Software Specification - Project 2

QS 2023/2024

Exercise 1: Static Analysis of C (6 val) In this exercise, you will analyse a thid-party C implementation of a `Hash table` data structure[1] provided in the `ex1` directory. You will use Infer and clang-analyzer to identify and fix bugs.

1. *Initial Analysis.* Run Infer and clang-analyzer on the Hash table program with additional analysis options. For Infer, use the `--pulse` option:

   ```
   infer run --pulse -- clang -c hashmap.c
   ```

   For clang-analyzer, use the additional checkers studied in the lab.

   Clearly identify the lines, types of bugs, and whether they are true positives or not in the `template.yml` file provided with the project script. Save this file as `<istid>_ex1.1.yml`.

2. *Manual Bug Identification.* Manually identify all bugs in the program. And, complete the bug report, saving it as `<istid>_ex1.2.yml`. Provide detailed descriptions of the bugs and specify whether the tools detected them or not. For example:

   ```
   report:
     - bug:
         type: Use-after-free
         lineno: 300
         class: TP (or FN if not found by anyone)
         description:
           Found with clang-analyzer. (or, Not found by any tool)
           In function `blah`, if the allocation of the variable `bleh`
           fails, the variable `blih` will be freed and later accessed.
   ```

3. *Bug Resolution.* Create a new version of the file, `hashmap-fixed.c`, fixing all the identified bugs while ensuring that neither Infer nor clang-analyzer produces any meaningful warnings. Add comments with `BUG-x` before each identified bug and `Fixes x` before the respective fixes in the code. Example:

   ```
   /* BUG-1 */
   void *ptr = malloc(length);
   /* Fixes 1*/
   if (!ptr) exit(1);
   ```

Hand-in Instructions: The solutions to Exercises 1.1 and 1.2 should be provided in separate files named `<istid>_ex1.1.yml` and `<istid>_ex1.2.yml`, respectively. The solution to Exercise 1.3 should be placed in a file named `hashmap-fixed.c`.

[1] `https://en.wikipedia.org/wiki/Hash_table`

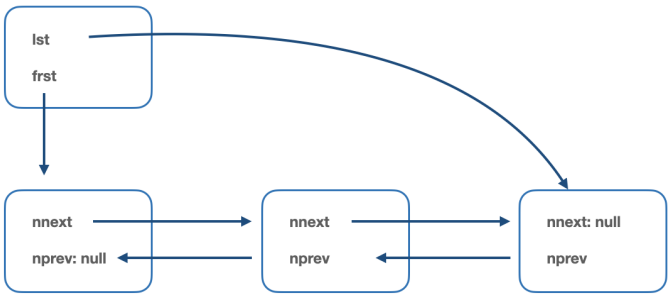

Figure 1: Doubly-Linked List

Exercise 2: Symbolic Testing TreeTables in C (6 val = 1 + 1 + 2 + 1.5 + 0.5) This exercise relies on third-party C implementation of a `TreeTable` data structure[2] provided together with the script of the project.

1. Implement a C function `int balanced(TreeTable* t)` that returns 1 if the tree table `t` is key balanced and 0 otherwise. In particular, this function should return 1 if for any given node the *height* of the left and right subtrees does not differ by more than 1.

2. Implement a C function `int sorted(TreeTable* t)` that returns 1 if the tree table `t` is key sorted and 0 otherwise. In particular, this function should return the value 1 when, for any given node, its key is greater than all the keys in the left subtree and smaller than all the keys in its right subtree.

3. Write a symbolic test suite for the given tree table implementation. The symbolic test suite should be *property-based*, meaning that each symbolic test should be aimed at checking a specific property of the given priority queue implementation (e.g. preservation of validity). Accordingly, each test should be annotated with a comment describing the property that it is meant to check. The created symbolic test suite should guarantee full code coverage of the functions: **(1)** `treetable_add()`; **(2)** `treetable_get()`; **(3)** `treetable_get_first_key()`; and **(4)** `treetable_get_greater_than()`.

4. Use Klee to run your symbolic test suite and create a concrete test suite using the concrete inputs generated by Klee.

5. Use the clang profiling tool to obtain a *line-oriented* code coverage report for the obtained concrete test suite.

Hand-in Instructions: The solutions to Exercises 2.1 and 2.2 should be written directly in the file `treetable.c` provided as part of the boilerplate of the project. The symbolic test suite produced as part of Exercise 2.3 should be placed inside a folder called `Ex2SymbTestSuite`, with each symbolic test being implemented in its own file. The concrete test suite produced as part of Exercise 2.4 should be placed inside a folder called `Ex3ConcTestSuite`; all concrete tests generated from the same symbolic test should be placed inside the same file, whose name should coincide with that of the corresponding symbolic test. Finally, the code coverage report obtained as part of Exercise 2.5 should be saved in a file called `Ex2CodeCoverage.txt`.

Exercise 3: Doubly-Linked Lists in Alloy (5 val = 1.5 + 0.5 + 2.5 + 0.5) Consider Figure 1 that illustrates a typical representation of a doubly-linked list. As expected, each node stores two pointers: one to the previous element in the list (`nprev`) and another to the next element (`nnext`). Additionally, for convenience, each doubly-linked list is associated with a head node that keeps a pointer to the first and last elements of the list.

[2] `https://github.com/GillianPlatform/collections-c-for-gillian/blob/master/lib/treetable.c`

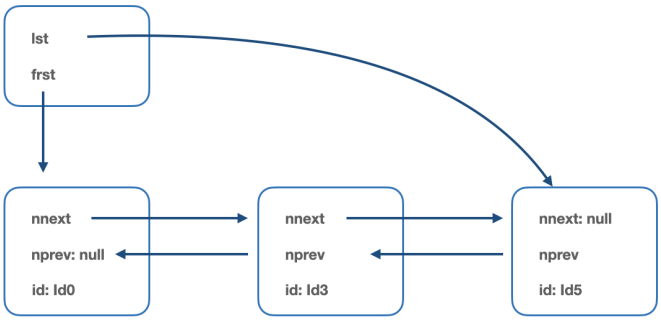

Figure 2: Ordered Doubly-Linked List

1. Define an Ellectrum Alloy model capturing the structure of doubly-linked lists as explained above and illustrated in Figure 1.

   Each *fact* of your model should be preceded by a comment describing the constraint that it is intended to model.

2. Use the Alloy `run` command to generate an instance of your model with at least two non-empty doubly-linked lists and five nodes.

3. Write state transformers for modelling the behaviour of the following operations:

   - `insert[n:Node,hn:HeadNode]` that inserts the node `n` into the doubly-linked list headed by the head node `hn`;
   - `remove[n:Node,hn:HeadNode]` that removes the node `n` from the doubly-linked list headed by the head node `hn`.

4. Use the Alloy `run` command to generate an Ellectrum Alloy trace for each transformer designed in 3.3. Each trace should be composed of at least two states.

Hand-in Instructions: The solution for Exercise 3 should be given in a file named `Ex3.als`. Within this file, please include the run commands used in Exercises 3.2 and 3.4. Additionally, you should provide two images displaying the generated models for Exercises 3.2 and 3.4, respectively, saved as `Ex3.2.png` and `Ex3.4.png`.

Exercise 4: Ordered Doubly-Linked Lists in Alloy (3 val = 0.75 + 0.25 + 1.5 + 0.5) The goal of this exercise is to model the behaviour of ordered doubly-linked lists. To achieve this goal, you should use the model created in Exercise 3 to define a new type of node, called an *ordered node*. Ordered nodes should include a additional field `id` of type `Id`. Note that the nodes in every ordered doubly-linked list are arranged in ascending id order, meaning that nodes with lower ids appear before nodes with larger ids. Refer to Figure 2 for an example of an ordered doubly-linked list.

1. Define an Ellectrum Alloy model capturing the structure of ordered doubly-linked lists as explained above and illustrated in Figure 2.

   Each *fact* of your model should be preceded by a comment describing the constraint that it is intended to model. Your model should make use of the model defined in Exercise 3 as well as the native Alloy `ordering` model.

2. Use the Alloy `run` command to generate an instance of your model with at least two non-empty ordered doubly-linked lists and five nodes.

3. Write state transformers for modelling the behaviour of the following operations:

- `insert[n:ONode,hn:OHeadNode]` that inserts the ordered node `n` into the ordered doubly-linked list headed by the head node `hn`;
- `remove[n:ONode,hn:OHeadNode]` that removes the ordered node `n` from the ordered doubly-linked list headed by the head node `hn`.

4. Use the Alloy `run` command to generate an Ellectrum Alloy trace for each transformer designed in 4.3. Each trace should be composed of at least two states.

Hand-in Instructions: The solution for Exercise 4 should be given in a file named `Ex4.als`. Within this file, please include the run commands used in Exercises 4.2 and 4.4. Additionally, you should provide two images displaying the generated models for Exercises 4.2 and 4.4, respectively, saved as `Ex4.2.png` and `Ex4.4.png`.

Instructions

Hand-in Instructions The project is due on the 27th of October, 2023. Be sure to create a `zip` file containing all the answer files and upload it in **Fenix**. Submissions will be closed at 23h59 on the 27th of October, 2023. Do not wait until the last few minutes for submitting the project.

Project Discussion After submission, you may be asked to present your work so as to streamline the assessment of the project as well as to detect potential fraud situations. During this discussion, you may be required to perform small changes to the submitted code.

Fraud Detection and Plagiarism The submission of the project assumes the commitment of honour that the project was solely executed by the members of the group that are referenced in the files/documents submitted for evaluation. Failure to stand up to this commitment, i.e., the appropriation of work done by other groups or someone else, either voluntarily or involuntarily, will have as consequence the immediate failure of this year's Software Specification course for all students involved (including those who facilitated the occurrence).