Constructing Verifiably Correct Java Programs Using OCL and CleanJava

Yoonsik Cheon and Carmen Avila

TR #13-15
February 2013; revised: May 2013

Keywords: correctness proof, functional program verification, intended function, CleanJava, Object Constraint Language.

1998 CR Categories: D.2.2 [*Software Engineering*] Design Tools and Techniques — Modules and interfaces, object-oriented design methods; D.2.3 [*Software Engineering*] Coding Tools and Techniques — Object-oriented programming; D.2.4 [*Software Engineering*] Software/Program Verification — Class invariants, correctness proofs, formal methods; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — invariants, pre- and post-conditions, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A.

Constructing Verifiably Correct Java Programs Using OCL and CleanJava

Yoonsik Cheon and Carmen Avila
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas, U.S.A.
ycheon@utep.edu; ceavila3@miners.utep.edu

Abstract—**A recent trend in software development is building a precise model that can be used as a basis for the software development. Such a model may enable an automatic generation of working code, and more importantly it provides a foundation for correctness reasoning of code. In this paper we propose a practical approach for constructing a verifiably correct program from such a model. The key idea of our approach is (a) to systematically translate formally-specified design constraints such as class invariants and operation pre and postconditions to code-level annotations and (b) to use the annotations for the correctness proof of code. For this we use the Object Constraint Language (OCL) and CleanJava. CleanJava is a formal annotation language for Java and supports Cleanroom-style functional program verification. The combination of OCL and CleanJava makes our approach not only practical but also suitable for its incorporation into existing object-oriented software development methods. We expect our approach to provide a practical alternative or complementary technique to program testing to assure the correctness of software.**

Keywords: correctness proof, functional program verification, intended function, CleanJava, Object Constraint Language.

I. Introduction

A recent software development trend is a shift of focus from writing code to building models [1]. The ultimate goal is to systematically generate an implementation from a model through a series of transformations. One key requirement of this model-driven development is the availability of a precise model to generate working code from it. A formal notation such as the Object Constraint Language (OCL) [2] can play an important role to build such a precise model. OCL is a textual, declarative notation to specify constraints or rules that apply to models expressed in various UML diagrams [3]. Modeling and specifying design constraints explicitly is also said to improve reasoning of software architectures and thus their qualities [4].

A formal design model can also provide a foundation for correctness reasoning of an implementation. In this paper we propose one such a method that takes advantage of formal design models to construct verifiably correct programs. The key idea of our approach is to derive code-level annotations from a formal design and to prove the correctness of code using a Cleanroom-style functional program verification technique. We use OCL as the notation for formally documenting design decisions and constraints and CleanJava as the notation for writing code-level annotations. CleanJava is a formal annotation language for the Java programming language to support Cleanroom-style functional program verification [5] (see Section II-B for an overview of CleanJava). A functional program verification technique such as Cleanroom [6] [7] views a program as a mathematical function from one program state to another and proves its correctness by essentially comparing two functions, the function computed by the program and its specification [8] [9] [10]. Since the technique uses equational reasoning based on sets and functions, it requires a minimal mathematical background, and unlike Hoare logic [11] it supports forward reasoning, reflecting the way programmers informally reason about the correctness of a program.

It is a known fact that software contains defects. Defects are introduced during software development and are often found through testing. However, studies indicate that testing can't detect more than 90% of defects; 10% of defects are never detected through testing. As stated by a famous computer scientist, testing has a fundamental flaw in that it can show the existence of a defect but not its absence. We expect our approach to provide a practical alternative or complementary technique to program testing to assure the correctness of software. We believe that the combination of OCL and CleanJava make our approach more practical and approachable by practitioners.

There has been an approach proposed to combine Cleanroom methodologies and formal methods [12], however there is no work done on combining OCL and functional program verification. Stavely described an approach to integrating the Z specification notation [13] into Cleanroom-style specification and verification [14]. One interesting aspect of his work is that a Z specification is converted to a constructive form, expressing state changes in an assignment notation. In this way, a Z specification can serve as a specification function for the program code to be developed, and the development can proceeds in Cleanroom style by verifying every section of code. Our approach also takes advantage of OCL constraints written constructively by translating them automatically to CleanJava annotations using a set of translation rules. However, we also learned that such constraints raise some interesting questions (see Section VI). Another related work is the translation of OCL to JML [15]. JML is a behavioral interface specification language for Java [16] [17]. In this work, JML is used as an assertion language for Java in that a subset of OCL constraints is translated into JML assertions for both static reasoning and runtime checks. One important contribution of this work is the

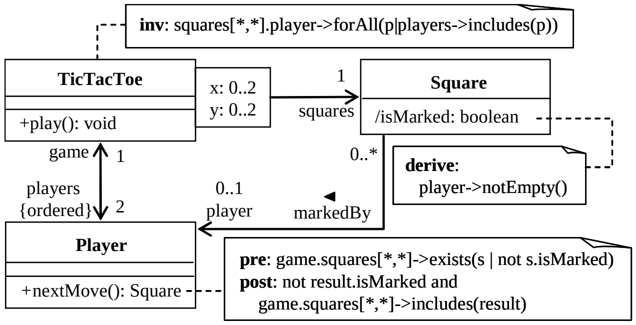


Fig. 1. UML class diagram with OCL constraints

```
//@ f0:[squares := Square[][]->any(sqs| isGameOver(sqs))]
  //@ f1:[p := nextPlayer()]
  Player p = nextPlayer();

  /*@ f2:[squares, p := Square[][]->any(sqs| isGameOver(sqs)
    @  && isSubState(squares, sqs)), anything] where
    @  isSubState(s1,s2) = (* s1 is substate of s2 *) @*/
  while (!isWonBy(p) && hasEmptySquare()) {
   /*@ f3:[sq.player, p := p, nextPlayer()]
     @  where sq = p.nextMove() @*/
    p = nextPlayer();
    Square sq = p.nextMove();
    sq.setPlayer(p);
  }
```

Fig. 2. Sample CleanJava code

translation rules from OCL to JML. Assertions are said to be more effective when derived from formal specifications, and several different techniques have been proposed for translating OCL constraints to runtime assertion checks [18].

The remainder of this paper is structured as follows. In Section II we briefly explain OCL and CleanJava using an example. In the subsequent two sections we first give an overview of our approach and then apply it to our running example. In Section V we describe our translation of OCL constraints to CleanJava annotations, and in Section VI we discuss some interesting aspects of our translation. In Section VII we provide a concluding remark.

II. Background

A. Object Constraint Language

The Object Constraint Language (OCL) [2] is a textual, declarative notation to specify constraints or rules that apply to UML models. OCL can play an important role in model-driven software development because UML diagrams lack sufficient precision to enable the transformation of a UML model to complete code. In fact, it is a key component of OMG's standard for model transformation for the model-driven architecture [19].

A UML diagram alone cannot express a rich semantics of and all relevant information about an application. The diagram in Figure 1, for example, is a UML class diagram modeling the game of tic-tac-toe. A tic-tac-toe game consists of 9 places in a 3×3 grid, and two players take turns to mark the places and win the game by marking three places in a horizontal, vertical, or diagonal row. However, the class diagram doesn't express the fact that a place can be marked only by the two players participating in the game. It is very likely that a system built based only on diagrams alone will be incorrect. OCL allows one to precisely describe this kind of additional constraints on the objects and entities present in a UML model. It is based on mathematical set theory and predicate logic and supplements UML by providing expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. The above-mentioned fact, for example, can be expressed in OCL as follows.

```
context TicTacToe
  inv: squares[*,*].player->forAll(p|players->includes(p))
```

This constraint, called an *invariant*, states a fact that should be always true in the model. The invariant is written using OCL collection operations such as `forAll` and `includes`; the `forAll` operation tests whether a given condition holds for every element contained in the collection, and the `includes` operation tests whether an object is contained in a collection.

It is also possible to specify the behavior of an operation in OCL. For example, the following OCL constraints specifies the behavior of an operation `Player::nextMove():Square` using a pair of predicates called *pre* and *postconditions*.

```
context Player::nextMove():Place
  pre: game.squares[*,*]->exists(s|not s.isMarked)
  post: not result.isMarked and
        game.squares[*.*]->includes(result)
```

The above pre and postconditions states that if invoked in a state that has at least one unmarked square the operation returns an unmarked square. In the postcondition, the keyword `result` denotes the return value.

B. CleanJava

CleanJava is a formal annotation language for the Java programming language to support Cleanroom-style functional program verification [5]. In the functional program verification, a program is viewed as a mathematical function from one program state to another. In essence, functional verification involves calculating the function computed by code, called a *code function*, and comparing it with the intention of the code written also as a function, called an *intended function* [8] [9] [10]. CleanJava provides a notation for writing intended functions. A *concurrent assignment* notation, $[x_1, x_2, \ldots, x_n := e_1, e_2, \ldots, e_n]$, is used to express these functions by only stating changes that happen. It states that x_i's new value is e_i, evaluated concurrently in the initial state—the state just before executing the code; the value of a state variable that doesn't appear in the left-hand side remains the same. For example, $[x, y := y, x]$ is a function that swaps the values of two variables x and y.

Figure 2 shows sample Java code annotated with intended functions written in CleanJava. It shows partial code of the `play` method of the TicTacToe class. Each section of code is annotated with its intended function. A CleanJava annotation is written in a special kind of comments either preceded by

`//@` or enclosed in `/*@` and `@*/`, and an intended function is written in the Java expression syntax with a few CleanJava-specific extensions. The first annotation labelled f_0 states that the new value of the `squares` field is an arbitrary value of a game-over state. In CleanJava, a type such as Square[][] can be used to denote the set of all values belonging to it, and `any` is a collection iterator that denotes an arbitrary value of a collection that satisfies a given condition; CleanJava defines several other collection iterators such as `forAll` and `exists`. The intended function labelled f_2 is interesting, as it shows several features of CleanJava. First, the keyword **`anything`** denotes an arbitrary value and its use indicates that one doesn't care about the final value of the local variable `p`. Second, a **`where`** clause introduces local definitions like the `isSubState` function. Third, in CleanJava one can escape from formality and mix a formal text such as a Java expression with an informal description, any text enclosed in a pair of `(*` and `*)`. For example, the notion of substate between two Square[][] objects—i.e., the `isSubState` function—is defined informally. The example also shows that one can omit the signature of a function introduced for use in annotations. It is automatically inferred by CleanJava and such a function typically defines a polymorphic function. The following is one possible formulation of the `isSubState` function with its signature completely specified.

```
boolean isSubState(Square[][] s1, Square[][] s2) =
  s1.length == s2.length &&
  CJSet{1..s1.length}->forAll(int i|
    s1[i].length == s2[i].length &&
    CJSet{1..s1[i].length}->forAll(int j|
      s1[i][j] == s2[i][j] &&
        (s1[i][j].isMarked ==>
          s1[i][j].getPlayer() == s2[i][j].getPlayer())))
```

If code is annotated with its intended function, its correctness can be proved formally. It would be instructive to sketch a correctness proof of the code shown in Figure 2. It requires the following proof obligations.

- Proof that the composition of functions f_1 and f_2 is correct with respect to, or a refinement ($\sqsubseteq$) of, f_0, i.e., $f_1; f_2 \sqsubseteq f_0$, where ";" denotes a functional composition.
- Proof that f_1, f_2, and f_3 are correctly refined by the corresponding code.

In functional verification, a proof is often trivial or straightforward because a code function can be easily calculated and directly compared with an intended function; for example, f_1 and f_3 are both code and intended functions. However, one often need to use different techniques such as a case analysis for an **`if`** statement and an induction for a **`while`** statement as in the proof of f_2 [9] [10]. Below we discharge the first proof obligation, where T is short for Square[][].

$f_1; f_2 \equiv$ [p := nextPlayer()];
[squares, p := $T \rightarrow$any(sqs| isGameOver(sqs) && isSubState(squares, sqs)), anything]
$\equiv$ [squares, p := $T \rightarrow$any(sqs| isGameOver(sqs) && isSubState(squares, sqs)), anything]
$\sqsubseteq$ [squares := $T \rightarrow$any(sqs| isGameOver(sqs) && isSubState(squares, sqs)]
$\sqsubseteq$ [squares := $T \rightarrow$any(sqs| isGameOver(sqs))]
$\equiv f_0$

III. Overview of Our Approach

The key idea of our approach is (a) to derive code annotations from formal designs and (b) to prove the correctness of code in Cleanroom-style functional verification by refining the derived annotations. We use OCL as the notation for formally documenting design decisions and details and CleanJava as the notation for writing code annotations. There are several advantages in using OCL as a formal design notation compared to more traditional formal specification languages such as Z [13]. It is a textual formal specification language that provide concise and precise expressions that have neither the ambiguities of natural language nor the inherent difficulty of using complex mathematics. As part of the standard modeling language UML, it allows one to specify and attach constraints and rules to various design models expressed in diagrams. From UML dynamic models with OCL constraints, e.g., state machine diagrams, it is also possible to derive working code (see Section IV for an example). There are also advantages in using CleanJava as the annotation notation and verification technique, compared to Hoare-style assertions. Unlike Hoare logic based on the first-order predicate logic, the technique requires a minimal mathematical background by viewing a program as a mathematical function from one program state to another and by using equational reasoning based on sets and functions. The reasoning in Hoare logic is backward in that one derives (weakest) preconditions from postconditions. This is similar to reading source code backward from the last line to the first. The functional program verification technique supports a forward reasoning by reflecting the way programmers reason about the correctness of a program informally. The combination of OCL and CleanJava will make our approach more approachable to Java programmers and practitioners.

The main steps of our approach are as follows.

1) Document a design using UML diagrams along with OCL constraints specifying design decisions and details.
2) Generate skeleton or working code from UML design models.
3) Translate OCL constraints to CleanJava intended functions to annotate the generated code.
4) Write algorithms to complete the skeleton code by refining the intended functions.
5) Verify the correctness of the algorithm code with respect to its intended function.

The last two steps may be performed simultaneously in a stepwise refinement fashion. In the next section, we will illustrate these steps in detail by applying them to our tic-tac-toe example.

IV. Illustration

In this section we illustrate our proposed approach by applying it to the running example. As sketched in the previous

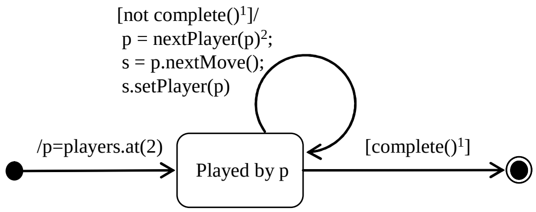
```
context TicTacToe
  inv: squares[*,*].player->forAll(p|players->includes(p))

context TicTacToe::TicTacToe()
  post: squares[*,*]->forAll(s|not s.isMarked)

context TicTacToe::play():void
  pre: squares[*,*]->forAll(s|not s.isMarked)
  post: isWonBy(players->at(1)) or isWonBy(players->at(2))
        or not hasEmptySquare()

context TicTacToe::isWonBy(p: Player): boolean
  body: Set{0..2}->exists(i|Set{0..2}->
        forAll(j|getSquare(i,j).isMarkedBy(p)))
    or Set{0..2}->exists(i|Set{0..2}->
        forAll(j|getSquare(i,j).isMarkedBy(p)))
    or Set{0..2}->collect(i|getSquare(i,i))->
       forAll(s|s.isMarkedBy(p))
    or Set{0..2}->collect(i|getSquare(i,2-i))->
       forAll(s|s.isMarkedBy(p))

context TicTacToe::hasEmptySquare(): boolean
  body: squares[*,*]->exists(s|not s.isMarked)

context TicTacToe::getSquare(i: int, j: int): Square
  pre: 0 <= i and i <= 2 and 0 <= j and j <= 2
  post: result = squares[i,j]

context Square::isMarked: boolean
  derive: player.notEmpty()

context Square::isMarkedBy(p: Player): boolean
  body: player = p

context Player::Player(g: TicTacToe)
  post: game = g

context Player::nextMove(): Square
  pre: game.hasEmptySquare()
  post: not result.isMarked and
        game.squares[*,*]->includes(result)
```

Fig. 3. OCL constraints for tic-tac-toe

section, the first step is to document a detailed design using UML diagrams along with OCL constraints.

1) Detailed design in UML and OCL: We elaborate our class diagram model by adding OCL constraints to the model and documenting detailed design decisions. Figure 3 shows OCL constraints for classes TicTacToe, Square, and Player along with several new operations introduced. In OCL, we document class invariants, operation pre and postconditions, values for derived attributes (e.g., `isMarked` of class Square), and return values of query operations (e.g., the `isWonBy` operation of class TicTacToe and the `isMarkedBy` operation of class Square). In addition to class invariants and operation pre and postconditions, OCL provides several other constructs, some of which are used in the example. The **`body`** construct defines the result of a query operation, and the **`derive`** construct specifies the value of a derived attribute or association end. The collection operation `at` appearing in the postcondition of the `play` operation returns the element at the given index; OCL uses 1-based index. The notation `Sequence{0..2}` denotes a sequence consisting of numbers from 0 to 2, inclusive.

It is also possible to define detailed algorithms for important operations using a combination of UML diagrams and OCL. For example, we can define an algorithm for the `play()` operation of the TicTacToe class using a UML state machine diagram, as shown below.



[1]complete() ≡ isWonBy(p) or not hasEmptySquare()
[2]nextPlayer(p) ≡ players->any(q | q <> p)

The state machine is called a *behavior state machine* and specifies that each player takes a turn to make a move—i.e., mark a square—until a play becomes completed. A play is complete if it is won by a player or there is no more empty square left. A behavior state machine can be used to derive implementation code (see below).

2) Skeleton code: The next step is to derive skeletal code from UML diagrams such as class diagrams. From a detailed class diagram, skeletal code such as shown below can be systematically or automatically generated.

```
public class TicTacToe {
  private Square[][] squares;
  private Player[] players;
  public TicTacToe() { ... }
  public void play() {... }
  public boolean isWonBy(Player p) { ... }
  public boolean hasEmptySquare() { ... }
  public Square getSquare(int i, int j) { ... }
}

public class Square {
  private Player player;
  public void setPlayer(Player p) { player = p; }
  public Player getPlayer() { return player; }
  public isMarkedBy(Player p) { ... }
  public boolean isMarked() { ... }
}

public class Player {
  private TicTacToe game;
  public Player(TicTacToe g) { ... }
  public Square nextMove() { ... }
}
```

For an association like `markedBy`, a pair of getter and setter methods (e.g., `getPlayer` and `setPlayer`) can also be automatically generated using the role names of the association ends (e.g., `player`). A derived attribute such as `isMarked` of class Square is translated to a query method.

This step may require making important implementation decisions such as deciding data structures. For example, we decided to represent the qualified association between TicTacToe and Square using a two-dimensional array. Such decisions often have impacts on the way we translate OCL constraints to CleanJava annotations in the following step, as CleanJava annotations are usually expressed in terms of concrete representation values.

3) OCL-to-CleanJava Translation: We next translate OCL constraints to CleanJava annotations and add them to the skeletal code. Figure 4 shows the skeletal code of class TicTacToe annotated in CleanJava. Most annotations are direct translations of the corresponding OCL constraints such as

```
public class TicTacToe {
 /*@ inv: [squares.length == 3 &&
   @      squares->forAll(Square[] sqs|sqs.length == 3)] @*/

 /*@ inv: [players.length == 2]

 /*@ inv: [squares->forAll(Square[] sqs|
   @       sqs->forAll(Square sq| !sq.isMarked() ||
   @       players->includes(sq.getPlayer()) @*/

  private Square[][] squares;
  private Player[] players;

 /*@ [square := Square[][]->any(Squares[][] sqs|
   @  isPristine(sqs))] @*/
  public TicTacToe() { ... }

  /*@ [isPristine(sqs) ->
    @  squares := Square[][]->any(Square[][] sqs|
    @   isGameOver(sqs))] @*/
  public void play() {... }

  /*@ [result : = isWonBy(squares, p)] @*/
  public boolean isWonBy(Player p) { ... }

  /*@ [result := squares->exists(Square[] sqs|
    @  sqs->exists(Square sq| !sq.isMarked())) @*/
  public boolean hasEmptySquare() { ... }

  /*@ [0 <= i && i <= 2 && 0 <= j && j <= 2 ->
    @  result := squares[i][j]] @*/
  public Square getSquare(int i, int j) { ... }

  /*@ fun boolean isPristine(Square[][] sqs) =
    @  sqs->forAll(Square[] sq|
    @   sq->forAll(Square s| !s.isMarked())) @*/

  /*@ fun boolean isGameOver(Square[][] sqs) =
    @  isWonBy(sqs, players[0])
    @  || isWonBy(players[1])
    @  || sqs->forAll(Square[] sq|
    @      sq->forAll(Square s| s.isMarked())) @*/

  /*@ fun boolean isWonBy(Square[][] sqs, Player p) =
    @  CJSet{0..2}->exists(int i| CJSet{0..2}->
    @    forAll(int j|sqs[i][j].isMarkedBy(p)))
    @  || CJSet{0..2}->exists(int i|CJSet{0..2}->
    @     forAll(int j|sqs[i][j].isMarkedBy(p)))
    @  || CJSet{0..2}->collect(int i|sqs[i][i]->
    @     forAll(Square s|s.isMarkedBy(p))
    @  || CJSet{0..2}->collect(int i|sqs[i][2-i])->
    @     forAll(Square s|s.isMarkedBy(p)) @*/
}
```

Fig. 4. Skeletal code with CleanJava annotations

invariants and pre and postconditions. However, the first two invariants are specific to the Java language and constraint the sizes of arrays. This is because the array size is not part of an array type in Java. As shown, OCL invariants are translated to CleanJava invariants [20], and pre and postconditions are translated to CleanJava intended functions. In general, pre and postconditions of the form **pre:** P **post:** Q are translated to an intended function of the form $[P' \rightarrow v_1, v_2, ..., v_n := E_i, ..., E_n]$, where P' is P written in the CleanJava syntax and v_i's and E_i's are derived from Q (see Section V for details). As shown, a concurrent assignment may have an optional condition or guard followed by an $\rightarrow$ symbol. This conditional concurrent assignment statement specifies a partial function that is defined only when the condition (P') holds. The example also shows that one can introduce mathematical functions (e.g., `isPristine`, `isGameOver`, and `isWonBy`) for the purpose of writing annotations.

4) Code Writing: Once a method is annotated with an intended function, the next step is to come up with working code—the method body. There are several possibilities here. It can be developed independently by referring to its pre and postconditions or the intended function. The intended function may be refined to working code in a stepwise refinement fashion. Yet another possibility is—if a detailed algorithm design was done and documented using a UML diagram such as a state machine diagram—to derive working code from a formal design model by systematically translating it. For example, it is straightforward to derive the following code for the `play()` method of the TicTacToe class from the behavior sate machine that describes its algorithm (see Section IV).

```
Player p = players[1];
while (!isWonBy(p) && hasEmptySquare()) {
  p = p == players[0] ? players[1] : players[0];
  Square sq = p.nextMove();
  sq.setPlayer(p);
}
```

5) Formal Verification: We verify the correctness of code by documenting each section of the code with an intended function and performing a functional program verification as described in Section II-B. We prove that the code is correct with respect to its intended function. If code was derived from a formally specified algorithm model such as a state machine and the algorithm was proved to be correct, the code may be correct by the way it was constructed provided that the algorithm model was transformed to code by following a set of transformation rules [21]. If a stepwise refinement was used to construct the code, the correctness proof may have already been performed as part of the refinement. In addition to intended functions and method bodies, we also need to prove the correctness of class invariants, if any. Essentially, we need to proved that each class invariant is established by the constructors of a class and preserved by all other methods of the class [20].

V. Translating OCL to CleanJava

An important component of our approach is translating OCL constraints to CleanJava annotations. We believe that this translation can be systematically done and even be automated by defining transformation rules. As an example, let's consider the invariant of the TicTacToe class shown below.

```
inv: squares[*,*].player->forAll(p|players->includes(p))
```

The constraint refers to two associations of class TicTacToe (`squares` and `players`) and an attribute of class Square (`player`). Remember that `squares` is the role name of a qualified association from TicTacToe to Square (see Figure 1 in Section II-A). If we know how these UML elements are reified in an implementation, we should be able to translate the OCL invariant to a CleanJava invariant by replacing UML/OCL elements with the corresponding Java/CleanJava elements. The following is one possible translation presented in the previous section.

```
inv: [squares->forAll(Square[] sqs|
      sqs->forAll(Square sq| !sq.isMarked() ||
      players->includes(sq.getPlayer())))]
```

However, a more direct and systematic translation would be to map each OCL construct to the corresponding CleanJava constructs. If there is no corresponding CleanJava construct, we can introduce a user-defined function for it (see below).

```
inv: [allPlaces(squares)->collect(Squares s| s.getPlayer())
      ->forAll(Player p|players->includes(p))] where
    CJSet<Square> allSquares(Square[][] sqs) = sqs->iterate(
      Square[] sq; CJSet<Square> r = new CJSet<Square>()|
        r.addAll(CJSet.fromArray(sq)))
```

In this translation, the reference to the qualified association end, `squares[*,*]`, is now translated to a user-defined function `allSquares` that, given a 2-dimensional array of squares, returns a set consisting of all the squares contained in the given array; the function is defined using the `iterate` collection operator. Also note that the dot notation in OCL when navigating an association (e.g., `squares[*,*].player`) is short for the `collect` iteration operator. Thus, it is translated to the CleanJava `collect` iteration operator.

The translation of pre and postconditions could be more involved depending on how they are written in OCL. This is because a functional program verification technique and notation is fundamentally different from an assertion-based technique and notation such as Hoare logic [11] and OCL. It is direct and constructive in that for each state variable such as a program variable one must state its final value explicitly. On the other hand, an assertion-based technique is indirect and constraint-based in that one specifies the condition that the final state has to satisfy by stating a relationship among state variables. The final value of a state variable isn't defined directly but instead is constrained and given indirectly by the specified condition.

As described in the previous section, pre and postconditions are translated to an intended function written using a conditional concurrent assignment. If there is a precondition, the translation produces a partial function of the form, $[P \rightarrow v_1, v_2, ..., v_n := E_i, ..., E_n]$, where P is the translation of the OCL precondition and v_i's and E_i's are derived from the OCL postcondition. For the translation of a postcondition, we can think of two different cases. If it is written in a constructive form, e.g, $x_1 = E_1$ and $x_2 = E_2$ and $\cdots$ and $x_n = E_n$, one possible translation would be $[x_i, x_2, ..., x_n := E'_1, E'_2, ..., E'_n]$, where E'_i is a CleanJava translation of E_i. An example is the postcondition of the `getSquare` operation of TicTacToe class, `result = squares[i,j]`, which is straightforwardly translated to `[result := squares[i][j]]`. If a postcondition is not written constructively, its translation is more involved and complicate. There are several such postconditions in our TicTacToe example, including that of the `nextMove` operation of class Player, shown below.

```
context Player::nextMove(): Square
  pre: game.hasEmptySquare()
  post: not result.isMarked and
        game.squares[*,*]->includes(result)
```

However, it is also possible to translate these postconditions systematically and perhaps even automatically. One possibility is to use the `any` iteration operator that returns an arbitrary element of a collection that meets a given condition. Consider a postcondition $P(x_1, x_2, \cdots, x_n)$, written in terms of mutable state variables x_i's like class attributes and the return value. The new values of x_i's collectively have to satisfy the constraint P. Thus, the postcondition can be translated to:

```
[x1, x2, ..., xn :=
  T1->any(T1 x1'|
   T2->any(T2 x2'|
     ...
      Tn->any(Tn x2'|P'(x1, x2, ..., xn)))))]
```

where P' is a CleanJava translation of P. For example, the pre and postconditions of the above `nextMove` operation can be translated to the following intended function.

```
[game.hasEmptySquare() ->
  result := Square->any(Square s| !s.isMarked() &&
    allSquares()->includes(s)) where
    allSquares() = /* ... */
```

VI. Discussion and Evaluation

There are a few interesting questions about translating OCL constraints to CleanJava annotations. OCL provides a special treatment for undefinedness of an expression and thus uses a three-valued (*true*, *false*, and *undefined*) propositional logic. This leads to an unpleasant consequence not only in correctness proof[1] but also in our translation of OCL constraints to CleanJava annotations. For example, the OCL disjunction operator (**or**) cannot be directly translated to the Java logical disjunction operator (||). In OCL, E_1 **or** E_2 is true even if E_1 is undefined as long as E_2 is true. In Java, however, the result of E_1 || E_2 is an exception (i.e., undefined) if the evaluation of E_1 throws an exception. Operationally the equivalent Java code is:

```
boolean result = false;
Exception first = null;
try { result = E1'; }
catch (Exception e) { first = e; }
finally {
  if (!result) result = E2';
  if (!result && e != null) throw first;
}
```

There seems to be no simple and natural way of translating this OCL expression to CleanJava that is faithful to the standard OCL semantics. One possibility is to introduce a CleanJava-specific conjunction operator with the same semantics as the standard OCL, but its usefulness in general is questionable.

We said in the previous section that if a postcondition is written in a constructive form, e.g., $x = E$, we translate it to an intended function of the form, $[x := E]$. But what if E is also a mutable state variable, say y, to give a postcondition of the form $x = y$? The assertion states that x and y have an equal value in the final state. Thus, in addition to the intended

[1]For example, a well-known law of propositional logic, $A \Rightarrow B = \neg A \vee B$, doesn't hold in OCL [22].

function $[x := y]$, $[y := x]$ is also a correct refinement. In fact, there are numerous correct implementations including $[x, y := 0, 0]$. However, we learned that in most cases when one writes an OCL constraint like $x = y$ the intention was in fact $x = y$ **and** $y = y$@pre. In OCL, y@pre denotes y's initial value, and such a conjunct is needed because OCL doesn't provide a special construct for stating a frame axiom or property. Thus, we think our translation scheme is reasonable. If a postcondition is not written constructively, we used the `any` iteration operator to translate it. This allows us to systematically and possibility automatically translate OCL constraints. However, the `any` operator is similar to the μ operator in Z [13], and the resulting expression is not in a form that is easy to manipulate in verification using equational reasoning. Fortunately, however, our empirical study indicates that a significant fraction of OCL constraints is written constructively; e.g., 67% of OCL constraints for our tic-tac-toe example were written constructively.

We are currently elaborating and refining our approach as well as formulating the OCL-CleanJava translation rules. We are also assessing and evaluating our approach using more realistic case studies. The preliminary result is very promising in that we were able to systematically translate OCL constraints to CleanJava annotations and to prove the correctness of implementation code. In fact we found that an intended function often times provided a good guidance to a possible implementation. For example, we coded CleanJava user-defined functions as (private) helper methods, and an iteration operator such as `forAll` triggered an introduction of a loop in implementation code. The structure and constructs of a CleanJava annotation are frequently reflected in the implementation code, providing an additional assurance that the code conforms to its design.

VII. Conclusion

In this paper we proposed a new method that can complement testing as a practical software verification and validation technique. Our approach takes advantage of recent emphasis and advances on software modeling and systematically translates formally-specified design constraints such as class invariants and operation pre and postconditions written in OCL to code-level annotations written in CleanJava. The translated CleanJava annotations are refined to correct implementations in a stepwise refinement fashion or used for the correctness proof of the implementation code using a Cleanroom-style functional program verification technique.

We believe that our combination of OCL and CleanJava provides several advantages. CleanJava supports Cleanroom-style functional program verification, where a program is viewed as a mathematical function from one program state to another and a correctness proof is done by essentially comparing two functions, the function computed by the program and its specification. Since the technique uses equational reasoning based on sets and functions, it requires a minimal mathematical background, and unlike Hoare logic it supports forward reasoning, reflecting the way programmers informally reason about the correctness of a program. Thus, our approach will be more approachable to Java programmers and practitioners. Since OCL is part of the standard modeling language UML, it would be easier to adopt our approach and incorporate or integrate into existing object-oriented software development methods.

Acknowledgment

This work was supported in part by DUE-0837567.

References

[1] A. W. Brown, "Model driven architecture: Principles and practice," *Software and System Modeling*, vol. 3, no. 4, pp. 314–327, Dec. 2004.
[2] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd ed. Addison-Wesley, 2003.
[3] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*, 2nd ed. Addison-Wesley, 2004.
[4] A. Tang and H. van Vliet, "Modeling constraints improves software architecture design reasoning," in *Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture 2009*, 2009, pp. 253–256.
[5] Y. Cheon, C. Yeep, and M. Vela, "The CleanJava language for functional program verification," *International Journal of Software Engineering*, vol. 5, no. 1, pp. 47–68, Jan. 2012.
[6] H. D. Mills, M. Dyer, and R. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, Sep. 1987.
[7] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering*. Addison Wesley, Feb. 1999.
[8] A. Stavely, *Toward Zero Defect Programming*. Addison-Wesley, 1999.
[9] Y. Cheon, "Functional specification and verification of object-oriented programs," Department of Computer Science, The University of Texas at El Paso, 500 West University Ave., El Paso, TX, 79968, Tech. Rep. 10-23, Aug. 2010.
[10] Y. Cheon and M. Vela, "A tutorial on functional program verification," Department of Computer Science, The University of Texas at El Paso, Tech. Rep. 10-26, Sep. 2010.
[11] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of ACM*, vol. 12, no. 10, pp. 576–580,583, Oct. 1969.
[12] Z. Langari and A. B. Pidduck, "Quality, cleanroom and formal methods," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.
[13] J. M. Spivey, *Understanding Z: a Specification Language and its Formal Semantics*. New York, NY: Cambridge University Press, 1988.
[14] A. M. Stavely, "Integrating Z and Cleanroom," in *LFM2000: Fifth NASA Langley Formal Methods Workshop*, Jun. 2000, pp. 141–150.
[15] A. Hamie, "Towards verifying java realizations of OCL-constrained design models using JML," in *6th IASTED International Conference on Software Engineering and Applications*, 2002.
[16] G. T. Leavens, "Tutorial on JML, the Java Modeling Language," in *ASE '07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2007, p. 573.
[17] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An overview of JML tools and applications," *International Journal on Software Tools for Technology Transfer*, vol. 7, no. 3, pp. 212–232, Jun. 2005.
[18] C. Avila, A. Sarcar, Y. Cheon, and C. Yeep, "Runtime constraint checking approaches for OCL, a critical comparison," in *Proceedings of SEKE 2010, The 22-nd International Conference on Software Engineering and Knowledge Engineering, July 1-3, 2010, San Francisco, CA*, 2010, pp. 393–398.
[19] D. S. Frankel, *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, Jan. 2003.
[20] C. Avila and Y. Cheon, "Functional verification of class invariants in CleanJava," in *Innovations and Advances in Computer, Information, Systems Sciences, and Engineering*, ser. Lecture Notes in Electrical Engineering, vol. 152. Springer-Verlag, Aug. 2012, pp. 1067–1076.
[21] K. Lano, *Model-Driven Software Development with UML and Java*. Course Technology, 2009.
[22] R. Hennicker, H. Hussmann, and M. Bidoit, "On the precise meaning of OCL constraints," in *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*. London, UK, UK: Springer-Verlag, 2002, pp. 69–84.