley.

Watson, C., Tufano, M., Moran, K., Bavota, G., & Poshyvanyk, D. (2020). On learning meaningful assert statements for unit test cases. *IEEE/ACM 42nd International Conference on Software Engineering (ICSE),* Seoul, Korea, 1398-1409.

Zammetti, F. (2019). *Practical Flutter: Improve Your Mobile Development with Google's Latest Open-Source SDK*. Apress.