Verifying the Rope Data Structure of Xi-Editor

Anna Scholtz
University of British Columbia
ascholtz@cs.ubc.ca

ABSTRACT

This paper describes a formal specification and verification of the rope data structure used for storing text in xi-editor. We are using the language and verifier Dafny for verifying the functional correctness and defining the specification. In our use of formal methods, we focus on proving that the rope data structure still conforms to the defined properties after applying several common operations.

CCS CONCEPTS

• **D.2.4 [Software Engineering]: Software/Program Verification**;

KEYWORDS

Formal Verification, Dafny, Rope Data Structure, Xi-Editor

1 INTRODUCTION

Xi-Editor is a novel text editor with a strong focus on performance. While there already exists a wide variety of text editor nowadays, xi-editor is using modern software engineering techniques to provide a fast and reliable editor. One of the core concepts in xi-editor that allows performant text processing, is the usage of a rope data structure for storing text.

Since this data structure is such a crucial part of this editor, it would be beneficial to verify that after applying certain operations, such as insertions or deletions, the rope data structure is still valid and conforms to certain defined properties. This is not only relevant in cases when a single user is working on some text, but also in a collaborative environment in which multiple users might edit the text in parallel. While xi-editor is designed to support collaborative editing, currently it is not implemented.

In this project, we are using formal verification methods to ensure the correctness of the rope data structure in xi-editor. For this we are first defining the properties of the rope data structure that will be validated, and provide an implementation in Dafny of the rope data structure as well as several common operations to verify that after applying these operations the rope is still conforming to the defined properties. Since the rope data structure used in xi-editor is derived but more complex version of the standard rope data structure [2], we decided to first verify the standard rope data structure and then apply the same verfication techniques to the more complex rope data structure used in xi-editor.

The rest of this paper is organized as follows: we provide some background information about xi-editor and the rope data structure in Section 2. Section 3 provides a definition of the properties of the standard rope data structure as well as rope data structure used in xi-editor and details their implementation and verification in Dafny. We finish the paper with related work in Section 5, future work in Section 6 and a conclusion in Section 7.

2 BACKGROUND

In the following we will provide some background information about xi-editor and details about the rope data structure that is used in xi-editor to store text.

2.1 Xi-Editor

Xi-Editor [1] is a text editor, that is currently in its early development stages, with a backend written in Rust. It is designed to be high-performant, reliable, developer friendly and supports frontends implemented in any language. For storing and processing text, xi-editor is using a specially adapted rope data structure. The editor is also designed to work in a collaborative environment with multiple users editing the same text. However, this functionality has not been implemented, yet.

2.2 Rope Data Structure

Rope data structures [2] are used for storing text in a way that common text operations, such as insertions or deletions, can be performed in a more performant way. Essentially, a rope is a binary tree in which only leaf nodes contain data. Each node has a weight value associated. The weight value for leaf nodes is equal to the length of the text value stored. Weight values for internal nodes are the sum of the weights of the nodes of the left subtree. Figure 1 shows a simplified example of how text can be stored in a rope.

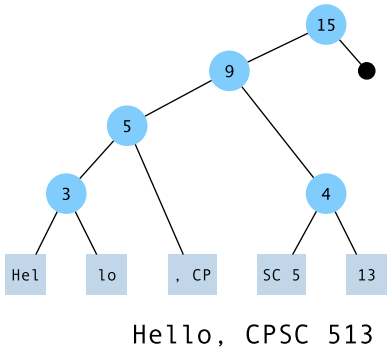

Figure 1: Example of text stored in a standard rope.

Xi-Editor uses a slightly modified rope data structure. Instead of having a structure similar to a binary tree, in xi-editor the rope data structure is based on a B-tree. A simplified example of what this data structure looks like is depicted in Figure 2.

Just like the standard rope data structure, the rope used in xi-editor only stores the text values in the leaf nodes. Internal nodes can have multiple children. The node weight, also referred to as length in xi, for leaf nodes is again the length of the stored text and for internal nodes the weight is, unlike in the standard rope,

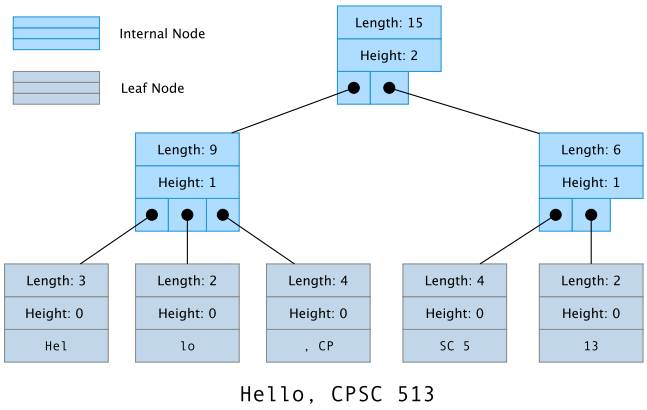

Figure 2: Text stored in a rope used in xi-editor.

the sum of the weights of all child nodes. Xi-Editor adds another attribute to the nodes which indicates the height of the node. This attribute is used for balancing the B-tree and making sure that leaves are at the same level.

2.3 Dafny

We are using the language and verifier Dafny [4] for verifying the functional correctness and defining the specification of the rope data structure. Dafny is an imperative, sequential language that supports verification through pre-conditions, post-conditions and loop invariants. Programs written in Dafny are first translated into the verification language Boogie 2 which is then used to generate first-order verification conditions that are passed to the SMT solver Z3.

Dafny allows to annotate implemented methods to ensure that certain properties hold. Supported annotations that are commonly used are post- and pre-conditions which can be added to the method's declaration, assertions which can be inserted within code and loop invariants. A full documentation of the dafny syntax and supported annotations is available at [3].

3 VERIFYING ROPES

This section describes the techniques applied to verify the standard rope data structure and the rope data structe used in xi-editor. We provide a specification, an overview of how the operations and data structure are validated and verification results for each implementation. For clarity, the specification will be stated as an English text and then transformed into code annotations.

We employed a Floyd-Hoare style approach to verify that the implementation of the rope data structure matches the specification. Since Xi-editor is implemented in Rust and not compatible with Dafny by default, we implemented a simplified version of the rope data structure and operations in Dafny. We manually added annotations, such as pre-conditions, post-conditions and loop invariants, which could be considered as lemmas, to ensure the correctness of the original implementation.

3.1 Standard Rope Verification

We start with verifying the standard rope data structure because its structure and properties are simpler and will serve as basis for verifying the more complex xi-editor rope.

3.1.1 Specification. The standard rope data structure used for storing text is based on a modified binary tree. It has the following properties:

(1) *Every node has at most two children.* It is also allowed that a node has only one child or no child at all.
(2) *Only leaves contain data.* The original text is split into chunks which are stored in the leaves.
(3) *Weight values of non-leaf nodes is the weights of all children in the left subtree.*
(4) *Weight values of leaf nodes is the length of the stored text.*

3.1.2 Verification. The rope data structure written in Dafny is shown in Listing 1. `Rope` is a tree which consists of two node types: `Leaf` and `InternalNode`. `Leaf` nodes contain text slices and `InternalNode` nodes are the internal nodes that can have up to two children. Each node has a specific weight that is stored in the `len` attribute.

```
1  datatype Node = Leaf(val: string) |
2      InternalNode(left: Rope?, right: Rope?)
3
4  class Rope {
5    ghost var Repr: set<object>
6
7    var len: int
8    var val: Node
9    // [...]
```

Listing 1: Standard rope data structure in Dafny

`Rope` has an extra attribute `Repr` which is not part of the actual implementation but only used for verification purposes and therefore denoted as `ghost` variable. Here, `Repr` is a set containing all the nodes that are stored in the rope.

The structure of the rope is defined in the `Valid()` predicate which is shown in Listing 2. In Dafny, predicates are functions that return a boolean value and that can be used as post-conditions and pre-conditions.

`Valid()` recursively verifies that internal nodes have at most two child nodes and validates each of these child nodes. `Repr` is used as a termination measure while recursively traversing the rope. Child nodes have a smaller `Repr` set than their parents, and the set consists of only one element for leaf nodes.

To verify that all nodes have correct weight values, we defined the predicate `ValidLen` which is depicted in Listing 3. It requires a valid structure of the rope and uses `Valid()` as a pre-condition to ensure this.

`ValidLen()` recursively traverses the rope and verifies that the weight of leaf nodes is equal to the length of the stored text, and of internal nodes is equal to the sum of the weights of the nodes in the left subtree. To sum up the node weights in a subtree, we defined

```
predicate Valid()
  reads this, Repr
{
  this in Repr &&
  (
    match this.val
    case Leaf(v) => true
    case InternalNode(left, right) =>
      (left != null ==>
        left in this.Repr &&
        this.Repr >= left.Repr &&
        this !in left.Repr &&
        left.Valid()
      ) &&
      (right != null ==>
        right in this.Repr &&
        this.Repr >= right.Repr &&
        this !in right.Repr &&
        right.Valid()
      )
  )
}
```

Listing 2: Predicate to validate the structure of the rope

```
predicate ValidLen()
  requires Valid()
  reads this, Repr
{
  match this.val
  case Leaf(v) =>
    this.len == |v|
  case InternalNode(left, right) =>
    (left != null ==>
      this.len == left.Len() &&
      left.ValidLen()
    ) &&
    (left == null ==> this.len == 0) &&
    (right != null ==> right.ValidLen())
}
```

Listing 3: Predicate to validate the weights of the nodes

a helper function `Len()` that traverses the subtree and sums up weight values of the nodes.

Both predicates are used as pre-conditions and post-conditions for implemented operations. We added additional conditions to these implemented methods to also verify that they are working correctly. Operations that are currently implemented are:

- `Report(): string` which returns the stored text,
- `Index(i: int) returns (charAtIndex: string)` which returns the character stored at index `i`,
- `Concat(rope: Rope) returns (concatenatedRope: Rope)` which concatenates the rope with another `rope` and returns the new resulting rope,
- `Split(i: int) returns (leftSplit: Rope?, rightSplit: Rope?)` which split the rope at index `i` and returns the two resulting ropes,
- `Insert(i: int, s: string) returns (newRope: Rope?)` which inserts text starting at index `i` and returns the updated rope,
- `Delete(i: int, j: int) returns (newRope: Rope?)` which removes text starting at index `i` and ending at index `j` and returns the updated rope.

Listing 4 shows the usage of the predicates as well as additional post-conditions for the `Insert()` method.

```
method Insert(i: int, s: string) returns (r: Rope?)
  requires Valid()
  requires ValidLen()
  ensures Valid()
  ensures ValidLen()
  ensures newRope != null ==> newRope.Valid()
  ensures newRope != null ==> newRope.ValidLen()
  ensures i < 0 || i >= this.Len()
    <==> newRope == null
{ [...] }
```

Listing 4: Definition of `Insert()` method

Running our implementation in Dafny 2.2.0.10923 results in no errors.

3.2 Xi-Editor Rope Verification

In the following we will provide a specification of the rope used in xi-editor that is based on the actual implementation. [todo: reference] Implemented operations will be simplified. We will focus solely on the text storage aspects and related operations, while in reality xi-editor has much more extensive functionality, for example to keep track of cursors, edit history or multiple views.

3.2.1 Specification. The rope data structure used in xi-editor is based on a B-tree and has the following properties:

(1) *Every node has at most MAX_CHILDREN children.* *MAX_CHILDREN* is constant that is by default set to 8.[1]
(2) *Every non-leaf node, except the root node, has at least MIN_CHILDREN child nodes. MIN_CHILDREN* is a constant this is by default set to 4.[2]
(3) *The root has at least two children, if it is not a leaf node.*
(4) *Only leaves contain data.* The original text is, again, split into chunks which are stored in the leaves.
(5) *The length of the text stored in leaf nodes is at most MAX_LEAF.* *MAX_LEAF* is a constant that is by default set to 1024.[3]
(6) *Weight values of non-leaf nodes is the sum of the children's weights.*
(7) *Weight values of leaf nodes is the length of the stored text.*
(8) *All leaves appear on the same level.*

[1] https://github.com/xi-editor/xi-editor/blob/master/rust/rope/src/tree.rs#L24 (3. December 2018)
[2] https://github.com/xi-editor/xi-editor/blob/master/rust/rope/src/tree.rs#L23 (3. December 2018)
[3] https://github.com/xi-editor/xi-editor/blob/master/rust/rope/src/rope.rs#L39 (3. December 2018)

3.2.2 Verification. The rope data structure as shown in Listing 5 has some similarity to the standard rope data structure. Like before, there are two different node types, `Leaf` and `InternalNode`, however `InternalNode` can have multiple children instead of just two. Furthermore, each node has an additional `height` attribute which indicates the level of the node. Leaf nodes, for example, have a height of 0 and their parents would have a height of 1. For verification purposes an additional ghost variable `Content` has been added.

```
datatype Node = Leaf(value: string) |
    InternalNode(children: seq<Rope>)

class Rope {
  ghost var Repr: set<Rope>
  ghost var Content: seq<string>

  var val: Node
  var len: int
  var height: int
  // [...]
```

Listing 5: Xi-editor rope data structure in Dafny

We defined two predicates `Valid()` and `ValidNonRoot()` to verify the correct structure of the rope. `Valid()` is almost identical to `ValidNonRoot()` which is shown in Listing 6, the only difference is that `Valid()` is used to verify the root node which has no *MIN_CHILDREN* threshold. Instead line 16 in Listing 6 is replaced by `|children| >= 2` in `Valid()`. When verifying the rope structure, the rope is traversed and it is ensured that the height attributes of the nodes are correct, that only leaves contain text content which must not exceed a certain threshold and that all children of internal nodes are valid.

`ValidLen()` is very similar to the predicate defined for the standard rope. It recursively traverses the rope and verifies that the weight of leaf nodes is equal to the length of the stored text, and of internal nodes is equal to the sum of all child nodes.

We implemented several of the operations used in xi-editor for modifying or processing the rope. Some of the operations used for the standard rope are not implemented in xi-editor. For example, there exists no `Delete` operation, instead to delete text the rope is split into three parts of which two parts are concatenated, leaving out the part that is to be removed. Furthermore, instead of having a `Report()` operation that returns the entire stored text, xi-editor has an operation that returns a slice of the text. Since only parts of the text are displayed to the user it is not necessary to transmit the entire text to the frontend which significantly improves performance, especially considering that after every edit the text needs to be updated.[4]

The operations that are implemented and verfied are:

- `SliceToString(i: int, j: int) returns (slice: string)` which returns a text slice,

[4]Xi-editor has additional more advanced mechanisms to handle edits in a performant way. For example, if there are single edits then only the deltas are sent and processed by the frontend to update the text. However, for reasons of simplicity, we did not consider deltas in our implementation.

```
predicate ValidNonRoot()
  reads this, Repr
  requires MAX_LEAF_LEN >= MIN_LEAF_LEN
  requires MIN_CHILDREN <= MAX_CHILDREN &&
           MIN_CHILDREN >= 2
{
  this in Repr &&
  (
    match this.val
    case Leaf(v) =>
      |v| <= MAX_LEAF_LEN &&
      Content == [v] &&
      height == 0
    case InternalNode(children) =>
      height >= 0 &&
      |children| >= MIN_CHILDREN &&
      |children| <= MAX_CHILDREN &&
      forall c: Rope :: c in children ==>
        c in Repr && this !in c.Repr &&
        c.Repr < Repr && c.ValidNonRoot() &&
        c.height == height - 1 &&
        |c.Content| <= |Content| &&
        forall co: string :: co in c.Content ==>
          co in this.Content
  )
}
```

Listing 6: Predicate to validate the structure of the rope in xi-editor

```
predicate ValidLen()
  requires Valid()
  reads this, Repr
{
  match this.val
  case Leaf(v) =>
    this.len == |v| &&
    ContentLen(this.Content) == |v| &&
    |Content| == 1
  case InternalNode(children) =>
    this.len >= 0 &&
    forall c: Rope :: c in children ==>
      c.len <= this.len &&
      c.ValidLen()
}
```

Listing 7: Predicate to validate the weights of the nodes

- todo

We used the previously defined predicates as pre-conditions and post-conditions for our implemented methods and added some additional conditions to ensure that the operations return correct results. All of the code is published on: https://github.com/scholtzan/cpsc-513-project.

Running our implementation in Dafny 2.2.0.10923 successfully verifies all conditions and results in no errors.

4 LESSONS LEARNED

This section describes the experiences and lessons we learned while verifying the rope data structure in xi-editor.

Since xi-editor is written in Rust and Dafny can by default only be integrated into programs written in C#, we had to reimplement the rope data structure and related operations in Dafny. On the one hand, this lead to a thorough review and more concise definition of specifications and could result in detecting bugs. Although no bugs were detected for xi-editor it definitely helped to understand and review existing code. On the other hand, since the code base in xi-editor for ropes is quite extensive, it was not possible to reimplement all functionality. Instead we had to simplify some aspects which could lead to missing important incorrect parts and still having undetected bugs.

While xi-editor has a lot of tests to ensure the correctness of the implementation, writing pre-conditions and post-conditions was surprisingly concise and expressive. Having something like Dafny for other programming languages, in this case for Rust, would make a very useful addition to common unit tests. Otherwise having two separate implementation of the same logic results in a lot of maintanance effort. It is quite likely that changes that are made in the xi-editor implementation will not be updated in the Dafny implementation since this would mean extra effort and maintainers having to familiarize themselves with Dafny on top of that.

At the beginning of this projects, we tried to use TLA+ and Pluscal for verifying ropes. However, specifying the system and data structure as axioms was quite complicated and took a lot of time. In comparison, defining these properties and modeling the rope data structure in Dafny was much faster and much more intuitive. While it is relatively easy to write code in Dafny, we noticed that due to the limited expressiveness the resulting code got quite long.

One major downside of using Dafny was its limited error handling. While it indicated which of the post- and pre-conditions failed, it did not show what the actual failure case was. If more extensive predicates are used as conditions, then only knowing which one failed still results in a lot of debugging effort to find out which part of the predicate is incorrect.

Often, some small code changes resulted in a timeout when running the verifier. These cases did not necessarily mean that code was incorrect. However, we often had to rewrite the implementation into something equivalent which the Dafny verifier could execute before a timeout occurred.

As mentioned before, Dafny lacks in powerfullness. For example, it only supports integer types `int` and `nat`, however a lot of the xi-editor implementation relies on unsigned integers. While it is possible to add additional checks to ensure the same behaviour, it resulted in a lot of additional and otherwise unnecessary conditions.

Currently, Dafny supports only verification of sequential code. One interesting aspect in xi-editor is that it is designed so that multiple users can edit text in parallel. While there have been some attempts to extend Dafny to verify parallel implementations [5] this is currently not possible.

Although there are some positives sides of verifying such a large project with Dafny, it is unlikely that it will be used for verification in production.

5 RELATED WORK

6 FUTURE WORK

7 CONCLUSION

The code is published on: https://github.com/scholtzan/cpsc-513-project

REFERENCES

[1] xi-editor. https://github.com/xi-editor/xi-editor.
[2] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
[3] R. L. Ford and K. R. M. Leino. Dafny reference manual. https://github.com/Microsoft/dafny/blob/master/Docs/DafnyRef/out/DafnyRef.pdf.
[4] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010.
[5] J. Mediero Iturrioz. Verification of concurrent programs in dafny. 2017.