Semantic Guidance and Feedback for the
Construction of Specifications and
Implementations

Paul C. Attie and Fadi A. Zaraket and Mohammad Fawaz and Mohamad Noureddine
American University of Beirut
Email: {pa07,fz11,mbf12,man17}@aub.edu.lb

Abstract

The problem of writing a specification which accurately reflects the intent of the developer has long been recognized as fundamental. We propose a method and a supporting tool to write and check a specification and an implementation using a set of use cases, i.e., input-output pairs that the developer supplies. These are instances of both good (correct) and bad (incorrect) behavior. We assume that the use cases are accurate, as it is easier to generate use cases than to write an accurate specification. We incrementally construct a specification (precondition and postcondition) based on semantic feedback generated from these use cases. We check the accuracy of the constructed specification using two proposed algorithms. The first algorithm checks the accuracy of the specification against an automatically generated specification from a supplied finite domain of use cases. The second checks the accuracy of the specification via reducing its domain to a finite yet equally satisfiable domain if possible. When the specification is mature, we start to also construct a program that satisfies the specification. However, our method makes provision for the continued modification of the specification, if needed. We illustrate our method with two examples; linear search and text justify.

I. INTRODUCTION

The derivation of programs from formal specifications, and the construction of a correctness proof hand-in-hand with the program has been advocated by Dijkstra [1], Hoare [2], Gries [3], and others. Central to this method is the a priori existence of a formal specification, that is assumed to be accurate, i.e., representing what the user requires. The task of constructing such a specification is addressed by the many requirements elicitation methods that have been presented in the literature [4], [5], [6], [7], [8], [9], [10], [11], [12], [13]. See Section VII below for a discussion of some of these.

a) Our contributions: We present a manual method for the construction of specifications for data-intensive, transformational, terminating sequential programs. Our method deals with sequential imperative programs, written in any standard sequential OO language, e.g., Java or C++. Currently, we support arithmetic and Boolean expressions, scalar and array data types, assignment, if, while, and procedure calls. It is a simple matter to extend our implementation to other constructs, see Section V. Specifications consist of a precondition, postcondition pair, written in first order logic. There is a single input, which is restricted by the precondition, and a single output, which is related to the input by the postcondition. Given a set of use cases, the method evaluates the specification and returns correction actions which the user executes to correct the specification.

We also present two algorithms for checking the resulting specification. The first checks the accuracy of the specification via reducing its domain to a finite domain which preserves satisfaction. The second
checks the accuracy of the specification against an automatically generated specification from a supplied finite domain of the input and output variables. In many cases, this will reveal deficiencies in the manually written specification.

We have implemented our method, available online, and all the examples given below were constructed using our implementation.

The rest of the paper is organized as follows. We present the method in Section II, and the linear search example in Section III. Section IV discusses additional implementation correction actions. We describe the supporting tool in Section V, provide the text justify example in Section VI, and discuss related work in Section VII. We conclude and discuss future work in Section VIII.

II. THE METHOD

A. Behaviors and specifications

The input of a program is given by a fixed set of input variables $\bar{x} = \langle x_1, \ldots, x_m \rangle$, e.g., actual parameters. Likewise, the output of a program is given by a fixed set of output variables $\bar{y} = \langle y_1, \ldots, y_n \rangle$, e.g., parameters that are reference types, or a single variable whose value is passed to a return statement. A variable can be both input and output, so the $\bar{x}$ and $\bar{y}$ are really meta-syntactic variables. Each $x_j$, $j = 1, \ldots, m$, takes values from a domain $I_j$, and each $y_j$, $j = 1, \ldots, m$, takes values from a domain $O_j$. Then $I = I_1 \times \cdots \times I_m$ is the domain of the inputs, and $O = O_1 \times \cdots \times O_n$ is the domain of the outputs.

A behavior is a pair $(i, o)$ where $i, o \in O$. $i = \langle iv_1, \ldots, iv_m \rangle$ is a tuple of the initial values of the input variables, and $o = \langle ov_1, \ldots, ov_n \rangle$ is a tuple of the final values of the output. We partition the set

1 http://webfea.fea.aub.edu.lb/fadi/dkwk/doku.php?id=speccheck
 Crucial to our method is the ability to evaluate a restriction on the relation between a partition, depending on how much of the input \( I \times O \) is included in \( S \), such that (1) \( f \) is atomic (i.e., a quantification or a relation symbol applied to terms) and free variable. Let \( f \) be a wff of first order logic. Then we write \( \text{vars}(f) \) for the set of free variables that occur in \( f \). We write \( f' < f \) iff \( f' \) is a subformula of \( f \). We write \( \text{at}(f) \) for the set of wff’s \( f' \) such that (1) \( f' \) is a subformula of \( f \), (2) \( f' \) is atomic (i.e., a quantification or a relation symbol applied to terms) and (3) there does not exist an atomic wff \( f'' \) such that \( f' < f'' < f \). In words, \( f' \) is a “maximal” atomic subformula of \( f \). We extend these notions to specifications: \( \text{vars}(S) \triangleq \text{vars}(P) \cup \text{vars}(Q) \), and \( \text{at}(S) = \text{at}(P) \cup \text{at}(Q) \). We assume, without loss of generality, that all the input and output variables occur free in \( S \), i.e., \( \bar{x} \cup \bar{y} \subseteq \text{vars}(S) \).

We will, as convenient, write the truth value of formula \( f \) within a valuation (model) \( V \) as \( V(f) \) or as \( f(V) \), i.e., sometimes we view valuations as mapping formulae to \( \{\text{true}, \text{false}\} \), and sometimes we view formulae as mapping valuations to \( \{\text{true}, \text{false}\} \). To reduce use of parenthesis, we sometimes use dot notation: \( V.f \), \( f.V \), respectively.

It is recognized that writing specifications is difficult because it involves formalizing an informal idea of behavior, given as a set of behaviors (use cases). Our aim in this paper is to bring formality and rigour to the task of writing specifications.

B. Definitions

**Definition 1.** A behavior \((i, o)\) satisfies a specification \( S = (P, Q) \) iff \( P(i) \Rightarrow Q(i, o) \). We write \((i, o) \models S\). We also write \([S] \triangleq \{(i, o) : (i, o) \models S\}\). A specification \( S \) is under-constrained iff there exists \((i, o) \in \text{bad}\) such that \((i, o) \models S\). A specification \( S \) is over-constrained iff there exists \((i, o) \in \text{good}\) such that \((i, o) \not\models S\). A specification is accurate iff it is not under-constrained and not over-constrained.

It is immediate that, for an accurate specification \( S \): \( \text{good} \subseteq [S] \) and \( \text{bad} \cap [S] = \emptyset \). There is no restriction on the relation between \([S]\) and \( \text{dontCare} \). Hence, given a partition of the behavior space \( I \times O \) into \( \text{good} \), \( \text{bad} \), and \( \text{dontCare} \), there are many possible specifications that correspond to this partition, depending on how much of the \( \text{dontCare} \) partition is included in \([S]\).

Crucial to our method is the ability to evaluate \( P(i) \) and \( Q(i, o) \) given \((i, o)\). This presents no problem, provided that all quantifications have finite (but unbounded, e.g., the size of an input array) range. We assume this restriction in the sequel.
Definition 2. Let $R, R'$ be predicates. $R'$ is stronger than $R$ iff $R' \Rightarrow R$ is valid. $R'$ is weaker than $R$ iff $R \Rightarrow R'$ is valid.

C. Correcting a specification using behaviors

We generate, by hand, several behaviors $(i, o)$, both good and bad. We evaluate $P(i)$ and $Q(i, o)$ for every pair $(i, o)$. The results generate correction actions, as given in Tables I and II. A correction action is a Boolean formula, where the propositions are basic correction actions, as follows. $R \downarrow v$ means strengthen predicate $R$ so that $R(v) = false$. $R \uparrow v$ means weaken $R$ so that $R(v) = true$. skip means leave $S$ unchanged.

Table I deals with the precondition $P$. The “Actual” column gives the outcomes from evaluating $P$, and the “Required” column gives the expected outcome. For good pairs, $P$ must hold and for bad pairs, $\neg P$ must hold. The “Correction Action” column gives the correction action that can be applied to $P$. The top half of the table shows the correction action for both good and bad behaviors, while the lower half shows the correction actions for the dontCares.

Table II shows similar information for the postcondition $Q$. The top half of the table deals with good behaviors while the lower half deals with bad behaviors. Note that we do not need correction actions in case of a dontCare behavior since by definition, the postcondition in a dontCare behavior can either be true or false.

Consider the first row in Table I. The precondition $P$ actually holds, which is also the required value. Therefore no correction action is needed and the user is instructed to skip, i.e. leave $P$ as it is. The second row shows the case where $P$ does not actually hold while it is required to do so. In this case, the correction action requires the user to weaken the precondition $P \uparrow i$ so that it holds over the input $i$. The rest of this table as well as Table II is similarly explained.

D. Correcting both a specification and an implementation using behaviors

We now add an implementation $S$. We assume that $S$ is sufficiently developed that it can generate outputs for any input. Define $g(i, o)$ iff $(i, o) \in$ good, and $S(i, o)$ iff $o$ is a result of execution $S$ with input $i$. Also define basic corrective action $S \downarrow (i, o)$: modify $S$ so that execution of $S$ with input $i$ does not produce output $o$.

Since $S$ generates output, we use inputs $i$ instead of behaviors $(i, o)$. Given input $i$ such that $P(i)$, we execute $S$ with input $i$, resulting in some output $o$. Table III gives the appropriate correction action for each case. The column labeled $g(i, o)$ gives the value of $g(i, o)$, which is input by an interactive query to the developer, who is the ultimate reference for good and bad. The column labeled $Q(i, o)$ gives the value obtained by evaluating $Q(i, o)$. For example, the third row has $\neg g(i, o)$ and $Q(i, o)$, i.e., $o$ is not a good output according to the developer, but it does satisfy the postcondition. The correction action $Q \downarrow (i, o) \land S \downarrow (i, o)$ thus requires that we change $S$ so that $o$ is not produced on input $i$, and we also change $Q$ so that $Q(i, o)$ is false, i.e., $Q$ reflects that $o$ is an incorrect output for input $i$.

III. THE LINEARSSEARCH EXAMPLE

We illustrate the method using a linear search example. Function linearSearch takes as input an array $a$, indices $l$ and $r$ that define left and right boundaries of the search, respectively, and element $e$. 
TABLE III
ADEQUACY OF \{P\}S\{Q\}\{B\} FOR P(i).

| g(i, o) | Q(i, o) | Correction Action |
|---------|---------|-------------------|
| true    | true   | skip              |
|         | false  | Q↑(i, o)          |
| false   | true   | Q↓(i, o) ∧ S↓(i, o) |
|         | false  | S↓(i, o)          |

linearSearch returns the index of e in a if e was found between l and r inclusive, and returns -1 otherwise. The following code listing shows the interface (or the empty definition) of linearSearch.

```
1 int linearSearch(int [] a, int l, int r, int e);
```

We start with the weakest specification, (false, true) (i.e., P = false and Q = true), which admits all behaviors. We repeatedly refine this specification by considering a single behavior, computing the actual values of P and then Q on the behavior, comparing with the expected value (depending on whether the behavior is good or bad), and then applying the corresponding correction action from Tables I and II for P and Q respectively.

**Pair 1** (good, required P ∧ Q):
input: a={1,2,3}, l=0, r=2, e=4
output: rv=-1
actual: ¬P ∧ Q
correction: (P↑) for P and skip for Q
User intuition dictates that Q(i, o) should hold, since the output rv=-1 correctly indicates that the value 4 is not present in a. From Table I, the correction action for P is to weaken it so that it holds for the given input. Array a and search element e can have arbitrary values, so we cannot constrain them in the precondition. We note that it makes sense for l to be ≤ r, since otherwise the search interval is empty. Thus we weaken P to l ≤ r. Q remains true since the correction action from Table II is to skip.

**Pair 2** (good, required P ∧ Q):
input: a={1,2,3,4,5}, l=0, r=4, e=2
output: rv=1
actual: P ∧ Q
correction: skip for both P and Q
No changes are made for this (i, o) pair as dictated by Tables I and II.

**Pair 3** (bad, required P ∧ ¬Q):
input: a={1,2,3}, l=0, r=2, e=4
output: rv=0
actual: P ∧ Q
correction: skip for P and Q↑(i, o) for Q
Now, we must strengthen Q so that Q(i, o) does not hold. Our informal description of the linear search problem states that when rv is not -1, then it must give a location of e in a. Hence we strengthen Q to:
$rv \neq -1 \Rightarrow a[rv] = e$. $P$ remains $l \leq r$.

**Pair 4** (bad, required $P \land \neg Q$):
- **input**: $a=\{5, 2, 7, 3, 6, 8\}$, $l=1$, $r=4$, $e=7$
- **output**: $rv=-1$
- **actual**: $P \land Q$
- **correction**: skip for $P$ and $Q(i, o)$ for $Q$

Now, we must strengthen $Q$ so that $Q(i, o)$ does not hold. Our informal description of the linear search problem states that when $rv$ is -1, then $e$ does not occur in $a$. Hence we strengthen $Q$ to:

$$(rv \neq -1 \Rightarrow a[rv] = e) \land (rv = -1 \Rightarrow (\forall k : 0 \leq k < a.size \Rightarrow e \neq a[k]))$$. $P$ remains $l \leq r$.

**Pair 5** (dontCare, required $\neg P$):
- **input**: $a=\{5, 2, 7, 3, 6, 8\}$, $l=4$, $r=1$, $e=7$
- **output**: $rv=-1$
- **actual**: $\neg P \land \neg Q$
- **correction**: skip for $P$, no action (i.e. skip) for $Q$

User intuition dictates that $P(i)$ should remain false, since the search interval is empty, which is dictated by Table I. No action should be taken on the postcondition $Q$.

**Pair 6** (good, required $P \land Q$):
- **input**: $a=\{5, 2, 7, 3, 6, 8\}$, $l=0$, $r=1$, $e=7$
- **output**: $rv=-1$
- **actual**: $P \land \neg Q$
- **correction**: skip for $P$ and $Q'(i, o)$ for $Q$

The user must now weaken the postcondition $Q$ so that $Q(i, o)$ holds. The conjunct $(rv \neq -1 \Rightarrow a[rv] = e)$ of $Q$ holds vacuously. The conjunct $(rv = -1 \Rightarrow (\forall k : 0 \leq k < a.size \Rightarrow e \neq a[k]))$. fails to hold since $e$ occurs in $a$, at position 2. However, the search interval is 0 to 1, and the occurrence of $e$ is outside of the interval. Hence, we should incorporate the search interval into $Q$. We might as well do this for both conjuncts now, even though this particular case does not require correction of the conjunct $(rv \neq -1 \Rightarrow a[rv] = e)$, other cases will. The revised $Q$ is $(rv \neq -1 \Rightarrow l \leq rv \leq r \land a[rv] = e) \land (rv = -1 \Rightarrow (\forall k : l \leq k \leq r \Rightarrow e \neq a[k]))$. $P$ is unchanged.

**Pair 7** (good, required $P \land Q$):
- **input**: $a=\{5, 2, 7, 3, 6, 8\}$, $l=-1$, $r=10$, $e=7$
- **output**: $rv=-1$
- **actual**: $P \land Q$ is undefined.

This brings up an issue that is not addressed by Tables I and II: what if $P$ or $Q$, as currently formulated, cannot be evaluated for some behavior? In this case, the problem is that the search interval $l, \ldots, r$ extends outside the index range of array $a$. For such cases, we define the appropriate correction to be that which results in $P$ and $Q$ being well-defined. Hence, we strengthen $P$ to $0 \leq l \leq r < a.size$, and we strengthen $Q$ to $0 \leq l \leq r < a.size \land (rv \neq -1 \Rightarrow l \leq rv \leq r \land a[rv] = e) \land (rv = -1 \Rightarrow (\forall k : l \leq k \leq r \Rightarrow e \neq a[k]))$.

Since the specification is mature, we add our implementation of LinearSearch annotated with the specification and the input-output pairs as shown in the following code listing.
We proceed to the \( \{ P \} S \{ Q \} \{ B \} \) checks of Table III. The input from Pair 2 invokes the correction \( S \downarrow(i,o) \) from the fourth row of Table III. Indeed our implementation computes the correct result but returns -1 instead in all cases. We correct the implementation and replace the `break;` with a `return i;` statement.

We now consider the use of `LinearSearch` as a priority queue where the key and the position of the elements have semantic significance and propose our expectation that `LinearSearch` returns the rightmost match of \( e \) if it exists using the following two input-output pairs.

```java
int LinearSearch(int [] a, int l, int r, int e) {
    int i = l;
    while ( i <= r ) {
        if (a[i] == e)
            break;
        i++;
    }
    return -1;
}
```

The program computes \( 2 \) as the index and the postcondition \( Q(i,o) \) evaluates to \( true \). However, \( \neg q(i,2) \) and Table III recommends the third row \( Q \downarrow(i,o) \land S \downarrow(i,o) \) requiring that we strengthen \( Q \) to refute \( 2 \) and strengthen the implementation to return \( 4 \). We strengthen the second conjunct of \( Q \) to \( (rv \neq -1 \Rightarrow (l \leq rv \leq r \land a[rv] = e) \land (\forall k. (k > rv \land k \leq r) \Rightarrow a[k] \neq e)) \). We also strengthen the implementation to search \( a \) from right to left as follows.

```java
int i = r;
while ( i >= l ) {
    if (a[i] == e)
        return i;
    i--;
}
return -1;
```

IV. THE OVER-IMPLEMENTATION CHECK

We use the negative inputs to check whether the precondition should be weakened, and the postcondition and the implementation should be strengthened.

In the following listing of `search` the precondition guarantees the passed array \( a \) to be sorted. This
TABLE IV
Adequacy of $\{P\}S\{Q\}\{B\}$ for $\neg P(i)$. 

| $g(i, o)$ | $Q(i, o)$ | Correction Action |
|-----------|-----------|-------------------|
| $true$    | $true$    | skip              |
| $false$   | $false$   | $Q\downarrow(i, o) \lor skip$ |
| $true$    | $false$   | $P\downarrow(i) \lor S\downarrow(i, o)$ |
| $false$   | $false$   | skip              |

gives a chance to write a binary search with a $O(\log n)$ running time instead of the $O(n)$ running time of linearSearch. However, our implementation is the same linearSearch implementation.

The bad behavior on Line 17 passes an array $a$ that is not sorted and thus fails Line 3 in the precondition. Also, the output passes the postcondition, however, the user deems the output as bad since he expects a binary search behavior. Table IV suggests the corrections for the inputs where $P(i)$ does not hold. In this case, the correction is either to weaken $P(i)$ to accept the unsorted array, or to modify the implementation. We choose the latter since we have not used the interesting precondition characteristics and over implemented sacrificing a logarithmic efficiency gain in this case.

V. THE SpecCheck TOOL

We built SpecCheck to evaluate our method. SpecCheck uses a simple ANTLR [14] based front end to parse implementations and specification triples ($\{P\}S\{Q\}\{B\}$) into a directed acyclic graph (DAG).
representation where nodes are programming elements, and edge weights describe operand relationships and statement order.

The algorithm of Figure 1 illustrates how SpecCheck computes the adequacy results for the properties. First, it picks a behavior $b \in B$, and assigns the corresponding input variables $i$ in the DAG to the input values $b_{input}$ of the behavior. Then it evaluates $P(i)$ with a recursive traversal of the DAG applying the semantics of every node in the DAG in the usual manner. SpecCheck applies the correction rules to $P$ in accordance with Table I, and waits for the user action to proceed. The user may modify $P$, $Q$, and $S$, and may also append new behaviors to $B$. SpecCheck recomputes the DAG after any intervention from the user. If the implementation $S$ is empty, SpecCheck assigns the corresponding output variables $o$ to the output values $b_{output}$ of the behavior, and evaluates $Q(i,o)$ similarly to $P(i)$ and the applies the $\{P\} - \{Q\} \{B\}$ rules from Table II and waits for the user action to proceed. Again, SpecCheck recomputes the DAG after any intervention from the user. If the implementation $S$ is not empty, SpecCheck computes $o$ by executing the implementation $S$, evaluates $Q(i,o)$, and then computes $g(i,o)$ by either querying the user, or checking whether the values of $o$ match those of an existing good behavior with the same $i$. Then SpecCheck applies the rules from Tables III and IV.

The precondition and postcondition accept a specification name and an associated predicate expression. No imperative constructs that modify variable values are allowed within the pre and post condition predicate expressions. SpecCheck supports a set of built-in predicates and computable properties of programming constructs such as scasize, scalpha, scnum, and scblank that compute the array size, whether a scalar represents an ASCII alphabetical character, an ASCII numerical character, and an ASCII space, respectively.

SpecCheck supports arithmetic and Boolean expressions, scalar and array data types, assignments, loops, and procedure calls including recursion. It is a simple matter to extend SpecCheck to support other constructs; currently we support them through syntactic sugar. For example, we represent an array of C++ objects of type ComplexNumber with two scalar data fields as two arrays of scalar numbers.
VI. THE JUSTIFY EXAMPLE

To evaluate SpecCheck, we used it to develop a more sophisticated program than linearSearch with its specifications from scratch. We considered the justify program that takes as input a single paragraph of left-justified English text, and transforms it into a fully justified (both left and right justified) version by modifying the whitespaces within the paragraph.

Informally, the fully justified paragraph must satisfy the following requirements.

1) The non-whitespace text is not changed.
2) The output has a uniform line length of $W$ characters, for all lines except possibly the last. This include spaces but not the newline character.
3) The number of spaces between words on the same line are as uniform as possible.
4) A line should not contain more “extra” blanks than the length of the first word of the following line.

We started from the above informal requirements and built a number of behaviors that intuitively satisfy and test against them. We quickly recognized that we need to define concepts such as blanks, words, lines, and paragraphs formally. We introduced a Boolean valued function for each concept and listed behaviors that satisfy and test against each of the concepts. Then we proceeded to build the pre and post conditions for each function.

We show below the code listing for the function sameWords that takes as input two strings $p_1$ and $p_2$ with two offsets $l_1$ and $l_2$, and claims that the two paragraphs starting at the offsets have the same non-whitespace contents. The two offsets serve well to define the function recursively. The full justify example is available online.

```java
1 boolean sameWords(int[] p1, int[] p2, int l1, int l2) {
  @pre swspec {
    subIsPara(p1, l1);
    subIsPara(p2, l2);
  }

  The swspec precondition guarantees that the text that starts at $l_1$ and $l_2$ in $p_1$ and $p_2$, respectively, is a paragraph. We define the subIsPara(int[] str, int l) function recursively with its specification and behaviors. The base case of subIsPara checks whether the str[l:str.size-1], the substring of str that starts at index l and ends at index str.size-1 inclusive, is a line, and the inductive case checks whether it is a line followed by a paragraph.

  To claim that $p_1$ and $p_2$ have the same contents, we invoke the function sameWords(p1,p2,0,0) as in the first good behavior below.
The `a` characters in the input strings designate the alphanumeric and punctuation characters, and the `N` character designates the newline character. We used the five good and two bad behaviors presented above in a fashion similar to `linearSearch` to come up with the following postcondition of `sameWords`.

The base case of `sameWords` claims that `p1` and `p2` are equal and consist of a word followed by a newline. It uses the function `subIsWord(int[] str, int l, int r)` that returns `true` when `a[l:r]` consists of alphanumeric characters only.

The inductive case partitions the substrings starting at `l1` and `l2` in `p1` and `p2`, respectively, into two ⟨hd, blank, tail⟩ tuples where the heads are equal, `p1[l1:w1] = p2[l2:w2]`, the blanks (`p1[w1+1:k1]` and `p2[w2+1:k2]`) may be of different lengths, and the tails have the same non-whitespace content. The function `headTailOfSub` returns `true` when its parameters define such a tuple.

The behaviors and the correction actions were instrumental to come up with the final version of the `sameWords` specification. In particular, the behaviors with varying blank spaces, helped fix mistakes with the indices. The behaviors with single words on a line helped us to refine our recursive definitions of a line and a paragraph (`subIsLine` and `subIsPara`) to use the single word as the base case rather than the empty string `ε`. This is good for two reasons:
Fig. 2. Snapshot of the Spec Check tool with the justify example.

- Allowing empty lines and paragraphs make some function definitions return more than one value, i.e., they would define a relation and not a function. This is awkward and requires repeatedly adding conjuncts such as \( w \neq \epsilon \) and \( \ell \neq \epsilon \) and results in longer, more verbose and prone to mistakes definitions. Behaviors directed us to abide by an important design principle since it is more concise to state the non-emptiness requirement in one place: the definitions of word and line.
- Empty lines and paragraphs do not correspond to our intuition of what a single paragraph looks like. We believe that the intuitiveness of the definition came as a result of the intuitiveness of the provided behaviors.

The use of substring indices in the function definitions helped eliminate some quantifiers from our predicates which in turns helps with the accuracy checks of the specification.

The implementation of the sameWords functions was exactly similar to the specification since it serves as a predicate function. This was not the case for the imperative functions such as int head(int []), and int tail(int []) that compute the indices of the head and tail of a given paragraph.

We briefly describe the rest of the justify specifications and implementation below. The precondition of the justify(int [] str, int W) function accepts single paragraphs or empty strings \( \text{str} = \epsilon \lor \text{subIsPara (str,0)} \). A behavior with a word on a line longer than \( W \) helped us modify the precondition to specify that all words must be of length \( \leq W \). This happened because we used \( W \) to be small in order to be able to practically interact with the tool. Another behavior with a word with length \( W - 2 \) followed by a word of length \( > 2 \) lead us to refine the precondition and only allow words with length \( < W/2 \).

In addition to the sameWords function, we used several other functions to define the postcondition of justify. For example, the minSep and maxSep functions compute the minimum and maximum inter-word separation of a line and the justLength Boolean function restricts their difference to be \( \leq 1 \). This expresses the space line uniformity.
The goodSep function recursively computes the total excess separation and the postcondition makes sure it is smaller than the length of the head of the next line. It uses numWords and totalSep to compute the number of words and the total number of blank characters on a line respectively. The function goodSep defines the excess separation as the difference as there must be at least \( n - 1 \) separators between \( n \) words on a line.

Figure 2 shows a snapshot of the SpecCheck graphical user interface (GUI) with results for the justify example. We provide the full implementation and the specification of justify along with SpecCheck online.

VII. RELATED WORK

Methods for checking specifications against behaviors have been presented in the literature [4], [5], [6], [7], [8]. These check a specification against behaviors by first writing the specification and then attempt to verify if the specification is accurate using a method such as animation, execution, model-checking, etc. In contrast, we go in the other direction: we write the specification from the behaviors, so that the specification is accurate by construction. In [15], a method for writing trace-based specifications is presented. The behaviors are sequences of atomic events. The technique is suitable for specifying reactive modules as “black boxes” that interact with an environment via events. Our method, in contrast, views programs as white boxes, and our specification are pre/postcondition pairs over the program state. Our method is intended for transformation programs that perform nontrivial computations on data, rather than reactive modules where control is the major issue. A method of writing temporal-logic based specifications using event traces (“scenarios”) is presented in [16]. This also applies to reactive systems and stresses control rather than data. in [9], a method for refining an initially simple specification using “elaborations” is presented. The elaborations are presented informally. A method of checking software cost reduction (SCR) specifications for consistency is presented in [10].

In none of the above works is there any analogue to our correction actions, and in particular to the construction of a correction action as a Boolean formula, which can then be carried out in several different ways (due to or and xor operators), depending on the developers intuition. The SPECIFIER [11] tool constructs formal specifications of data types and programs from informal descriptions, but uses schema, analogy, and difference-based reasoning, rather than our approach which is directly based on input-output behaviors. The Larch [12] approach enables the verification of claims about specifications, using the Larch prover, which improves the confidence in the specification’s accuracy. In [13], a method for testing preconditions, postconditions, and state invariants, using mutation analysis, is presented. It would be interesting to compare this method with our approach of checking a specification against a finite-domain version of the same specification.

VIII. CONCLUSION

We presented a method to manually construct a specification from use cases and two algorithms for checking the accuracy of the resulting check. We illustrated our method with two examples; linearSearch and justify. Future theoretical development consists of extending the class of formulae that can be checked formally by the reduction algorithm. Currently this class is expressive enough to specify properties such as sortedness, injectivity, and selectedness. Future applications consist of using the SpecCheck tool in undergraduate courses to introduce students to formalism and to help them write formal specifications correctly.
REFERENCES

[1] E. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice-Hall Inc., 1976.

[2] C. Hoare, “An axiomatic basis for computer programming,” *Commun. ACM*, vol. 12, no. 10, pp. 576–580, 583, 1969.

[3] D. Gries, *The Science of Programming*. New York: Springer Verlag, 1981.

[4] M. Heimdahl, M. Whalen, and J. Thompson, “Nimbus: a tool for specification centered development,” in *Requirements Engineering Conference, 2003. Proceedings. 11th IEEE International*, Sept. 2003, p. 349.

[5] D. Hazel, P. Strooper, and O. Traynor, “Requirements engineering and verification using specification animation,” in *Automated Software Engineering. Proceedings. 13th IEEE International Conference on*, oct 1998, pp. 302 –305.

[6] C. Heitmeyer, J. Kirby, J., B. Labaw, M. Archer, and R. Bharadwaj, “Using abstraction and model checking to detect safety violations in requirements specifications,” *Software Engineering, IEEE Transactions on*, vol. 24, no. 11, pp. 927 –948, nov 1998.

[7] E. M. Kim, S. Kusumoto, and T. Kikuno, “An approach to safety and correctness verification of software design specification,” in *Software Reliability Engineering*, oct 1995, pp. 78 –83.

[8] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese, “Model checking large software specifications.” *Software Engineering, IEEE Transactions on*, vol. 24, no. 7, pp. 498 –520, jul 1998.

[9] M. Feather, “Constructing specifications by combining parallel elaborations,” *Software Engineering, IEEE Transactions on*, vol. 15, no. 2, pp. 198 –208, feb 1989.

[10] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, “Automated consistency checking of requirements specifications,” *ACM Trans. Softw. Eng. Methodol.*, vol. 5, pp. 231–261, July 1996.

[11] K. Miriyala and M. Harandi, “Automatic derivation of formal software specifications from informal descriptions,” *Software Engineering, IEEE Transactions on*, vol. 17, no. 10, pp. 1126 –1142, oct 1991.

[12] S. Garland, J. Guttag, and J. Horning, “Debugging larch shared language specifications,” *Software Engineering, IEEE Transactions on*, vol. 16, no. 9, pp. 1044 –1057, sep 1990.

[13] T. Miller and P. Strooper, “A framework and tool support for the systematic testing of model-based specifications,” *ACM Transactions on Software Engineering Methodologies*, vol. 12, pp. 409–439, October 2003.

[14] T. J. Parr and R. W. Quong, “Antlr: A predicated-ll(k) parser generator,” *Software Practice and Experience*, vol. 25, pp. 789–810, 1994.

[15] S. Prowell and J. Poore, “Foundations of sequence-based software specification,” *Software Engineering, IEEE Transactions on*, vol. 29, no. 5, pp. 417 – 429, may 2003.

[16] A. van Lamsweerde and L. Willemet, “Inferring declarative requirements specifications from operational scenarios,” *Software Engineering, IEEE Transactions on*, vol. 24, no. 12, pp. 1089 –1114, dec 1998.