UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Contract-based Analysis and Dynamic Verification of C code

**FINAL DEGREE PROJECT**

Degree in Computer Engineering

*Author:* Coroban, Raul-Ionut

*Tutors:* Alpuente Frasnedo, María
Villanueva García, Alicia

Academic Year 2017-2018

# Resum

En Engiyneria de Software, el concepte de contracte està relacionat amb l'especificació del comportament d'un programa emprant termes formals com precondicions, postcondicions i invariants. L'estat de l'art actual permet derivar propietats concisses que poden ser usades com entrada per analitzadors de codi. No obstant això, aquests contractes automàticament derivats poden no ser completament concissos o correctes, el que ens porta a el que coneixem com "contractes abstractes", que poden contindre *axiomes candidats*.

En aquest document proposem dos mètodes per al refinament de dits contractes, els quals en el nostre cas estàn generats per la ferramenta d'inferència automàtica d'especificacions denominada KindSpec 2.0. La primera proposta es basa en la realització de proves amb la ferramenta de generació automàtica de dades QuickCheck. La segona proposta, traduix els axiomes candidats a fòmules E-ACSL que son dinàmicament verificades per Frama-C. Explotant la sinergia dels mètodes, el contracte abstracte pot ser refinat i generar contractes software correctes (i sovint complets).

**Paraules clau:** Proves software automàtiques, Verificació dinàmica, Descobriment d'especificacions, Propietats d'un programa, Mètodes formals en Enginyeria Informàtica

# Resumen

En Ingeniería de Software, el concepto de contrato está relacionado con la especificación del comportamiento de un programa utilizando axiomas formales como precondiciones, postcondiciones e invariantes. El estado del arte actual permite derivar propiedades concisas que pueden ser usadas como entrada para analizadores de código con funcionalidad creciente. Sin embargo, estos contratos derivados automáticamente pueden no ser completamente precisos o correctos, correspondiendo a lo que se conoce como "contratos abstractos" que contienen *axiomas candidatos*.

En este documento proponemos dos métodos para el refinamiento de dichos contratos, los cuales en nuestro caso están generados por la herramienta de inferencia automática de especificaciones para código C llamada KindSpec 2.0. La primera técnica propuesta se basa en la realización de pruebas con la herramienta de generación automática de juegos de datos QuickCheck. La segunda técnica propuesta traduce los axiomas candidatos a fórmulas E-ACSL que son verificadas dinámicamente por Frama-C. Gracias a la sinergia entre las dos técnicas es posible refinar contratos abstractos para derivar de ellos contratos correctos y, en muchas ocasiones, completos.

**Palabras clave:** Pruebas software automáticas, Verificación dinámica, Descubrimiento de especificaciones, Propiedades de un programa, Métodos formales en Ingeniería Informática

# Abstract

In Software Engineering, software contracts allow the program behavior to be specified using formal axioms such as preconditions, postconditions and invariants. The current state of the art makes it possible to derive, from the program code, concise properties that can be then used as an input for program analyzers. However, such automatically derived contracts might not be fully precise and/or correct, leading to what is known as "abstract contracts", which may contain *candidate axioms*.

In this project we propose two methods for the refinement of automatically inferred contracts, which in our case are generated by the automatic inference tool KindSpec 2.0.

The first proposed technique is based on testing by using the automatic data generation tool QuickCheck. The second proposed technique translates candidate axioms into E-ACSL formulas that are dynamically verified by Frama-C. By exploting the synergy of the two methods, the abstract contract can be refined into correct (and often complete) software contracts.

**Key words:** Automated Software Testing, Dynamic verification, Specification Discovery, Program Properties, Formal Methods in Computer Science

# Contents

# List of Figures



# List of Tables

# CHAPTER 1
# Introduction

Over the last years, the software development industry has grown at unexampled rate. It has developed tools and applications for almost every aspect of our lives, thus is not a surprise we can find computer systems almost anywhere.

Such a rise should have been controlled in order to ensure the delivery of quality products that fit the purpose they have been created for and on which depend not only user's satisfaction, but also quotidian activities, from commercial flights to stock exchange transactions.

Software engineering has evolved in parallel with this grow in order to control the craft of Software development, which covers from the analysis of needs a software product must satisfy to the software deployment and maintenance. Software development is composed by many activities that work together to mature a concept into a software product: requirement analysis, ellicitation, design, implementation, testing, and product development.

However, nowadays development teams tend to work separately and remotely, often on small components of a complex product. This fact increases the probability (high *per se* even in experimented programmers) of introducing a bug in the code. For this reason, tools that are able to analyze the implemented code and to check the correct performance of the software product are of great importance. They not only help to obtain a quality product that can be exploited in a real-life context, but also help to reduce the maintenance time required to solve potential/future errors.

One of the main challenges in software development is to guarantee that the product is correct and free of errors. Correctness is an essential but extremely difficult property to ensure. Past failures in software history already caused fatal consequences. One of them is the well-known case of the Therac-25 radiotherapy machine [Por12] that emitted 100 times higher beta radiation doses to the patients due to a problem in data validation. Particularly, the machine had two operative modes, one using electrons and one using photons. The amount of radiation needed for photons to produce the same levels of output as the electrons is much higher. As there were no explicit security boundaries nor validation steps for the input data, a mistake made when inserting the values in the interface, using the electron mode, had been fatal for the patients.

As we can see, defects in software development can cause all kind of negative consequences, from core system breakdowns that eventually derive in losses of billions of dollars, to the fatal case that may harm human lives.

It is thus very important to use methods that that are able to remove or prevent mistakes in the code before its deployment. One of the best established approaches to accomplish this is to use *formal methods*, a collection of notations and techniques for describing, analyzing and ensuring system's properties. They are based on mathematical theories

and their main aim is to enhance software quality by verifying whether a system satisfies its specification, which significantly reduces the eventual damage caused by uncaught errors.

Nevertheless, even assuming we can use techniques based on formal artifacts that can assure a program is correct, yet the mathematical development to formalize the requiered intended specifications is often laborious. A challenghing solution relies on the possibility to automate the process.

In program analysis and verification, the user intent is expressed by some sort of specification (e.g. logical assertions, functional specifications, reference implementations, program contracts, summaries, models, passing and failing tests, etc.)

This project focuses on the process of analysing and verifying software contracts that assert some properties and are automatically generated from a C program. This is achieved in two ways: 1) By using the automatic test case generation tool QuickCheck to refine the contracts by getting rid of falsified axioms , and 2) by using the runtime verification plug-in E-ACSL that is based on program annotations of the Frama C framework to get confidence by dynamically verifying the assertions. By combining these two techniques, we are able to check that the generated contracts are reliable by using exhaustive test cases, and furthermore, we can check the corresponding E-ACSL assertions do not fail at runtime.

## 1.1 State of Art

One of the main applications of formal methods is finding bugs that cause a program not to meet its specification. The problem we might face occurs when we do not have any specification for the program or when the existing ones do not define it completely. Both issues make the task of locating bugs harder.

To address these challenges, *specification mining* techniques have been developed. They essentially consist of examining execution traces of a program in order to infer models or properties that the program satisfies. This term has been first used by Ammons et al. in [ABL02] and one of the first papers describing this topic is Cook and Wolf [CW95]. This technique solves the problem of writing program specifications and developing the software in accordance with it, and allows integration with other tools that may enhance its capabilities.

To infer properties from a software piece, some specification discovery systems such as KindSpec 2.0 [APV15, APV16] employ symbolic execution, yet other approaches eagerly search for frequent patterns using finite state machines. As well, CLIPER [LKL07] searches for common patterns through single and multiple program traces, including algorithms that increase the effectiveness of the pruning strategy. Other authors like Shoham et al. propose an inferring method that uses intreprocedural static analysis and abstract interpretation [SYFP08].

Finally, also related to this work is Daikon[1], a dynamic invariant-detection tool. The generation of invariants at runtime allows one to describe data structures and algorithms, helping their design and future maintenance. Nowadays, this kind of formal specifications is missing in most software applications due to the resources needed to invest in its generation. Moreover, manually translating properties to annotations might become a hard task. Daikon's approach is very attractive as inverts the task, inferring the property from the data structure instead of asking for a manual implementation.

[1] https://plse.cs.washington.edu/daikon/

On the other hand, formal methods are a rising trend nowadays as more and more industries include it to shield their software against failures. An example of this inclusion in the industry world is SPARK Pro [2], a tool designed to analyze software architectural requirements using formalisms. This kind of tools ease developer's tasks of preventing errors like incorrect input, integer overflow, improper initialization, array out-of-bounds errors and other unforeseen leaks that make the product vulnerable.

Also, platforms like Cardano[3] and Tezos[4] were built using formal specification which allows them to specify smart contracts, protocols that facilitates contracts negotiation on the web.

## 1.2 Objectives of this work

A program contract specifies the meaning of the program methods or subroutines, that is, the task that methods perform. It is defined in a formal, accurate and verifiable way.

The program contracts are automatically generated by the contract synthesis tool KindSpec 2.0 that relies on symbolic execution [Oak79] and abstract subsumption [ASV08]. This is done for programs written in a non-trivial subset of C that supports functions, pointer-based structures and heap manipulation. Due to the abstraction process that is applied in order to ensure termination of the symbolic inference of contracts in KindSpec 2.0, some inferred axioms cannot be guaranteed to be correct and are simply delivered as *candidate axioms* to be falsified or validated subsequently.

The main objective of this project is to develop a software system that can help to refine and validate program contracts that are automatically inferred at runtime.

## 1.3 Proposal

We aim to generate refined and verified contracts by taking advantage of two testing tools for C. First, candidate axioms generated by KindSpec 2.0 are tested by randomly generating automatic test cases that are aimed to falsify them. To accomplish this, a suitable translation is given between the assertion output language of the contract generation tool KindSpec 2.0 and the input property language of the testing tool QuickCheck. This translation is done automatically by using our first tool, and yields a set of tests so that, if any axiom is falsified by a given test, it is ruled out.

The remaining non-falsified axioms can be subsequently translated into the E-ACSL notation, a subset of the ACSL language, for runtime verification by using our second tool. Finally, those candidate axioms that cannot be either falsified or verified using the E-ACSL plug-in from Frama-C are kept as candidate or are simply discarded.

## 1.4 Project Structure

After this introduction, Chapter 2 analyzes the problem and state of the technology from which this project starts, which is the output of the KindSpec 2.0 tool. To this goal, first a running example is described. Chapter 3 presents QuickCheck approach for C programs and the automated tool AutoQCC, which is the tool we use to falsify axioms together with the translation from KindSpec axioms into QuickCheck properties. Chapter 4 is devoted to the verification of non-falsified axioms. We first present the Frama-C project then describe the translation from KindSpec axioms to E-ACSL formulas in order

[2] https://www.adacore.com/sparkpro
[3] https://www.cardano.org/en/home/
[4] https://tezos.com

to be verified by Frama-C, and finally present the automatic tool AutoEACSL. Next, in Chapter 5 we describe the full system which is used in final Chapter 7 to test the experiment.

CHAPTER 2

# KindSpec 2.0: Inference of contracts by automated synthesis

## 2.1 Automatic discovery of program properties

In this chapter we analyze the problem we are addressing, taking in account all the factors that may influence our strategy and describing the terms needed to comprehend the flow of the project.

Assertion checking techniques are an effective method for program validation, which is why they are making their way in Software Industry. Essentially, a contract, in terms of well-known software notions, consists of a set of requisites that are imposed to arguments and results when functions are defined.

Due to its interest, recently a great effort has been invested to endow programs with exhaustive contracts, although current contract inference tools are still immature in practice.

KindSpec 2.0 (KS2 f.n.o) is a tool that allows automatic contract generation for a program that is written in a non-trivial fragment of C, called KERNEL-C, which includes functions, input/output pointers, dynamic memory allocation and pointer manipulation. Contract generation in KS2 is based on a distinction between modifier methods that change the program state (in terms of a finite-state machine or FSM), and observer methods that only monitor it without doing any changes.

Starting from a Kernel-C program and a modifier method of interest, KS2 computes a suitable contract for the method which consists of a *precondition* (required for a correct behaviour of the program) and a *postcondition* that is expressed as a set of axioms (that form a declaration or a postulate which is supposed to be true in the program's context). Said axioms possess a great relevance for program analysis, since they completely define the method behaviour.

For the axiom generation, KS2 relies on *symbolic execution* [Kin76], bounded by *abstract subsumption*. Information loss associated to abstraction causes KS2 to generate two types of axioms: axioms that are correct by construction, when abstraction is not needed, and *candidate axioms* whose correction can not be guaranteed because of abstraction.

If an exhaustive checking were performed, candidate axiom correctness could be eventually either proved or falsified, and the latter might lead to a refinement process guided by counterexample generation. Although exhaustive checking is unaffordable in general, in order to acquire more confidence on the (non-falsified) candidate axioms, some ex-post verification can also be performed.

## 2.2 The contract synthesis tool KindSpec 2.0

KindSpec 2.0 is a tool that can help mitigate the specification effort as it implements a specification inference technique for heap-manipulating programs that are written in a non-trivial fragment of C. It relies on the rewriting logic semantic Framework $\mathbb{K}$ which facilitates the development of executable semantics of programming languages and also allows formal analysis tools for the defined languages to be derived with minimal effort.

Specification inference consists in discovering high-level specifications that closely describe the program behavior. Given a program $P$, any function $m$ (called a modifier) in $P$ that uses I/O primitives and/or modifies the state of encapsulated dynamic data structures defined in the program is likely to be included in the property synthesis. The intended specification for $m$ is to be cleanly expressed by using any combination of the non-modifier functions of $P$ (i.e., functions, called observers), which inspect the program state and return values expressing some information about the encapsulated data.

The key idea behind the inference procedure is that, given a modifier procedure for which we desire to obtain a specification, KS2 starts from an initial symbolic state $s$ and symbolically evaluates $m$ on $s$ to obtain as a result a set of pairs $(s,s')$ of initial and final symbolic states, respectively. Then, the observer methods in the program are used to explain the computed final symbolic states. More precisely, for each pair $(s,s')$ of initial and final states, a pre/post statement is synthesized where the precondition is expressed in terms of the observers that explain the initial state $s$, whereas the postcondition contains the observers that explain the final state $s'$. This is the final form of an axiom, which we illustrate in Figure 2.4.

In order to describe how the specification extraction technique works, let us consider a C program which implements the insertion of an element $x$ in a set $s$, according to the Listings 2.1, 2.2 and 2.3.

On one side, listing 2.1 contains the two necessary structures to build an arraylist data set that has a custom size and capacity:

1. arraylist: Contains the attributes of the data structure that indicate the current size of the set and its capacity. Also has a pointer to the array that stores the data.

2. lnode*: Structure that stores an integer item and the pointer to the next node. If the node is the last one in the list, it points to null.

The main idea of the arraylist is that the list grows in elements as long as there is enough capacity in the data structure, i.e., $capacity < sizze$.

```
1  struct arraylist {
2     int sizze;
3     int capacity;
4     struct lnode* body;
5  };
6
7  struct lnode {
8     int data;
9     struct lnode* next;
10 };
```

**Listing 2.1:** Data structures of *arraylist_insert.c*

On the other side, Listing 2.2 contains the modifier method of interest *arraylist_insert*. The method starts checking the validity of the main pointer. If the pointer is not null and

the element *item* has not already been inserted in the structure the flow of the execution continues. Then, the next condition checks if the structure has enough space for a new element. If this results true, the new item is added at the end of the structure.

```
11 int arraylist_insert(struct arraylist* l, int item)
12 {
13    struct lnode* n;
14
15    if(!l) {
16       return 0;
17    }
18
19    if(arraylist_find(l, item)) {
20       return 0;
21    }
22
23    if(l->capacity > l-> sizze && l -> body) {
24       n = l->body;
25       while(n->next) {
26          n = n->next;
27       }
28       n->next = (struct lnode*)malloc(sizeof(struct lnode));
29       n->next->data = item;
30       n->next->next = NULL;
31       (l->sizze)++;
32       return 1;
33    } else {
34       return 0;
35    }
36 }
```

**Listing 2.2:** Modifier method of *arraylist_insert.c*

Last, in Listing 2.3 we can find the observer methods of the running example:

- arraylist_find(list, element): Searches the element 'element' in the list 'list'.
- arraylist_isFull(list): Checks if list 'list' has reached full capacity.
- arraylist_isNull(list): Checks if list 'list' is initialised.
- arraylist_isEmpty(list): Checks if list 'list' has no elements inserted.
- arraylist_size(list): Returns the length of the list 'list'.

```
37 /* Finds element in list list */
38 int arraylist_find(struct arraylist* list, int element)
39 {
40    struct lnode* n;
41    int found = 0;
42
43    if(!list) return 0;
44    n = list->body;
45
46    while(n) {
47       if(n->data == element) {
48          found = 1;
49       }
50       n = n->next;
```

```
51    }
52    return found;
53 }
54
55 int arraylist_isFull(struct arraylist* list) {
56    if(!list) return 0;
57    return list->capacity <= list-> sizze;
58 }
59
60 int arraylist_isNull(struct arraylist* list) {
61    if(!list) return 1;
62    return 0;
63 }
64
65 int arraylist_isEmpty(struct arraylist* list) {
66    if(!list) {
67       return 0;
68    } else {
69       return !list->body;
70    }
71 }
72
73 int arraylist_size(struct arraylist* list) {
74    if(!list) {
75       return 0;
76    } else {
77       return list->sizze;
78    }
79 }
```

**Listing 2.3:** Observer methods of *arraylist_insert.c*

In Figure 2.4 we show a fragment of the KS2 output for the insert method of our running example. More specifically, although the output is more extensive, we only show an excerpt, a precondition and two axioms that are part of the the inferred method *Postcondition*:

```
80 PRECONDITION P:
81 (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze ^
       arraylist_find(l,item)=0 ^ arraylist_isFull(l)=1) ||
82 ...
83 ------------------------------------------------------------------------------
84 ------------------------------------------------------------------------------
85 POSTCONDITION Q:
86 AXIOMS:
87 A1: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
       ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=1) =>
88 (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze ^
       arraylist_find(l,item)=0 ^ arraylist_isFull(l)=1 ^ ret=0) ^
89
90 A2: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
       ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=0) =>
91 (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze + 1
       ^ arraylist_find(l,item)=1 ^ ret=1) ^
92 ...
```

**Listing 2.4:** KindSpec 2.0 output for the running example *insert.c*

The first axiom (A1) specifies that, provided the set $l$ is not empty or uninitialized, is full, and the element *item* is not originally contained in $l$, then after the insertion the set is not empty nor uninitialized, the size of the set is the same, the element $x$ is not contained in the set and the modifier function returns 0. Roughly speaking, the set is full and we cannot add any other element into it.

The second axiom (A2) in Listing 2.4 corresponds to the case when the list is not full, not empty, has length *i1* and does not contain the element $x$. Then, after the execution of the modifier, the length is incremented by one and the element is in the list.

As we anticipated, the synthesis of such a specification is done by combining *symbolic execution* [KaaV76], *lazy initialisation* [KaaV03], and *abstract subsumption* [KsaV08] techniques. Informally, symbolic execution analyses the program's execution by using symbolic variables (with no value assigned). Lazy initialisation, in turn, is based on delaying dynamic memory allocation for object initialisation until the first moment it is accessed, either for reading its value or for setting a new one. Finally, abstract subsumption is used to reduce the total computation space of the program (frequently used in loops to avoid termination problems associated to symbolic execution).

In this way, for each pair *(s, s')* of initial and final states in the symbolic execution tree for a modifier method, a $p \implies q$ implication is generated, where both $p$ and $q$ are expressed in terms of observer methods from the program. The axiom $p \implies q$ can be thought of as the $s$ and $s'$ explanation.

## 2.3 The correctness problem due to abstraction

In Computer Science, abstraction is a technique frequently used to hide the complexity of a system, so that the user that needs to interact with it can focus on the higher or more abstract layer rather than the implementation of the underlying layers. For example, a programmer does not need to know how the data representation works at machine level to write code because the level of abstraction has been raised in order to ease the coding task.

When it comes to Software Engineering, abstraction is really helpful in analysis and verification tasks. When we use symbolic execution, we may encounter loops and recursion in the analyzed code, that cause infinite branches in the execution path. However, not exploring all the generated branches leads to a lack of correctness in the analysis.

One solution to mitigate this problem is employing *subsumption* techniques that detect if a particular path or a similar one have been already executed. If so, the symbolic execution is halted, assuming that the given branch is covered. Nevertheless, when recursive data structures are present, this subsumption checking technique might lead to infinite loops. This is why *abstract subsumption* is proposed as a solution that can ensure termination even with complex data structures by calculating an approximation of the generated data structures, easing the subsumption task whose aim is to stop the symbolic run.

In our project, using abstraction and subsumption techniques to stop the symbolic runs involve loosing the ability to ensure the axioms inferred by KS2 are correct. In the next chapters we propose two refinement techniques that aim to obtain more precise and complete contracts.

# CHAPTER 3
# Contract refinement by using QuickCheck

QuickCheck (QC f.n.o.) is a software tool that automatically provides random test cases for program properties. It can be used to test the veracity and consistency of the given properties thanks to the the libraries that it offers for different data types. While it was originally defined for Haskell, it has been lately adapted to a multitude of programming languages, which includes C, Java, Prolog, Ruby, etc.

When one works with QC, the first step is to define the properties that must be checked. For example, Listing 3.1 contains the integer multiplication commutativity property in QuickCheck syntax for C.

```
93 QCC_TestStatus mulCommutativity(QCC_GenValue **vals, int len, QCC_Stamp **stamp)
94 {
95   int a = *QCC_getValue(vals, 0, int*);
96   int b = *QCC_getValue(vals, 1, int*);
97
98   return a*b == b*a;
99 }
```

**Listing 3.1:** QuickCheck for C - Property Example

In this code *multCommutativity* receives 3 parameters: *vals* is a double pointer that contains all the integer random values that are automatically generated by QC libraries, *len* indicates how many values were generated, and *stamp* contains information about the test outcome (passed, failed or unknown).

Thanks to the **QCC_getValue()* macro[1], the random value can be extracted, indicating the source, position and expected type of data. In this case, the first two random integers are extracted from the *vals* structure that is received as a parameter. Finally, the commutativity property is checked in terms of a boolean expresion and the result is retuned in terms of *QCC_TestStatus* which is detailed later.

QC is able to infer from the heading of the properties which data must be generated and then executes the tests and informs if tests have succeeded or not. If any of the tests fails, we conclude the program does not fulfill the property. The generated output reports how many tests were performed and its outcome. Besides, if it returns a negative result, a counterexample is generated. Figure 3.2 shows the result for the commutativity of multiplication example.

[1] A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro

```
100 multCommutativity Testing: 1000 test passed (Failed 0) - Not sure 0!
```

**Listing 3.2:** Axiom test result

It must be mentioned that, given that QC is coupled with the developed systems we must respect the syntax and semantics of the source language. For this reason, depending on the considered language, the syntax for specifying properties varies.

In this project, we focus on QuickCheck for C (QCC f.n.o.), and for its proper use we proceeded to a thorough study of its capabilities. Its understanding is relatively easy with basic knowledge about pointers, memory allocation and other common features of imperative programming languages.

The QCC system provides random value generators for all the existing primitive types in C (`int, long, float, double, char and boolean`), and also generators for arrays that contain primitive type values. In both cases, a range of values can be established for the desired tests, e.g., generate an array of integers which values range between 20 and 250. On the other hand, the main ability of the system is automatic test generation and execution, which allows eventual axiom falsification (up to the given tests) by using data generators. The result of the tests is a tag (declared as "Stamp") which indicates if the test has been passed (QCC_OK), if it has failed (QCC_FAIL) or if it has an unknown result (QCC_NOTHING). Additionally, the system supports logic operators (AND, OR, NOT and XOR) for these tags that allow defining more complex properties by combining individual test results.

The execution of all the tests admits as a parameter the number of times we want the tests to be executed, and each time it is carried out with independent random sets.

## 3.1 From KindSpec 2.0 contracts into QuickCheck program properties

As shown in Figure 3.1 the contract synthesis performed by KS2 returns a plain text and a Java object. This output must be transformed into testing structures, which later are being executed alongside QCC libraries (that include the random generators). In later sections the method for completing the translation task properly is described.

We illustrate the process by means of our running example. First, at “POSTCONDITION Q:“ headline of Figure 2.4 we see all the implications the program must satisfy, so we can deduce which tests QCC must generate.

In Listing 2.3 we can appreciate the anatomy of each method, that is to say, its name, its call and the return result when executed. Said result is just a boolean indicator flag (‘0‘ = false, ‘1‘ = true) that informs if the consulted property, characterized by observer methods, is true or false or if the operation execution has been correctly done (or any error may have arisen) in the case of modifier methods. I also retrieves relevant information in cases like *length*.

The test generation for any axiom (which needs a C function for each of them) is then carried out step-wisely as follows:

1. First, we create a testing structure in QCC by defining the name of the checking function, the number of arguments (resources it needs) and the type of data of these. For example, for an axiom A we could create a checking function that uses three arguments - two *Integers* and one *Character*.

2. Then, we extract the generated data from where QCC stored them.

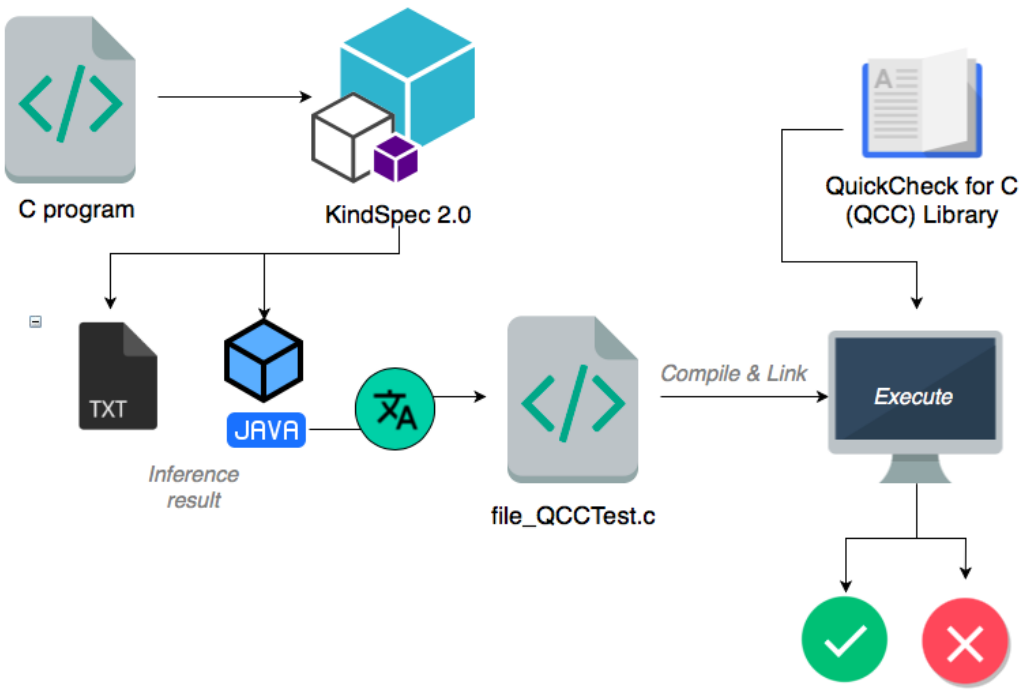


**Figure 3.1:** QCC translation flow

3. Finally, we translate the KS2 output into QCC syntax to proceed with the checking step. Each axiom corresponds to a QCC property. We note that the axiom contains an implication, which divides the clause in two parts: an *antecedent*, which must be satisfied before running the program method, and a *consequent*, which specifies the correct behavior of the execution.

In order to illustrate the proposed methodology, we analyze the following axiom:

```
101 arraylist_isFull(list)=0 ^ arraylist_size(list)=0 ^ arraylist_find(list,
        element)=0 ^ arraylist_isNull(list)=0 ^ arraylist_isEmpty(list)=1)
102                     =>
103 (arraylist_size(list)=1 ^ arraylist_find(list, element)=1 ^
        arraylist_isNull(list)=0 ^ arraylist_isEmpty(list)=0 ^ ret=1)
```

Program Listing 3.3 exemplifies the result of the procedure we have just explained for our example. First we define *axiom1* as the check function with the following default arguments in the header:

1. *len*: Indicates the size of the automatically generated test set.
2. ***vals*: Contains the automatically generated random data.
3. ***stamp*: Specifies the tags for returning the function result. It is used for other purposes not related to our work that we will not describe in this document.

Then, we extract the generated Integer value with **QCC_getValue* from the ***vals* argument. Finally, we write the following correspondence between the axiom and the code: The first conditional block"IF" corresponds to the antecedent of the implication, calling all the necessary observer methods. For example, if the axiom indicates the structure

is not full yet, with *isfull(s)=0*, in the QCC implementation a call to the method which verifies said condition must be done, in this case, *arraylist_isfull(l)*, being *l* the given data structure.

```
QCC_TestStatus axiom1(QCC_GenValue **vals, int len, QCC_Stamp **stamp) {
  arraylist* l = arraylist_create();
  int elementToContain = *QCC_getValue(vals, 1, int*); int ret; int length = 1;

  // Antecedent
  if(!arraylist_isfull(l) &&
    size(l) == 0 &&
    !contains(&elementToContain, l) &&
    !isnull(l) &&
    isempty(l)) {
      ret = insert(l, &elementToContain);

            // Consequent
            return (size(l) == 1 &&
            contains(l, &elementToContain) &&
            !isnull(l) &&
         !isempty(l) && ret);
     } else {
      return QCC_NOTHING;
     }
}
```

**Listing 3.3:** Running example translation

If the condition is true (the antecedent is satisfied), the "IF" block content, which is essentially the execution of the target method, is executed. For this example, a random data has been added to the structure. If the addition has been done correctly, a boolean indicator flag "1" is returned, or a "0" otherwise. This returned value is stored to use it in the consequent for the QCC property.

The consequent of the axiom is represented in the return block of the QCC code. Again, observer calls from the data structure are used to check if the implication holds. If all goes as expected, the test is passed, and QCC returns an affirmative stamp QCC_OK, or a QCC_FAIL otherwise. It should be mentioned a special case that arises when the property cannot be checked because the guard in the conditional is not satisfied and a conclusive result cannot be inferred, which is identified by QCC_NOTHING and does not contribute to QCC statistics when analyzing the test results. This corresponds to the case when the antecedent is not satisfied for the test case that is run, and corresponds to the "ELSE" block. All the stamps are collected by QCC and the global result of the execution is returned, indicating the number of successful results and the failures (Figure 3.5).

The last part of the methodology requires a *main* function to call the QCC library, which executes the above generated axiom test constructions. As well as each axiom has its own function defined, it also has its own function call in the *main*.

Listing 3.4 shows the function call for our running example. The first step to build it is to initialize the random() function for C, as it is the top function of QC. *QCC_init(0)* performs the initialization using an integer parameter. Then, the *QCC_testForAll* starts the automatic random test generator for the specified axiom. It requires a minimum of 5 arguments:

1. Number of tests to perform. The higher the value, the more security we win.

2. Number of maximum failures. If the number of errors reaches this value, the execution halts. In our particular case, if one error arises, the axiom can be ruled out immediately.

3. Name of the function to test. References the function to be tested with random values.

4. Number of generator(s) needed. It is related to the next argument.

5. Name of at least one generator. This is a variable arguments list which allows to specify as many generators as the User needs.

```
104 int main(int argc, char **argv) {
105    QCC_init(0);
106     printf("Axiom 1 Testing: ");
107    QCC_testForAll(1000, 10, axiom1, 2, QCC_genArrayIntLRD, QCC_genInt);
108 }
```

**Listing 3.4:** *Main* function for the running example

In this example, the axiom defined in Listing 3.3 named "axiom1" is tested with 1000 test cases, allows 10 errors and calls 2 generators: an array of integers generator and an integer value generator. After compiling and linking the C file, the execution of the test can be carried out:

```
109 Axiom 1 Testing: 1000 test passed (Failed 0) - Not sure 0!
```

**Listing 3.5:** Results of the running example

To conclude the example, the candidate axiom has passed all the tests with no failures so, in terms of the massive random testing performed by QCC, this can be considered a reasonable warranty of correctness.

Let us show an example of the output provided by QCC when a property does not hold. Assume that we modify Listing 3.3 at *"arraylist_size(l) == 0"* with *"arraylist_size(l) == 1"*, so that we obtain a negative result, which indicates how many tests must have been carried out to find the error, and the data set which made it fail. In this case, the axiom is false as witnessed by the following input data (showed in square brackets) and the value "454578811" that is inserted in the given test.

```
110 Axiom 1 Testing: Falsifiable after 1 test
111 [596, 658, 451, 540, 655, 53, 210, 180, 645, 91, 703, 905, 269, 593, 367, 164,
     825, 935, 712, 450, 526, 271, 47, 554, 222, 876, 768, 965, 348, 190, 514,
     917, 820, 961, 434, 452, 986]
112 454578811
```

**Listing 3.6:** Running example fail test

## 3.2 Automatic translation to QCC - AutoQCC Tool

The main objective of this module is to automatically generate a new file written in C which contains QCC structures and which the user is able to execute in order to verify candidate axioms. We named this tool AutoQCC, which stands for "Automatic QuickCheck for C".

Our starting point is the automatic inference tool KS2 output, which issues a file containing all the inferred axioms in a TXT format and a Java Object. At first sight, the

information the TXT file provides is not helpful for our translators, so we must search for a greater source of knowledge. We know the KS2 tool had been build in Java, and it has a complex open-source structure we are going to describe in the following sections. Knowing the inner formations of the program, a method to extract all the information about the result of the execution on a test program helps us to build our testing structures.

The solution applied in this project is to use the standard file returned as a Java Object. As the translation language to perform the automatic translation is also Java, the common standard object which serves as source of knowledge for the translator has a SER format. Among the many advantages of this format, we find how simple is for the majority of software to process it. In particular, in Java, if a class A implements the *Serializable* class, the generated objects of this A class can be easily exported to a file formated as SER.

In this way, KS2 exports a SER formated file each time it infers properties for a new program, and this contains all the objects created to automatically generate structures for QCC and EACSL. As anticipated, we need to perform an automatic translation from the SER file to generate C executable files.

The automatic inference tool KS2 is composed by many modules which work together with $\mathbb{K}$ framework to generate a program contract. In this project study a part of these modules in order to understand its functioning and to be able to explore the classes properly. The module shown in Figure 3.2, called "Inference" contains a library of tools KS2 uses to build the full inferred contract.

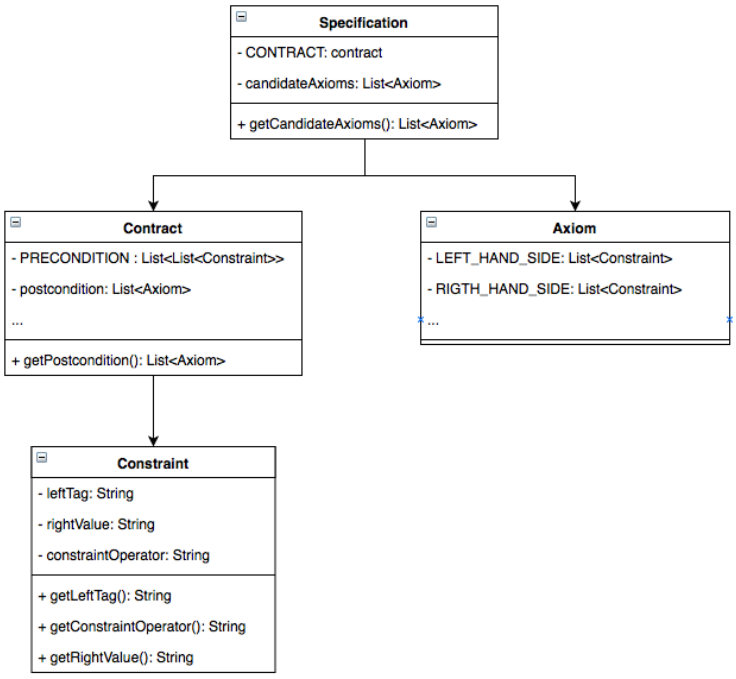


**Figure 3.2:** *Inference* module in KindSpec 2.0

We can consider *Specification* class is our starting node for our translation. It has two attributes that are used to separate the automatically generated properties: *CONTRACT* (from *Contract* class), which describes the inferred contract of a program, and *candidateAxioms*, a Java List of *Axioms* that contain the inferred candidate axiom.

On one hand, a *CONTRACT* object contains the preconditions and the postconditions similar to the ones we could see in Listing 2.4 stored as Lists of *Axioms*. Since the content of this attribute is a list of verified axioms, we do not include them in the translation.

On the other hand, *candidateAxioms* is also a List of axioms, so the analysis of this list is not different from the one applied on the *Postcondition*.

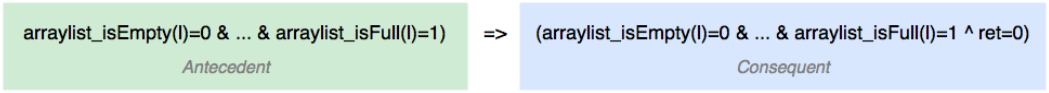


**Figure 3.3:** Axiom Structure

The structure of an *Axiom* object is simple: it contains one right side and one left side separated by an implication ($\Rightarrow$), which we call the antecedent and the consequent, respectively. In KS2 implementation, each of these attributes is a List of Constraints. We can see their structure in Figure 3.3.

A Constraint is also easy to describe, as we can see in Figure 3.4. It is formed by:

- *leftTag*: the name of the observer method.
- *constraintOperator*: the operator that relates both sides.
- *rightValue*: the expected returned value of the observer method.

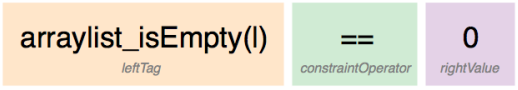


**Figure 3.4:** *Constraint* object structure

The second item we focus on is *SymbolicExecution*. Its task during the automatic property infer is to store all the methods (observers and modifier) analyzed for purposes not related to this project. They are saved in a Java List as *FunctionProfile* objects, class that collects all the parameters that define a method:

- *name*: indicates the name of the method.
- *returnType*: specifies the type of data the function returns.
- *arguments*: stores all the arguments the method needs to operate as *Argument* objects.

For example, if we consider "int isFull(struct set *s)", *name* would be "isFull", return type is an integer, and there is only a user defined struct as argument.

A fragment of the second module we are studying is shown in Figure 3.5, and it is tagged as "symbolic". It contains all the classes needed to perform the symbolic execution, but for this project we are only considering the ones displayed in the Figure.

*FunctionProfile* class helps us define all the attributes a method has, like its name, return types and a list of arguments. Using this class, a *SymbolicExecution* object can completely define all the attributes of a modifier method, and all the observer methods stored in its "programFunctions" attribute.

Knowing this information about the structure we can deduce both *SymbolicExecution* and *Specification* contain all the necessary information for us to generate the testing files.

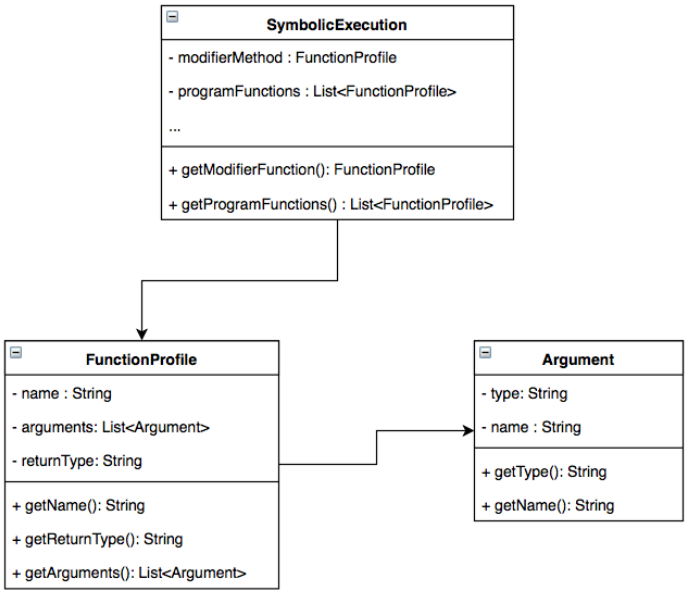


**Figure 3.5:** *Symbolic* module in KindSpec 2.0

Now, all we have to do is, once the KS2 tool has inferred a program contract, to export these two Java objects as SER formatted files and start the translation. The SER file is the necessary input file for our tools, and serves as a link of both with KS2.

The full source code of AutoQCC tool has been attached to this document as Appendix file A. The following is a step by step explanation of the program execution flow.

To perform the translation we first must create an output file where to print the result. Then, the writing can begin.

The initial step in all programs written in C is to declare the *#include* tags at the beginning of the file. By default, we import *stdio.h*, *string.h* and the QC library *quickcheck4c.h*.

Next, we import the SER formatted file and extract the object contained in it. *Specification* and *SymbolicExecution* classes contain all the information needed for the translators to perform their tasks, therefore its necessary to recover them before starting the *candidate axiom* translation.

*SymbolicExecution* object stores all the functions present in the contract and AutoQCC tool use it as source. Generally, as we build a translation, we only need the method's name, its symbolic parameter and its expected return value, all of them recoverable from *Specification*.

To begin the *candidate axiom* translation, we first need to declare variables for different purposes:

- Counters: We need to iterate over Java Lists and, although we use the efficient version of the For clause (*For Each*), we still need to enhance the output information, for example, return to the user the number of axioms and candidate axioms present in the contract. Other variables are used to count the number of types, the number of tests to be performed, etc.

```
113         int axiomCounter = 1;
114         int typesCounter = 0;
115         int numberOfTests = args[0];
```

- Data Structures: According to Listing 3.3, all the needed used in the QCC structure must be declared at the beginning of each generated method. Besides, the *main* method of the program needs to know which types of data QC needs to generate. Both *variablesToDeclare* and *typesOfData* lists are stored in a Java *ArrayList* data structure.

```
116     List<List<FunctionType>> typesOfData = new ArrayList();
117     List<Argument> variablesToDeclare = new ArrayList();
```

- Auxiliary Data Structures: KS2 uses Java List or classes that implement Java List to store information and perform its tasks. When recovering the data from either *Specification* or *SymbolicExecution*, the same data structures are used.

```
118     List<FunctionProfile> functions = se.getProgramFunctions();
119     List<Axiom> post = spec.getContract().getPostcondition();
120     List<Axiom> candidateAxioms = spec.getCandidateAxioms();
```

- Strings: As reading the recovered data from the SER file, the information is stored in plain text in order to be printed in the final file. To accomplish this, many String variables keep this data until the end of the iterations. Along the described code the *String* values include escape sequences which add tabulations (\t) and line breaks (\n) to the final code in order to improve readability.

```
121     String finalString = "";
122     String main = "";
```

In order to perform the *candidate axiom* translation, the tool needs to iterate over the lists containing the axioms of the postcondition and the candidate axioms. On each iteration it follows the same steps:

1. Sets the header of the method including the type of axiom (candidate or not), the number of the axiom (based on a counter variable we declared at the beginning) and the arguments, explained above the Listing 3.3.

```
123   // Start of the For Each iteration
124   for (Axiom a: candidateAxioms) {
125      String headerString = "QCC_TestStatus axiom" + axiomCounter +
           "(QCC_GenValue **vals, int len, QCC_Stamp **stamp) {\n";
```

2. Stores the left hand side and the right hand side of the axiom, namely, the antecedent and the consequent, which correspond to the IF clause and the Return clause from 3.3. They are both lists of *Constraint* objects, so it needs to iterate over them using a *For Each* clause too.

```
126     List<Constraint> left = a.getLeftHandSide();
127     List<Constraint> right = a.getRightHandSide();
128
129       //Nested For Each loop start
130        for (Constraint c : left) {
```

3. For the antecedent part (IF clause):

(a) On each of the inner iteration the tool saves the name, the operator and the value of each *Constraint* object. If the value of the observer method is not an integer (which is usually a boolean indicator 0 or 1, except for observer methods like *length()* or *size()*), it means KS2 set a symbolic variable during the inferring task, and it must add it to the *variablesToDeclare* list as an *Argument* object. In such case, it need further information about the function it is adding to the data structure, so it searches for it in the functions the *SymbolicExecution* object provides. As depicted function when calling "findFunction(...)" from Listing 3.7, doing this query returns a *FunctionProfile* object which contains the full data about it. The required argument for this query are a list of all the available observer methods and the name of the method we need to find.

```
131         String name = c.getLeftTag();
132         String value = c.getRightValue();
133         String co = c.getConstraintOperator();
134         counter++;
135 
136         try {
137             int newValue = Integer.parseInt(value);
138 
139             body += name + " " + co + "= " + newValue;
140             if(counter < left.size()-1) body+= " &&\n\t\t";
141             if(counter == left.size()-1) body+= ")";
142         }
143         catch(Exception e) {
144             FunctionProfile fpaux = findFunction(functions, name);
145             variablesToDeclare.add(new
                    Argument(fpaux.getReturnType(), value));
146 
147             body += name + " " + co + "= " + value;
148             if(counter <= left.size()) {
149                 body+= " &&\n\t\t";
150             } else body+= ")\n";
151         }
```

```
152   public static FunctionProfile findFunction(List<FunctionProfile>
          funcs, String functionName) {
153      for (FunctionProfile fp: funcs) {
154          if(functionName.contains(fp.getName())) return fp;
155      }
156      return null;
157   }
```

**Listing 3.7:** Auxiliary method to enhance observer method information

(b) Finally, the type of data the observer method returns is read and is added to the *typesOfData* list according to QCC types. Remember QCC has its own types of data, for example "QCC_Int" corresponds to the *int* type of C.

```
158         List<String> tys = getFunctionTypes(functions, name);
159         List<FunctionType> aux = new ArrayList();
160         for (String s: tys) {
161             aux.add(new FunctionType(s, counter));
162         }
163         types.add(typesCounter, aux);
164         typesCounter++;
165       } // End of the nested For Each
```

4. Includes the modifier method. Again, the *SymbolicExecution* object is used to enhance the data to be printed on the file, such as, the name of the function and its arguments. The latter are stored into the *variablesToDeclare* list unless they have not been already added.

```
166     body += " {\n\t\t\tret = " + se.getModifierProfile().getName() + "(";
167
168     List<Argument> ars = se.getModifierProfile().getArguments();
169     int arsCount = 0;
170     for (Argument ar : ars) {
171        variablesToDeclare.add(ar);
172        body += ar.getName();
173
174        if(arsCount <= ars.size()) body += ",";
175     }
176     body += "); //Modifier function\n";
177
178     initArgs = addArgsToBegining(variablesToDeclare);
```

5. For the consequent part (RETURN clause) the tools repeats the same steps as in 3.

6. Adds all the gathered info to a String variable whose content is printed in the file at the end of all iterations.

```
179   writer.println(finalString);
```

Last, the only remaining part to generate is the *main* function of the program. Its aim is to call all the created test functions using "QCC_testForAll" (responsible for the tests execution in QCC) indicating:

- The number of tests to be performed. This is passed as an argument by the User when executes the tool.
- The number of allowed errors. This value is set by default to "1" because with one failure we can abort the testing as there is at least one case when the axiom is not correct.
- The types of generated data needed. To do so we shall retrieve the information from the *typesOfData* list and pretty print the variables that must be declared, i.e, build a correct declaration following C syntax of all the variables stored in the *variablesToDeclare* list. This function, which is shown in Listing 3.8, has been declared outside the main one in order to keep the readability of the axiom building process.

```
180   public static String prettyPrintTypes(List<FunctionType> types) {
181       String returnString = "";
182       int counter = types.size();
183
184       for (FunctionType ft: types) {
185           returnString += ft.getQCCtype();
186        if(counter-1 > 0) returnString += ",";
187           counter--;
188       }
189       return returnString;
190   }
```

**Listing 3.8:** Pretty print function for variables needed in the testing structure

Again, as we need a call function for each axiom individually, an iteration must be carried out.

```
191     String main = "int main(int argc, char **argv) {\n\tQCC_init(0);\n";
192       main += "\tprintf(\"QCC is testing AXIOMS... \");\n\n";
193
194       int i = 0;
195       while(axiomCounter >= i) {
196           main += "\tprintf(\"Axiom " + i + ": \");\n";
197           main += "\tQCC_testForAll(" + numberOfTests + ", 10, axiom" + i + ",
                  " + types.get(i).size() + ", " + prettyPrintTypes(types.get(i)) +
                  ");\n\n";
198           i++;
199       }
200       main += "\n}";
```

**Listing 3.9:** *main function of the QCC testing structure*

To complete the translation, the *main* function and all the method calls are printed into the file.

```
201     writer.println(main);
202     writer.close();
```

At this point we obtain a C file with all the axioms translated. The last step to complete the testing structure must be performed by the user. Since the structures in C offer such a vast variety of possible constructions, the automatically generated data QCC delivers needs to be adapted to said structures. A full example to depict this is shown in the Section **??** of this document.

# CHAPTER 4
# Property checking with E-ACSL

Frama-C is a suite of tools devoted to the analysis of the source code of software written in C. It offers a static analysis (a computation of the code without executing it) which aim to perform a more in-depth look at the source code.

The C analysis platform Frama-C has wide range of extensions which broaden the capabilities of the tool and allow deeper understanding and control of the code. The E-ACSL plug-in is one of these extensions, which supports runtime verification of the C code. This task is done by translating the annotated C source code program $p$ into another program $p'$ that will be verified once is run with a test case and which will fail at runtime if any of the assertions defined by the annotations is violated. If no annotation is violated, it concludes that $p$ has the same functional behaviour as $p'$. We can consider this tool as an expansion of the Frama-C code analyzer as it performs dynamic verification aside from the static one.

The E-ACSL plug-in works with a subset of the ACSL [1] notation, which can express a wide range of functional properties thus, in order to take advantage of the tool, we first need to translate into E-ACSL syntax the output from KindSpec 2.0. ACSL is a formal specification language designed expressly to write program properties following a function contract template. Besides, it can be used to express complete or partial specifications, from a low level ("the function expects an integer as return") to high level ("the linked list returns the mean of the values stored in the odd positions"). This fact makes it a perfect candidate to use in this project.

The E-ACSL plug-in uses first-order logic to create inner annotations. First-order logic can use quantified variables over non-logical objects to express existence (∃ - *exists* symbol) and universality (∀ - *forall* symbol) and also allows the use of sentences that contain variables. For example, we can express "Toby is a dog" as "∃ a dog X, and X is a dog".

The source code of the program follows C syntax and its annotations can be written at any point of the program as long as is delimited by a comment block and start with the "@" symbol, i.e, "/*@ ... */". Also, E-ACSL provides a set of built-in predicates and logic functions that may be handy for complex asserts.

Let us introduce an example whose annotations aim to check that the value of $x$ is equal to zero during the execution. Evidently, the code in Listing 4.1 does not fail at runtime as the value assigned to $x$ is 0, while the code in Listing 4.2 causes an execution abort:

[1] http://www.frama-c.com/acsl.html

```
203 int main(void) {
204   int x = 0;
205   /*@ assert x == 0; */
206   return 0;
207 }
```

**Listing 4.1:** E-ACSL correct example

```
208 int main(void) {
209   int x = 0;
210   /*@ assert x == 1; */
211   return 0;
212 }
```

**Listing 4.2:** E-ACSL error example

When annotating a program using E-ACSL language, we must compile the program using Frama-C, and specify the E-ACSL plug-in has to be included, otherwise the annotation is ignored because the GCC compiler does not consider comment blocks. However, the compiler used by Frama-C is an extension of GCC which has been enhanced to read the comment blocks tagged with an "@" symbol.

## 4.1 Translating KindSpec 2.0 contracts into EACSL program specifications

The ACSL notation has a paramount notion among its functional properties, which is the ability to define function contracts. These contracts express the behaviour of a function in formal terms, following Hoare's style [Hoa69] where preconditions, postconditions and/or invariants can be defined using clauses like *requires*, *ensures* and *invariant*, respectively.

Remember that we are working with axioms that express a program property, and our main objective is to eventually ensure its correctness.

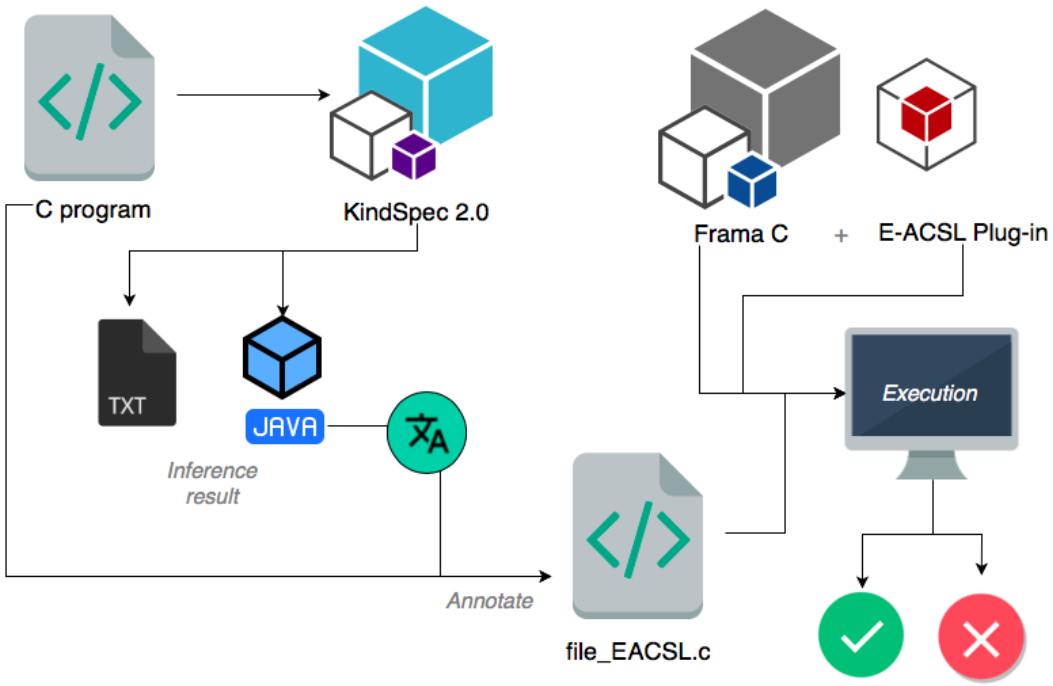


**Figure 4.1:** E-ACSL translation flow

Following the diagram from Figure 4.1 we can see the translation flow is similar to the QCC one except for two reasons: First, the libraries used come from Frama-C tool and the expansion plug-in E-ACSL. Then, the generated executable file takes the source code as input too because the system is performing an annotation into this code.

If we take as starting point the output generated by the refinement process described in Section 3.1, we have seen the final form of axioms correspond to an implication that relates preconditions $p$ and postconditions $q$. Now, a translation from the KS2 output to ACSL notation is needed in order to be able to use the E-ACSL plug-in. This annotation is placed above the modifier method for readability purposes, but it could be put anywhere in the code as is typed inside a comment block.

The notation offers a syntax based on formal assertions, where properties are easy to define. Besides, it has built-in predicates and logic functions, most of them based in C pointers, all of them marked with a backslash, both to indicate they are already defined and to prevent them to be overwritten. As an example consider *\valid(p)* that checks if pointer $p$ has a valid memory allocation, and *\result*, that retrieves a function result.

Let us assume one of the previous axioms generated by KS2:

```
213 arraylist_isFull(list)=0 ^ arraylist_size(list)=0 ^ arraylist_find(list,
        element)=0 ^ arraylist_isNull(list)=0 ^ arraylist_isEmpty(list)=1)
214                     =>
215 (arraylist_size(list)=1 ^ arraylist_find(list, element)=1 ^
        arraylist_isNull(list)=0 ^ arraylist_isEmpty(list)=0 ^ ret=1)
```

The translation from KS2 axiom to ACSL formulas is quite immediate as we only need to adapt the logic connectors (change "^ by "&&" and the boolean values (the plug-in does not interpret 0's and 1's as boolean values as C does). Also, if we are working with pointers, the clause *\valid(s)* is needed to check the validity of the argument's memory allocation. The ACSL annotation obtained from the previous KS2 axiom is the following:

```
216 /*@\valid(list);
217 ensures
218 (arraylist_isFull(list)==\false && arraylist_size(list)==\false &&
        arraylist_find(list, element)==\false && arraylist_isNull(list)==\false &&
        arraylist_isEmpty(list)==\true)
219               ==>
220 (arraylist_size(list)==\true && arraylist_find(list, element)==\true &&
        arraylist_isNull(list)==\false && arraylist_isEmpty(list)==\false &&
        \result==1);
221 */
```

**Listing 4.3:** E-ACSL Example Translation

Note a one-to-one correspondence of the implication seen in the KS2 output and the translation in Listing 4.3. The *ensures* clause encompass all commands until it finds a semicolon. In this case the clause is used because we want to guarantee that, provided the precondition is satisfied, the postcondition holds in runtime.

It is possible that the result of the used functions is stored in a variable for which we do not know its value. For example, if the list's length was $x$ before the element insertion and the process is not successful (\result == 0), after performing the operation, the length will be the same. In this case we have to introduce universal quantifiers in order to ensure this property. Besides, we use built-in functions from E-ACSL which save the variables values before the *ensures* implication *(\old)* and after it *(\at(variable_name, Post))*:

```
/*@\valid(list);
   ensures \forall integer i1, i2, i3;
         i1 >= 0 && (i2 == 1 || i2 == 0) && (i3 == 1 || i3 == 0) ==>
(arraylist_isFull(list)==\false && arraylist_size(list)==\false &&
    arraylist_find(list, element)==\false && arraylist_isNull(list)==\false &&
    arraylist_isEmpty(list)==\true)
          ==>
(arraylist_size(list)==\true && arraylist_find(list, element)==\true &&
    arraylist_isNull(list)==\false && arraylist_isEmpty(list)==\false &&
    \result==1);
          ==>
      \old(i1) == \at(i1, Post) &&
      \old(i2) == \at(i2, Post) &&
      \old(i3) == \at(i3, Post)
*/
```

**Listing 4.4:** EACSL Example Translation

Remember the KS2 output consists of a set of axioms that describe the program's behavior as independent axioms, that capture the method behavior for one particular case (the structure is empty, or has already the element contained, is null...). To express individual cases, E-ACSL allows defining function contracts based on behaviors using the keyword *\behavior*. We already know how to translate an axiom, so all boils down to structure them in the following way:

```
/*@ \valid(list)
behavior ListEmpty:
   <ensures axiom1 applied on list>;

behavior ElementIncluded:
   <ensures axiom2 applied on list>;
...
behavior ListNull:
   <ensures axiomN applied on list>;

complete behaviors
disjoint behaviors
*/
```

**Listing 4.5:** Set of axioms in ACSL

When the contract is structured in behaviors, it is not required to express a "complete" set of behaviors, that is, some cases might not be covered. Similarly, it is not required that two distinct behaviors do not overlap. If any of these conditions are desirable they can be specified at the end of the annotation block and become an extra check of the contract:

- Complete behaviors: Specifies that a set of behaviors covers all the possible conducts the program might have. The clause can depict which behaviors from the specified set are the one that make this condition true. If none is specified, the plug-in concludes all do.
- Disjoint behaviors: Specifies that a set of behaviors are pairwise disjoint, that is, the possible conducts the program might have are expressed individually and do not overlap. Similarly to the previous one, the clause can depict which behaviors make this condition true and if none is specified, all of them are considered.

Finally, in the same way as in previous sections, the elements used to define the property are observer functions from the analyzed code. For the current version of E-ACSL

this usage is not available, so a translation of these must be performed, according to the ACSL notation.

All user-defined functions in this formal language need a header declaration, where the return type, arguments (and their type) and function call are indicated. Then, E-ACSL axioms (different from the ones we are generating) must be defined as rules the function must follow to perform the function task.

Here is an example of the *isFull(s)* ACSL function:

```
246 /*@ axiomatic IsFull {
247 logic boolean isFull(struct arraylist *list) = isFull(list);
248
249 axiom NotFull:
250    \forall struct arraylist *list;
251    list->size == list->capacity ==> \true;
252
253 axiom Full:
254    \forall struct arraylist *list;
255    list->size < list->capacity ==> \ false;
256 }*/
```

**Listing 4.6:** Translation to EACSL

It should be mentioned that the created *axiomatic* structure is used to include and organize the axioms and functions, but should not be used to call the user-defined function body as is not recognized by the compiler.

## 4.2 Translated ACSL specification for the running example

In our running example described in Section 2.2 we started with the following axioms:

```
257 A1: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
    ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=1) => (arraylist_isEmpty(l)=0 ^
    arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze ^ arraylist_find(l,item)=0 ^
    arraylist_isFull(l)=1 ^ ret=0) ^
258
259
260 A2: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
    ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=0) => (arraylist_isEmpty(l)=0 ^
    arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze + 1 ^
    arraylist_find(l,item)=1 ^ ret=1) ^
261
262
263 A3: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
    ^ arraylist_find(l,item)=1) => (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=0 ^
    arraylist_size(l)=?l_sizze ^ arraylist_find(l,item)=1 ^ ret=0) ^
264
265
266 A4: (arraylist_isEmpty(l)=1 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
    ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=1) => (arraylist_isEmpty(l)=1 ^
    arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze ^ arraylist_find(l,item)=0 ^
    arraylist_isFull(l)=1 ^ ret=0) ^
267
268
269 A5: (arraylist_isEmpty(l)=1 ^ arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze
    ^ arraylist_find(l,item)=0 ^ arraylist_isFull(l)=0) => (arraylist_isEmpty(l)=1 ^
    arraylist_isNull(l)=0 ^ arraylist_size(l)=?l_sizze ^ arraylist_find(l,item)=0 ^
    arraylist_isFull(l)=0 ^ ret=0) ^
```

```
270
271
272 A6: (arraylist_isEmpty(l)=0 ^ arraylist_isNull(l)=1 ^ arraylist_size(l)=0 ^
    arraylist_find(l,item)=0 ^ arraylist_isFull(l)=0) => (arraylist_isEmpty(l)=0 ^
    arraylist_isNull(l)=1 ^ arraylist_size(l)=0 ^ arraylist_find(l,item)=0 ^
    arraylist_isFull(l)=0 ^ ret=0)
```

**Listing 4.7:** Start axioms

After translating them into QCC properties, we ran the tests with the automatic test case generator QuickCheck, and none of them had been falsified.

Then, for their dynamic verification we proceeded to translate them into E-ACSL formulas following the methodology described in the previous section. In Listing **??** we show the final ACSL specification, where the annotation is depicted in a big comment block placed before the *insert* method. First, user defined logic functions that are needed for constructing the properties are described in structures tagged as *axiomatic*. Each of these are part of the manual implementation the user needs to due to the impossibility to generate automatic mathematical specifications. Then, each axiom is defined as a behavior. In some cases, as we mentioned before, we might need to introduce the universal quantifiers *forall* to build a correct assertion. The code from Listing 4.8 shows a fragment from the final contract written in E-ACSL. Appendix C contains the full:

```
273 /*@
274 // ------- USER DEFINED LOGIC FUNCTIONS -------
275 axiomatic IsNull {
276 logic boolean isnull(struct arraylist *l) = isNull(l);
277 axiom Null:
278    \forall struct arraylist *l;
279    l->body == \null ==> \false;
280 axiom NotNull:
281    \forall struct arraylist *l;
282    l->body != \null ==> \true;
283 }*/
284
285 /*@ axiomatic IsEmpty {
286 logic boolean isempty(struct arraylist *l) = isEmpty(s);
287 axiom Empty:
288    \forall struct arraylist *l;
289    l->size == 0 ==> \true;
290 axiom NotEmpty:
291    \forall struct arraylist *l;
292    l->size > 0 ==> \false;
293 }*/
294
295 /*@ axiomatic IsFull {
296 logic boolean isfull(struct arraylist *l) = isFull(s);
297 axiom NotFull:
298    \forall struct arraylist *l;
299    l->size == l->capacity ==> \true;
300 axiom Full:
301    \forall struct arraylist *l;
302    l->size < l->capacity ==> \false;
303 }*/
304
305 /*@ axiomatic Size{
306 logic boolean size{L}(struct arraylist *l, integer a) = length{L}(s, a);
307 axiom True:
```

```
308    \forall struct arraylist *l, integer a;
309    l->size == a ==> \true;
310 axiom False:
311    \forall struct arraylist *l, integer a;
312    l->size !=a ==> \false;
313 }*/
314
315
316 /*@ axiomatic Contains {
317 logic integer contains(integer x, struct arraylist *l) = contains(l, x);
318 axiom Found:
319    \exists struct arraylist *l, integer i, integer x;
320    l[i] == x ==> \true
321
322 axiom NotFound:
323    \forall struct arraylist *l, integer i, integer x;
324    l[i] != x ==> \false;
325 }*/
326
327 // ------- BEHAVIOURS DEFINITION -------
328 /*
329 requires \valid(l);
330 behavior A:
331    ensures \forall integer ?l_sizze;
332       (?l_sizze >= 0) ==>
333    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
       ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1) ==>
334    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
       ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1 &&
       \result) ==>
335    \old(?l_sizze) == \at(?l_sizze, Post)
336
337
338 behavior B:
339    ensures \forall integer ?l_sizze;
340       (?l_sizze >= 0) ==>
341    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
       ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0) ==>
342    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
       ?l_sizze + 1 && arraylist_find(l,item) == 1 && \result) ==>
343    \old(?l_sizze) == \at(?l_sizze, Post)
344 ...
345 behavior F:
346    ensures arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 1 &&
       arraylist_size(l) == 0 && arraylist_find(l,item) == 0 && arraylist_isFull(l)
       == 0) ==>
347    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 1 && arraylist_size(l) ==
       0 && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0 && \result)
348
349
350 complete behaviors
351 disjoint behaviors
352 */
353 int arraylist_insert(arraylist* l, void* item) {...}
```

**Listing 4.8:** Final contract for arraylist_insert.c

To check whether the candidate axioms generated by KS2 are correct or not, one must run this code with the E-ACSL plug-in from Frama-C. As stated before, the whole pro-

gram is compiled by an extension of GCC and outputs an execution file if the syntax is correct. During execution, if no assert clause is violated and no error is returned, it does not fail at runtime, and we can assure the contract is correct. Otherwise, the assert clause that does not hold during execution is identified.

After performing both tests we can conclude the following based on the results:

1. If all the tests for all the inferred candidate axioms pass the QCC tool, the checking of the axioms can go through the second tool. One single error here would mean there is one case where the property does not hold, and the axiom is not correct.

2. If the program is successfully executed with using the Frama-C plug-in E-ACSL, it would mean the added annotation is correct. Therefore, we can consider the axiom as correct and include it in the final axiom list that swell the contract

Last, it should be mentioned that the user-defined logic functions are defined by the tester. For the running project, we manually inserted the functions. This task requires an in-depth knowledge and skill in Mathematics and Logic as one have to express in formal terms the methods declaration. The result of the second checking tool using E-ACSL plug-in tells us the inferred candidate axioms hold using the User-defined logic functions, but it cannot assure their definition has been done in a proper way. For example, "IsFull" logic function from Listing 4.6 checks if a *struct* member, namely "size" is equal to or less than another member called "capacity". If this logic function had been incorrectly defined and had made the test pass by chance (not failing at runtime), it would have resulted in a false negative, that is, asserting the program is error free when in fact contains an error.

Also, it can be even the opposite and deliver a false positive: abort the execution because it fails at runtime when the axiom is correct. In this case it might be the logic function definition the one not being accurate.

## 4.3 Automatic translation to EACSL - AutoEACSL Tool

Similarly to the previous tool AutoQCC described in Section 3.2, the main objective of this module is to automatically generate a new file written in C. It uses exactly the same structures shown in 3.2 and 3.5. Again, the source code is displayed in Appendix B.

Now, the file contains the source code KS2 analyses and a ACSL annotation is added to it. Remember EACSL is printed in the original program as a comment block "/*@...*/". The execution performance results unaffected as the comment block is ignored by the C compiler and is only readable by the EACSL plug-in.

The strategy followed is almost the same as in AutoQCC, except this time we read two objects instead: The SER file to gather the execution information, and the source code we are analyzing (whose contract we are verifying) is needed in order to merge the existing code and the annotation.

An overview of the generated annotation which results in a code similar to Listing **??** would be: First the user-defined methods (tagged as "axiomatic") to use in the *ensure* clauses are created. Then, "behaviors" are defined, described in Listing 4.5 according to the number of axioms. Each behavior includes an *ensures* clause which checks the implication specified in the axiom. Finally, if any of the items specified either in the antecedent or the consequent is a variable, a *forall* clause is added at the beginning of the *ensures* clause.

Our starting point for this second tool is the source code. For readability purposes, the ACSL annotation should be placed right above the modifier function of the analyzed

program. To do so, first the source code has to be read and then printed to the new file the tool is going to generate until it reaches the modifier function.

Then, the SER file is imported and both *Specification* and *SymbolicExecution* objects are extracted.

After this the building of the user-defined functions of the annotation starts. As we mentioned in Section 4.1, these functions are required by the annotation as E-ACSL does not support already-defined program methods call yet. To read all the observer methods used in the contract, *SymbolicExecution* object is used. As it iterates over the list of methods, it builds the structure of the method in terms of logic functions. It should be mentioned that this functions cannot be automatically generated, since each of them depends on the original observer method definition. The tester must manually complete this part in order to make this logic functions available.

To generate the *ensures* clauses we follow the same strategy as in AutoQCC. This part also requires auxiliary variables line counters, data structures and Lists. At each iteration over the list of axioms the tool:

1. Stores the *Constraint* methods that form its antecedent and its consequent.
2. Builds both the antecedent and the consequent by concatenating the *Constraint* content using "&&".
3. Detects if any of the expected results of the previous two constructions includes a variable. If so, it is stored in an auxiliary structure which collects the variables we need to add to the *forall* clause.
4. If there is at least one variable stored in the auxiliary structure (else clause from Listing 4.9, it builds the *forall* clause, specifying that any variable must be equal or greater than zero, as the expected results are 0 and 1 (when the observer methods return a boolean value) or a positive integer number (in observer methods like "length"). Also, if the variable relates the antecedent and the consequent, an extra check which tests if the value before (*old* ) and after (*post*) the program execution is the same is needed. For example, consider the running example of this project. If an element is successfully inserted in the data structure, the length before the execution is "x" and after it is "x+1". Assuming "x" is the Integer number 4, "x+1" would be equal to 5. This part starts after the comment block "OLD-POST" in Listing 4.9.
5. Builds the final structure of each behavior adding the *ensures* clause, the *forall* clause (if applicable), the antecedent and the consequent, and the *old* / *post* clause.

```
354     if (variablesToEnsure.isEmpty()) {
355        finalAxiomsBehaviors += header + axiomBehavior + "\n\n";
356     } else {
357        int varSize = variablesToEnsure.size();
358
359        pre += "\\forall integer ";
360        for (int i = 0; i < varSize; i++) {
361           pre += variablesToEnsure.get(i);
362              if (i < varSize-1) pre += ",";
363               }
364               pre += ";\n\t\t";
365
366
367               for (int i = 0; i < varSize; i++) {
368                   pre += "(" + variablesToEnsure.get(i) + " >= 0)";
369                   if (i < varSize-1) pre += " && ";
```

```
370                 }
371                 pre += " ==>\n";
372
373             /*
374             OLD-POST: Final ensures clause that makes sure the variables have
                    the same value before and after the execution
375             */
376                 for (int i = 0; i < varSize; i++) {
377                     post += "\\old(" + variablesToEnsure.get(i) + ") ==
                            \\at(" + variablesToEnsure.get(i) + ", Post)";
378                     if (i < varSize-1) pre += "&&";
379                 }
380                 post += "\n";
381
382                 finalAxiomsBehaviors += header + pre + "\t" + axiomBehavior
                        + "==>\n" + "\t" + post + "\n\n";
383             }
```

**Listing 4.9:** *forall* constraint building for variables

After the automatic generation of each behavior per axiom, the final step is to print the remaining part of the original source code into the final generated file.

# CHAPTER 5
# Full System Integration

In this chapter we show the final system structure where we integrate the two software artifacts into the contract synthesis methodology.

Figure 5.1 shows the full system integration, where the merge between AutoQCC and AutoEACSL systems are described.

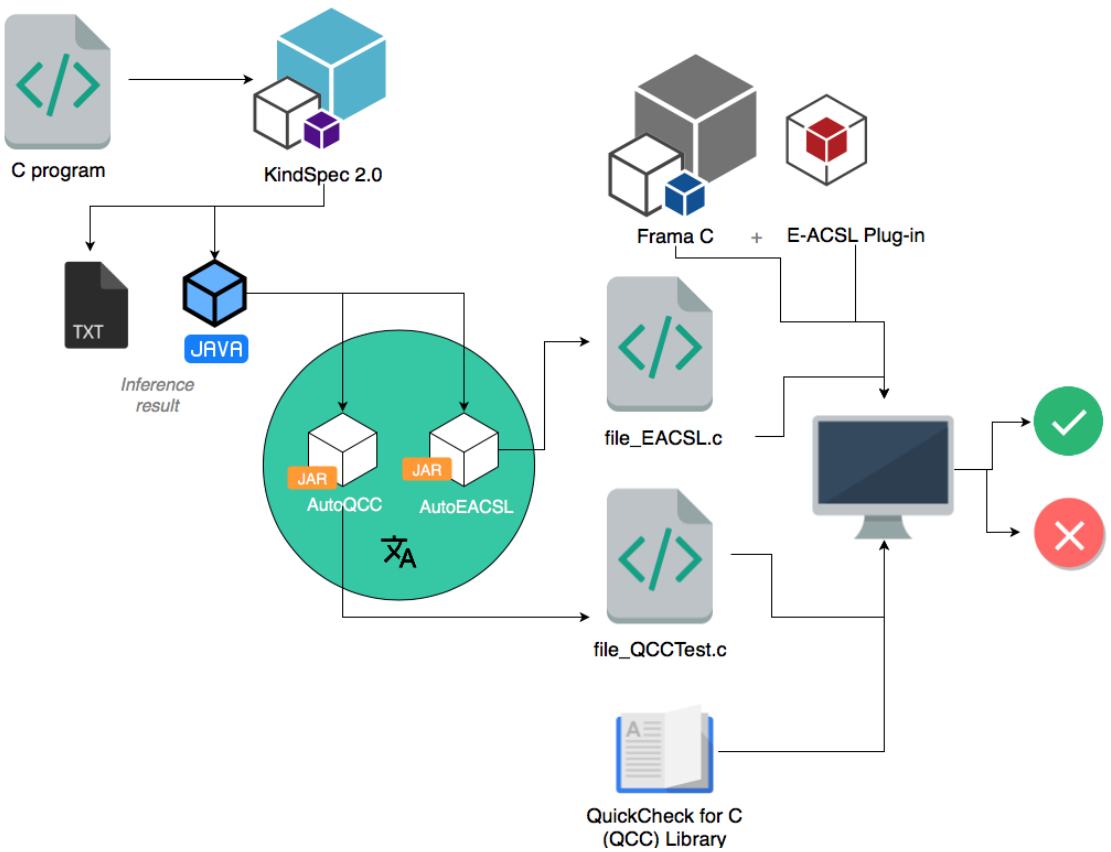


**Figure 5.1:** Full system translation flow

## 5.1 *Automatica.sh* - Shell script for tool execution

Both tools have been built in Java language so, in order to execute them we can export them individually to JAR format. In this way, we only need to execute the JAR file with the right arguments.

To ease the use of these tools the system uses command-line interface (CLI) which allows the User to execute each tool from the terminal. The script has been developed

in Shell language for Unix machines. It has a simple structure as it only has to call the JAR files and pass them the necessary arguments. Most of the work done for the script is to normalize the input and show detailed errors when the requirements are not met. Listing 5.1 contains an example of the variables used for the check: *numberFormat* is used to assure an argument is an integer value, while *fileFormat* trims the input file name in two parts[1], the name and the extension, keeping only the second one.

```
384 #!/bin/bash
385 numberFormat='^[0-9]+$'
386 inputFileFormat=$(echo "$1" |cut -d"." -f2);
```

**Listing 5.1:** "Automatica" script for AutoQCC and AutoEACSL

To execute the translators the User needs to:

1. Indicate the name of the script and the input serialized file (SER format).
2. Choose the tool wanted to use by indicating a flag.
   - "-quickcheckforc" or "-qcc" selects the AutoQCC tool.
   - "-eacsl" selects the AutoEACSL tool.
   - "-full" or "-f" uses both tools to perform the two translations.
3. Pass one argument, depending on the selected tool. AutoQCC needs to know the number of tests it must perform. AutoEACSL asks for the source file into which it has to add the annotation. If both tools are called with the "-full" flag, these two arguments have to be passed, first the number of tests and then the source code file.

The *Automatica.sh* script from checks in its first block if the file input format is a SER formatted file (KSS[2] is also allowed) that contains a Java Serializable object. As the input file is the first argument of the script call, we have to analyze the $1 variable extension name (remember in Shell $0 is the name of the script).

Then, the second block checks the number and type of arguments it has been passed. Taking into account we need at least 4 options and the maximum allowed arguments to properly call the script is 5, this block returns detailed instructions of the script call if the constraints are violated. Also, whenever the AutoQCC tool is called, we use the *$re* to check it is an integer value.

```
387 case $2 in
388      -qcc | -quickcheckforc)
389                          if [[ $3 =~ $re ]]; then
390                             java -jar AutomaticaQCC.jar $file $3
391                          fi ;;
392      -eacsl)
393                          java -jar AutomaticaEACSL.jar $file $3 ;;
394      -f | -full)            if [[ $3 =~ $re ]]; then
395                             java -jar AutomaticaQCC.jar $file $3
396                             java -jar AutomaticaEACSL.jar $file $4
397                          else
398                             exit
399                          fi ;;
400   esac
```

**Listing 5.2:** "Automatica" script for AutoQCC and AutoEACSL

[1] *-d* flag for *cut* command splits the name of the file by the indicated delimiter. *-fN* collects the *N*th fragment generated by the splitter

[2] The KSS file type is primarily associated with 'FabTrol MRP' by FabTrol Systems, Inc.

Finally, a *case* command is used in Listing 5.2 to execute the proper translator. As stated in the above list, the tools can be used individually, if we only want to perform a verification part, or together, which is the purpose of this methodology. Each of them generates a C file that contains the translations which is saved in the same directory as the JAR file.

We must mention that this script returns useful information when the input is not correct, but this part has not been included in this Chapter. Appendix D contains the full script content.

## 5.2 Towards completely automated translators

The created tools automatically generate files whose purpose are testing the contract's candidate axioms. The translation from candidate axioms to testing structures is mostly automated but, as the libraries QC offers only produce test cases for primitive variables, the translation is not executable. We invite the tester to check in depth the generated clauses and complete parts considered defective or unsound.

For the AutoQCC tool, the IF-RETURN-ELSE clause is built starting from the list of axioms provided by the SER file. This part is completely automatic. Remember this structure declares variables at the top of each function whose value come from the automatic random test generator QC. In most of the cases, the needed values are primitive, but if we find a user-defined structures (very common in C due to the language's versatility), QC does not generate a suitable test case for it. During the axiom list analysis on the AutoQCC execution we may encounter a non-primitive variable, and this will not be added to the QC functions call (described in Section 3.1) located in the *main* function.

A good solution, which we reserve for future work is to expand QC's libraries in order to obtain suitable generators for any kind of structures. Adding this capability to our system, we can develop a fully automated tool.

# CHAPTER 6
# Experiment

In this chapter we test the tools we have developed in order to assure they can be used to fulfill the objective we established. Beside, we aim to prove the project has not been developed to fit the necessities of one single program. The AutoQCC can translate any SER file generated by the automatic inferring process KS2 into C code. Remember this translation needs thorough checking and might need improvements which are up to the tester to perform.

To check the proper functioning of the tool, in the next section we expose the results for a linked list data structure example.

## 6.1 Testing linked list data structure - Insert.c

This experiment tests the *candidate axioms* of the linked list data structure shown in Listing 6.1. The program includes one modifier method called *insert* and observer methods related to the set.

```
401    struct lnode {
402        int value;
403        struct lnode *next;
404    };
405
406    struct set {
407        int capacity;
408        int sizze;
409        struct lnode *elems;
410    };
411
412    //Modifier method
413    int insert(struct set *s, int x) {...}
414
415    //Observer methods
416    int isnull(struct set *s) {...}
417
418   int isempty(struct set *s) {...}
419
420   int isfull(struct set *s) {...}
421
422   int contains(struct set *s, int x) {...}
423
424   int length(struct set *s) {...}
```

After automatically performing the contract inference by using the KS2 tool, we obtain the following candidate axioms list which we aim to falsify with AutoQCC:

```
POSTCONDITION Q:
AXIOMS:
A1: (isfull(s) = 0 ^ length(s) = ?l0 ^ contains(s,x) = 0 ^ isnull(s) = 0 ^
    isempty(s) = 0 ^ ?l0 >= 2) => (length(s) = l0 + 1 ^ contains(s,x) = 1 ^
    isnull(s) = 0 ^ isempty(s) = 0 ^ l0 >= 2 ^ ret))

A2: (isfull(s) = 0 ^ length(s) = ?l0 + 1 ^ contains(s,x) = 1 ^ isnull(s) = 0 ^
    isempty(s) = 0 ^ ?l0 >= 2) => (isfull(s) = 0 ^ length(s) = ?l0 + 1 ^
    contains(s,x) = 1 ^ isnull(s) = 0 ^ isempty(s) = 0 ^ ?l0 >= 2 ^ ret)

A3: (isfull(s) = 0 ^ length(s) = ?l0 + 1 ^ contains(s,x) = ?c ^ isnull(s) = 0 ^
    isempty(s) = 0 ^ ?l0 >= 2) => (isfull(s) = 0 ^ length(s) = ?l0 + 1 ^
    contains(s,x) = ?c ^ isnull(s) = 0 ^ isempty(s) = 0 ^ ?l0 >= 2 ^ ret);
```

**Listing 6.1:** KindSpec 2.0 output for the running *insert.c*

Following the strategy described in previous chapters, we analyze the SER file and translate it to C language. Let us show an example of the generated code by AutoQCC. The first axiom (A1) has been translated in he following way:

```
    int ret = 0;
    int ?l0 + 1 = *QCC_getValue(vals, 0, int*);
    struct set * s = *QCC_getValue(vals, 1, int*);
    int x = *QCC_getValue(vals, 2, int*);

   //Left Hand Side - Antecedent of the Axiom
   if (isfull(s) == 0 &&
      length(s) == ?l0 + 1 &&
      contains(s,x) == 0 &&
      isnull(s) == 0 &&
      isempty(s) == 0 &&
      ?l0 >== 2) {
         ret = insert(s,x); //Modifier function

         //Right Hand Side - Consequent of the Axiom
         return (length(s) == ?l0 + 2 &&
               contains(s,x) == 1 &&
               isnull(s) == 0 &&
               isempty(s) == 0 &&
               ?l0 >== 2 &&
               ret == 1);
   } else {
      return QCC_NOTHING;
   }
```

Next we must check the code for possible mistakes in order to create an executable C file. This step is necessary to avoid both small and big errors that might alter the normal execution. Also, we can add some improvements to the code that can prevent errors. For example, Listing 6.1 has the following errors:

1. The antecedent shown in describes the result of the *length* observer method as '?l0 + 1', while the consequent says '?l0 + 2'. '?l0' variable stores the length of the list 's'. It is obvious then that executing this version of the code results in an error, because the execution path never satisfies the guard of the IF clause.

2. Variables in C only allow letters, digits and underscores, so '?l0' has to be redefined.

3. As described in Section 3.2, a *Constraint* object allows to store the *constraintOperator*, which relates the function's name and its expected value. Normally it only includes an equal sign (=), but sometimes can specify other relation, like greater or equal than (>=). The AutoQCC's parser cannot infer when this happens, so it adds an extra equal sign (resulting in a '>==') every time, because in C syntax comparisons are expressed as '=='.

Finally, as we are working with a data structure which has no generator in QC libraries. As QC can generate arrays of random numbers we can manually add this data. Listing 6.1 shows the final state of the translation, which is fully executable:

```
456  int ret = 0;
457  int* items = *QCC_getValue(vals, 0, int*);
458  int x = *QCC_getValue(vals, 1, int*);
459
460  struct set *s = new(50);
461
462   int i;
463
464   for (i = 0; i < vals[1]->n; i++) {
465       insert(s, ((uint8_t *)items) + (i*sizeof(int)));
466   }
467
468  int l0 = length(s);
469  //Left Hand Side - Antecedent of the Axiom
470  if (isfull(s) == 0 &&
471     length(s) == l0 &&
472     contains(s,x) == 0 &&
473     isnull(s) == 0 &&
474     isempty(s) == 0 &&
475     l0 >= 2) {
476        ret = insert(s,x); //Modifier function
477
478        //Right Hand Side - Consequent of the Axiom
479        return (length(s) == l0 + 1 &&
480              contains(s,x) == 1 &&
481              isnull(s) == 0 &&
482              isempty(s) == 0 &&
483              l0 >= 2 &&
484              ret == 1);
485  } else {
486     return QCC_NOTHING;
487  }
```

The result of the execution for the testing procedure of the inferred candidate axioms of the linked list data structure is depicted in Table 6.1:

| Structure Tested | #Tests | #QCC_OK | #QCC_FAIL | #QCC_NOTHING | Test passed? |
|---|---|---|---|---|---|
| axiom1 | 1000 | 1000 | 0 | 42 | Yes |
| axiom2 | 1000 | 1000 | 0 | 20 | Yes |
| axiom3 | 1000 | 1000 | 0 | 14 | Yes |

**Table 6.1:** Insert.c candidate axioms test results

Analyzing the obtained outcome, we can see the number of positive results (#QCC_OK) matches the number of tests performed, thus we can conclude the *candidate axioms* have passed the first checking test. Remember that if the *candidate axiom* is falsified by one

single test, that is, if we find one case when the axiom is not true for a valid input, it can be ruled out. This is tagged by the #QCC_FAIL flag. Also we observe each test had several #QCC_NOTHING flags. These are the cases when QC generated random data which was considered not valid by the IF clause.

Finally, the last step to fully determine if these axioms can be promoted to solid axioms is to dynamically check the axioms by executing the AutoEACSL annotated program using the Frama C E-ACSL plug-in. The generated program, as mentioned in Section4.2, also needs user modification to define the logic functions the plug-in needs to run. Unfortunately, the current version of E-ACSL does not support user-defined logic function call, fact that makes impossible for us to perform the final check. For this reason, we look forward to fully complete the checking procedure once the plug-in supports the mentioned feature.

# CHAPTER 7
# Conclusions

To conclude, this project develops a double methodology to refine automatically synthesized contracts generated by the discovery tool KindSpec2.0, and accomplishes a dynamic verification of the properties of the considered C program. These are the objectives we met:

- Translation of KindSpec's output to a form QuickCheck can formally analyze.
- Testing and static verification of candidate axioms to obtain a refined list of final axioms.
- Translation of KS2 axioms to ACSL annotations that are integrated into the considered code to be dynamically verified with the E-ACSL plug-in.
- Coupling and partial automation of the translation tools.

Let us mention that the developed system enhances and refines the capabilities of an existing software, and also relies on external libraries by delegating some core tasks. This prevents us from dealing with complex data structures such as graphs, which are not supported by the external coupled systems.

Automatic techniques are a key factor for reducing the laborious tasks of both specifying contracts and creating test cases that check the sound behavior of the program. The combination of automatic systems like KS2 and and the tools developed in this project can help to this cause. Also, as for the future of Software Testing, collaboration between automatic tools can result in an efficient way of performing automatic tests, either because a partitioning of the task among the collaborators is produced or because of the synergy among several complementary tools.

Along the project development, we had to face some challenges. Both tools required previous comprehension of the environment where they were built. QCC has been developed in C language and its source code has been made efficient by using all the expressive power of the language, fact that enlarged its study time. During the testing part of this module problems related with C language arose. As mentioned in this document, the AutoQCC tool is not fully automated, and may require modifications, which are not trivial to make if the knowledge of the C language is not very extensive (mainly related to pointer access and macro usage). For this reason we strongly recommend the user to acquire some skill in C language.

The E-ACSL tool, on the other hand, required learning both the coupling between Frama-C and the E-ACSL plug-in and the ACSL notation needed to perform translations. In this part, knowledge related to Logics and Mathematics is required in order to correctly complete the structures (skeletons) from the annotations which AutoEACSL automatically adds to the source file.

One of the biggest problems we encountered also comes from the Frama C plug-in. User-defined logic functions are needed to adapt the testing structures to the source code. Not supporting this ability prevented us from fully check the candidate axioms of an inferred contract. The issue itself could not have been avoided, but if had arose earlier, we might have had time to change our strategy in other direction.

Finally, in relation to the Computer Science this project combines several skills obtained during the four-year training. Developing code and adapting to any programming language are the top two abilities among a Computer Scientist basic competences, and they have been widely applied, both in developing the tools (AutoQCC and AutoEACSL) and in generating the proper translations for the final testing files (written in C and ACSL notation).

Regarding the Software Engineering field, many other abilities have been exploited. All the training related to logical and mathematical terms was needed to understand in first place the objective of the project, and also to develop a suitable strategy to accomplish such approach. Another principal task of the project was to build a software formed by coupled modules which, at the same time, must communicate with other tools.

Overall, the project required many different yet complementary transversal competences. Analyzing the problem related to Software testing we are facing is crucial to offer a practical (and suitable) solution (CT_02, CT_03). Besides, since the field of study is reserved to the research environment in Software Engineering, manipulation of specific tools (CT_13) is needed.

# CHAPTER 8
# Future work

As already mentioned in Chapter 7, the developed tools are not considered fully automated as the generated files result of their execution are not directly executable. This is because the required extended library is not available for QC, which would allow us to offer automatic random testing for C structures.

In the future, we plan to develop a general test generator for C that can develop test cases for any C structure by using the already defined primitive type generators. We are based on the principle that, eventually, every *struct* will have a primitive type for its members, that is, each element of the composite data type can be a primitive type or a User-defined type, which in turn will have primitive types or more similar structures.

Taking this into account, QCC libraries could be expanded and include general automatic random test generators that come handy when the C source code includes complex structures. Besides, as this approach takes advantage of the already built generators the efficiency would only depend on the depth and breadth of the pointers, that is, as more complex a *struct* grows, pointers reference more and more values (memory locations), which need a random value to generate a test case.

However, the presented idea could not result efficient enough to work with complex data structures like linked lists or hash tables. A second extension of the AutoQCC tool could be including libraries that generate random values for this kind of structures.

If a general generator were integrated to the AutoQCC tool, we can fully automate it, as we will be able to create automatic random test cases for any C data type.

On the other hand, even though AutoEACSL has not the same potentiality, we still can provide a library for the most common user-defined functions. With an intermediate interface, the User can select which functions work best to substitute the observer methods, and add them automatically to the annotation. However, this library has to be checked and probably modified by the tester as each data structure can be developed following a different strategy each time.

Besides, as we mentioned at the end of the Chapter 6, E-ACSL plug-in does not support user-defined logic functions call yet, but their documentation concludes saying it is coming soon. We look forward to complete our testing procedure when the capabilities of the plug-in are enhanced.

# Bibliography


[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. 7, 2002.

[APV15] María Alpuente, Daniel Pardo, and Alicia Villanueva. Automatic inference of specifications in the k framework. 2015, 1512.06941. doi:10.4204/EPTCS.200.1.

[APV16] María Alpuente, Daniel Pardo, and Alicia Villanueva. Symbolic abstract contract synthesis in a rewriting framework, 2016, 1608.05619.

[ASV08] Saswat Anand, Corina S., and Păsăreanu Willem Visser. Symbolic execution with abstraction. 11:53–57, 2008. doi:10.1007/s10009-008-0090-1.

[CW95] Jonathan E. Cook and Alexander L. Wolf. Automating process discovery through event-data analysis. pages 73–82, 1995. doi:10.1145/225014.225021.

[Hoa69] Tony Hoare. An axiomatic basis for computer programming. 12, 1969. doi:10.1007/978-1-4612-6315-9_9.

[KaaV76] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Symbolic execution and program testing. 19:385–394, 1976. doi:10.1145/360248.360252.

[KaaV03] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. page 553–568, 2003.

[Kin76] James C. King. Symbolic execution and program testing. 19:385–394, 1976. doi:10.1145/360248.360252.

[KsaV08] Sarfraz Khurshid, Corina S. Pšăreanu, and Willem Visser. Symbolic execution with abstraction. 11:53–67, 2008. doi:10.1007/s10009-008-0090-1.

[LKL07] David Lo, Siau-Cheng Khoo, and Chao Liu. Efficient mining of iterative patterns for software specification discovery. pages 460–469, 2007. doi:10.1145/1281192.1281243.

[Oak79] J.D. Oakley. *Symbolic Execution of Formal Machine Descriptions*. CMU-CS. Department of Computer Science, Carnegie-Mellon University, 1979. URL https://books.google.es/books?id=2l_irQEACAAJ.

[Por12] Anne Marie Porrello. Death and denial: The failure of the therac-25, a medical linear accelerator. 2012. URL http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/THERAC25.html.

[SYFP08] Sharon Shoham, Eran Yahav, Stephen J. Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. 32:651 – 666, 2008. doi:10.1109/TSE.2008.63.

# APPENDIX A
# AutoQCC source code

```
488 import gui.InferenceData;
489 import inference.Axiom;
490 import inference.Constraint;
491 import inference.Specification;
492 import symbolic.Argument;
493 import symbolic.FunctionProfile;
494 import symbolic.SymbolicExecution;
495
496 import java.io.*;
497 import java.util.ArrayList;
498 import java.util.List;
499
500 public class AutoQCC {
501     public static void main(String args[]) {
502
503       /*
504       Collect KindSpec information
505       */
506         InferenceData inferenceData = getObject(args);
507         if (inferenceData == null) return; // Refactor
508         SymbolicExecution se = inferenceData.getExecutionInfo();
509         Specification spec = inferenceData.getSpecification();
510
511         System.out.println("Serialized file read!");
512
513       /*
514       Declaration of necessary structures
515       */
516         List<List<FunctionType>> typesOfData = new ArrayList(); // Position 'x'
              of the arraylist contains a list with the types for the 'x'th axiom
517         List<Argument> variablesToDeclare = new ArrayList(); // Contains all the
              variables that must be declared at the beginning of an axiom structure
518
519         List<FunctionProfile> functions = se.getProgramFunctions();
520         List<Axiom> post = spec.getContract().getPostcondition();
521         List<Axiom> candidateAxioms = spec.getCandidateAxioms();
522
523
524       /*
525       Program variables
526       */
527         String finalString = "";
528         int axiomCounter = 1;
529         int typesCounter = 0;
```

```
530         int numberOfTests = 0;
531         if (args.length > 1) numberOfTests = Integer.valueOf(args[1]);
532 
533 
534 
535       /*
536       Starting to write on the file
537       */
538         PrintWriter writer = null;
539         try {
540             writer = new PrintWriter(se.getModifierProfile().getName() +
                    "_QCCTest.c", "UTF-8");
541         } catch (FileNotFoundException e) {
542             e.printStackTrace();
543         } catch (UnsupportedEncodingException e) {
544             e.printStackTrace();
545         }
546 
547         System.out.print("Creating automated file...");
548         System.out.println("Done!");
549 
550         writer.println("#include \"quickcheck4c.h\"\n#include <stdio.h>\n#include
                <string.h>");
551         writer.println("// Include HERE all the necessary .h references to
                correctly link this file \n\n");
552         writer.println("/* SMALL API");
553         writer.println(" - Arguments for each function are the following:\n");
554         writer.println("     1. '**vals' is an array that includes all the
                generated values. Use the following syntax to retrieve the value:");
555         writer.println("          > *QCC_getValue(vals_argument,
                POSITION_IN_ARRAY, C_TYPE_OF_VALUE*);");
556         writer.println("     2. 'len' tells us how many data has been generated");
557         writer.println("     3. '**stamp' is used to manage the flags QCC_OK,
                QCC_FAIL and QCC_NOTHING\n*/\n\n\n");
558 
559 
560         System.out.print("Reading axioms...");
561         /*
562       Iteration for each Axiom on the Postcondition list
563       */
564         for (Axiom a : candidateAxioms) {
565             String headerString = "QCC_TestStatus axiom" + axiomCounter +
                    "(QCC_GenValue **vals, int len, QCC_Stamp **stamp) {\n";
566             String initArgs = "";
567             String body = "\n\t//Left Hand Side - Antecedent of the Axiom\n\tif
                    (";
568 
569 ////////////////////////////////////////////////////
570 /* IF_RETURN Clauses - Axiom definition */
571 ////////////////////////////////////////////////////
572 
573             List<Constraint> left = a.getLeftHandSide();
574             List<Constraint> right = a.getRightHandSide();
575 
576             int counter = -1;
577 
578             // IF Clause - LeftHandSide
579             for (Constraint c : left) {
580                 String name = c.getLeftTag();
```

```
581                 String value = c.getRightValue();
582                 String co = c.getConstraintOperator();
583                 counter++;
584 
585                 // If 'value' is a variable, it adds it to the declaration list
586                 try {
587                     int newValue = Integer.parseInt(value);
588 
589                     body += name + " " + co + "= " + newValue;
590                     if (counter < left.size() - 1) {
591                         body += " &&\n\t\t";
592                     }
593 
594                     if (counter == left.size() - 1) {
595                         body += ")";
596                     }
597                 } catch (Exception e) {
598                     FunctionProfile fpaux = findFunction(functions, name);
599                     if (fpaux == null) System.out.print("Function Profile not
                         found");
600 
601                     variablesToDeclare.add(new Argument(fpaux.getReturnType(),
                         value));
602 
603                     // Writes in the body part
604                     body += name + " " + co + "= " + value;
605                     if (counter <= left.size()) {
606                         body += " &&\n\t\t";
607                     } else {
608                         System.out.println("B" + counter);
609                         body += ")\n";
610                     }
611                 }
612 
613                 //Adds types
614                 List<String> tys = getFunctionTypes(functions, name);
615                 List<FunctionType> aux = new ArrayList();
616                 for (String s : tys) {
617                     aux.add(new FunctionType(s, counter));
618                 }
619                 typesOfData.add(typesCounter, aux);
620                 typesCounter++;
621             }
622 
623 
624 ////////////////////////////////////////////////////
625 /* MODIFIER Function */
626 ////////////////////////////////////////////////////
627 
628             body += " {\n\t\t\tret = " + se.getModifierProfile().getName() + "(";
629 
630             //Adding all arguments
631             List<Argument> ars = se.getModifierProfile().getArguments();
632             int arsCount = 0;
633             for (Argument ar : ars) {
634                 variablesToDeclare.add(ar);
635                 body += ar.getName();
636 
637                 if (arsCount <= ars.size()) {
```

```
638                     body += ",";
639                 }
640             }
641             body += "); //Modifier function\n";
642 
643             initArgs = addArgsToBegining(variablesToDeclare);
644 
645 
646 ////////////////////////////////////////////////////
647 /* RETURN Clause - RightHandSide */
648 ////////////////////////////////////////////////////
649 
650             body += "\n\t\t\t//Right Hand Side - Consequent of the
                    Axiom\n\t\t\treturn (";
651 
652             for (Constraint c : right) {
653                 String name = c.getLeftTag();
654                 String value = c.getRightValue();
655                 String co = c.getConstraintOperator();
656 
657                 body += name + " " + co + "= " + value + " &&\n\t\t\t\t\t\t";
658 
659                 //Adds types
660                 List<String> tys = getFunctionTypes(functions, name);
661                 List<FunctionType> aux = new ArrayList();
662                 for (String s : tys) {
663                     aux.add(new FunctionType(s, counter));
664                 }
665                 typesOfData.add(typesCounter, aux);
666                 typesCounter++;
667             }
668 
669             body += "ret == " + a.getReturnValue() + "); \n\t} else {\n\t\treturn
                    QCC_NOTHING;\n\t}\n}\n\n\n";
670 
671             axiomCounter++;
672             finalString += headerString + initArgs + body;
673             variablesToDeclare.clear();
674 
675         } // END of the FORE axiom iterator
676 
677         writer.println(finalString);
678 
679         System.out.println("Done! " + axiomCounter + " axioms found.");
680 
681 
682 ////////////////////////////////////////////////////
683 /* MAIN - Testing Structures */
684 ////////////////////////////////////////////////////
685 
686         writer.println("/* MAIN API");
687         writer.println("\tThe testing structure has the following parameters:");
688         writer.println("\t\tQCC_testForAll(NUMBER_OF_TESTS,
                    NUMBER_OF_SUPPORTED_ERRORS, FUNCTION_TO_TEST, NUMBER_OF_GENERATORS,
                    GENERATORS...)");
689         writer.println("These are the available generators:");
690         writer.println("\t\t'QCC_genInt' - Generates an Integer value");
691         writer.println("\t\t'QCC_genDouble' - Generates a Double value");
692         writer.println("\t\t'QCC_genFloat' - Generates a Float value");
```

```
693         writer.println("\t\t'QCC_genBool' - Generates a Boolean value");
694         writer.println("\t\t'QCC_genChar' - Generates a Character");
695         writer.println("\t\t'QCC_genString' - Generates a String");
696         writer.println("\t\t'QCC_genArray' - Generates an Array");
697 
698         writer.println("\n\tATTENTION! The automatically generated testing
                structures might be incomplete. Feel free to add,");
699         writer.println("\treplace or delete the generators.");
700         writer.println("*/");
701 
702         String main = "int gui(int argc, char **argv) {\n\tQCC_init(0);\n";
703         main += "\tprintf(\"QCC is testing AXIOMS... \");\n\n";
704 
705 
706         int i = 0;
707         while (axiomCounter >= i) {
708             main += "\tprintf(\"Axiom " + i + ": \");\n";
709             main += "\tQCC_testForAll(" + numberOfTests + ", 100, axiom" + i + ",
                    " + typesOfData.get(i).size() + ", " +
                    prettyPrintTypes(typesOfData.get(i)) + ");\n\n";
710             i++;
711         }
712         main += "\n}";
713 
714         writer.println(main);
715         writer.close();
716     }
717 
718 
719     /*
720     *
721     *
722     *  HELPER METHODS
723     *
724     *
725     * */
726 
727     /*
728     Adds types to the gui call of QCC.
729     */
730     public static String prettyPrintTypes(List<FunctionType> types) {
731         String returnString = "";
732         int counter = types.size();
733 
734         for (FunctionType ft : types) {
735             returnString += ft.getQCCtype();
736 
737             if (counter - 1 > 0) {
738                 returnString += ",";
739             }
740             counter--;
741         }
742         return returnString;
743     }
744 
745     /*
746     Changes normal types to QCC Types
747     */
```

```
748     public static List<String> getFunctionTypes(List<FunctionProfile>
            programFunctions, String functionName) {
749         List<String> finalReturn = new ArrayList();
750 
751         for (FunctionProfile fp : programFunctions) {
752 
753             if (functionName.contains(fp.getName())) {
754                 List<String> args = fp.getArgumentTypes();
755 
756                 for (String s : args) {
757                     if (s.equals("int")) {
758                         finalReturn.add("QCC_genInt");
759 
760                     } else if (s.equals("Integer")) {
761                         finalReturn.add("QCC_genInt");
762 
763                     } else if (s.equals("double")) {
764                         finalReturn.add("QCC_genDouble");
765 
766                     } else if (s.equals("float")) {
767                         finalReturn.add("QCC_genFloat");
768 
769                     } else if (s.equals("boolean")) {
770                         finalReturn.add("QCC_genBool");
771 
772                     } else if (s.equals("char")) {
773                         finalReturn.add("QCC_genChar");
774 
775                     } else if (s.equals("String")) {
776                         finalReturn.add("QCC_genString");
777 
778                     } else {
779                         finalReturn.add("Custom Type" + s);
780                     }
781                 } // 2nd FOR
782 
783             } // IF
784         } // 1st FOR
785 
786         if (finalReturn.isEmpty()) {
787             finalReturn.add("//Add here all the custom data generator you may
                    need. Separate them by commas");
788             return finalReturn;
789         } else {
790             return finalReturn;
791         }
792     }
793 
794 
795     /*
796     *   Prepares variables declaration at the beginning of the file. Uses the
            arguments collected from modifier function.
797     * */
798     private static String addArgsToBegining(List<Argument> argumentsToAdd) {
799         String stringToComplete = "\tint ret = 0;\n";
800         int counter = 0;
801 
802         for (Argument a : argumentsToAdd) {
```

```
803             stringToComplete += "\t" + a.getType() + " " + a.getName() + " =
                    *QCC_getValue(vals, " + counter + ", int*);\n";
804             counter++;
805         }
806         return stringToComplete;
807     }
808
809     /*
810     *   Finds FunctionProfile by its name and returns it.
811     * */
812     private static FunctionProfile findFunction(List<FunctionProfile> funcs,
            String functionName) {
813         for (FunctionProfile fp : funcs) {
814             if (functionName.contains(fp.getName())) return fp;
815         }
816         return null;
817     }
818
819     /*
820     *   Retrieves the .ser file to start the axiom translation
821     * */
822     private static InferenceData getObject(String[] args) {
823         ObjectInputStream in = null;
824         try {
825             in = new ObjectInputStream(new FileInputStream(args[0]));
826         } catch (IOException e) {
827             e.printStackTrace();
828         }
829
830         try {
831             try {
832                 InferenceData obj = (InferenceData) in.readObject();
833                 return obj;
834             } catch (IOException e) {
835                 e.printStackTrace();
836             }
837
838         } catch (ClassNotFoundException e) {
839             System.out.println("Can't read object: " + e);
840         }
841
842         return null;
843     }
844
845 }
```

# APPENDIX B
# AutoEACSL source code

```
846 import main.InferenceData;
847 import inference.Specification;
848 import symbolic.Argument;
849
850 import java.io.FileInputStream;
851 import java.io.IOException;
852 import java.io.ObjectInputStream;
853
854 import inference.Axiom;
855 import inference.Constraint;
856 import symbolic.FunctionProfile;
857 import symbolic.SymbolicExecution;
858
859
860 import java.io.*;
861 import java.util.ArrayList;
862 import java.util.List;
863
864
865 public class AutoEACSL {
866
867     public static void main(String args[]) {
868
869       /*
870       Collect KindSpec information
871       */
872         InferenceData inferenceData = getObject(args);
873         if (inferenceData == null) return; // Refactor
874         SymbolicExecution se = inferenceData.getExecutionInfo();
875         Specification spec = inferenceData.getSpecification();
876
877         System.out.println("Serialized file read!");
878
879
880       /*
881       Declaration of necessary structures
882       */
883         List<String> variablesToEnsure = new ArrayList();
884
885         List<FunctionProfile> functions = se.getProgramFunctions();
886         List<Axiom> postcondition = spec.getContract().getPostcondition();
887
888       /*
889       Program variables
```

```
890       */
891         int axiomCounter = 0;
892         char[] AXIOM_NAMES = {'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J',
                'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
                'Y', 'Z'};
893         String finalAxiomsBehaviors = "";
894         String fileLinesBefore = "";
895         String fileLinesAfter = "";
896
897
898
899       /*
900       Starting to write on the file
901       */
902         PrintWriter writer = null;
903         try {
904             writer = new PrintWriter(se.getModifierProfile().getName() +
                    "_EACSL.c", "UTF-8");
905         } catch (FileNotFoundException e) {
906             e.printStackTrace();
907         } catch (UnsupportedEncodingException e) {
908             e.printStackTrace();
909         }
910
911         /*
912         Read C file
913         */
914         try {
915             FileReader fileReader = new FileReader(args[1]);
916             BufferedReader bufferedReader = new BufferedReader(fileReader);
917
918             while((fileLinesBefore = bufferedReader.readLine()) != null) {
919                 if (!fileLinesBefore.contains(se.getModifierProfile().getName()))
                        {
920                     writer.println(fileLinesBefore);
921                 } else {
922                     fileLinesAfter += fileLinesBefore + "\n";
923                     break;
924                 }
925             }
926             while((fileLinesBefore = bufferedReader.readLine()) != null) {
927                 fileLinesAfter += fileLinesBefore + "\n";
928             }
929
930         }
931         catch(FileNotFoundException ex) {
932             System.out.println("Unable to open file");
933         }
934         catch(IOException ex) {
935             System.out.println("Error reading file ");
936         }
937
938         writer.print("/*@ ");
939
940
941 //-------------------------------------------------------------------------------------
942 /* AXIOMATIC */
943 //-------------------------------------------------------------------------------------
944         for (FunctionProfile f : functions) {
```

```
945             List<Argument> profileArguments = f.getArguments();
946             String head = "axiomatic " +
                    Character.toUpperCase(f.getName().charAt(0)) +
                    f.getName().substring(1) + " {\n";
947             String axiomArgs = "";
948             String axiomArgsNames = "";
949 
950             int axiomArgsCounter = 0;
951 
952             head += "\tlogic " + f.getReturnType() + " " + f.getName() + "(";
953 
954             for(Argument argument : profileArguments) {
955                 axiomArgs += argument.getType() + " " + argument.getName();
956                 axiomArgsNames += argument.getName();
957 
958                 if (axiomArgsCounter < profileArguments.size()-1) {
959                     axiomArgs += ",";
960                     axiomArgsNames += ",";
961                 }
962                 axiomArgsCounter++;
963             }
964             axiomArgs += ") = " + f.getName() + "(" + axiomArgsNames + ");\n";
965             writer.print(head + axiomArgs + "\t\t//Enter here your axiom
                    declaration\n}\n\n");
966 
967         }
968 
969 //-------------------------------------------------------------------------------------------------
970 /* ENSURES */
971 //-------------------------------------------------------------------------------------------------
972 
973         for (Axiom a : postcondition) {
974             String header = "behavior " + AXIOM_NAMES[axiomCounter] +
                    ":\n\tensures ";
975             String axiomBehavior = "";
976             String pre = "";
977             String post = "";
978 
979             List<Constraint> left = a.getLeftHandSide();
980             List<Constraint> right = a.getRightHandSide();
981 
982             int counter = -1;
983 
984 ////////////////////////////////////////////////////////////////////////////////////////////////
985 /* Building the precedent */
986 ////////////////////////////////////////////////////////////////////////////////////////////////
987             for (Constraint c : left) {
988                 String name = c.getLeftTag();
989                 String value = c.getRightValue();
990                 String co = c.getConstraintOperator();
991                 counter++;
992 
993                 // If 'value' is a variable, it adds it to the forall list
994                 try {
995                     int newValue = Integer.parseInt(value);
996                     axiomBehavior += name + " " + co + "= " + newValue;
997                 } catch (Exception e) {
998                     variablesToEnsure.add(value);
999                     axiomBehavior += name + " " + co + "= " + value;
```

```
1000                }
1001
1002                if (counter < left.size()-1) axiomBehavior += " && ";
1003                if (counter == left.size()-1) axiomBehavior += ")";
1004            }
1005
1006
1007            axiomBehavior += " ==> \n \t";
1008
1009
1010 ////////////////////////////////////////////////////////////////////////////////////////////////////
1011 /* Building the consequent */
1012 ////////////////////////////////////////////////////////////////////////////////////////////////////
1013
1014            for (Constraint c : right) {
1015                String name = c.getLeftTag();
1016                String value = c.getRightValue();
1017                String co = c.getConstraintOperator();
1018
1019                axiomBehavior += name + " " + co + "= " + value + " && ";
1020            }
1021
1022            axiomBehavior += "\\result) ";
1023
1024
1025      /*
1026      If any variable is found, it must be added to the formal spec at the
            beginning and at the end, using a
1027      forall clause and old-post clause, respectively
1028      */
1029            if (variablesToEnsure.isEmpty()) {
1030                finalAxiomsBehaviors += header + axiomBehavior + "\n\n";
1031            } else {
1032         /*
1033         FORALL: Ensures all the variables are boolean indicators
1034         */
1035                int varSize = variablesToEnsure.size();
1036                //Adds items to forall
1037                pre += "\\forall integer ";
1038                for (int i = 0; i < varSize; i++) {
1039                    pre += variablesToEnsure.get(i);
1040                    if (i < varSize-1) pre += ",";
1041                }
1042                pre += ";\n\t\t";
1043
1044                //Declare forall bounds
1045                for (int i = 0; i < varSize; i++) {
1046                    pre += "(" + variablesToEnsure.get(i) + " >= 0)";
1047                    if (i < varSize-1) pre += " && ";
1048                }
1049                pre += " ==>\n";
1050
1051         /*
1052         OLD-POST: Final ensures clause that makes sure the variables have the
               same value before and after the execution
1053         */
1054                for (int i = 0; i < varSize; i++) {
1055                    post += "\\old(" + variablesToEnsure.get(i) + ") == \\at(" +
                           variablesToEnsure.get(i) + ", Post)";
```

```
                    if (i < varSize-1) pre += "&&";
                }
                post += "\n";

                finalAxiomsBehaviors += header + pre + "\t" + axiomBehavior +
                    "==>\n" + "\t" + post + "\n\n";
            }

            variablesToEnsure.clear();
            axiomCounter++;
        }

        writer.println(finalAxiomsBehaviors);
        writer.println("complete behaviors\ndisjoint behaviors");
        writer.println("*/\n");
        writer.print(fileLinesAfter);
        writer.close();


    }

    /*
     *   Finds FunctionProfile by its name and returns it.
     * */
    public static FunctionProfile findFunction(List<FunctionProfile> funcs,
        String functionName) {
        for (FunctionProfile fp: funcs) {
            if(functionName.contains(fp.getName())) return fp;
        }
        return null;
    }

    /*
     *   Retrieves the .ser file to start the axiom translation
     * */
    private static InferenceData getObject(String[] args) {
        ObjectInputStream in = null;
        try {
            in = new ObjectInputStream(new FileInputStream(args[0]));
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            InferenceData obj = (InferenceData) in.readObject();
            return obj;
        } catch (IOException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        return null;
    }
}
```

# APPENDIX C
# AutoEACSL inference result

The following Listing contains testing file E-ACSL aims to test at runtime. Since the logic functions cannot be fully translated to ACSL notation, the ones appearing in this code have been implemented by our team.

```
1109 /*@
1110 // ------- USER DEFINED LOGIC FUNCTIONS -------
1111 axiomatic IsNull {
1112 logic boolean isnull(struct arraylist *l) = isNull(l);
1113 axiom Null:
1114    \forall struct arraylist *l;
1115    l->body == \null ==> \false;
1116 axiom NotNull:
1117    \forall struct arraylist *l;
1118    l->body != \null ==> \true;
1119 }*/
1120
1121 /*@ axiomatic IsEmpty {
1122 logic boolean isempty(struct arraylist *l) = isEmpty(s);
1123 axiom Empty:
1124    \forall struct arraylist *l;
1125    l->size == 0 ==> \true;
1126 axiom NotEmpty:
1127    \forall struct arraylist *l;
1128    l->size > 0 ==> \false;
1129 }*/
1130
1131 /*@ axiomatic IsFull {
1132 logic boolean isfull(struct arraylist *l) = isFull(s);
1133 axiom NotFull:
1134    \forall struct arraylist *l;
1135    l->size == l->capacity ==> \true;
1136 axiom Full:
1137    \forall struct arraylist *l;
1138    l->size < l->capacity ==> \false;
1139 }*/
1140
1141 /*@ axiomatic Size{
1142 logic boolean size{L}(struct arraylist *l, integer a) = length{L}(s, a);
1143 axiom True:
1144    \forall struct arraylist *l, integer a;
1145    l->size == a ==> \true;
1146 axiom False:
1147    \forall struct arraylist *l, integer a;
1148    l->size !=a ==> \false;
```

```
}*/


/*@ axiomatic Contains {
logic integer contains(integer x, struct arraylist *l) = contains(l, x);
axiom Found:
   \exists struct arraylist *l, integer i, integer x;
   l[i] == x ==> \true

axiom NotFound:
   \forall struct arraylist *l, integer i, integer x;
   l[i] != x ==> \false;
}*/

// ------- BEHAVIOURS DEFINITION -------
/*
requires \valid(l);
behavior A:
   ensures \forall integer ?l_sizze;
      (?l_sizze >= 0) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1 &&
   \result) ==>
   \old(?l_sizze) == \at(?l_sizze, Post)


behavior B:
   ensures \forall integer ?l_sizze;
      (?l_sizze >= 0) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze + 1 && arraylist_find(l,item) == 1 && \result) ==>
   \old(?l_sizze) == \at(?l_sizze, Post)


behavior C:
   ensures \forall integer ?l_sizze;
      (?l_sizze >= 0) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 1) ==>
   arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 1 && \result) ==>
   \old(?l_sizze) == \at(?l_sizze, Post)


behavior D:
   ensures \forall integer ?l_sizze;
      (?l_sizze >= 0) ==>
   arraylist_isEmpty(l) == 1 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1) ==>
   arraylist_isEmpty(l) == 1 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
   ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 1 &&
   \result) ==>
   \old(?l_sizze) == \at(?l_sizze, Post)


```

```
1198 behavior E:
1199    ensures \forall integer ?l_sizze;
1200       (?l_sizze >= 0) ==>
1201    arraylist_isEmpty(l) == 1 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
        ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0) ==>
1202    arraylist_isEmpty(l) == 1 && arraylist_isNull(l) == 0 && arraylist_size(l) ==
        ?l_sizze && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0 &&
        \result) ==>
1203    \old(?l_sizze) == \at(?l_sizze, Post)
1204
1205
1206 behavior F:
1207    ensures arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 1 &&
        arraylist_size(l) == 0 && arraylist_find(l,item) == 0 && arraylist_isFull(l)
        == 0) ==>
1208    arraylist_isEmpty(l) == 0 && arraylist_isNull(l) == 1 && arraylist_size(l) ==
        0 && arraylist_find(l,item) == 0 && arraylist_isFull(l) == 0 && \result)
1209
1210
1211 complete behaviors
1212 disjoint behaviors
1213 */
```

**Listing C.1:** Final contract for arraylist_insert.c

# APPENDIX D
# *Automatica.sh* Script

```
1214 #!/bin/bash
1215 re='^[0-9]+$'
1216 file=$1;
1217 format=$(echo "$1" |cut -d"." -f2);
1218
1219 if [[ $format != "ser" ]] || [[ $format != "kss" ]]; then
1220    echo "------------- File Format Error -------------"
1221    echo "Please, select a file with a serialized object. These tend to have
         '.ser' or '.kss' formats."
1222    echo "The selected file has a '."$format"' format"
1223    echo ""
1224    exit
1225 fi
1226
1227 if [[ $# -lt 3 ]] || [[ $# -gt 5 ]]; then
1228    echo "------------- Input Error -------------"
1229    echo "Please use the following call of the tool:"
1230    echo " ./Automatics.sh input_file [-qcc #tests | -eacsl source_code | -full
         #tests source_code]"
1231    echo ""
1232    echo "  -qcc       Calls 'AutoQCC' tool. Its only argument is
         #number_of_tests"
1233    echo "  -eacsl       Calls 'AutoEACSL' tool. Its argument is the name of the
         'source_code' file"
1234    echo "  -full      Calls both 'AutoQCC' and 'AutoEACSL'. It needs both
         arguments #number_of_tests and 'source_code'"
1235    echo ""
1236 else
1237    case $2 in
1238       -qcc | -quickcheckforc)
1239                            if [[ $3 =~ $re ]]; then
1240                               echo "------------------ Starting translation to
                                     QuickCheck... -----------------"
1241                               java -jar AutomaticaQCC.jar $file $3
1242                               echo "Translation successful. C file has been
                                     generated, please check it."
1243                            else
1244                               echo "Please provide a correct number for the #tests
                                     parameter of QCC"
1245                            fi
1246                            ;;
1247       -eacsl)
1248                            echo "------------------ Starting translation to
                                 EACSL... -----------------"
```

```
1249                        java -jar AutomaticaEACSL.jar $file $3
1250                        echo "Translation successful. EACSL file has been
                                generated, please check it."
1251                        ;;
1252
1253      -f | -full)           if [[ $3 =~ $re ]]; then
1254                            java -jar AutomaticaQCC.jar $file $3
1255                            sleep 2
1256                            java -jar AutomaticaEACSL.jar $file $4
1257                            echo "Translation successful. EACSL and QCC files
                                    have been generated, please check them."
1258                        else
1259                            echo "Please provide a correct number for the #tests
                                    parameter of QCC"
1260                        fi
1261                        ;;
1262      *)
1263                        echo "Please select one of the following options:"
1264                        echo "  -qcc | -quickcheckforc  Calls 'AutoQCC' tool.
                                Its only argument is #number_of_tests"
1265                        echo "  -eacsl          Calls 'AutoEACSL' tool. Its
                                argument is the name of the 'source_code' file"
1266                        echo "  -f | -full      Calls both 'AutoQCC' and
                                'AutoEACSL'. It needs both arguments
                                #number_of_tests and 'source_code'"
1267                        echo ""
1268
1269    esac
1270 fi
```