

KINDSPEC 2.2 *Quick Start Guide*

DSIC-VrAIn, Universitat Politècnica de València, Spain

June, 2020

Contents

1	What is KINDSPEC 2.2?	1
2	Starting KINDSPEC 2.2	2
3	KINDSPEC 2.2 interface	3
3.1	The top menu bar	3
3.2	The Console panel	5
3.3	The Program File panel	6
3.4	The Inference Parameters panel	6
4	A typical contract inference session	8
4.1	The example program	8
4.2	Setting the interface	8
4.3	Reading the contract	10
4.4	Axiom refinement	11
A	Appendix: KERNELC Syntax for KINDSPEC 2.2	17

1 What is KINDSPEC 2.2?

KINDSPEC 2.2 is an automated tool for inferring software contracts from programs that are written in a non-trivial fragment of C, called KERNELC, that supports pointer-based structures and heap manipulation. By relying on the standard distinction between *modifier* and *observer* program functions, KINDSPEC 2.2 generates a method contract for any routine by using other (observer) functions in the same program. The synthesized contract essentially consists of logical axioms that express pre- and post-condition assertions that define the precise input/output behavior of the C routine. The inferred axioms include default (general) rules and exceptions to these rules that specify exceptional or error behavior (e.g., undesirable use cases or execution side effects).

Roughly speaking, KINDSPEC 2.2 relies on a semantic definition of KERNELC in the \mathbb{K} framework that is used for symbolically executing the program functions and synthesizes program properties by interpreting the results. In order to avoid non-termination of loops and recursion, the semantic definition is enriched with a widening operator that is based on abstract interpretation. However, because of abstraction, some inferred axioms cannot be guaranteed to be correct; hence, they are kept apart as candidate (or overly general) axioms. KINDSPEC 2.2 can then perform a post-processing refinement that allows the user to trust some axioms (marking them as correct)

and to falsify those candidate axioms (thereby removing the falsified axioms from the axiom set). KINDSPEC 2.2 also filters out any redundant axiom from the contract.

This guide explains how to use the features of the KINDSPEC 2.2 tool, which are:

- Automatic inference of axiomatic contracts for KERNELC programs from source code
- Post-processing refinement, which consists of three phases: trusting, falsification, and removal of redundant axioms

2 Starting KINDSPEC 2.2

KINDSPEC 2.2 is available as a desktop application with a graphical interface. To access the KINDSPEC 2.2 tool, enter the website http://safe-tools.dsic.upv.es/kindspec2_2/ and click on the tab “Download”. Then, click on the link in “Download a desktop version of KINDSPEC 2.2”, as indicated in Figure 1.

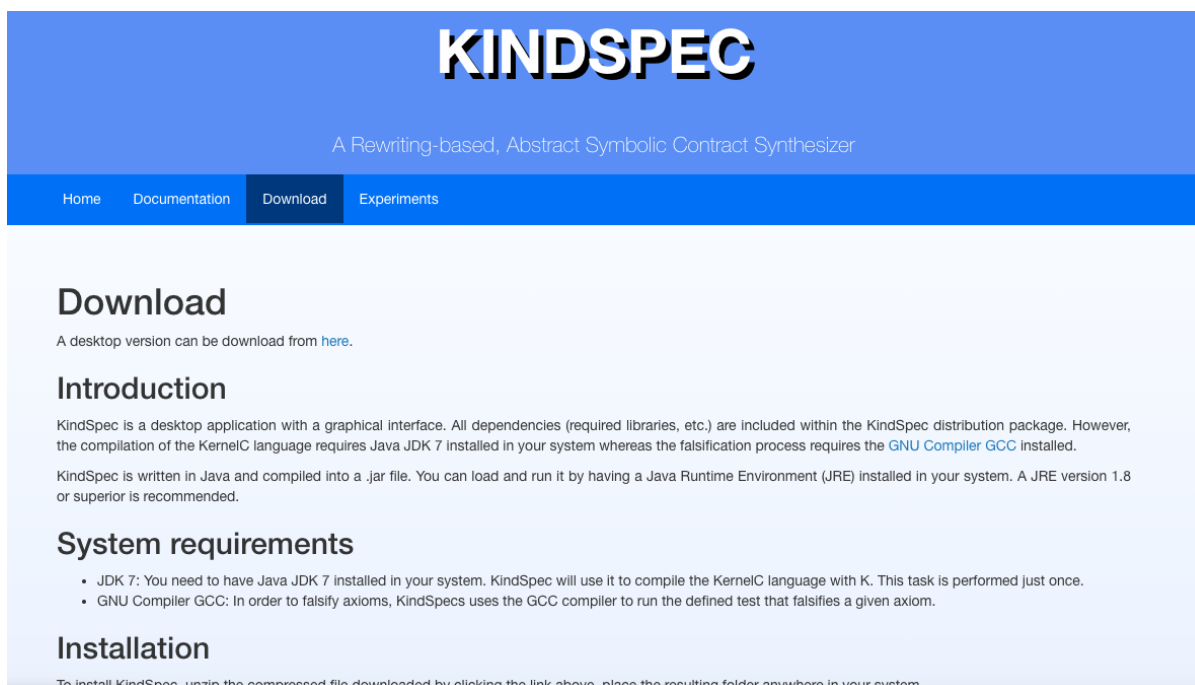


Figure 1: KINDSPEC 2.2 download webpage.

The distribution package of KINDSPEC 2.2 is fully standalone, which means that all required dependencies are comprised except for the following two ones: 1) the Java Runtime Environment (JRE)¹, which is required to run KINDSPEC 2.2; and 2) the GNU Compiler GCC², which is used for axiom falsification and must be installed in your system (with the `gcc` command being usable from the command line.)

To execute KINDSPEC 2.2, unzip the compressed distribution file, place the resulting folder anywhere in your system, and then run the executable file `KindSpec.jar`.

¹The latest JRE version is available at <https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html>. Please note that KINDSPEC 2.2 is not compatible with some versions of Java, specifically with Java 10.

²Available at <https://gcc.gnu.org/install/>

The first time you run the program you need run a last step to finish the installation. That last step allows the dependencies included in the package to be automatically installed (see Figure 2). Please, follow carefully the installation instructions on that page. You need to provide the path to JDK7 on your system and then confirm that you want to install the dependencies (see Figure 2). The installation process may take several minutes since it compiles the `KERNELC` language semantics that is encapsulated in a `ℳ` framework script. When the setup is completed, another dialog confirms that the installation was successful. The main interface of KINDSPEC 2.2 will finally launch after clicking on “OK” or closing the dialog.

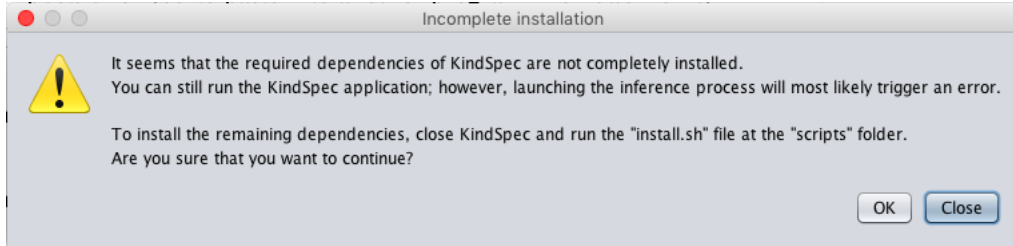


Figure 2: Setup dialog.

3 KINDSPEC 2.2 interface

When KINDSPEC 2.2 is opened, the main window of the application interface appears. As illustrated in Figure 3, the interface is composed of five, easily identifiable parts: 1) the top menu bar, with classical **File** and **Help** buttons; 2) the **Console** panel, in the left-hand side section of the window, which organizes all the information that is generated during the contract inference process; 3) the **Program File** panel, in the top part of the right-hand side section; and 4) the **Inference Parameters** panel, below the Program file panel on the right-hand side section. In the following, we explain in detail all of the features.

3.1 The top menu bar

The menu bar at the top of the interface window provides the **File** and **Help** (sub-)menus. The **File** menu allows the user to manage the inferred contracts and to export any information that is generated during the inference session. The **File** options are illustrated in Figure 4. We enumerate them as follows:

- **Reset session:** Cleans up and resets the application to the initial (default) state.
- **Export contract:** Stores the current contract into a serialized file.
- **Load contract:** Restores a previously saved contract without running the inference process again.
- **Save console:** Stores the contents of the **Console** area in plain text files. The specific tab to be saved in a file can be selected through a dialog with radio buttons, as shown in Figure 5. Only one tab can be saved at a time (but the menu option can be accessed again).
- **Close:** Closes up the KINDSPEC 2.2 interface.

The **Help** submenu contains a **Contact** option; clicking on it will show a dialog with contact addresses to report any issue found during the execution of KINDSPEC 2.2.

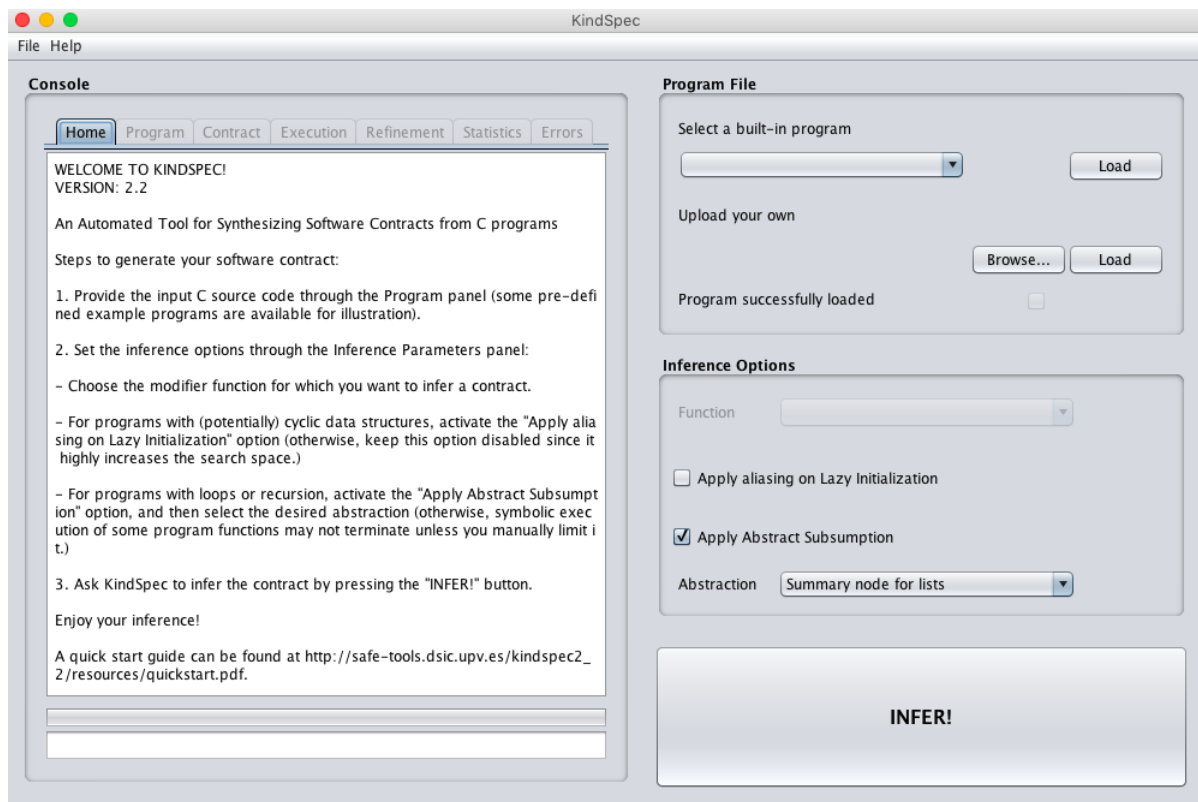


Figure 3: Main window of the KINDSPEC 2.2 interface.

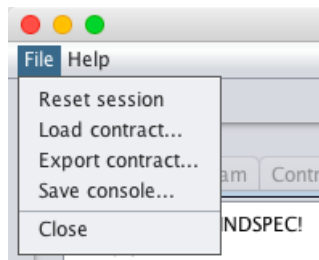


Figure 4: File submenu in the top menu bar.

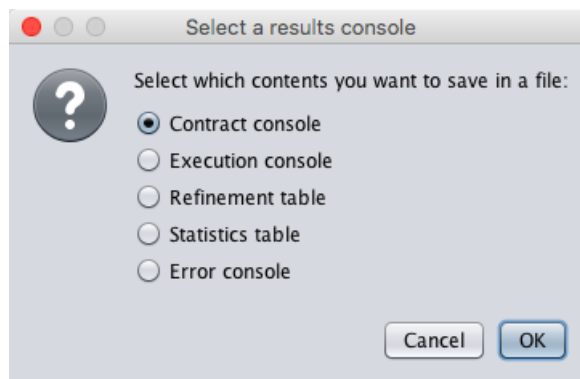


Figure 5: Save console dialog.

3.2 The Console panel

This area provides access to the different outcomes of the contract inference process. The different contents can be accessed through several tabs. All of the tabs (except for the Home tab) are initially deactivated and become enabled only when some output is made accessible through them. The purpose of each tab is explained below:

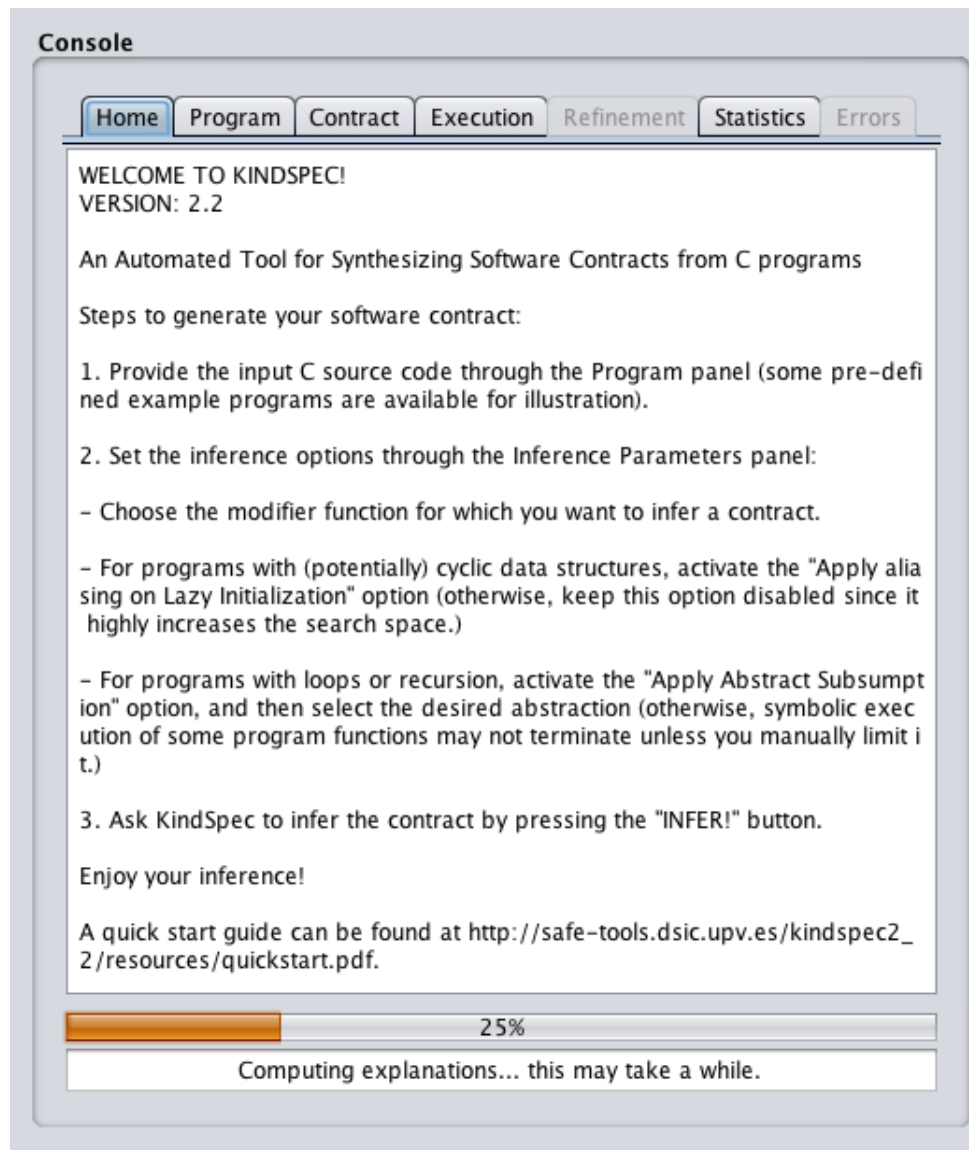


Figure 6: Console panel with progress bar and control text area.

1. The Home tab briefly provides some instructions for using the interface in a standard session. It is always enabled and it never gets modified.
2. The Program tab shows the source code of the input program that is currently loaded. Note that the program source code is not modifiable from this area.
3. The Contract tab outputs the resulting contract at the end of the whole inference process.

4. The **Execution** tab shows all of the intermediate results; for instance, the final states of the symbolic execution tree or the raw, initially extracted axioms and candidate axioms (before any refinement). The contents of this tab are updated several times during the inference process.
5. The **Refinement** tab lists the inferred candidate axioms in a table. The user can select any candidate axiom from the table either for **Trusting** it or for attempting its **Falsification** through dedicated buttons. The whole refinement process is described in Section 4.
6. The **Statistics** tab summarizes some practical information of the inference session; for example, the number of symbolic execution paths deployed, the number of different axioms distilled from these paths, and the elapsed inference time.
7. The **Errors** tab is never activated in a successful inference session. It is only enabled whenever an exception or error arises during KINDSPEC 2.2's execution, reporting them in a log that can be accessed through this tab. When this tab is activated, the execution of KINDSPEC 2.2 generally stops.

Finally, the lower part of this panel also contains a progress bar and a text area that are both shown in Figure 6. The progress bar is active when the inference process (or the refinement process) starts and provides an estimation of the percentage of work left to complete the task. The text area outputs control information about the current task.

3.3 The Program File panel

This area allows a KERNELC program³ to be loaded, which can be done in two ways, as illustrated in Figure 7. First, a dropdown menu is available at the top of this panel, which allows the user to choose among a set of predefined benchmark programs and load the selected program by clicking the **Load** button at the right-hand side of the dropdown menu. Second, the **Browse** button can be used to search for (and select) a C program file on the user's computer. Once selected, the path to the file will appear next to the **Upload your own** label shown in Figure 7. The program can then be loaded by pressing the adjacent **Load** button. In the case that a valid program is provided, the **Program** tab in the **Console** area becomes active as well as the **Function** selector in the **Inference Parameters** panel, which is described in the next section.

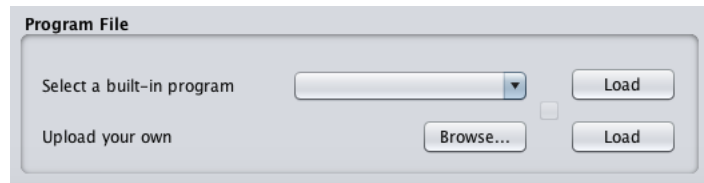


Figure 7: Program File panel.

3.4 The Inference Parameters panel

This panel can be used to set the key inference parameters. First, the **Function** dropdown menu in Figure 8 allows the function for which a contract must be inferred to be selected from the functions found in the input program. Note that a program must have been loaded first in order for this selector to be enabled. Also, two checkboxes are provided for setting two specific parameters:

³The KERNELC syntax accepted by KINDSPEC 2.2 is summarized in Appendix A.

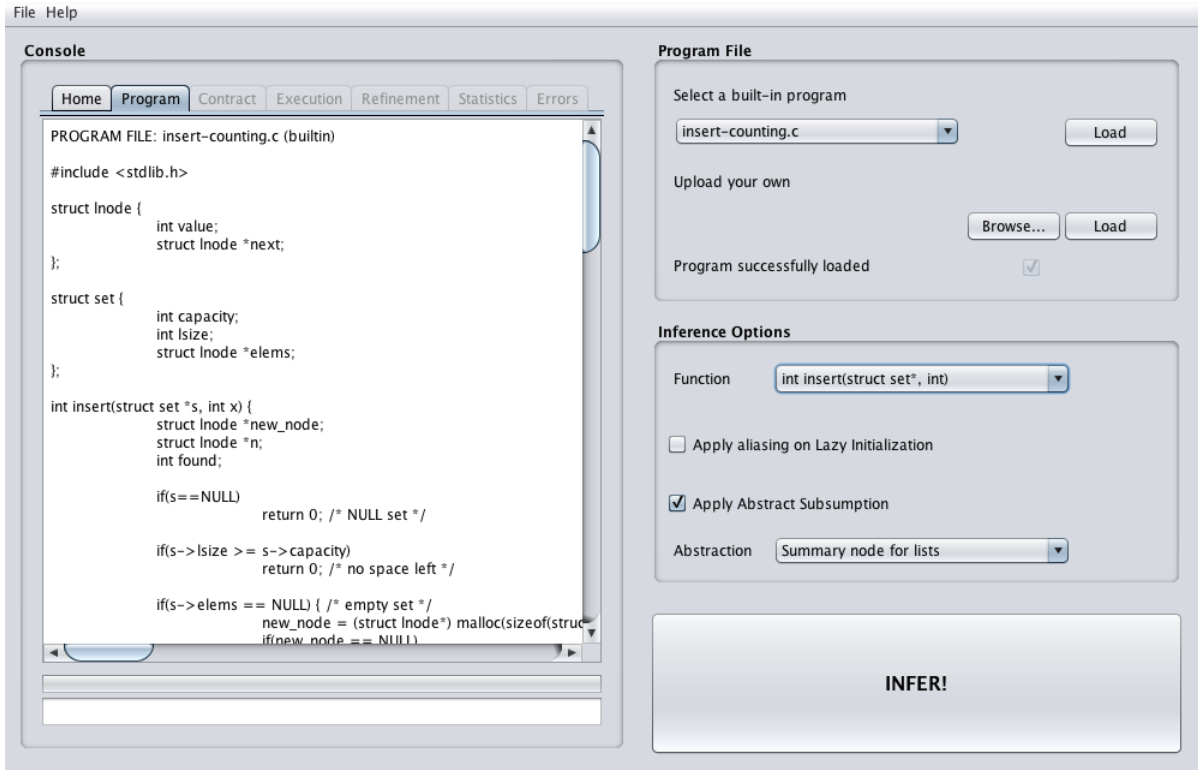


Figure 8: Interface configuration for the `insert(s,x)` example.

Apply aliasing on Lazy Initialization As mentioned above, KINDSPEC 2.2 relies on symbolic execution, which consists of running functions with symbolic values as input values instead of using specific data values as in standard execution. These symbols represent unbounded values of their respective types. *Symbolic values* are then constrained as a result of exploring the symbolic execution paths. Specifically, the accumulated constraint along a given path represents the properties that concrete input data needs to satisfy in order to exercise the considered path. In order to support potentially dynamic data objects (`struct` types in C), standard symbolic execution is enriched with the *lazy initialization* technique. Roughly speaking, when an instruction performs a first access to a symbolic object reference field, the symbolic execution forks the current state into three different heap configurations, in which the field is respectively initialized to: 1) null; 2) a reference to a new object with all symbolic attributes; and 3) a previously introduced concrete object of the desired type. With these initialized cases, conditions on these objects can be added to the accumulated constraint⁴.

Lazy initialization avoids requiring any a priori bound size for symbolic input structures. However, note that in case 3, lazy initialization generates a new path for every object of the same type that already exists in the heap. This linking is called *aliasing* and can cause state blow-up after some iterations. Fortunately, aliasing is not necessary if the program does not deal with cyclic structures (such as circular lists). For this reason, aliasing is disabled by default and can be enabled on demand by selecting the **Apply aliasing on Lazy Initialization** option. For efficiency reasons, we recommend keeping this option unselected unless the input program operates with cyclic structures.

⁴Further information about lazy initialization can be found in: S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. In: *Proc. TACAS '03*. ACM, pp. 553-568, 2003.

Apply Abstract Subsumption Symbolic execution of code that contains loops or recursion may result in an infinite number of paths whenever the termination condition depends on symbolic data. A classical solution is to establish a bound to the depth of the symbolic execution tree by specifying the maximum number of unfoldings for each loop and recursive function. However, KINDSPEC 2.2 implements a more accurate technique that is based on abstract subsumption⁵ and determines the length of the symbolic execution paths in a dynamic way by using abstraction on the memory structures. The use of such abstract subsumption to terminate the symbolic execution of the program can be enabled/disabled through the **Apply Abstract Subsumption** checkbox. KINDSPEC 2.2 also provides predefined abstractions for several data structures that can be selected in the **Abstraction** combo box. The classical **Summary** node for lists is selected by default.

4 A typical contract inference session

Let us illustrate a classical inference session with KINDSPEC 2.2.

4.1 The example program

The example program that is used in this walkthrough is shown in Figure 9 and consists of the KERNELC implementation of an abstract data type for representing sets by using linked lists.

Roughly speaking, we define set operations over a data structure (**struct set**) that records the number of elements contained in the set (field **lsize**), the maximum number of elements that can be held (field **capacity**), and a pointer to a list that stores the set elements (field **elems**). Each node of the list is a record data structure (**struct lnode**) that contains an integer value (field **value**) and a pointer to the subsequent list element (field **next**).

The program is composed of six methods: a modifier function (**insert**), and five observer functions (**isnull**, **isempty**, **isfull**, **contains**, and **length**). A call **insert(s,x)** to the insert function proceeds as follows. It first checks that the pointer **s** to the set structure is different from **NULL**, that the set is not full, and that **x** is not in the set yet. Then, a new list node ***new_node** is allocated, filled with the value **x**, and inserted as the first element of the list; also, the size of the set is increased by 1 and the call returns 1. Otherwise, 0 is returned and **s** is not modified. The following observers return 0 (which in C stands for the boolean value *false*) unless explicitly stated otherwise. The observer function **isnull(s)** returns 1 (which stands for the boolean value *true*) only if the pointer **s** references to **NULL** memory; **isempty(s)** returns 1 if **s** points to an empty list (i.e., **s->elems** is **NULL**); **isfull(s)** returns 1 if **s** is not null and not empty and if the size of **s** is greater than or equal to its capacity; and **contains(s,x)** returns 1 if the value **x** is found in **s**. Finally, the function **length(s)** counts up the number of elements in the set.

The user can ask KINDSPEC 2.2 to infer a contract for the **insert(s,x)** function by using the **isnull**, **isempty**, **isfull**, **contains**, and **length** functions to interpret the symbolic states before and after the function execution.

4.2 Setting the interface

After the KINDSPEC 2.2 Java application has been launched as indicated in Section 2, we can proceed to load the example program in KINDSPEC 2.2, configure the interface, and start the inference process for the **insert(s,x)** function as follows.

⁵For more information on abstract subsumption, we direct the reader to:
S. Anand, C. S. Păsăreanu, and W. Visser. Symbolic execution with abstraction. *STTT*, 11(1):53–67, 2009.


```

1  #include <stdlib.h>
2
3  struct lnode {
4      int value;
5      struct lnode *next;
6  };
7
8  struct set {
9      int capacity;
10     int lsize;
11     struct lnode *elems;
12 };
13
14 int insert(struct set *s, int x) {
15     struct lnode *new_node;
16     struct lnode *n;
17     int found;
18
19     if(s==NULL)
20         return 0; /* NULL set */
21
22     if(s->lsize >= s->capacity)
23         return 0; /* no space left */
24
25     if(s->elems == NULL) { /* empty set */
26         new_node = (struct lnode*) malloc(sizeof(struct
27             lnode));
28         if(new_node == NULL)
29             return 0; /* no memory left */
30         new_node->value = x;
31         new_node->next = NULL;
32
33         s->elems = new_node;
34         s->lsize = 1;
35
36         return 1;
37     }
38     n = s->elems;
39     found = 0;
40     while(n != NULL) {
41         if(n->value == x) {
42             found = 1;
43         }
44         n = n->next;
45     }
46
47     if(found) {
48         return 0; /* element already in the set */
49     }
50
51     /* Creation of new node */
52     new_node = (struct lnode*) malloc(sizeof(struct
53         lnode));
54     if(new_node == NULL)
55         return 0; /* no memory left */
56     new_node->value = x;
57     new_node->next = NULL;
58
59     n = s->elems;
60     s->elems = new_node;
61     s->lsize = s->lsize + 1;
62
63     return 1; /* element added */
64 }
65
66 int isNull(struct set *s) {
67     if(s==NULL)
68         return 1;
69     return 0;
70 }
71
72 int isEmpty(struct set *s) {
73     if(s==NULL)
74         return 0;
75     if(s->elems==NULL)
76         return 1; /* s is empty */
77     return 0;
78 }
79
80 int isFull(struct set *s) {
81     if(s==NULL)
82         return 0;
83     if(s->lsize >= s->capacity)
84         return 1; /* s is full */
85     return 0;
86 }
87
88 int contains(struct set *s, int x) {
89     struct lnode *n;
90
91     if(s==NULL)
92         return 0; /* s is NULL */
93
94     n = s->elems;
95     while(n != NULL){
96         if(n->value == x)
97             return 1; /* element found */
98         n = n->next;
99     }
100
101     return 0; /* element NOT found */
102 }
103
104 int length(struct set *s) {
105     struct lnode *n;
106     int count;
107
108     if(s==NULL)
109         return 0; /* s is NULL */
110
111     count = 0;
112     n = s->elems;
113     while(n != NULL){
114         count = count + 1;
115         n = n->next;
116     }
117
118     return count;
119 }

```

Figure 9: KERNELC implementation of a set data type through linked lists.

The example program is available as a built-in benchmark program, thus it can be easily selected by opening the drop-down combo box in the **Program File** panel and clicking on `insert-counting.c`. Then, load it into KINDSPEC 2.2 by pressing the **Load** button at the right-hand side of the drop-down menu. The focus in the **Console** panel will immediately change to the **Program** tab, where the code of the program is displayed.

Now let us configure the inference parameters in the Inference Parameters panel as shown in Figure 8. Open the Function selector and scroll down to the profile of the `insert(s,x)` function, which is `int insert(struct set*, int)`. Since the program deals with acyclic (singly-linked) lists, we leave the Apply aliasing on Lazy Initialization option deactivated. We also keep the Apply Abstract Subsumption option with the default Summary node abstraction selected in the Abstraction selector.

In order to run the inference process, click on the **INFER!** button in the lower right area of the window. The Contract, Execution, and Statistics tabs in the Console area become accessible. Also, the progress bar and the control text area below the bar are now active. All of the remaining elements of the interface will be disabled while the inference process is being carried out. Since the example program contains many observer functions, symbolic execution is costly and the application generally takes a bit of time. In this case, the control text area will show the message “... this may take a while”. Meanwhile, the enabled tabs in the Console area can be freely navigated.

Once the inference process is finished, the focus in the Console panel will automatically change to the Contract tab that gets populated with the resulting contract, as illustrated in Figure 10. Since

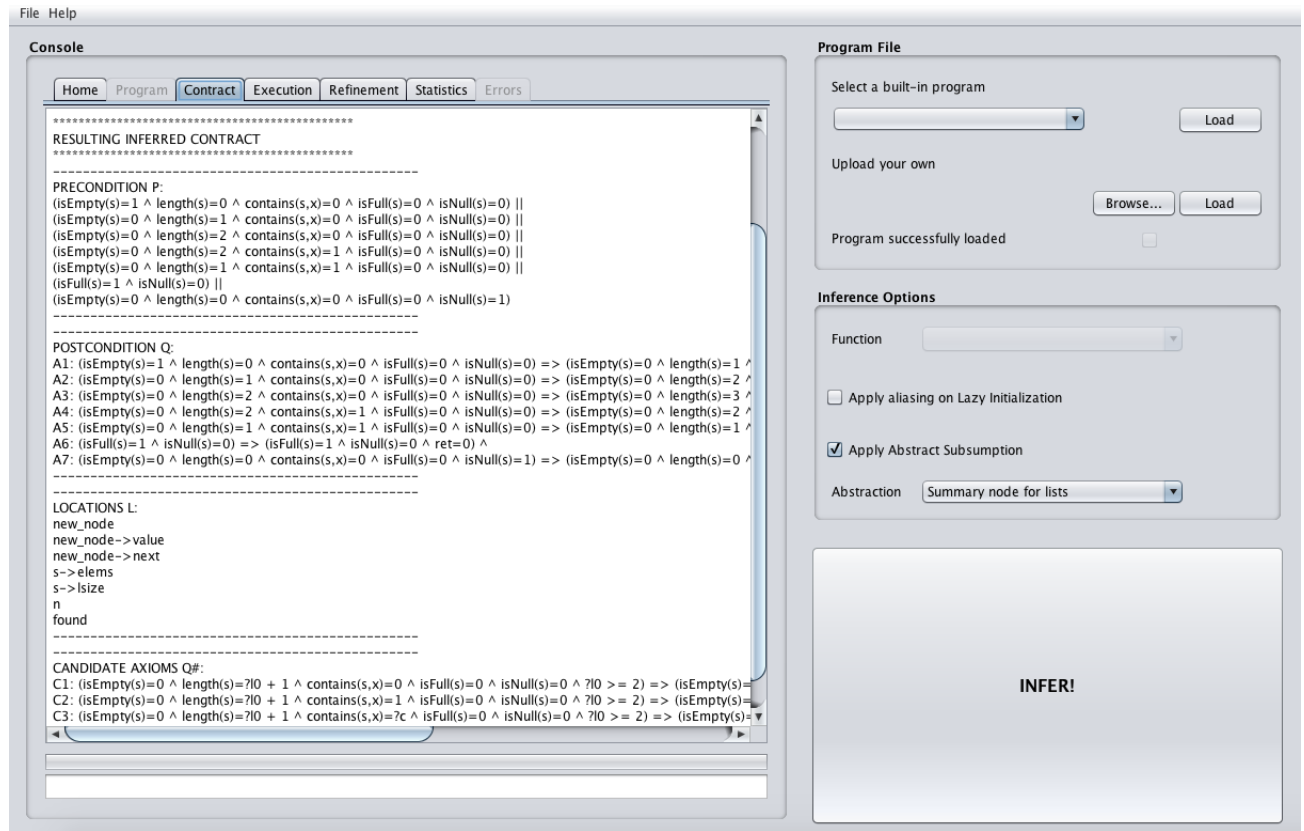


Figure 10: Contract tab in the Console panel.

three candidate axioms have been synthesized, the Refinement tab is enabled. Also, KINDSPEC 2.2 fills in the Statistics table, as shown in Figure 11.

4.3 Reading the contract

The contract obtained after running the inference process on the `insert(s,x)` function is shown in Figure 12 and consists of four main elements:

1. the function precondition P , which is given by the disjunction of all of the antecedents of the

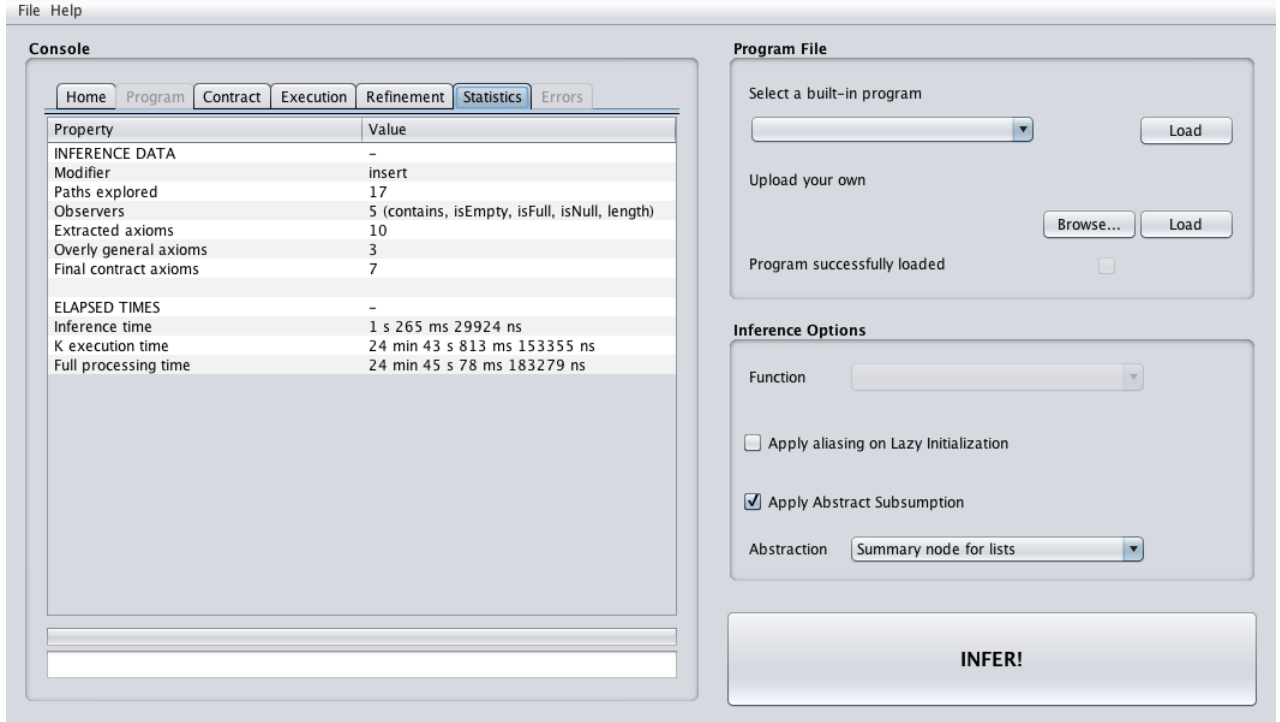


Figure 11: Statistics table and interface at the end of the inference process.

inferred axioms $(p \Rightarrow q)$;

2. the function postcondition Q , which is given by the set of inferred axioms;
3. the set L of references to memory locations (function parameters and data-structure pointers and fields) whose value might be affected by the function execution;
4. the set of candidate axioms $Q\#$, which will be explained in the following subsection.

For instance, axiom **A2** in the function postcondition can be read as follows: if the set \mathbf{s} is not null, not full, and not empty, and the value in the only node in the list is not \mathbf{x} , then, after execution, the set remains non-null and non-empty, the value \mathbf{x} is now in the set, the length is increased by 1, and the call to `insert(s,x)` returns 1, which denotes a successful insertion.

Now we can optionally save the generated contract by clicking on the top File submenu, and then choosing the **Export contract** option of the menu, which saves the serialized file that represents the contract (with `.kss` extension) in the selected directory of the file system. We can also save the contract in textual format (with `.txt` extension) to be able to access it at any time. This is done by choosing the **Save console** option of the File submenu, then choosing the **Contract console** (in the console selection dialog), and finally clicking OK.

4.4 Axiom refinement

Due to the application of abstract subsumption, the inferred contract has three axioms that are uncertain and potentially spurious. These overly general axioms are called *candidate* axioms and can be polished through the **Refinement** tab.

Let us initiate a new KINDSPEC 2.2 session (or continue it if you didn't finish it) to illustrate the refinement process by analyzing the obtained candidate axioms.

```

*****
*****
Inference performed April 08 2020
Selected modifier function: insert
From file: ../build/examples/insert_length_counting.c
*****
*****

*****
RESULTING INFERRED CONTRACT
*****
-----
PRECONDITION P:
(isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isFull(s)=1 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1)
-----
POSTCONDITION Q:
A1: (isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) =>
    (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A2: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) =>
    (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A3: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) =>
    (isEmpty(s)=0 ^ length(s)=3 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A4: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) =>
    (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^ ret=0) ^
A5: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) =>
    (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^ ret=0) ^
A6: (isFull(s)=1 ^ isNull(s)=0) => (isFull(s)=1 ^ isNull(s)=0 ^ ret=0) ^
A7: (isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1) =>
    (isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1 ^ ret=0)
-----
LOCATIONS L:
new_node
new_node->value
new_node->next
s->elems
s->lsize
n
found
-----
CANDIDATE AXIOMS Q#:
C1: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) =>
    (isEmpty(s)=0 ^ length(s)=?l0 + 2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=1) ^
C2: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) =>
    (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0) ^
C3: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2) =>
    (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^ isFull(s)=0 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0)
-----

```

Figure 12: Inferred contract for the `insert(s,x)` example.

If you have exported the contract and closed the previous session, open the KINDSPEC 2.2 application as explained in Section 2. Then, click on the File submenu and select the Load contract option. Look for the serialized .kss file saved in the previous section and select it. Once done, the Contract, Execution, and Refinement tabs will be enabled and all of the corresponding contract information will also be loaded into them. Navigate to the Refinement tab, shown in

Figure 13, and take a look⁶ at the candidate axioms. For our running example, these candidate axioms can be seen in Figure 12.

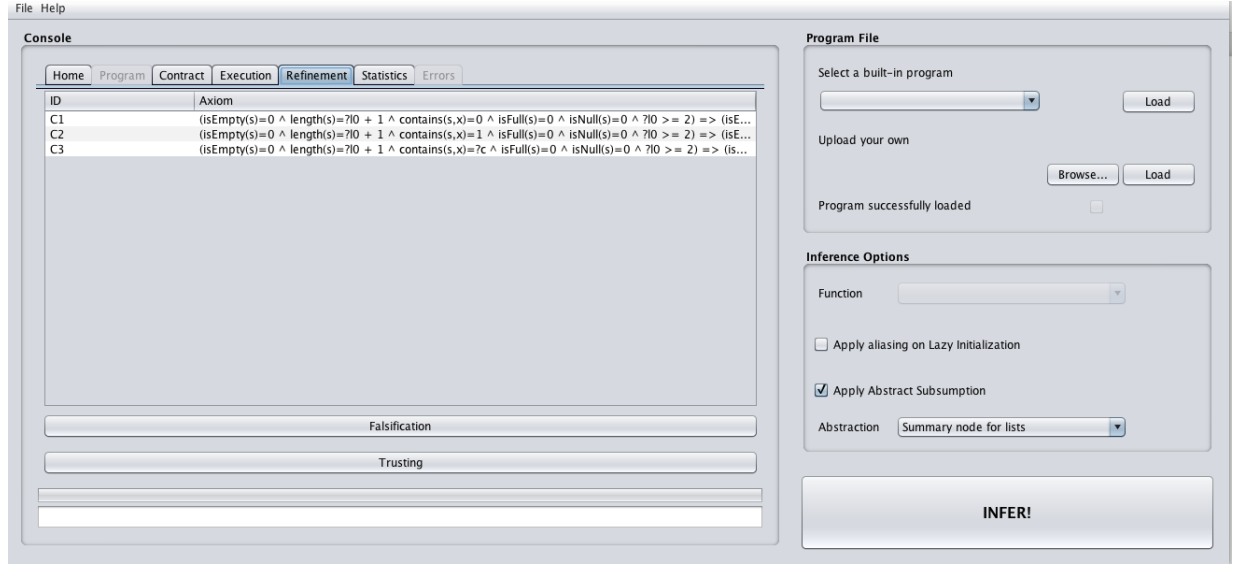


Figure 13: Refinement console.

The candidate axiom C1 can be read as follows: if the set s is not full, null, or empty, it does not contain the element x , and its length is any number greater than or equal to 2 (+ 1) before the execution of `insert(s,x)`, then after the execution s remains non-null and non-empty, its length has been increased by 1, and now it contains the element x . Additionally, the call `insert(s,x)` returned the value 1, which stands for successful insertion. Note that this axiom represents correct information according to our example program: it stands for the general case where the element does not belong to the set.

To add this candidate axioms to the contract, just select it and click on the **Trusting** button below the list of candidate axioms. The system asks the user for confirmation (Figure 14). Upon

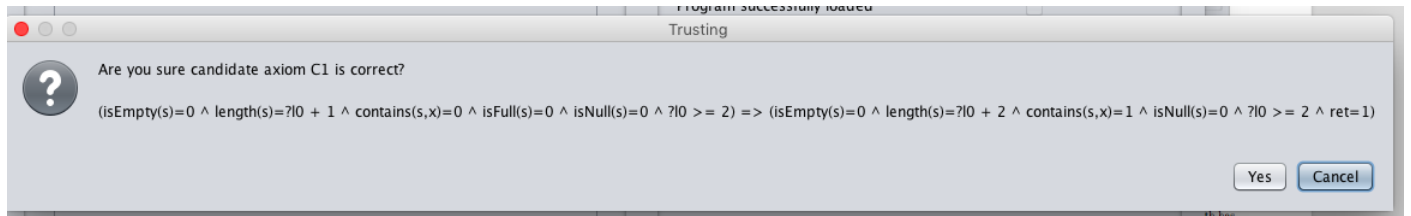


Figure 14: Confirmation before incorporating the axiom to the contract.

confirmation, a generalization and simplification process starts that tries to generate a more general axiom that subsumes some of the contract axioms, including the trusted one (Figure 15).

Then, the system might offer an hypothesis axiom that subsumes some of the axioms in the contract. If KINDSPEC 2.2 were not able to automatically check that the set of solutions of the hypothesis coincides with the solution set of the subsumed axioms, it informs the user that the hypothesis could not be verified and its confirmation is pending; otherwise it informs the range was verified (Figure 16).

⁶Note that the columns of the table can be freely redimensioned.

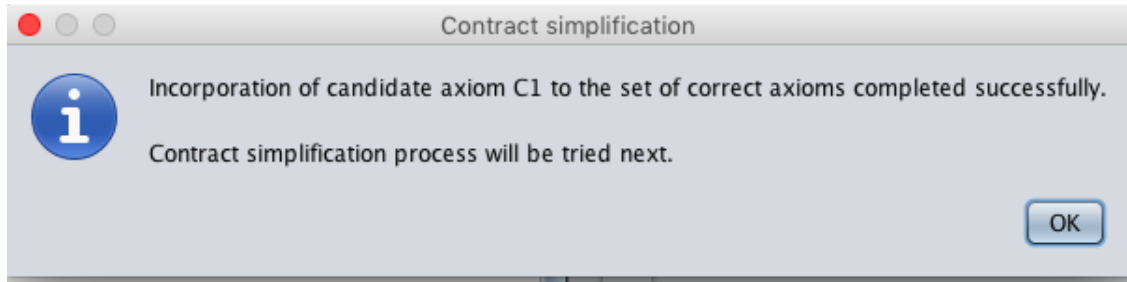


Figure 15: KindSpec informs about the steps executed.

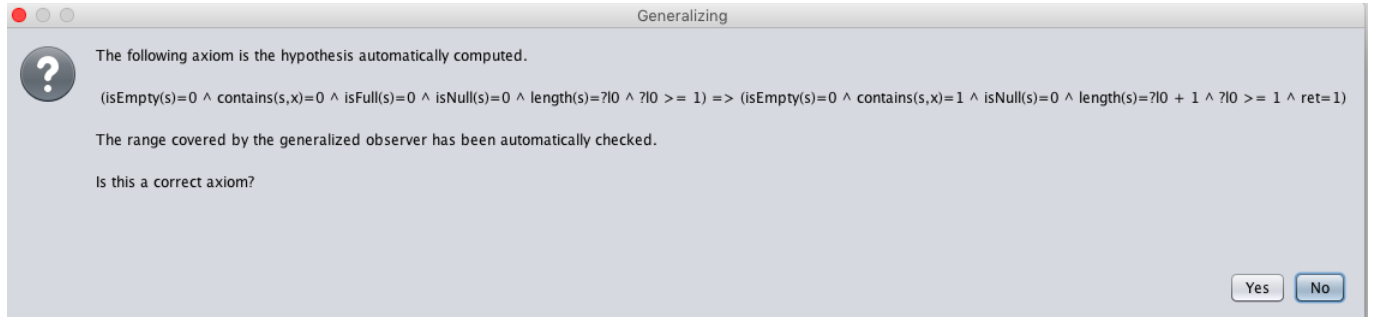


Figure 16: KindSpec shows the hypothesis computed for generalizing some axioms of the contract.

The same reasoning applies to axiom C2. It correctly represents the case when the element x was already in the set before the execution of `insert`, hence nothing is inserted in the list and `insert` returns 0. A trusting process can be then undertaken that is similar to the one for C1.

The residual contract that results from the axiom refinement process is shown in Figure 17.

Axiom C3 can be easily identified to be false since the symbolic value `?c` in the `contains(s,x)=?c` equation incorrectly states that the returned value is irrelevant, and moreover, the `insert` function returns 0 and does not modify the input list regardless of this value. As discussed above, this is not the real behavior of the `insert` function; if the list does not contain the element x , the list is modified and `insert` returns 1 as in axiom C1. We can then conclude that this axiom is incorrect, and a counterexample can be provided so that it can be finally discarded.

This can be done by first selecting axiom C3 in the table and then clicking on the `Falsify` button to start the falsification subprocess. A dialog prompts that asks the user whether he wants an automatic falsification process to be run to try refute the axiom. In the case when the user selects “No” or the automatic falsification fails, a window opens to manually provide a counterexample for the axiom, which is entered by suitably instantiating the arguments of the `insert(s,x)` function. This window is shown in Figure 18. In order to refute a candidate axiom, the provided counterexample instance is required to satisfy the precondition while the final state that is reached (after the execution of `insert`) violates the postcondition. For the specific axiom C3, it suffices to provide an instantiation where the value of x is not initially contained in a set s that has initial length ≥ 3 . Once the argument instantiation is provided, both conditions will automatically be tested by KINDSPEC 2.2.

For example, the following counterexample can be provided, which achieves the candidate axiom C3 to be falsified:

```
struct set * s = (struct set *) malloc(sizeof(struct set));
s->capacity = 7;
s->lsize = 3;
s->elems = (struct lnode *) malloc(sizeof(struct lnode));
```

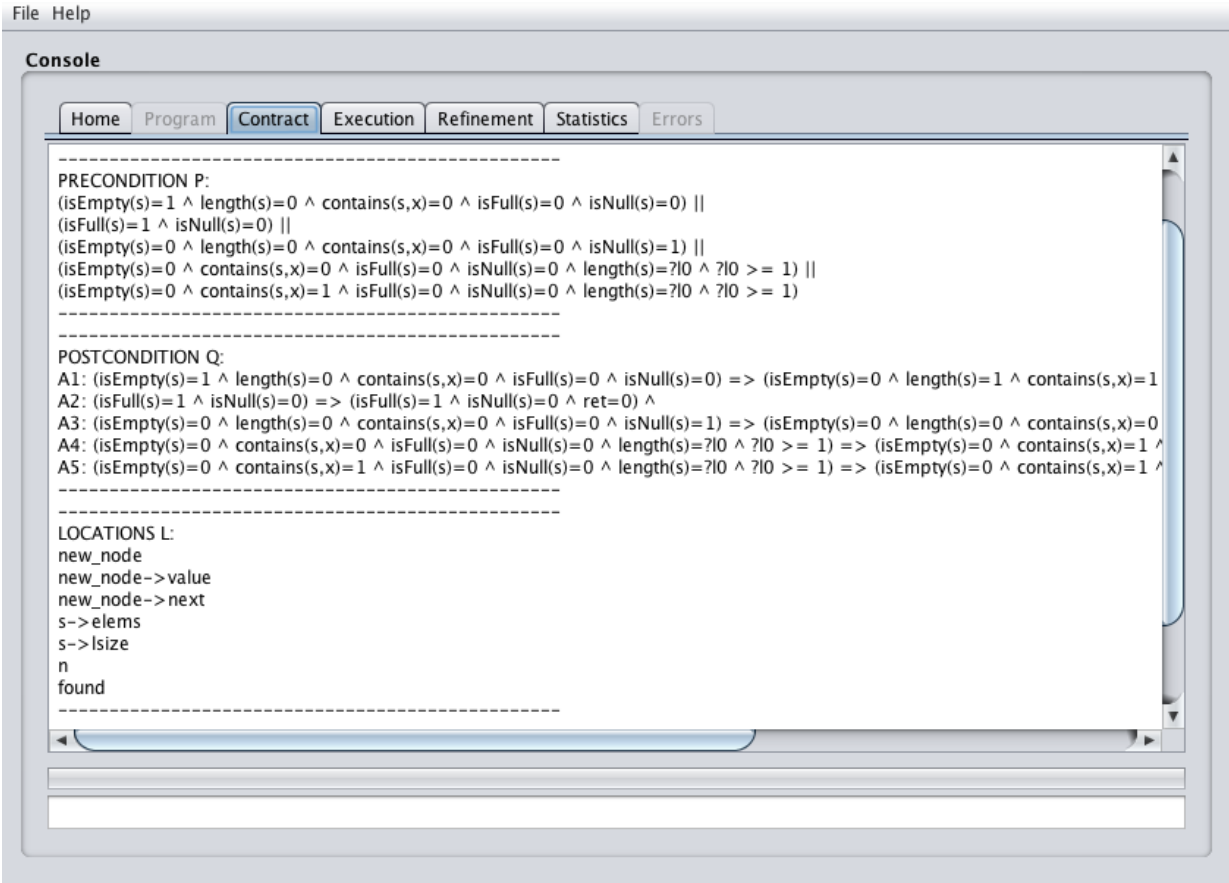


Figure 17: Contract after trusting axioms.

```

s->elems->value = 3;
s->elems->next = (struct lnode *) malloc(sizeof(struct lnode));
s->elems->next->value = 4;
s->elems->next->next = (struct lnode *) malloc(sizeof(struct lnode));
s->elems->next->value = 5;
s->elems->next->next = NULL;
int x = 8;

```

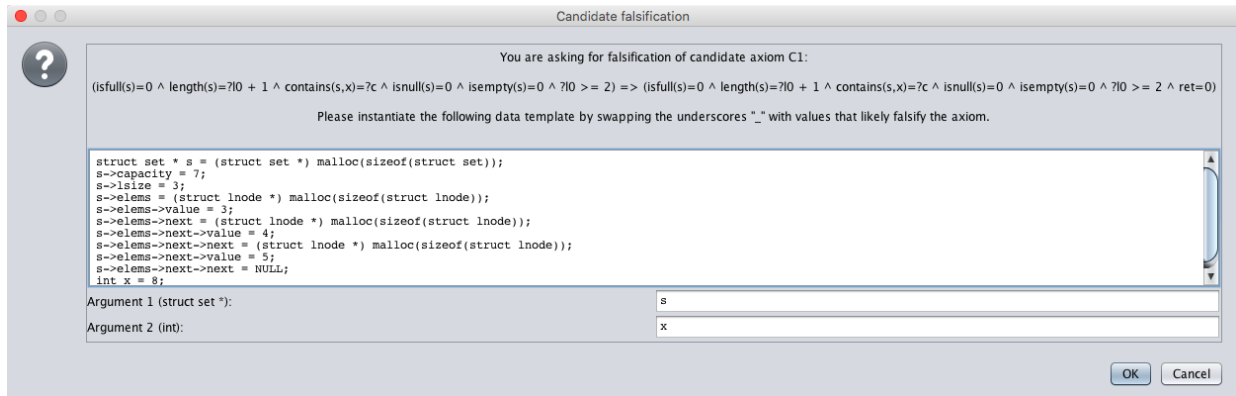


Figure 18: Axiom falsification dialog.

Given this argument instantiation, KINDSPEC 2.2 generates C code that attempts to execute: 1) the observers in the initial state as to check that the return values coincide with those stated in the candidate axiom precondition; 2) the `insert` modifier in order to obtain the final execution state; and 3) the observers in the final state so that the system can check that the return values do not coincide with those stated in the precondition. The generated code is automatically compiled and run by using the GNU Compiler GCC. If all checks are as expected and the instantiation is proved to be an effective counterexample, the axiom is ruled out and a success message is output to the user. This message also provides paths to the generated C files that falsified the axiom.

After refinement, the resulting final contract can be retrieved from the **Contract** tab of the **Console** panel and it can also be exported to a serialized file or saved in text format, as you did previously with the preliminary contract.

A Appendix: KERNELC Syntax for KINDSPEC 2.2

In the following, we present the syntactic elements that are supported by KINDSPEC 2.2's language specification for KERNELC. Roughly speaking, KERNELC is a subset of C that includes:

- Structured type definition and use;
- Function definition and calling;
- Variables and pointers of types `int` and `void`;
- Usual imperative program statements, such as variable assignment, function return, conditional and iterative instructions (by means of `if` and `while` constructs), statement blocks, etc.;
- Integer values, variable identifiers (through the predefined \mathbb{K} sorts *Int* and *Id*, respectively), and the `NULL` constant;
- Access to structured object fields by means of the `.` and `->` operators;
- Type castings;
- Access to memory content and memory addresses with the unary `*` and `&` operators;
- Usual arithmetic, relational and logical expressions for primitive (integer) values; and
- Built-in `sizeof`, `malloc`, and `free` functions for memory allocation and deallocation.

In contrast, the current KERNELC language implemented in KINDSPEC 2.2 lacks the following features:

- Type definitions (`typedef`);
- Non-integer primitive types (`float`, `double`, `char...`);
- Pointer arithmetic (this is due to the abstract memory model of KINDSPEC 2.2, which assumes that each memory cell is independent of the rest and there is no sequence between one object and another);
- Array support; and
- External library importation.

The current BNF syntax of supported KERNELC language terms (terminals and non-terminals) is presented below. Note that the $List\{S, s\}$ construct is a predefined \mathbb{K} terminal that represents a sequence of elements of sort *S* split by the separation character *s*.

File ::= *Globals*

Globals ::= $List\{Global, ""\}$

Global ::= *StructDeclaration*
 | *FunctionDeclaration*
 | *FunctionDefinition*
 | `#include <stdio.h>`
 | `#include <stdlib.h>`

StructDeclaration ::= `struct Id{ VariableDeclarations } ;`

FunctionDeclaration ::= `Type Id(ParameterDeclarations) ;`

FunctionDefinition ::= `Type Id(ParameterDeclarations) StatementBlock`

VariableDeclarations ::= $List\{ VariableDeclaration, ""\}$

VariableDeclaration ::= `Type Id ;`

ParameterDeclarations ::= $List\{ ParameterDeclaration, ", " \}$

ParameterDeclaration ::= `Type Id`

Type ::= `int`
 | `void`
 | `struct Id`
 | `Type *`

StatementBlock ::= $\{ Statements \}$

Statements ::= $List\{ Statement, "" \}$

Statement ::= *VariableDeclaration*
 | `Type Id = Expression ;`
 | `Expression = Expression ;`
 | `return Expression ;`
 | `return ;`
 | `Expression ;`
 | `;`
 | `Expression += Expression ;`
 | `Expression -= Expression ;`
 | `Expression *= Expression ;`
 | `Expression /= Expression ;`
 | `Expression ++ ;`
 | `Expression -- ;`
 | `if (Expression) Statement else Statement`
 | `if (Expression) Statement`
 | `while (Expression) Statement`
 | *StatementBlock*

<i>Expression</i> ::= <i>Int</i>		<i>Expression</i> < <i>Expression</i>
<i>Id</i>		<i>Expression</i> <= <i>Expression</i>
NULL		<i>Expression</i> > <i>Expression</i>
(<i>Expression</i>)		<i>Expression</i> >= <i>Expression</i>
<i>Expression</i> . <i>Id</i>		<i>Expression</i> == <i>Expression</i>
<i>Expression</i> -> <i>Id</i>		<i>Expression</i> != <i>Expression</i>
<i>Id</i> (<i>Arguments</i>)		! <i>Expression</i>
sizeof (<i>Type</i>)		<i>Expression</i> && <i>Expression</i>
(<i>Type</i>) <i>Expression</i>		<i>Expression</i> <i>Expression</i>
- <i>Expression</i>		
* <i>Expression</i>		
& <i>Expression</i>		
<i>Expression</i> * <i>Expression</i>		
<i>Expression</i> / <i>Expression</i>		
<i>Expression</i> + <i>Expression</i>		
<i>Expression</i> - <i>Expression</i>		

Arguments ::= *List*{*Expression*, “,”}

Id ::= **main**
| **malloc**
| **free**