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. The language and verifier Dafny is used for verifying the functional correctness and defining the specification. In the use of formal methods, the focus is 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 editors nowadays [1, 5, 6, 17], xi-editor is using modern software engineering techniques to provide a fast and reliable editor. One of the core concepts which 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, formal verification methods are used to ensure the correctness of the rope data structure in xi-editor. For this, the following contributions are made:

- A definition of the properties of the rope data structure that will be validated.
- An implementation 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 in Dafny [14].
- A summary of the good and bad parts of employing this verification approach as well as possible improvements and other insights.

Since the rope data structure used in xi-editor is derived but a more complex version of the standard rope data structure [8], the approach in this project was to first verify the standard rope data structure and then apply the same verification techniques to the more complex rope data structure used in xi-editor.

The rest of this paper is organized as follows: some background information about xi-editor, the rope data structure and Dafny is provided in Section 2. Section 3 presents 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. The paper finishes with a summary of lessons learned in Section 4, related work in Section 5, future work in Section 6 and a conclusion in Section 7.

2 BACKGROUND

In the following, some background information about xi-editor, details about the rope data structure that is used in xi-editor to store text and Dafny is provided.

2.1 Xi-Editor

Xi-Editor [7] 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 [8] 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.
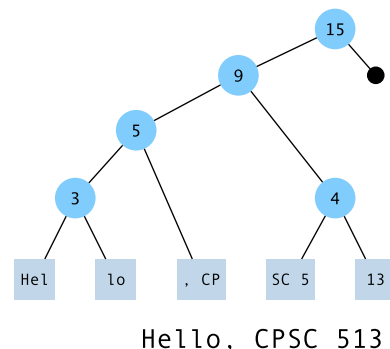

Figure 1: Example of text stored in a standard rope.

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.
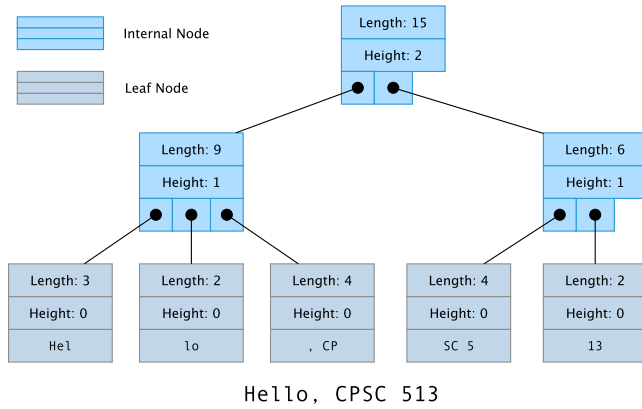

Hello, CPSC 513

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

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, 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

The language and verifier Dafny [14] is used 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 annotating 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 [11].

3 VERIFYING ROPES

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

A Floyd-Hoare style approach was employed 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, first a simplified version of the rope data structure and operations was implemented in Dafny and then extended to resemble xi-editors rope data structure. Annotations, such as pre-conditions, post-conditions, and loop invariants, were added manually which could be considered as lemmas, to ensure the correctness of the original implementation.

3.1 Standard Rope Verification

The first step taken was verifying the standard rope data structure because its structure and properties are simpler and will serve as a 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 are the weights of all children in the left subtree.*
(4) *Weight values of leaf nodes are 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, the predicate `ValidLen` has been defined which is depicted in Listing 3. It requires

```
1   predicate Valid()
2     reads this, Repr
3   {
4     this in Repr &&
5     (
6       match this.val
7       case Leaf(v) => true
8       case InternalNode(left, right) =>
9         (left != null ==>
10          left in this.Repr &&
11          this.Repr >= left.Repr &&
12          this !in left.Repr &&
13          left.Valid()
14        ) &&
15        (right != null ==>
16          right in this.Repr &&
17          this.Repr >= right.Repr &&
18          this !in right.Repr &&
19          right.Valid()
20        )
21    )
22  }
```

Listing 2: Predicate to validate the structure of the rope

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, a helper function `Len()` needed to be written that traverses the subtree and sums up the weight values of the nodes.

```
1   predicate ValidLen()
2     requires Valid()
3     reads this, Repr
4   {
5     match this.val
6     case Leaf(v) =>
7       this.len == |v|
8     case InternalNode(left, right) =>
9       (left != null ==>
10        this.len == left.Len() &&
11        left.ValidLen()
12      ) &&
13      (left == null ==> this.len == 0) &&
14      (right != null ==> right.ValidLen())
15  }
```

Listing 3: Predicate to validate the weights 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()` which returns the stored text,
- `Index(i: int)` which returns the character stored at index i,
- `Concat(rope: Rope)` which concatenates the rope with another rope and returns the new resulting rope,
- `Split(i: int)` which splits the rope at index i and returns the two resulting ropes,
- `Insert(i: int, s: string)` which inserts text starting at index i and returns the updated rope,
- `Delete(i: int, j: int))` 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.

```
1   method Insert(i: int, s: string) returns (r: Rope?)
2     requires Valid()
3     requires ValidLen()
4     ensures Valid()
5     ensures ValidLen()
6     ensures newRope != null ==> newRope.Valid()
7     ensures newRope != null ==> newRope.ValidLen()
8     ensures i < 0 || i >= this.Len()
9       <==> newRope == null
10  { [...] }
```

Listing 4: Definition of `Insert()` method

Running the implementation in Dafny 2.2.0.10923 results in no errors.

3.2 Xi-Editor Rope Verification

The following will provide a specification of the rope used in xi-editor that is based on the actual implementation. Implemented operations will be simplified. The focus will be 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. Currently, there is no official specification of the rope data structure. To derive a formal specification the existing Rust implementation and some existing documentation [2] was used. 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.

---

[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)

(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.*

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.

```
1  datatype Node = Leaf(value: string) |
2      InternalNode(children: seq<Rope>)
3
4  class Rope {
5    ghost var Repr: set<Rope>
6    ghost var Content: seq<string>
7
8    var val: Node
9    var len: int
10   var height: int
11   // [...]
```

Listing 5: Xi-editor rope data structure in Dafny

Two predicates `Valid()` and `ValidNonRoot()` were defined 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.

Several of the operations used in xi-editor for modifying or processing the rope are available in the implementation. 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

```
1  predicate ValidNonRoot()
2    reads this, Repr
3    requires MAX_LEAF_LEN >= MIN_LEAF_LEN
4    requires MIN_CHILDREN <= MAX_CHILDREN &&
5            MIN_CHILDREN >= 2
6  {
7    this in Repr &&
8    (
9      match this.val
10     case Leaf(v) =>
11       |v| <= MAX_LEAF_LEN &&
12       Content == [v] &&
13       height == 0
14     case InternalNode(children) =>
15       height >= 0 &&
16       |children| >= MIN_CHILDREN &&
17       |children| <= MAX_CHILDREN &&
18       forall c: Rope :: c in children ==>
19         c in Repr && this !in c.Repr &&
20         c.Repr < Repr && c.ValidNonRoot() &&
21         c.height == height - 1 &&
22         |c.Content| <= |Content| &&
23         forall co: string :: co in c.Content ==>
24           co in this.Content
25   )
26 }
```

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

```
1  predicate ValidLen()
2    requires Valid()
3    reads this, Repr
4  {
5    match this.val
6    case Leaf(v) =>
7      this.len == |v| &&
8      ContentLen(this.Content) == |v| &&
9      |Content| == 1
10   case InternalNode(children) =>
11     this.len >= 0 &&
12     forall c: Rope :: c in children ==>
13       c.len <= this.len &&
14       c.ValidLen()
15 }
```

Listing 7: Predicate to validate the weights of the nodes

significantly improves performance, especially considering that after every edit the text needs to be updated.[4]

The operations that are implemented and verified are:

---

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

---

[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, deltas were not further considered in this project.

- `SliceToString(i: int, j: int)` which returns a text slice,
- `Index(i: int)` which returns the character stored at index i,
- `Concat(rope: Rope)` which concatenates the rope with another `rope` and returns the new resulting rope,
- several helper methods to merge nodes or determine weights.

The previously defined predicates are used as pre-conditions and post-conditions for the 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 the implementation in Dafny 2.2.0.10923 successfully verifies all conditions and results in no errors.

4 LESSONS LEARNED

This section describes some experiences and lessons learned from 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#, it was necessary to reimplement the rope data structure and related operations in Dafny. On the one hand, this leads 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, some aspects needed to be simplified 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 implementations of the same logic results in a lot of maintenance 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, an attempt was made 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, often it was necessary to rewrite the implementation into something equivalent which the Dafny verifier could execute before a timeout occurred.

As mentioned before, Dafny lacks in powerfulness. 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 behavior, 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 [16] 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 xi-editor.

5 RELATED WORK

There exists a wide variety of text editors. Some of them, such as Vi [6] or Sam [17], rely on older techniques and store text in a buffer that is represented as an array of lines. Other text editors, such as Atom [1] are rich on features and functionality, however, show a lack in performance when processing large files [9]. GTK+ [3], which is a toolkit for creating graphical user interfaces, provides a framework for text editing which uses a rope data structure quite similar to the one used in xi-editor [4].

There does not seem to be any work related to using formal verification methods for verifying text editor implementations. In most cases, the implementation correctness is checked by providing a range of unit tests.

Dafny has been used to prove the correctness of distributed systems [12], verify operating system implementations [15] or secure systems [13].

6 FUTURE WORK

Currently, only a few aspects important for text storage in xi-editor have been verified. It would make sense to verify operations `Split()` and `Insert()` which have currently not been considered. Additionally, it might make sense to reduce some simplifications. For instance, currently, the `Concat()` operation requires the ropes to be the same height in order to simplify the re-balancing. This simplification was made for the scope of this project, but since the original implementation can concatenate arbitrary ropes, this should be extended in the future. Future work could focus on verifying additional parts of xi-editor, for example, it would be useful to also verify that moving through the edit history (eg. undo and redo) is working correctly. The current implementation also ignores edit deltas and cursors which could be part of future work.

Another aspect that could be investigated is collaborative editing. Especially situations in which text is edited concurrently by multiple users might result in bugs or inconsistencies. Since Dafny currently only supports verification of sequential programs, it might be necessary to employ other verification tools or languages, such as TLA+ [10].

It would be useful to have something like Dafny which integrates with other programming languages, for example, Rust. Then

developers would not need to reimplement their whole system in Dafny for verification.

7 CONCLUSION

This paper describes a verification approach for the standard rope data structure and the rope data structure used in xi-editor using Dafny. Besides providing specifications for data structures and the verification approach and implementation details were presented. Additionally, experiences of using Dafny for verifying a large, existing code base were summarized.

REFERENCES

[1] Atom. https://atom.io/.
[2] Crdt - the xi text engine. https://xi-editor.github.io/xi-editor/docs/crdt-details.html.
[3] The gtk+ project. https://www.gtk.org/.
[4] Gtktextbtree.c. https://github.com/GNOME/gtk/blob/master/gtk/gtktextbtree.c.
[5] Sublime text: A sophisticated text editor for code, markup and prose. https://www.sublimetext.com/.
[6] The traditional vi. http://ex-vi.sourceforge.net/.
[7] xi-editor. https://github.com/xi-editor/xi-editor.
[8] H.-J. Boehm, R. Atkinson, and M. Plass. Ropes: an alternative to strings. *Software: Practice and Experience*, 25(12):1315–1330, 1995.
[9] M. Brunsfeld. The state of atom's performance. https://blog.atom.io/2018/01/10/the-state-of-atoms-performance.html, 2018.
[10] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the tla+ proof system. In *International Joint Conference on Automated Reasoning*, pages 142–148. Springer, 2010.
[11] R. L. Ford and K. R. M. Leino. Dafny reference manual. https://github.com/Microsoft/dafny/blob/master/Docs/DafnyRef/out/DafnyRef.pdf.
[12] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17. ACM, 2015.
[13] C. Hawblitzel, J. Howell, J. R. Lorch, A. Narayan, B. Parno, D. Zhang, and B. Zill. Ironclad apps: End-to-end security via automated full-system verification. In *OSDI*, volume 14, pages 165–181, 2014.
[14] 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.
[15] H. Mai, E. Pek, H. Xue, S. T. King, and P. Madhusudan. Verifying security invariants in expressos. In *ACM SIGPLAN Notices*, volume 48, pages 293–304. ACM, 2013.
[16] J. Mediero Iturrioz. Verification of concurrent programs in dafny. 2017.
[17] R. Pike. The text editor sam. *Software: Practice and Experience*, 17(11):813–845, 1987.