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.

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
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 $3 \times 3$ 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 supports 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.

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

Fig. 1. UML class diagram with OCL constraints

This constraint, called an invariant, states a fact that should always be 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.

```ocl
context Player::nextMove():Square
  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
In functional verification, a proof is often trivial or straight-forward because a code function can be easily calculated and verified by using 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. 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 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

```java
// 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.

```java
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 needs 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[][]`

\[
f_1; f_2 \equiv \{ p := nextPlayer() ;[
  [squares, p := T->any(sqs) isGameOver(sqs) \&\&
  isSubState(squares, sqs), anything]
  \equiv [squares, p := T->any(sqs) isGameOver(sqs) \&\&
  isSubState(squares, sqs), anything]
  \sqsubseteq [squares := T->any(sqs) isGameOver(sqs) \&\&
  isSubState(squares, sqs), anything]
\]
\]

```java
isSubState(squares, sqs))
\sqsubseteq [squares := T->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
For example, we can define an algorithm for the operations using a combination of UML diagrams and OCL.

Sequence\{0..2\} denotes a sequence consisting of numbers at the given index; OCL uses 1-based index. The notation play at construct specifies the value of a derived attribute derive construct defines the result of a query operation, and the constructs, some of which are used in the example. The body pre and postconditions, OCL provides several other of class Square). In addition to class invariants, operation pre and postconditions, along with several new operations introduced. In OCL, we decided to represent the qualified association between class Square and the isMarkedBy 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.

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 getSquares(int i, int j) { ... }
}

public class Square {
    private Player player;
    public Square getPlayer() { 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
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 intended functions. In general, pre and postconditions are translated to CleanJava invariants [20], and pre and postconditions 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 intended functions. In general, 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 state machine that describes its algorithm (see Section IV).

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.

```java
public class TicTacToe {
    /*@ inv: [squares.length == 3 && squares->forall(Square sq| sq.isMarkedBy(player)) || squares->forall(Square sq| !sq.isMarked()) ] */
    private Square[][] squares;
    private Player[] players;

    /*@ {squares := Square[][]->any(Square[][] sqs| sqs->forall(Square sq| sqs.length == 3)) } */
    private void initSquares(){
        squares := Square[][]->any(Square[][] sqs| sqs->forall(Square sq| sqs.length == 3));
    }

    /*@ {squares := Square[][]->any(Square[][] sqs| isPristine(sqs)) } */
    private void initGame(){
        squares := Square[][]->any(Square[][] sqs| isPristine(sqs));
    }

    /*@ {isPristine(sqs) } */
    private boolean isPristine(Square[][] sqs){
        return !sqs->exists(Square sq| sq.isMarkedBy(player));
    }

    /*@ {isWinning(sqs) } */
    private boolean isWinning(Square[][] sqs){
        return !sqs->exists(Square sq| !sq.isMarked());
    }

    /*@ {isGameOver(sqs) } */
    private boolean isGameOver(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWonBy(player) } */
    private boolean isWonBy(Player player){
        return squares->exists(Square sq| sq.isMarkedBy(player));
    }

    /*@ {result := isWonBy(squares, player) } */
    public boolean play(Player player){
        return isWonBy(player);
    }

    /*@ {isPristine(sqs) } */
    public TicTacToe(){
        initSquares();
        initGame();
    }

    /*@ {isPristine(sqs) } */
    public boolean isPristine(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWinning(sqs) } */
    public boolean isWinning(Square[][] sqs){
        return !sqs->forall(Square sq| !sq.isMarked());
    }

    /*@ {isGameOver(sqs) } */
    public boolean isGameOver(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWonBy(player) } */
    public boolean isWonBy(Player player){
        return squares->exists(Square sq| sq.isMarkedBy(player));
    }

    /*@ {result := isWonBy(squares, player) } */
    public boolean play(Player player){
        return isWonBy(player);
    }

    /*@ {isPristine(sqs) } */
    public TicTacToe(){
        initSquares();
        initGame();
    }

    /*@ {isPristine(sqs) } */
    public boolean isPristine(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWinning(sqs) } */
    public boolean isWinning(Square[][] sqs){
        return !sqs->forall(Square sq| !sq.isMarked());
    }

    /*@ {isGameOver(sqs) } */
    public boolean isGameOver(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWonBy(player) } */
    public boolean isWonBy(Player player){
        return squares->exists(Square sq| sq.isMarkedBy(player));
    }

    /*@ {result := isWonBy(squares, player) } */
    public boolean play(Player player){
        return isWonBy(player);
    }

    /*@ {result := isPristine(sqs) } */
    public boolean isPristine(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWinning(sqs) } */
    public boolean isWinning(Square[][] sqs){
        return !sqs->forall(Square sq| !sq.isMarked());
    }

    /*@ {isGameOver(sqs) } */
    public boolean isGameOver(Square[][] sqs){
        return !sqs->forall(Square sq| sq.isMarked());
    }

    /*@ {isWonBy(player) } */
    public boolean isWonBy(Player player){
        return squares->exists(Square sq| sq.isMarkedBy(player));
    }

    /*@ {result := isWonBy(squares, player) } */
    public boolean play(Player player){
        return isWonBy(player);
    }

    /*@ {squares->forall(Square sq| !sq.isMarked()) } */
    public void play(){
        while (!isPristine(squares) && !isWinning(squares)){
            if (isPristine(squares)) {
                player p = players[1];
                while (isPristine(p) && hasEmptySquare(p)) {
                    p = p == players[0] ? players[1] : players[0];
                    Square sq = p.nextMove();
                    sq.setPlayer(p);
                }
            }
        }
    }
}
```

Fig. 4. Skeletal code with CleanJava annotations
inv: [squares->forAll(Square[] sqs) sqs->forAll(Square sq | !sq.isMarked) \&\& players->includes(sq.getPlayer())]


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).

\[
\text{nextMove} = \begin{cases} 
\text{null} & \text{if } \text{game.hasEmptySquare}() \\
\text{result} & \text{catch (Exception e) ( first = e; )} \\
\text{finally} \{ 
\text{if (result) result = E1; }
\text{if (result \&\& !first) null throw first; }
\}
\end{cases}
\]

However, it is also possible to translate these postconditions systematically and perhaps even automatically. One possibility is to use the \text{any} iteration operator that returns an arbitrary element of a collection that meets a given condition. Consider a postcondition \(P(x_1, x_2, \ldots, x_n)\), 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:

\[
[x_1, x_2, \ldots, x_n := T_1 -> \text{any} \{T_1 x_1\} \\
T_2 -> \text{any} \{T_2 x_2\} \\
\cdots \\
T_n -> \text{any} \{T_n x_2\} P'(x_1, x_2, \ldots, x_n))
\]

where \(P'\) is a CleanJava translation of \(P\). For example, the \text{pre} and postconditions of the above \text{nextMove} operation can be translated to the following intended function.

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

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

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

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

\[
\text{context Player:nextMove(): Square} \\
\text{pre: game.hasEmptySquare()} \\
\text{post: not result.isMarked and game.squares[*,*]->includes(result]}
\]
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 \texttt{any} iteration operator to translate it. This allows us to systematically and possibility automatically translate OCL constraints. However, the \texttt{any} operator is similar to the \(\mu\) 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 \texttt{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