Enhancing Design-by-Contract with Frame Specifications

Yoonsik Cheon and Benjamin Good
Department of Computer Science, The University of Texas at El Paso, El Paso, Texas, U.S.A.

Keywords: Annotation, Design by Contract, Frame Condition, Runtime Check, Dart.

Abstract: This paper introduces an annotation-based approach to extending Design by Contract (DbC) with support for specifying and enforcing frame properties at runtime. Frame specifications, also known as *frame conditions* or *frame properties*, define which parts of a program's state may be modified during execution. Our approach models object states as abstract tuples, ensuring that runtime checks do not introduce unintended side effects. We implement a proof-of-concept prototype in Dart, utilizing compile-time instrumentation and runtime reflection to accommodate optional typing. By automating contract enforcement, this approach reduces the need for manual assertions, simplifies code maintenance, and enhances clarity by separating program logic from runtime checks. We evaluate its effectiveness in a cross-platform mobile application, comparing it to traditional assertion-based methods.

1 INTRODUCTION

Design by Contract (DbC) is a well-established software correctness methodology that emphasizes clear specifications of obligations and guarantees between software components (Meyer, 1992; Meyer, 1997; Mitchell and McKim, 2001; Ozkaya, 2019). By embedding formal contracts, such as preconditions, postconditions, and invariants, within the code, DbC ensures that software behaves as expected during execution. The fundamental premise is that the client and provider of a method must adhere to agreed-upon conditions, thereby fostering robustness and reliability.

However, while DbC effectively specifies what a method must achieve (via postconditions) and what constraints must hold before execution (via preconditions), it does not specify what parts of the program state may change. This missing aspect, known as *frame conditions*, is crucial for reasoning about program behavior modularly (Borgida et al., 1995). Frame conditions help ensure that methods do not inadvertently modify unrelated state, a critical property for maintaining system integrity—especially in complex applications where unintended side effects can lead to subtle and hard-to-detect errors. Indeed, unforeseen state changes have been identified as a root cause of serious software failures in both industrial and safety-critical systems (Amusuo et al., 2022; Gazzola et al., 2017). This highlights the importance of making state change constraints explicit and verifiable as part of the contract.

This paper introduces an annotation-based approach to integrating frame specifications into DbC for runtime verification. Our approach enables developers to specify, at the method level, which parts of an object or program state may be modified during execution. We model objects as tuples of abstract state components, allowing for fine-grained control over permissible modifications without unintended side effects. By leveraging compile-time instrumentation and runtime reflection, our approach enforces frame conditions dynamically, improving software reliability and maintainability. Additionally, we address challenges inherent to optionally typed languages, ensuring that our technique remains flexible and applicable across diverse programming environments.

For this study, we chose Dart (Bracha, 2016) as our implementation platform due to its increasing popularity in mobile app development and its unique language features, such as optional typing. To the best of our knowledge, no existing DbC framework provides built-in support for specifying and enforcing frame conditions at runtime. While prior research has explored static analysis and verification of frame properties (Hatcliff and Dwyer, 2001; Leino and Nelson, 2002; Marche et al., 2004; Kassios, 2006; Leino and Muller, 2006; Darvas and Leino, 2007; Hahnle et al., 2007), our work is the first to integrate runtime support for frame conditions within a DbC-based runtime checking system. A runtime-based technique

Cheon, Y. and Good, B.
Enhancing Design-by-Contract with Frame Specifications.
DOI: 10.5220/0013578400003964
In *Proceedings of the 20th International Conference on Software Technologies (ICSOFT 2025)*, pages 378-385
ISBN: 978-989-758-757-3; ISSN: 2184-2833

is especially important for dynamically or optionally typed languages like Dart, where static analysis alone may be insufficient or ineffective.

Existing work on DbC in Dart includes Chalin's Dart Contracts proposal (Chalin, 2014), which introduces mechanisms for specifying function and class behavior but lacks provisions for defining or enforcing frame properties. Our annotation-based approach addresses this gap by allowing developers to explicitly specify frame conditions in Dart and verify them at runtime. Specifically, besides `@Modifies` annotation for declaring frame conditions, our approach introduces the `@State` annotation, which provides an abstract representation of an object's state to facilitate frame checks (see Section 4.2). This concept is analogous to model variables in Behavioral Interface Specification Languages (BISLs) (Cheon et al., 2005), which define specification-only variables based on concrete program variables. Furthermore, the `parts` parameter within the `@State` annotation functions similarly to data groups (Leino, 1998), allowing developers to group mutable fields under an abstract state element for more structured and modular specification of frame conditions.

The rest of this paper is organized as follows. Section 2 reviews Design by Contract. Section 3 outlines key challenges in specifying frame conditions. Section 4 describes our annotation-based approach, including state modeling and verification. Section 5 details the prototype implementation. Section 6 presents our evaluation. Section 7 concludes with a summary and future work.

2 DESIGN BY CONTRACT

Design by Contract (DbC) is a programming methodology that enhances software robustness by explicitly defining formal agreements, or "contracts," between system components (Meyer, 1992; Mitchell and McKim, 2001; Ozkaya and Kloukinas, 2014). These contracts specify the mutual obligations and guarantees of components, focusing on preconditions, postconditions, and class invariants to ensure correctness and maintainability.

At the core of DbC are preconditions and postconditions, which define the expected conditions before and after a method's execution, respectively. Preconditions specify the constraints that must hold before a method is invoked. If these are satisfied, the method is guaranteed to fulfill its postconditions. Postconditions describe the expected state of the system after execution, ensuring that the method behaves as intended. Consider a simple bank account class with a method responsible for money withdrawal. The contract for this method can be expressed as follows:

```
@Requires('amount > 0 && amount <= balance')
@Ensures('balance == balance@pre - amount')
void withdraw(long amount) {
  balance -= amount;
}
```

In this example, the `withdraw()` method includes a precondition, denoted by the `@Requires` annotation, which states that the withdrawal amount must be positive (`amount > 0`) and must not exceed the current balance (`amount <= balance`). The postcondition, indicated by `@Ensures`, guarantees that the balance is correctly updated by deducting the withdrawn amount. The notation `balance@pre` refers to the account balance before the method execution, providing a reference for validation after the method completes. By explicitly defining these expectations and guarantees, DbC simplifies debugging, maintenance, and software evolution. Enforcing such contracts makes software behavior more predictable and reliable, reducing the risk of errors due to violated assumptions.

DbC also support class invariants—properties that must always hold in an object's stable state, typically before and after any public method execution. Invariants define integrity constraints such as "the balance must never be negative" and they apply across all methods that can affect an object's state.

While DbC enforces preconditions, postconditions, and invariants, it does not specify how state may change during a method's execution. This is where frame conditions come into play. Frame conditions define which parts of the state can change between a method's entry and exit, ensuring that methods do not inadvertently modify unrelated parts of the state. For example, in a bank account class, an invariant might ensure the balance is never negative, whereas a frame condition for the `withdraw()` method would restrict changes to the balance field and no other part of the state. Invariants describe a single state, while frame conditions relate two states: the pre-state and post-state. Additionally, not all frame properties can be captured as invariants, especially when expressing the absence of side effects, which is crucial for modular reasoning about code behavior.

3 MOTIVATION AND CHALLENGES

We aim to enhance DbC by integrating frame specifications, which go beyond traditional preconditions and postconditions to explicitly define which parts of a system's state can change during method execution. This added expressiveness improves contract clarity, reduces unintended side effects, and enables automatic runtime verification. While DbC traditionally ensures that specific conditions are satisfied before and after execution, it does not specify what must remain unchanged. Frame conditions fill this gap by listing all variables that can change—known as "and nothing else changes" (Borgida et al., 1995)—indirectly asserting that all other variables remain the same. By clearly stating which variables or structures may be modified, frame conditions help prevent subtle bugs caused by unintended state modifications, contributing to more robust and maintainable software. They also support automated runtime checks to verify that these constraints are upheld. If a method violates its frame conditions, violations are detected immediately, improving debugging and maintainability. Overall, frame conditions strengthen DbC by providing clearer and more enforceable specifications.

Although conceptually simple, implementing runtime frame checks involves significant challenges. Capturing an object's initial state and comparing it post-execution requires true cloning, which is nontrivial due to reference semantics. For example, in the `withdraw()` method:

```
void withdraw(long amount) {
  var preState = captureState(this);
  ...
  assert(ensureFrame(
    preState, captureState(this)));
}
```

Assigning `this` to `preState` only stores a reference, not a copy. Modifications affect both, so accurate preservation requires cloning, which many classes lack. Without built-in support, developers must implement custom cloning, which is error-prone and inconsistent.

Dynamic and optional typing in modern languages like Dart complicates frame checks further. Lacking explicit type information hinders compile-time generation of reliable check code. For example, enforcing immutability of unspecified fields in `withdraw()` becomes difficult without clear type declarations. Frame conditions also become complex with nested or composite objects. Determining whether to clone just the top-level object or its entire reference graph impacts both correctness and performance. For instance, verifying that only the `balance` changes while the associated customer object remains untouched requires deep cloning and precise specification. This also involves deciding between reference and value equality. Finally, runtime frame checks can introduce significant overhead. Capturing and comparing object states—especially when dealing with deep or complex structures—may impact performance, particularly in systems that require high throughput. Balancing the need for rigorous frame enforcement with performance considerations remains a key challenge, although this issue is not addressed in the current study.

4 OUR APPROACH

We introduce custom annotations to specify frame conditions for methods and translates these conditions into runtime checks. These annotations allow programmers to declare which elements of a program's state may be modified during method execution while ensuring that all other elements remain unchanged. For example, consider the annotated `withdraw()` method and its instrumented code:

```
@Modifies('balance')
void withdraw(long amount) {
  var preState = state;
  state.frame('this.balance');
  ...
  assert(preState == state);
}

get state => State({'this':
  {'balance': balance, ... }});
```

In this example, the `@Modifies` annotation specifies that only the `balance` field may be modified, while all other elements of the state must remain unchanged. The skeletal code generated from the annotation shows how the annotation enables a runtime check. Before the method executes, the current state is captured and stored in a local variable `preState`. A getter method, `state`, constructs a state model, or a snapshot of the current state, consisting of name-value pairs. The `frame()` method of the `State` framework class is used to capture information about the mutability of state elements. After the method execution, the stored initial state (`preState`) is compared with the final state to ensure compliance with the frame conditions. The overridden equality (`==`) operator is used to compare only those state elements that should not have been changed—i.e., elements not listed in the `@Modifies` annotation. We can also introduce class-specific state model classes along with methods to specify and check frame conditions (see Section 4.3). One limitation of our approach is that intermediate state changes cannot be detected. While this is acceptable in sequential environments, it may be unsuitable for concurrent settings where intermediate changes could be observed.

4.1 State and Object Models

Frame condition checks compare a program's state before and after method execution to verify compliance with specified constraints. The pre-state captures the program state before execution; the post-state, after. A frame condition declares which parts of the pre-state may change—implicitly asserting that the rest must remain unchanged. In our approach, the pre-state acts as a pattern against which the post-state is matched. Mutable parts are explicitly marked, and runtime checks confirm the post-state conforms to this pattern. The `State` class constructs abstract state snapshots, with its `frame()` method marking mutable fields, ensuring side-effect-free runtime check.

A program state includes snapshots of all accessible objects within a method's scope: the receiver (`this`), parameters, and global references. While the receiver's state can often be generated at compile time (refer to Section 4.3), dynamic or optional typing may require constructing states for parameters and globals at runtime. We model a program state as a mapping from variable names to object states. For example, in a board game method such as `placeStone(Player player, int x, int y)`, the relevant program state might include (see Section 6):

$$\{this : this, player : player, x : 10, y : 12\}$$

Objects such as boards or players may have nested structures. A program state captures top-level objects, while sub-objects represent nested states. Each object state maps field names to values or other object states, forming a hierarchical model. For example, a `Game` object with a `Board` and two `Player` objects (one of whom has the turn) appears in Figure 1.

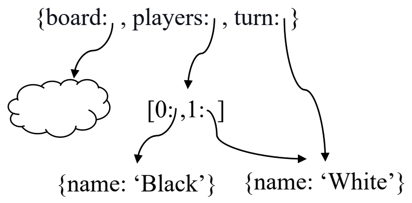

Figure 1: An object with nested structures.

The pre-state model serves two main purposes: (a) identifying which fields may be modified and (b) preserving the values of fields that must remain unchanged. To indicate that a field is mutable, we use a wildcard object (* to represent any value, which simplifies the post-state comparison.

$$\{board : board, players : players, turn : *\}$$

In this example, the pre-state model—augmented with frame conditions—allows changes to *turn*, while requiring *board* and *players* to remain unchanged.

4.2 Custom Annotations

We introduce custom annotations to specify frame conditions. The primary one, `@Modifies`, declares which parts of the program state a method may modify. All other state parts are implicitly required to remain unchanged. State parts are referenced using field notation, with comma-separated entries:

```
@Modifies('this.places[x][y]')
void placeStone(Player player, int x, int y)
```

As explained, we model objects abstractly as name-value maps: $\{f_1 : v_1, \ldots, f_n : v_n\}$. Names identify observable parts of an object's state. For example, `this.places[x][y]` accesses a specific element in a nested list. In Dart, object fields can also be accessed through user-defined getters and query methods. Our approach supports referencing such methods in `@Modifies`, making frame conditions applicable regardless of how the state is accessed.

```
@Modifies('this.getPlace(x,y)')
void placeStone(Player player, int x, int y)
```

4.2.1 Composite Objects

Objects often have nested structure. We use path expressions like `x.y.z` to refer to subcomponents. To improve expressiveness, we may allow limited wildcards—e.g., `x.p*` for all fields of `x` starting with p. Future work may explore the benefits of such patterns.

4.2.2 Collections

Collections—arrays, lists, sets, maps—are accessed using index notation (e.g., `a[10]`), which maps to getter/setter methods. Dart uses lists for arrays, making both interchangeable. To reference collection elements in frame conditions, we allow: (a) specific elements: `a[1, 10, size]`, (b) ranges: `a[1..10]`, and (c) all elements: `a[*]`. This syntax can be extended to cover insertions and deletions, capturing dynamic updates.

4.2.3 State Annotations

At runtime, a frame condition ensures that all framed state parts remain unchanged:

$$\forall x \bullet x \in \mathrm{State}_{\mathrm{pre}} \land x \in \mathrm{Framed} \Rightarrow x_{\mathrm{post}} \equiv x_{\mathrm{pre}}$$

In practice, the relevant objects are those accessible in the pre-state, such as `this` and parameters. Deep references via nested calls are harder to track and often irrelevant. To manage this, we provide annotations to include or exclude specific state elements: (a) explicit state such as public fields are included by default and (b) implicit state such as getters and observer methods must be explicitly annotated. Supported annotations include: (a) `@State.includes` and `@State.excludes` to control what counts as state and (b) `@State('name', 'expression')` to define custom state elements. For example, if player stats are fetched externally:

```
@State('stat', 'getStat(player)')
@Modifies('getPlace(x,y), stat.moves')
void placeStone(Player player, int x, int y)
```

Annotations can also define abstract state properties that encapsulate multiple fields into a single logical unit. For example, a player's full name—composed of first, middle, and last name fields—can be treated as one abstract state called `name`. This abstraction simplifies frame conditions by allowing related fields to be referenced collectively—for instance, referring to `name` instead of listing each individual component.

```
@State('name', '"$_first $_middle $_last"',
  parts: '_first, _middle, _last')

@Modifies('name')
void changeName( ...)
```

4.3 Translation to Runtime Checks

We translate frame conditions into runtime checks through a combination of compile-time instrumentation and runtime reflection. This hybrid approach accommodates Dart's optional typing system, where types may be implicit or only known at runtime. At compile time, methods annotated with frame conditions are identified, and instrumented code is inserted to enforce checks. These checks capture the method's initial state and verify it afterward against the specified frame constraints. When static type information is available (e.g., for the receiver `this`), it guides the generation of efficient code to access the object's state. Otherwise, the code relies on reflection to infer and access the structure dynamically.

Instrumentation performs four key steps: (a) identify accessible objects by determining which objects the method can access (receiver, parameters, relevant globals), using scope analysis and annotations, (b) capture pre-state by recording the initial state of all identified objects, (c) apply frame constraints by marking which parts of the state are allowed to change based on the annotation, and (d) verify post-state by comparing the final state with the initial state to ensure only allowed changes occurred.

Consider a method annotated with a frame condition:

```
@Modifies('board.getPlace(x,y),
  currentPlayer')
void placeStone(int x, int y)
```

This method modifies only a specific board position and the current player. The instrumented code might look like:

```
void placeStone(int x, int y) {
  // capture the pre-state.
  State preState = State({'this':
    {'board': board, 'players': players}});
  assert(preState.capture({'x':x, 'y':y})));

  // augment with frame condition.
  assert(preState.frame(
    'this.board.getPlace(x,y),'
    'this.currentPlayer'));

  // execute original code

  // verify the post-state.
  assert(preState == State(...)
    ..capture({'x': x, 'y': y}));
}
```

The code begins by capturing the pre-state using a `State` object, which stores the initial values of relevant objects. A call to `preState.capture()` extends this snapshot to include parameters `x` and `y`, although this is technically unnecessary for value types like `int` in Dart. This capture occurs within an `assert` statement to avoid runtime overhead in production. Frame conditions are then applied to identify mutable elements—in this case, `this.board.getPlace(x, y)` and `this.currentPlayer`—which are marked using wildcard objects. These wildcards, handled by the `frame()` method, signal permissible changes (see Section 5). After executing the method body, the instrumented code captures the post-state and asserts that only allowed modifications occurred, using the overriden `==` operation.

4.3.1 Object Comparison and Equality

To detect changes in object states, we compare pre- and post-state values using a customizable equality mechanism. Dart supports both reference equality (`identical()`) and value equality (`==`), and our system allows developers to choose the method for each field or object via annotations:

```
@State('currentPlayer', '_currentPlayer',
  equality=identity)
final Player _currentPlayer;
```

The `equality` parameter accepts built-in comparisons (==, `identical()`) or custom functions. Developers can also specify defaults at the class or application level. The equality choice affects how object states are captured: (a) reference equality stores only object identities, (b) value equality requires shallow or deep copies, and (c) custom equality may need tailored cloning, which can also be annotated. Snapshots for comparison may be captured eagerly (during pre-state initialization) or lazily (when applying frame constraints), optimizing storage. Choosing a default comparison is also important. Although many developers expect value equality, reference equality is faster and avoids object copying. Since Dart treats all values as objects, capturing everything as value state would be costly. We thus default to reference equality, allowing overrides as needed.

5 IMPLEMENTATION

We developed a proof-of-concept prototype for frame annotations in Dart, targeting cross-platform mobile apps. Dart was chosen for its platform reach and language features like optional typing, which pose both challenges and opportunities for design-by-contract (DbC) and frame conditions. The prototype currently supports basic annotations such as `@Modifies` and `@State`. Our implementation leverages several public Dart packages: (a) `source_gen` for automating code generation from frame annotations via custom generators, (b) `build` for orchestrating generator execution during compilation, and (c) `analyzer` for performing static code analysis.

A key challenge was Dart's optional typing. We addressed this with a hybrid approach that combines compile-time instrumentation and runtime reflection. When type information is available, the tool infers object structures and generates optimized code. Otherwise, it falls back on runtime inspection. For example, given known type information, the tool generates:

```
State preState = State({'this':
  {'board': board, ...}});
```

The structure of `this` is inferred at compile time. Its fields (e.g., `board`) are included as name-value pairs. If deeper abstraction is needed at runtime, reflection can construct nested state models.

To enforce frame conditions, we compare the final program state against the initial snapshot (`preState`). To facilitate this comparison, fields marked mutable in the initial state are replaced with wildcards using the `frame()` method of the `State` class, which applies transformations via reflection. For example, for the `@Modifies('this.board.getPlace(x,y)')` clause, the following code is generated:

```
preState.frame('this.board.getPlace(x,y)');
```

This method traverses the `preState` state, locating the specified path and replacing it with a wildcard. If intermediate objects like `board` are not already modeled as states, they are dynamically converted:

```
'board': State({'getPlace(x,y)': *, ...})
```

This ensures only declared changes are allowed, and any undeclared mutations can be detected. If a composite object isn't pre-modeled, its state is built at runtime to preserve consistency and isolation of side effects. At the core is the `State` class, which captures and compares program states. It supports composite structures, applies wildcard logic for mutable fields, and verifies frame compliance.

For production, all instrumented code is encapsulated within `assert` statements. Dart removes these in release mode (`--release`), ensuring no runtime overhead. Tree-shaking further eliminates unused code, resulting in efficient builds.

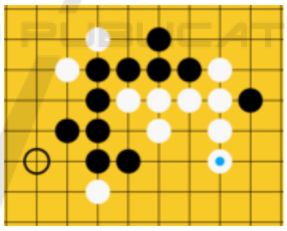
Figure 2: Screenshot of Omok game.

6 PRELIMINARY EVALUATION

To evaluate the effectiveness of our annotation-based approach, we conducted a small case study using a Dart/Flutter implementation of Omok (also known as Gomoku), a two-player strategy game played on a 15×15 grid (see Figure 2). Players take turns placing stones in an attempt to form an unbroken line of five—vertically, horizontally, or diagonally. We compared our approach to a previously proposed assertion-based method (Cheon et al., 2024), in which frame specifications are manually encoded as executable assertions within the source code to enable runtime checks.

The game logic was encapsulated in three main classes—`Board`, `Player`, and `Game`—totaling 256 source lines of code (SLOC). These classes defined 21 methods. To compare the annotation and assertion approaches, we specified frame properties for all methods, even though full coverage is uncommon in practice. Manually adding frame assertions increased code size by 59% to 408 SLOC, while the annotation-based version required only 284 SLOC—an 11% increase (see Table 1).

Table 1: Size complexities.

	SLOC	Increase (%)
Base code	256	N/A
Assertion	408	59
Annotation	284	11

These results show that annotations significantly reduce the overhead of specifying frame properties, making the code more concise and maintainable. The declarative style eliminates much of the boilerplate required by manual assertions.

Below is an excerpt of the `Game` class using our annotation-based approach to specify which parts of the state may be modified:

```
class Game {
  @State('board')
  final Board _board; // other fields ...

  @Modifies('this.* except players')
  void newGame() { ... }

  @modifiesNothing
  Player get currentPlayer => _currentPlayer;

  @Modifies('currentPlayer')
  Player changePlayerTurn() { ... }

  @Modifies('players[1]')
  set opponentPlayer(Player opponent) { ... }

  @Modifies('board.places[x][y],
    board.winRow, currentPlayer')
  Outcome makeMove(int x, int y) { ... }

  // other methods ...
}
```

In this example, the `newGame()` method modifies all fields except `players`. The `@modifiesNothing` shorthand is used for the read-only `currentPlayer` getter. Given that a majority of methods (57% in our prior study (Cheon et al., 2024)) are observers, this annotation simplifies specifications while improving clarity. Methods like `changePlayerTurn()` and `opponentPlayer()` illustrate how annotations can precisely limit modifications to specific fields, including individual elements (e.g., `players[1]`). The `makeMove()` method demonstrates fine-grained specification by allowing updates to only relevant board state components—`places[x][y]`, `winRow`—and `currentPlayer`. Composite objects such as `board` are treated abstractly, referencing only select internal state elements rather than the entire object.

Although our preliminary evaluation focused only on code size, the results demonstrate that annotations offer a practical and scalable approach for specifying and enforcing frame properties in Dart. By reducing source lines of code and decoupling frame specifications from method bodies, annotations improve code maintainability. This is particularly advantageous in optionally typed languages like Dart, where annotations provide a flexible mechanism for modeling state and mutability. When embedded directly in the codebase, frame annotations promote systematic management of state changes, thereby enhancing code clarity and minimizing unintended side effects.

Despite these benefits, several open challenges remain. One issue involves modeling state using getters, which is often necessary when dealing with encapsulated internal state—especially in third-party libraries or cloud-backed objects. However, using multiple getters to represent related aspects of state can lead to redundancy or ambiguity. A promising direction is to group related getters into unified state abstractions, simplifying frame conditions and improving readability. Another challenge lies in handling call-by-value semantics. In Dart, primitive types like `int` are passed by value, and any modifications within a method are local. However, reference types still allow mutation of the underlying object, which can result in visible side effects outside the method scope. Frame specifications must therefore distinguish between changes to parameter references and mutations to the objects they reference. Additional complexity stems from the distinction between reference and value models. Frame conditions must clarify whether a reference itself (e.g., a field like `board`) may be reassigned, or whether only its internal state (e.g., `places` and `winRow`) may be modified. This distinction is crucial for accurately expressing permissible state changes and maintaining consistent behavior.

Inheritance further complicates the specification of frame properties. It raises design questions such as whether subclasses should only extend the abstract state defined by their superclasses or be allowed to modify inherited state components. Similarly, when overriding methods, should subclasses be required to specify frame conditions that are at least as restrictive as those in the superclass? Balancing extensibility with contract integrity is essential for safe and predictable object-oriented design.

7 CONCLUSION

We presented an annotation-based approach for integrating frame properties into Design by Contract (DbC) in Dart, with a focus on runtime verification. A key contribution of our approach is the use of an abstract representation of object states, modeled as tuples, to ensure that runtime checks do not introduce unintended side effects. This abstraction maintains a clean separation between program logic and state management. Additionally, our method supports both statically and dynamically typed languages, enabling optional typing through a hybrid of compile-time and runtime mechanisms. By adding frame conditions in annotations, our approach can reduce code complexity, enhance maintainability, and simplifyes the verification process.

As part of future work, we plan to evaluate our technique on larger and more complex codebases to better assess its effectiveness, scalability, and broader impact on verification and validation, beyond just reducing code size. We also intend to explore how well the approach extends to key object-oriented features such as inheritance, method overriding, and polymorphism to ensure its robustness in diverse design contexts. Finally, we aim to refine our prototype into a practical and user-friendly tool to support developers in writing safer, more maintainable code.

REFERENCES

Amusuo, P. C., Sharma, A., Rao, S. R., Vincent, A., and Davis, J. C. (2022). Reflections on software failure analysis. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1615–1620, Singapore.

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

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

Chalin, P. (2014). Ensuring that your Dart will hit the mark: An introduction to Dart contracts. In *International Conference on Information Reuse and Integration*, pages 369–377, San Francisco, CA.

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

Cheon, Y., Liu, B., and Rubio-Medrano, C. (2024). Asserting frame properties. In *International Conference on Software Technologies (ICSOFT 2024)*, pages 145–152, Dijon, France.

Darvas, A. and Leino, K. R. M. (2007). Practical reasoning about invocations and implementations of pure methods. In *International Conference on Fundamental Approaches to Software Engineering*, pages 336–351.

Gazzola, L., Mariani, L., Pastore, F., and Pezze, M. (2017). An exploratory study of field failures. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, pages 67–77.

Hahnle, R., Schmitt, P. H., and Beckert, B. (2007). *Verification of Object-oriented Software: The Key Approach*. Springer.

Hatcliff, J. and Dwyer, M. (2001). Using the Bandera tool set to model-check properties of concurrent java software. In *CONCUR 2001–Concurrency Theory: International Conference*, pages 39–58, Aalborg, Denmark.

Kassios, I. T. (2006). Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods: 14th International Symposium on Formal Methods*, pages 268–283, Hamilton, Canada.

Leino, K. R. M. (1998). Data groups: Specifying the modification of extended state. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 144–153.

Leino, K. R. M. and Muller, P. (2006). A verification methodology for model fields. In *Programming Languages and Systems: 15th European Symposium on Programming*, pages 115–130, Vienna, Austria.

Leino, K. R. M. and Nelson, G. (2002). Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 24(5):491–553.

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

Meyer, B. (1992). Applying 'design by contract'. *IEEE Computer*, 25(10):40–51.

Meyer, B. (1997). *Object-Oriented Software Construction*. Prentice Hall, Upper Saddle River, NJ, 2nd edition.

Mitchell, R. and McKim, J. (2001). *Design by Contract, by Example*. Addison-Wesley, Boston, MA.

Ozkaya, M. (2019). Teaching design-by-contract for the modeling and implementation of software systems. In *International Conference on Software Technologies (ICSOFT)*, pages 499–507.

Ozkaya, M. and Kloukinas, C. (2014). Design-by-contract for reusable components and realizable architectures. In *ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE '14)*, pages 129–138.