UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

Traducción de contratos software a un lenguaje de especificación estándar

Trabajo Fin de Máster

Máster Universitario en Ingeniería y Tecnología de Sistemas Software

AUTOR/A: Rodríguez José, Luis Carlo

Tutor/a: Villanueva García, Alicia

CURSO ACADÉMICO: 2023/2024

Este Trabajo Fin de Máster se ha depositado en el Departamento de Sistemas Informàticos y Computación de la Universitat Politècnica de València para su defensa.

Trabajo Fin de Máster
Máster Universitario en Ingeniería y Tecnología de Sistemas Software

Título: Traducción de contratos software a un lenguaje de especificación estándar

Julio - 2024

Autor(a): Luis Carlo Rodíguez José
Director(a): Alicia Villanueva García
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

Agradecimientos

Deseo expresar mi más sincero agradecimiento a ValgrAI (Valencian Graduate School and Research Network of Artificial Intelligence) por su generoso apoyo financiero. La contribución de ValgrAI me ha proporcionado la posibilidad de tener a mi disposición todas las herramientas necesarias para completar el Máster Universitario en Ingeniería y Tecnología de Sistemas Software de forma eficiente y efectiva. Este apoyo no solo ha enriquecido mi experiencia educativa, sino que también ha aumentado mi motivación y ha reforzado mi compromiso con la excelencia en este campo de estudio.

Resum

Este treball final de màster està enfocat en la traducció de contractes programari d'un format d'axioma lògic a un llenguatge d'especificació estàndard. Entenem per contractes programari els requisits essencials que s'imposen sobre els arguments i els valors retornats quan s'invoquen funcions en un programa programari. Estos requisits han de complir-se per al correcte funcionament de les funcions, a causa d'això és important verificar els contractes programari per a aconseguir garantir la fiabilitat del programari. D'altra banda els llenguatges d'especificació estàndard són ferramentes formals utilitzades per a definir i documentar requisits, comportaments i propietats dels sistemes de programari de manera precisa i no ambigua. Estos llenguatges permeten descriure el funcionament esperat d'un sistema, facilitant la verificació i validació del programari. Tindre contractes programari en un llenguatge d'especificació estàndard permet utilitzar ferramentes per a realitzar anàlisis i verificacions formals del codi amb l'objectiu de comprovar que les condicions especificades en els contractes de programari es complixen durant l'execució del programa.

Paraules clau: *contractes de programari, llenguatges d'especificació estàndard, ACSL*

Resumen

Este trabajo final de máster está enfocado en la traducción de contratos software de un formato de axioma lógico a un lenguaje de especificación estándar. Entendemos por contratos software los requisitos esenciales que se imponen sobre los argumentos y los valores retornados cuando se invocan funciones en un programa software. Dichos requisitos deben cumplirse para el correcto funcionamiento de las funciones, debido a esto es importante verificar los contratos software para lograr garantizar la fiabilidad del software. Por otra parte los lenguajes de especificación estándar son herramientas formales utilizadas para definir y documentar requisitos, comportamientos y propiedades de los sistemas de software de manera precisa y no ambigua. Estos lenguajes permiten describir el funcionamiento esperado de un sistema, facilitando la verificación y validación del software. Tener contratos software en un lenguaje de especificación estándar permite utilizar herramientas para realizar análisis y verificaciones formales del código con el objetivo de comprobar que las condiciones especificadas en los contratos de software se cumplen durante la ejecución del programa.

Palabras clave: *contratos de software, lenguajes de especificación estándar, ACSL*

Abstract

This master's thesis focuses on the translation of software contracts from a logical axiom format to a standard specification language. Software contracts are understood to be the essential requirements imposed on the arguments and return values when functions in a software program are invoked. These requirements must be met for the functions to operate correctly; therefore, it is important to verify software contracts to ensure software reliability. On the other hand, standard specification languages are formal tools used to define and document requirements, behaviors, and properties of software systems in a precise and unambiguous manner. These languages allow for the expected functionality of a system to be described, facilitating software verification and validation. Having software contracts in a standard specification language enables the use of tools to perform formal analysis and verification of the code to ensure that the conditions specified in the software contracts are met during program execution.

Keywords: *software contracts, standard specification languages, ACSL*

Tabla de contenidos

Índice de figuras

Capítulo 1

Introducción

La verificación de la fiabilidad y corrección del software es un desafío que prevalece en el desarrollo de sistemas de información. Los contratos de software, que definen requisitos esenciales sobre las funciones de los programas, son fundamentales para asegurar que el software funcione según lo esperado. Sin embargo, la verificación de estos contratos de manera manual es una tarea muy costosa y propensa a errores. Tenemos a nuestra disposición los lenguajes de especificación estándar que se destacan por su facilidad de uso para los desarrolladores y analistas, y su habilidad para documentar y verificar los requisitos y comportamientos de los sistemas de software.

1.1. Motivación y Antecedentes

Actualmente se cuenta con una herramienta llamada *KindSpec* [8] desarrollada en *Java* y *K* (para el lenguaje de especificación *KernelC*) que infiere contratos software a partir de programas en el lenguaje *C* o una parte significativa de ellos. Debido a la gran utilidad y diversas aplicaciones de los contratos software es necesario poder razonar con ellos. Para esto se ha optado utilizar herramientas de análisis existentes de las cuales algunas utilizan una notación estándar (lenguajes estándar de especificación).

Una de estas herramientas y en la que este trabajo se enfoca en utilizar es *FramaC* [1] la cual es un marco de trabajo (*framework*) de análisis de código para programas escritos en el lenguaje de programación *C* y proporciona una plataforma modular que permite la integración de diversos análisis estáticos y dinámicos, lo que ayuda a detectar errores, garantizar la corrección del software y mejorar la seguridad y fiabilidad del código *C*. *FramaC* utiliza *ACSL* [2] como lenguaje estándar de especificación para la verificación de programas. *ACSL* permite escribir contratos formales en el código *C* expresados como comentarios con una notación específica.

KindSpec genera contratos en formato de axiomas lógicos que representan las condiciones necesarias para el correcto funcionamiento de las funciones. Sin embargo, para aprovechar al máximo estas inferencias, es crucial traducir estos axiomas a un formato estándar que pueda ser utilizado por herramientas de análisis y verificación existentes.

En este contexto, se busca transformar los contratos inferidos por *KindSpec* en contratos formales utilizando una notación estándar. Esta transformación permiti-

rá emplear herramientas para realizar análisis y verificaciones formales del código. Además, esto proporcionará una validación adicional de los contratos inferidos por *KindSpec*.

1.2. Objetivos

El objetivo de este Trabajo Final de Máster (TFM) se enfoca en traducir los contratos inferidos por *KindSpec*, los cuales están en un formato de axioma lógico, en contratos formales en la notación estándar, específicamente *ACSL*. Al tener los contratos inferidos por *KindSpec* en la notación estándar (*ACSL*) de una herramienta de verificación (*FramaC*) se podría utilizar dicha herramienta para realizar análisis y verificaciones formales del código, facilitando la detección de errores, validación del código para confirmar que cumple con sus especificaciones y por defecto validación de la certitud de los contratos inferidos. Este último punto es especialmente importante ya que *KindSpec* infiere contratos correctos hasta cierto punto, pero en determinados casos hace hipótesis que el usuario debe confirmar y autorizar, esto da entrada a la posibilidad de tener inferencias incorrectas ya que el usuario puede equivocarse.

1.3. Estructura del documento

En los apartados anteriores se ha planteado un resumen del trabajo, la motivación, objetivos y el problema al cual se busca aportar una solución efectiva. En los siguientes capítulos se expone el contexto, desarrollo, resultados y conclusiones sobre el trabajo realizado. De forma resumida, este documento se estructura de la siguiente manera: Capítulo 2 expone los preliminares sobre la herramienta *KindSpec* y la estructura de los contratos inferidos que genera. Capítulo 3 introduce el lenguaje de especificación *ACSL* y explican las características utilizadas en este trabajo. Capítulo 4 detalla la definición de la Librería de Predicados en *ACSL* necesaria para interpretar los predicados de los contratos inferidos por *KindSpec*. Capítulo 5 describe la metodología utilizada para la traducción de los contratos inferidos a la notación estándar *ACSL*. Capítulo 6 plantea la implementación del servicio de traducción desarrollado como solución. Capítulo 7 presenta los resultados obtenidos tras realizar las traducciones con la solución desarrollada. Finalmente, el Capítulo 8 detalla las conclusiones y trabajos futuros.

Capítulo 2

Preliminares

Es esencial tener conocimiento de los conceptos, tecnologías y herramientas subyacentes para comprender plenamente los objetivos y métodos empleados a lo largo del proyecto. En esta sección de preliminares, se presenta el contexto y las bases fundamentales que constituyen el punto de partida de este proyecto y se detallarán los principios teóricos y prácticos que sustentan el desarrollo del proyecto.

2.1. KindSpec

KindSpec es una herramienta automatizada que infiere contratos de software a partir de programas escritos en un fragmento significativo de *C* incluyendo estructuras basadas en punteros, manipulación de *heap* y recursividad. *KindSpec* utiliza *KernelC* como semántica del lenguaje *C* en el marco semántico *K*, de esta forma *KindSpec* aprovecha las capacidades de ejecución simbólica de *K* para explicar axiomáticamente cualquier función del programa. Para lograr esto *KindSpec* utiliza rutinas observadoras en el mismo programa para caracterizar los estados del programa antes y después de la ejecución de una determinada función [8].

La técnica de inferencia de *KindSpec* se basa en el esquema de clasificación desarrollado para las abstracciones de datos, donde una función (método) puede ser un constructor, un modificador o un observador. Un constructor devuelve un nuevo objeto de datos desde cero (es decir, sin tomar ningún objeto como parámetro de entrada); un modificador altera un objeto existente (es decir, cambia el estado de uno o más de sus atributos); y un observador inspecciona el objeto y devuelve un valor que caracteriza uno o más de sus atributos de estado. Dado que el lenguaje *C* no impone encapsulación de datos, no se puede suponer la pureza de ninguna función; por lo tanto, no se asume la premisa tradicional que establece que las funciones observadoras no causan efectos laterales. En otras palabras, cualquier función puede ser potencialmente un modificador, y simplemente se define un observador como cualquier función cuyo tipo de retorno sea diferente de *void* [7].

Los contratos inferidos se expresan en forma de axiomas lógicos que especifican el comportamiento preciso de entrada/salida de las rutinas en *C*, incluyendo tanto axiomas generales para el comportamiento por defecto como axiomas de excepción para especificar el comportamiento en caso de errores [8].

2.2. Contratos Inferidos por KindSpec

Los contratos inferidos por *KindSpec* están definidos siguiendo el concepto de precondiciones y postcondiciones en una rutina basándose en el principio general *design-by-contract* [11]. Debido a la abstracción que utiliza el marco de definición del lenguaje de programación *K*, no se puede garantizar que algunos de los axiomas inferidos son correctos por lo que *KindSpec* categoriza estos axiomas como Axiomas Candidato.

KindSpec obtiene los contratos de la siguiente manera, dado un conjunto de axiomas inferidos ($IA = \{p_1 \Rightarrow q_1, \ldots, p_n \Rightarrow q_n\}$) y un subconjunto de axiomas de excepción ($EA \subseteq IA$) los cuales pertenecen al conjunto de axiomas inferidos, obtenemos un conjunto de axiomas por defecto (DA) el cual contendrá la lista de todos axiomas inferidos exceptuando a los axiomas excepcionales ($DA = (IA - EA)$). El contrato resultante se define como **<*Pre, Post, Loc*>**, donde: *Pre* es la precondición de la función ($\vee p \mid (p \implies q) \in DA$) y representa los datos de entrada admisibles del la función; *Post* es la postcondición de la función (IA) dada por el axioma inferido; y *Loc* es un conjunto de referencias a ubicaciones de memoria cuyo valor podría verse afectado por la ejecución de la función esto puede incluir los parámetros de la función, punteros y campos de estructuras de datos. Por lo que el contrato que obtenemos de *KindSpec* está estructurado en las siguientes secciones: 1) la precondición, 2) las postcondiciones (IA), 3) el conjunto de referencias a ubicaciones de memoria afectadas y 4) una lista de axiomas de excepción (EA) [8].

2.2.1. Estructura de los contratos inferidos

Para poder traducir los contratos inferidos primero debemos entender su estructura la cual está en formato de axioma lógico.

A continuación, se expone un ejemplo del resultado de un contrato inferido.

Este contrato en concreto fue inferido para la función `collapseC` en la Figura 2.2.1, que pertenece al programa `cyclic_lists.c` en la Figura 2.2.1, y completo en el anexo *.1*

```
**********************************************
**********************************************
Inference performed May 26 2020
Selected modifier function: collapseC
From file: examples/cyclic_lists.c
**********************************************
**********************************************

**********************************************
RESULTING INFERRED CONTRACT
**********************************************
-------------------------------------------------
PRECONDITION P:
(lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1) ||
(lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})) ||
(lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=2 ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0)
-------------------------------------------------
-------------------------------------------------
POSTCONDITION Q:
A1: (lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0
    ^ ret=(NPE, {21})) ^
A2: (lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1) => (lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1 ^ ret=0) ^
```

```
A3: (lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})) => (lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})
    ^ ret=0) ^
A4: (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1) ^
A5: (lenC(c)=2 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1) ^
A6: (lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1)
-------------------------------------------------
-------------------------------------------------
LOCATIONS L:
t
n
head
head->next
c->lsize
-------------------------------------------------
-------------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (lenC(c)=?l0 + 2 ^ isN(c)=0 ^ isE(c)=0 ^ ?l0 >= 2) => (lenC(c)=?l0 ^ isN(c)=0 ^ isE(c)=0 ^
     ?l0 >= 2 ^ ret=1) ^
C2: (lenC(c)=?l0 + 2 ^ isN(c)=0 ^ isE(c)=0 ^ ?l0 >= 2) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^
    ret=1)
-------------------------------------------------
```

Contrato función `collapseC`

```
1  #include <stdlib.h>
2
3  struct clist {
4   int lsize;
5   struct lnode *elems;
6  };
7
8  struct lnode {
9   int value;
10  struct lnode *next;
11 };
12
13 int isPrec(struct clist *c, struct lnode *n, struct lnode *t) {
14   . . .
15 }
16
17 int collapseC(struct clist *c) {
18  struct lnode *n;
19  struct lnode *t;
20  struct lnode *head;
21
22  if(c == NULL) return 0;
23  if(c->elems == NULL) return 0;
24
25  t = c->elems;
26  n = c->elems->next;
27
28  while (t != n && !(isPrec(c, n, t))) {
29   t = n;
30   n=n->next;
31  }
32
33  head = n;
34  n = head->next;
35  while(n != head) {
36   t = n;
37   n = n->next;
38   free(t);
39   c->lsize--;
40  }
41
42  head->next = head;
43
44  return 1;
45 }
```

```
46
47 int isN(struct clist *c) {
48  . . .
49 }
50
51 int isE(struct clist *c) {
52  . . .
53 }
54
55 int lenC(struct clist *c) {
56  . . .
57 }
```

Código del programa `cyclic_lists.c`

Como podemos observar en el ejemplo anterior, que el contrato consiste en cuatro elementos principales [10]:

1. `PRECONDITION P`
   Precondición P, que contiene la disyunción de todos los antecedentes de los axiomas inferidos $(\vee p \mid p \Rightarrow q)$
2. `POSTCONDITION Q`
   Postcondición Q, que contiene el conjunto de axiomas inferidos $(p \Rightarrow q)$
3. `LOCATIONS L`
   Ubicaciones L, que contiene las referencias a ubicaciones de memoria (parámetros de función, punteros y campos de estructuras de datos) cuyo valor podría verse afectado por la ejecución de la función
4. `CANDIDATE AXIOMS Q#`
   Axiomas candidatos Q#, que contiene el conjunto de axiomas candidatos

Para la traducción nos podemos enfocar en los elementos de postcondición `Q` y axiomas candidato `Q#` ya que estos incluyen las precondiciones y postcondiciones $(p \Rightarrow q)$ para cada comportamiento. Cada axioma está etiquetado con su propio identificador único, para los axiomas `Q` la letra *A* seguido por el número secuencial (es decir, A1, A2, A3, ...) y para los axiomas `Q#` la letra *C* seguido por un número secuencial (es decir, C1, C2, C3, ...). Estos identificadores serán fundamentales para llevar a cabo la traducción.

Capítulo 3

ACSL: ANSI/ISO C Specification Language

El Lenguaje de Especificación *ANSI/ISO C* (*ACSL*, por sus siglas en ingles) [2] es un lenguaje de especificación de interfaz de comportamiento (*BISL*, por sus siglas en ingles) implementado en el marco de *FramaC*. *FramaC* [1] es un marco de trabajo (*framework*) de análisis de código para programas escritos en el lenguaje de programación *C*. *FramaC* proporciona una plataforma modular que permite la integración de diversos análisis estáticos y dinámicos de código *C*, lo que ayuda a detectar errores, garantizar la corrección del software y mejorar la seguridad y fiabilidad del código *C*. *ACSL* ha sido el lenguaje de especificación que se ha optado por usar debido a que esta funcionalidad que permite a escribir contratos formales y propiedades de una función con un formato específico de anotaciones/comentarios directamente en el código *C*.

3.1. Reglas léxicas

En *ACSL* las especificaciones se dan como anotaciones en comentarios escritos directamente en los archivos fuente de *C*, de modo que los archivos fuente no pierden la habilidad de compilar. Estos comentarios deben comenzar con `/*@` o `//@` y terminar como es habitual en *C*. Por simplicidad, *ACSL* solo describe anotaciones en el estilo usual de comentarios `/*@ ... */`. Las anotaciones de una sola línea en los comentarios `//@` son similares. Sin embargo, es importante notar que dos comentarios consecutivos, independientemente de su estilo, se consideran como dos anotaciones independientes. En particular, no es posible en general dividir una anotación de varias líneas en varios comentarios `//@`, para este caso es obligatorio usar comentarios `/*@ ... */`.

El texto del lenguaje de especificación que se coloca dentro de comentarios especiales de C cuenta con una estructura léxica que sigue principalmente la de *ANSI/ISO C*. A continuación, un ejemplo.

```
1 //@ type point = struct { real x; real y; };
2 //@ type triangle = point[3];
3 //@ logic point origin = { .x = 0.0 , .y = 0.0 };
4 /*@ logic triangle t_iso = { [0] = origin,
5   @                         [1] = { .y = 2.0 , .x = 0.0 }
```

```
6   @                              [2] = { .x = 2.0 , .y = 0.0 }};
7   @*/
```

Se deben notar algunas diferencias [2]:

- El signo de arroba (`@`) es equivalente a un carácter de espacio, excepto cuando indica el comienzo de una anotación *ACSL*.
- Los identificadores pueden comenzar con el carácter de barra invertida (`\`).
- Algunos caracteres *UTF8* pueden usarse en lugar de algunos *construct*.
- Los comentarios pueden colocarse dentro de las anotaciones *ACSL*. Utilizan el formato de *C++*, es decir, comienzan con `//` y se extienden hasta el final de la línea actual. Los comentarios que comienzan con `/*` no pueden anidarse dentro de los comentarios de especificación *ACSL*. Las anotaciones anidadas que comienzan con `//@` se analizan como si el `//@` fuera reemplazado por un espacio en blanco. Una anotación *ACSL* que contiene solo espacios en blanco (después del preprocesamiento) se ignora.
- *ACSL* utiliza algunos elementos gramaticales de *C*, como literales, expresiones de tipo, declaraciones y sentencias. La mayoría de estos elementos se identifican como tales por prefijos *C-* en las figuras que presentan la gramática.

3.2. Expresiones lógicas

El lenguaje de expresiones que se puede usar en las anotaciones se llaman expresiones lógicas. Estas corresponden a expresiones puras de *C*, con *constructs* adicionales. La gramática para los *constructs* básicos de las expresiones lógicas distingue entre predicados y términos, siguiendo la distinción habitual entre proposiciones y términos en la lógica clásica de primer orden.

ACSL utiliza estas expresiones lógicas para los contratos formales de especificación debido a su precisión y rigor formal, lo que elimina ambigüedades y asegura una descripción exacta del comportamiento del software. Estas expresiones lógicas son interpretables por herramientas de verificación automática lo que permite una comprobación eficaz de las condiciones antes y después de la ejecución de una función. Debido a que los contratos de *ACSL* utilizan la lógica formal se facilita la documentación clara y la definición de condiciones complejas, lo que mejora la comprensión y mantenimiento del código.

En *ACSL* existe una distinción entre booleanos y predicados. La expresión `x < y` en posición de término es un booleano, y la misma expresión también está permitida en posición de predicado. A diferencia de *C*, en *ACSL* existe una distinción entre booleanos y enteros. Hay una conversión implícita de enteros a booleanos, por lo que se puede escribir `x && y` en lugar de `x != 0 && y != 0`, aunque si nos quedamos con la segunda expresión tendríamos una lógica más explicita. Si se necesita realizar la conversión inversa, se requiere un *cast* explícito, por ejemplo, `(int)(x > 0) + 1`, donde `\false` se convierte en `0` y `\true` se convierte en `1`.

ACSL cuenta con *constructs* adicionales, uno de ellos es la cuantificación. En *ACSL* La cuantificación universal se denota por `\forall` y la cuantificación existencial por `\exists`. La cuantificación se puede hacer sobre cualquier tipo (tipos lógicos y tipos

C). La cuantificación sobre punteros debe usarse con cuidado, ya que depende del estado de la memoria donde se realiza la desreferenciación [2].

3.3. Contratos simples en ACSL

Los contratos simples en *ACSL* se utilizan para especificar de manera precisa y no ambigua el comportamiento esperado de una función. Un contrato simple se compone de precondiciones, efectos laterales y postcondiciones.

La semántica de los contratos simples de funciones en *ACSL* que solo contienen cláusulas simples, se define de la siguiente manera:

```
1 /*@ requires P1; requires P2; ...
2   @ assigns L1; assigns L2; ...
3   @ ensures E1; ensures E2; ...
4   @*/
```

Las cláusulas `requires`, `assigns`, y `ensures` se utilizan para definir las precondiciones, efectos laterales y postcondiciones respectivamente en los contratos formales de funciones y proporcionan una manera precisa para describir las funciones en *C*. A continuación, se detalla cada una de las cláusulas.

requires. Se utiliza para especificar las precondiciones de una función. Las precondiciones deben cumplirse antes de que la función se ejecute. Si las precondiciones no se cumplen cuando la función es invocada el comportamiento correcto de la función no está garantizado.

Por ejemplo, consideremos la siguiente función `foo` que recibe como parámetro `x`, el valor de `x` debe ser mayor que `0` para que garantizar el funcionamiento correcto de `foo`. Esto se expresaría de la siguiente manera en *ACSL* utilizando la cláusula `requires`:

```
1 /*@ requires x > 0;
2   @*/
3 void foo(int x);
```

assigns. Se utiliza para especificar los efectos laterales de una función, en otras palabras define las ubicaciones de memoria que la función puede modificar durante su ejecución. Se incluyen las variables globales, parámetros de entrada y cualquier otra ubicación de memoria accesible por la función. Si la función no realiza la modificación de alguna ubicación de memoria se puede utilizar el "*built-in construct*"de *ACSL* `\nothing`, de tal forma que tendríamos la cláusula `assigns \nothing` la cual es equivalente a `assigns \empty`.

Por ejemplo, consideremos que la siguiente función `foo`, supongamos que esta función puede modificar la memoria a la que apunta `pntr`. Esto se expresaría de la siguiente manera en *ACSL* utilizando la cláusula `assigns`:

```
1 /*@ assigns *pntr;
2   @*/
3 void foo(int *pntr);
```

ensures. Se utiliza para especificar las postcondiciones de una función. Las postcondiciones son las condiciones que deben ser verdaderas después de que la función haya terminado de ejecutarse. Las postcondiciones dependen tanto de las precondiciones como de los cambios realizados durante la ejecución de la función. Cabe mencionar que una cláusula `ensures` no implica que la función necesariamente retornará algo.

Por ejemplo, consideremos la siguiente función `increment`, supongamos que la postcondición de esta función será que el resultado de la función (`\result`) será igual a `x + 1` después que la función haya terminado de ejecutarse. Esto se expresaría de la siguiente manera en *ACSL* utilizando la cláusula `ensures`:

```
1 /*@ ensures \result == x + 1;
2   @*/
3 int increment(int x);
```

Las clausulas se pueden agrupar en conjunciones unidas por el operador lógico `AND` de *ACSL* que es equivalente al operador del lenguaje *C*, `&&`.

Por ejemplo, el siguiente contrato simple:

```
1 /*@ requires P1;
2   @ requires P2;
3   @ requires P3;
4   @ assigns L1;
5   @ assigns L2;
6   @ assigns L3;
7   @ ensures E;
8   @*/
```

Es equivalente a:

```
1 /*@ requires P1 && P2 && P3;
2   @ assigns L1 && L2 && L3;
3   @ ensures E;
4   @*/
```

A continuación, un ejemplo completo de un contrato simple de una función simple:

```
1 /*@ requires x >= 0;
2   @ assigns \nothing;
3   @ ensures \result == x + 1;
4   @*/
5 int increment(int x) {
6     return x + 1;
7 }
```

El contrato se describe de la siguiente forma, la precondición está especificada por `requires x >= 0`, en esta cláusula el valor de `x` debe ser mayor o igual a `0` antes de que la función `increment` sea invocada. Los efectos secundarios se denotan en la cláusula `assigns \nothing`, la cual asegura que la función no modifica ninguna ubicación de memoria. Por último, la postcondición está especificada en la cláusula `ensures \result == x + 1`, la cual garantiza que el resultado de la función es igual al valor de `x` incrementado en `1`.

Los contratos simples en *ACSL* son fundamentales para definir las condiciones de entrada, los efectos secundarios y las condiciones de salida de las funciones de manera precisa y verificable. Esta formalización ayuda a mejorar la fiabilidad y la seguridad del software, facilitando tanto su desarrollo como su mantenimiento.

3.4. Contratos con Comportamientos Definidos en ACSL

Los contratos con comportamientos definidos en *ACSL* permiten especificar diferentes comportamientos de una función específica bajo distintas posibles condiciones. Esto se logra mediante la definición de varias cláusulas `behavior` (comportamientos) dentro de un único contrato. Cada cláusula `behavior` tiene sus propias precondiciones, efectos secundarios y postcondiciones.

Los contratos con comportamientos definidos se utilizan cuando una función puede tener distintos modos de operación o respuestas dependiendo de las condiciones de entrada. Los nombres de los comportamientos deben ser distintos (únicos) en el contrato de la función (o declaración) dada ya que funcionan como un identificador único. La estructura de los contratos con comportamientos definidos en *ACSL* se define de la siguiente manera:

```
/*@ requires P;
  @ behaviour b1:
  @   assumes A1;
  @   requires R1;
  @   assigns L1;
  @   ensures E1;
  @ behaviour b2:
  @   assumes A2;
  @   requires R2;
  @   assigns L2;
  @   ensures E2;
  @*/
```

A continuación se describe cada una de las cláusulas de los contratos con comportamientos definidos.

requires. Al igual que los contratos simples, `requires` se utiliza para especificar una precondición, que debe cumplirse antes de que se invoca la función. En los comportamientos definidos esta cláusula puede ser común para todos los `behaviour` o también puede ser definida a nivel de comportamiento. Si se utiliza de forma global se define al principio antes de la primera declaración de comportamiento y si se utiliza a nivel de comportamiento se define dentro de los `behaviour`.

behavior. Es el inicio del comportamiento y lo define con un nombre único. Por ejemplo, en el caso de la cláusula `behaviour b1`, esta define el comportamiento con el nombre `b1`, por otra parte, `behavior b2` define otro comportamiento distinto llamado b2, cada una de estas definiciones cuenta con sus propias precondiciones, efectos laterales y postcondiciones.

assumes. Se utiliza para especificar una condición adicional que sirve para refinar aún más las precondiciones introduciendo suposiciones adicionales que deben cumplirse para que un comportamiento específico se aplique.

assigns. Se utiliza para especificar las ubicaciones de memoria que se pueden modificar y al igual que la cláusula `requires` se pueden especificar globalmente antes de

la primera declaración de comportamiento o dentro de los comportamientos.

ensures. Se utiliza para especificar las postcondiciones específicas para el comportamiento.

Los contratos con comportamientos definidos se puede utilizar incluso cuando solo hay un único comportamiento. Por ejemplo, el siguiente contrato simple:

```
1 /*@ requires P; assigns L; ensures E; */
```

Es equivalente a un único comportamiento definido:

```
1 /*@ requires P;
2   @ behavior <unique name>:
3   @   assumes \true;
4   @   assigns L;
5   @   ensures E;
6   @*/
```

Una de las prácticas recomendadas en *ACSL* para la especificación de contratos de funciones es proporcionar contratos completos y claros para facilitar la verificación del software. Esto se debe a que contratos incompletos o ausentes pueden llevar a suposiciones predeterminadas que dificultan la verificación precisa del comportamiento del programa por medio de las herramientas de verificación de código. Una función puede no tener contrato o puede tener un contrato con sin ciertas cláusulas definidas. Expandiendo un poco sobre esto último, si un contrato no tiene definidas las cláusulas `requires` y `ensures` por defecto se consideran como `\true`. Si no se proporciona una cláusula `assigns`, esta permanece sin especificar. Si una función del programa solo está declarada pero no tiene un cuerpo, entonces significa que potencialmente modifica "todo", por lo tanto, al validar esta función será imposible verificar cualquier cosa sobre los programas que llamen a esa función. Por ende, puede considerarse como obligatorio darle un contrato a esa función. Alternativamente, si esa función tiene un cuerpo el no dar una cláusula `assigns` en el contrato significa que se le atribuye a las herramientas de verificación el trabajo de computar una sobre-aproximación de los conjuntos de ubicaciones modificadas.

A continuación, se expone un ejemplo de una función `division` que tiene dos comportamientos definidos: uno para cuando la división es exacta (`exact`) y otro para cuando hay un resto (`with_remainder`).

```
1  /*@ requires y != 0;
2    @ behavior exact:
3    @   assumes x % y == 0;
4    @   ensures \result == x / y;
5    @   assigns \nothing;
6    @ behavior with_remainder:
7    @    assumes x % y != 0;
8    @    ensures \result == x / y;
9    @    assigns \nothing;
10   @*/
11 int division(int x, int y) {
12     return x / y;
13 }
```

En este ejemplo, el contrato cuenta con una precondición global `requires y != 0`, está definida justo al principio y asegura que la variable `y` no sea `0` antes de eje-

cutar la función `division`. Si la variable `y` es `0`, la división no está definida. El contrato tiene dos comportamientos definidos, el primero es la definición del comportamiento cuando la división es exacta, nombrado `exact`, contiene tres clausulas: 1) `assumes x % y == 0`, se cumple cuando `x` es divisible por la variable `y` sin resto; 2) `ensures \result == x / y`, asegura que el resultado de la función es la división exacta; 3) `assigns \nothing`, especifica que este comportamiento no modifica ninguna ubicación de memoria. El segundo es la definición del comportamiento cuando después de la división hay un resto, nombrado `with_remainder`, también contiene tres cláusulas: 1) `assumes x % y != 0`, este comportamiento se cumple cuando la variable `x` no es divisible por la variable `y` sin resto; `ensures \result == x / y`, asegura que el resultado de la función es la parte entera de la división; `assigns \nothing`, especifica que este comportamiento no modifica ninguna ubicación de memoria.

3.5. Declaración de predicados y funciones

ACSL permite la declaración de nuevas funciones y predicados las cuales se definen con expresiones lógicas explicitas. Estas expresiones se especifican después del signo igual =. Esta funcionalidad permite ampliar el lenguaje de expresiones lógicas utilizado en las anotaciones mediante declaraciones de nuevos tipos lógicos, así como de nuevas constantes, funciones lógicas y predicados. Poder definir nuevas funciones y predicados permite a los usuarios expresar propiedades complejas y específicas que no están cubiertas por las construcciones lógicas estándar. Esto es especialmente útil en la verificación formal y en la especificación de propiedades del software, ya que permite una mayor precisión y claridad en la descripción del comportamiento esperado de los programas.

Tomemos como ejemplo el siguiente código donde se define un nuevo predicado `is_positive` con un parámetro de tipo entero y una función lógica `get_sign` con un parámetro de tipo real y retorna un entero.

```
1 //@ predicate is_positive(integer x) = x > 0;
2 /*@ logic integer get_sign(real x) =
3   @   x > 0.0 ? 1 : ( x < 0.0 ? -1 : 0);
4   @*/
```

En *ACSL* también se pueden definir *lemmas* los cuales son afirmaciones adicionales que los usuarios pueden definir para ayudar el proceso de verificación de programas. Herramientas como demostradores de teoremas pueden utilizar estos lemas para demostrar que las especificaciones de *ACSL* se cumplen.

Por ejemplo, en una función que calcula el máximo común divisor de dos números un lema podría ser la propiedad matemática conocida sobre el máximo común divisor que puede usarse para ayudar a probar que la implementación de la función es correcta. Podríamos definir que el máximo común divisor es conmutativo:

```
1 /*@ lemma gcd_commutative: \forall int x, y; gcd(x, y) == gcd(y, x);
2   @*/
```

Este lema podría ayudar a un demostrador de teoremas a verificar si la función que calcula el máximo común divisor es correcta.

3.6. Características experimentales utilizadas

ACSL en su *versión 1.20* contiene ciertas características que se consideran como *experimentales* lo cual significa que la sintaxis y semántica no son estables aún por lo que es posible que las herramientas de razonamiento esperen para incorporarlas a su versión final. De estas características experimentales este trabajo utiliza la declaración de *tipos lógicos concretos* y *módulos de especificación*.

Tipos lógicos concretos
Los tipos lógicos concretos son tipos de datos que no solo pueden declararse, sino que también pueden definirse con estructuras específicas. Estos tipos lógicos permiten representar estructuras de datos más complejas y detalladas dentro de las especificaciones lógicas. A continuación los diferentes tipos lógicos concretos y sus características.

Tipos de Registro (Record Types)
Los *Record Types* son un tipo de registro en *ACSL* que se definen como una colección de campos, cada uno con un nombre y un tipo. Para acceder a los campos de un registro se utiliza la notación `p.x` donde `p` es una instancia del registro y `x` es el nombre del campo.
Ejemplo:

```
1 /*@ type point = { integer x; integer y; };
2   @*/
3 /*@ logic integer get_x(point p) = p.x;
4   @*/
```

Tipos de Sumativos / Enumerados (Sum Types)
Los *Sum Types* son un tipo de suma en *ACSL* que representan un dato que puede ser una de varias alternativas etiquetadas, similar a una enumeración extendida con tipos asociados. Para trabajar con los tipos de sumativos, se utiliza una construcción de coincidencia de patrones (`match`) que permiten definir diferentes comportamientos para cada posible variante del tipo de suma.
Ejemplo:

```
1 /*@   type shape = Circle(integer radius) | Rectangle(integer width, integer height);
2   @*/
3 /*@  logic integer area(shape s) = \match s {
4   @      case Circle(r): \pi * r * r;
5   @      case Rectangle(w, h): w * h;
6   @  };
7   @*/
```

Tipos Producto (Product Types)
Los *Product Types* son un tipo de producto en *ACSL* y es similar a una tupla, donde se agrupan varios valores, cada uno con su tipo.
Ejemplo

```
1 /*@ type pair = { integer fst; integer snd; };
2   @*/
3 /*@ logic pair make_pair(integer x, integer y) = { x, y };
4   @*/
```

Tipos Funcionales (Function Types) Los *Function Types* en *ACSL* se utilizan para

declarar funciones lógicas que pueden tomar argumentos y devolver valores. Se declaran con `logic` seguido por el tipo del resultado, la lógica está definida después del signo de igual ==
Ejemplo:

```
/*@ logic integer factorial(integer n) =
  @   (n == 0) ? 1 : n * factorial(n - 1);
  @*/
```

Estas funciones también pueden ser recursivas.
Ejemplo:

```
/*@ type list = Nil | Cons(integer head, list tail);
  @*/
/*@ logic integer length(list l) = \match l {
  @   case Nil: 0;
  @   case Cons(h, t): 1 + length(t);
  @ };
  @*/
```

En el ejemplo anterior, la primera declaración define un tipo algebraico `list` (lista) en *ACSL*. Un tipo algebraico es una forma de definir tipos compuestos mediante la combinación de otros tipos. Aquí, `list` puede ser uno de dos constructores una lista `list` puede ser una lista vacía (`Nil`) o una lista construida por un entero seguido de otra lista (`Cons`). La segunda declaración es la función lógica `length` que calcula la longitud de una lista `list l`. La función está definida usando un patrón de coincidencia (`\match`), que permite descomponer y examinar las diferentes formas que puede tomar la lista `l`. Si la lista `l` es `Nil` (es decir, una lista vacía), la longitud es `0`, si la lista `l` es `Cons(h, t)` (es decir, tiene un elemento `h` seguido por una sublista `t`), la longitud es `1` más la longitud de la sublista `t`. Esta es una definición recursiva, donde la función `length` se llama a sí misma con la sublista `t` hasta llegar al caso base `Nil`.

3.6.1. Módulos de especificación

Los módulos de especificación se pueden ser paramétricos y se utilizan para encapsular varias definiciones lógicas en un solo bloque. Esto facilita la organización y reutilización de las especificaciones lógicas a lo largo de diferentes partes de un programa.

Un ejemplo de los módulos de especificación dado en la documentación de *ACSL* [2] es el siguiente.

```
/*@ module List {
  @
  @   type list<A> = Nil | Cons(A , list<A>);
  @
  @   logic integer length<A>(list<A> l) =
  @     \match l {
  @       case Nil : 0
  @       case Cons(h,t) : 1+length(t) } ;
  @
  @   logic A fold_right<A,B>((A -> B -> B) f, list<A> l, B acc) =
  @     \match l {
  @       case Nil : acc
  @       case Cons(h,t) : f(h,fold_right(f,t,acc)) } ;
```

```
14  @
15  @   logic list<A> filter<A>((A -> boolean) f, list<A> l) =
16  @     fold_right((\lambda A x, list<A> acc;
17  @       f(x) ? Cons(x,acc) : acc), Nil) ;
18  @
19  @ }
20  @*/
```

El módulo de este ejemplo se puede proveer a un programa como una librería de especificaciones algebraicas predefinidas guardadas en un fichero `List.acsl` e importándolas de la siguiente manera en el código:

```
1 //@ import List;
```

Adicionalmente para poder acceder a los componentes del módulo se debe utilizar la notación `List::length`.

Capítulo 4

Definición de la Librería de Predicados en notación ACSL

Los contratos inferidos por *KindSpec* definen en sus precondiciones y postcondiciones predicados a los cuales se les da una interpretación basada en la semántica *KernelC* durante el proceso de inferencia. En los casos en los que *KindSpec* recurre a la interpretación abstracta [8] para generar los axiomas candidatos, se recurre a una semántica abstracta definida específicamente, es por esto que se obtienen axiomas candidatos que necesitan confirmación. Además de estos predicados, en *KindSpec* se han añadido las excepciones más comunes del lenguaje de programación *C*. Los comportamientos indeseables o erróneos más comunes que pueden ocurrir al ejecutar un programa han sido catalogados con un código de error único, como se muestra en la Figura 4.1. Utilizando este conjunto de errores los valores de retorno de error de las funciones tienen la forma `(errorCode, {pc})`, donde el contador de programa `pc` identifica la declaración precisa del fragmento de código en el programa que causó el error [8].

Cuando se quieran utilizar las herramientas de análisis para verificar los contratos que obtendremos en la notación de *ACSL*, se debe partir con la asunción de que contamos con una interpretación correcta en la lógica de los predicados que utilizamos en el contrato. Debido a esto es crucial disponer de una librería con la interpretación de estos predicados. Como ya hemos visto, *ACSL* permite definir predicados, tipos de datos concretos y lemas en un módulo de especificación e importarlos como librería en el programa donde estará el contrato. Para cada predicado debemos definir un cuerpo y una semántica con expresiones lógicas. Además, para los códigos de error comunes que se encuentran en los axiomas del contrato como `(NPE, {45})`, se debe dar una interpretación equivalente en *ACSL*.

4.1. Identificación de los Predicados

KindSpec cuenta con un conjunto de programas de prueba de rendimiento predefinidos, expuestos en el anexo .3, los cuales se aprovecharán para este trabajo. Utilizaremos los contratos inferidos de cada programa para identificar los predicados a interpretar, expuestos en el anexo .4.

Al analizar el código de cada uno de estos programas, podemos observar que están

Error code	Exception	Description
`NPE`	*Null Pointer Error*	Null dereferencing
`DBZ`	*Division By Zero*	Division of any number by 0
`VVA`	*Void Value Access*	Access to a non-pointer value of type `void`
`NMS`	*Non-valid Malloc Size*	Calling `malloc` with a negative or zero object size
`NOD`	*Null Object Destruction*	Calling `free` over a `null` reference
`UMA`	*Undefined Memory Access*	Access to an undefined memory segment (e.g., immediately after a pointer declaration)
`OOS`	*Out Of Scope*	Access to a variable that is out of scope
`IRT`	*Incorrect Return Type*	Type of return value does not match the function profile
`IAT`	*Incompatible Assign Types*	Mismatch between type of variable and assigned value
`NEF`	*Non-Existing Function*	The called function is not defined or declared
`UAC`	*Unsuitable Call Arguments*	Function call does not match the function profile

Figura 4.1: Excepciones más comunes en *C*

relacionados hasta cierto punto y están diseñados para la gestión de listas en *C*. Se definen diferentes tipos de listas, como listas simples, listas cíclicas y listas de conjuntos, junto con sus respectivas funciones de validación y manipulación. Estos programas proporcionan una base para comprender el objetivo de los predicados de los contratos inferidos y nos ayudaran a dar una definición robusta de los predicados en los módulos *ACSL*. Otro punto a destacar es que todos los predicados reciben un puntero como parámetro, por lo que debemos tomar esto en consideración para la definición. Además, todos los predicados retornarán valores enteros aunque se utilicen para validaciones booleanas lógicas, se utilizará el `0` para representar `\false` y el `1` para representar `\true`.

A continuación, se detallan los diferentes componentes de los distintos programas ejemplo:

- `listNode`

```
1 struct listNode
2 {
3     int val; // value of the node
4     struct listNode *next; // pointer to the next node
5 };
```

  - Observadores:
    - isNull: para validar si el nodo es nulo
    - length: para obtener el tamaño de la lista
    - firstElem: para obtener el primer nodo de la lista
    - lastElem: para obtener el último nodo de la lista

- `list`

```
1 struct list
2 {
3     struct listNode *elems; // pointer to the first node
4 };
```

  - Observadores:
    - isNull: para validar si la lista es nula
    - length: para obtener el tamaño de la lista
    - firstE: para obtener el primer elemento de la lista
    - lastE: para obtener el último elemento de la lista

- `cycList`

```
1 struct cycList
2 {
3     int lsize; // size of the list
4     struct listNode *elems; // pointer to the first node
5 };
```

  - Observadores:
    - isN: para validar si la lista cíclica es nula
    - isE: para validar si la lista cíclica está vacía, retorna `NPE` si la lista es nula
    - isEmpty: para validar si la lista cíclica está vacía, retorna `0` si la lista es nula
    - lenC: para obtener el tamaño de la lista cíclica, retorna `NPE` si la lista está vacía o es nula
    - lengthCircular: obtener el tamaño de la lista cíclica, retorna `0` si la lista está vacía o es nula

- `setList`

```
1 struct setList
2 {
3     int capacity; // maximum number of elements
4     int lsize; // number of elements in the set
5     struct lnode *elems; // pointer to the first node
6 };
```

  - Observadores:
    - isNull: para validar si la lista es nula
    - isEmpty: para validar si la lista está vacía, retorna `NPE` si la lista es nula
    - isFull: para validar si la lista está llena, retorna `NPE` si la lista es nula
    - contains: para validar si un nodo está contenido en una lista
    - length: para obtener el tamaño de la lista

4.2. Definición de la lógica de los Predicados

Antes de iniciar a definir la lógica de los predicados, debemos definir los códigos de errores. Para identificar y gestionar de manera efectiva los errores en los predicados del módulo, se optó por utilizar números negativos para cada código de error. Se ha optado definirlos de esta manera debido a que los predicados en que se definirán serán expresiones booleanas que retornarán un valor entero, `0` para representar `\false` y `1` para representar `\true` . Al ser los valores de error un entero negativo se facilitará la diferenciación clara entre valores de error y valores válidos, minimizando las confusiones durante la ejecución del programa. De esta manera, se asegura una respuesta coherente y predecible de los predicados ante situaciones de error.

Iniciaremos la definición del módulo con la declaración de estos códigos de errores.

```
/*  @ module Pred {
    @ // Error codes for known exceptions
    @   // ----- Error codes -------------------------
    @   type errorCode = int;
    @   logic errorCode NPE = -1; // Null Pointer Error
    @   logic errorCode DBZ = -2; // Division By Zero
    @   logic errorCode VVA = -3; // Void Value Access
    @   logic errorCode NMS = -4; // Non-valid Malloc Size
    @   logic errorCode NOD = -5; // Null Object Destruction
    @   logic errorCode UMA = -6; // Undefined Memory Access
    @   logic errorCode OOS = -7; // Out Of Scope
    @   logic errorCode IRT = -8; // Incorrect Return Type
    @   logic errorCode IAT = -9; // Incompatible Assign Types
    @   logic errorCode NEF = -10; // Non-Existing Function
    @   logic errorCode UAC = -11; // Unsuitable Call Arguments
    @ }
    @*/
```

Para la definición de los predicados iremos definiéndolos acumulativamente por cada tipo de lista.

Iniciaremos por el tipo *listNode*
Tal y como hemos identificado, el tipo `listNode` cuenta con su definición y sus observadores. Primero debemos definir el tipo `listNode` en la librería de la siguiente forma:

```
    @   // ----- List Nodes -------------------------
    @   type listNode = struct {
    @      int val; // Value of the node
    @      struct listNode *next; // Pointer to the next node
    @   };
```

Esta definición crea una estructura llamada `listNode` que representa un nodo en una lista enlazada. Cuenta con los componentes `int val`, un entero que almacena el valor del nodo y `struct listNode *next`, un puntero que apunta al siguiente nodo en la lista.

Luego se definen los predicados`isNull`, `length`, `firstElem`, `lastElem` del tipo `listNode`. A continuación, se dará una explicación de la lógica definida para cada uno de los predicados definidos. Todos los predicados reciben como parámetro un puntero `lst` de tipo `listNode`.

- `isNull`

- Este predicado verifica si un nodo de la lista es nulo. Si el puntero `lst` es nulo (`\null`), devuelve `1`, indicando que el nodo es nulo. De lo contrario, devuelve `0`, indicando que el nodo no es nulo.

```
1       @   logic integer isNull(listNode *lst) =
2       @       (lst == \null) ? 1 : 0;
```

- `length`
  - Este predicado calcula la longitud de la lista a partir de un nodo dado. Si el nodo `lst` es nulo (`\null`), devuelve `0`, indicando que no hay más nodos en la lista. De lo contrario, devuelve `1` más la longitud de la lista a partir del siguiente nodo (`l->next`), calculando así recursivamente la longitud total de la lista.

```
1       @   logic integer length(listNode *lst) =
2       @       (lst == \null) ? 0 : 1 + length(lst->next);
```

- `firstElem`
  - Este predicado obtiene el valor del primer elemento de la lista. Si el nodo `lst` es nulo (`isNull(l) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, devuelve el valor del nodo `lst` (`l->val`).

```
1       @   logic integer firstElem(listNode *lst) =
2       @       (isNull(lst) == 1) ? NPE : lst->val;
```

- `lastElem`
  - Este predicado obtiene el valor del último elemento de la lista. Si el nodo `lst` es nulo (`isNull(l) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. Si el siguiente nodo (`l->next`) es nulo (`isNull(l->next) == 1`), devuelve el valor del nodo actual (`l->val`), indicando que este es el último nodo. De lo contrario, llama recursivamente a `lastElem` con el siguiente nodo (`l->next`), continuando así hasta encontrar el último nodo de la lista.

```
1       @   logic integer lastElem(listNode *lst) =
2       @       (isNull(lst) == 1) ? NPE : (isNull(lst->next) == 1) ? lst->val :
            lastElem(lst->next);
```

Continuamos con el tipo *list*

Tal y como hemos identificado, el tipo `list` cuenta con su definición y sus observadores. Se define el tipo `list` en la librería de la siguiente manera:

```
1   @   // ----- Lists ------------------------------
2   @   type list = struct {
3   @       listNode *elems; // Pointer to the first element of the list
4   @   };
```

Esta definición crea una estructura llamada `list` que representa una lista enlazada.

Cuenta con el componente `listNode *elems`, un puntero al primer elemento (nodo) de la lista.

Definición de los predicados `isNull`, `length`, `firstE`, `lastE` del tipo `list`. A continuación, se dará una explicación de la lógica definida para cada uno de los predicados definidos. Todos los predicados reciben como parámetro un puntero `lst` de tipo `list`.

- `isNull`
  - Este predicado verifica si una lista es nula. Si el puntero `lst` es nulo (`\null`) o el primer elemento de la lista es nulo (`isNull(l->elems) == 1`), devuelve `1`, indicando que la lista es nula. De lo contrario, devuelve `0`, indicando que la lista no es nula.

```
1       @   logic integer isNull(list *l) =
2       @       (l == \null || isNull(l->elems) == 1) ? 1 : 0;
```

- `length`
  - Este predicado calcula la longitud de la lista. Si la lista `lst` es nula (`isNull(l) == 1`), devuelve `0`, indicando que la lista no tiene elementos. De lo contrario, devuelve la longitud de los nodos de la lista (`length(l->elems)`), utilizando el predicado `length` definido para `listNode`.

```
1       @   logic integer length(list *l) =
2       @       (isNull(l) == 1) ? 0 : length(l->elems);
```

- `firstE`
  - Este predicado obtiene el valor del primer elemento de la lista. Si la lista `lst` es nula (`isNull(l) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, devuelve el valor del primer nodo de la lista (`firstElem(l->elems)`), utilizando el predicado `firstElem` definido para `listNode`.

```
1       @   logic integer firstE(list *l) =
2       @       (isNull(l) == 1) ? NPE : firstElem(l->elems);
```

- `lastE`
  - Este predicado obtiene el valor del último elemento de la lista. Si la lista `lst` es nula (`isNull(l) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, devuelve el valor del último nodo de la lista (`lastElem(l->elems)`), utilizando el predicado `lastElem` definido para `listNode`.

```
1       @   logic integer lastE(list *l) =
2       @       (isNull(l) == 1) ? NPE : lastElem(l->elems);
```

Continuamos con el tipo *cycList*
Tal y como hemos identificado, el tipo `cycList` cuenta con su definición y sus observadores. Se define el tipo `cycList` en la librería de la siguiente manera:

```
1     @   // ----- Cyclic Lists ------------------------
2     @   type cycList = struct {
3     @       int lsize; // Size of the list
4     @       listNode *elems; // Pointer to the first element of the list
5     @   };
```

Esta definición crea una estructura llamada `cycList` que representa una lista cíclica. Cuenta con los componentes `int lsize`, un entero que almacena el tamaño de la lista y `listNode *elems`, un puntero al primer elemento (nodo) de la lista.

Además de los predicados identificados `isN`, `isE`, `isEmpty`, `lenC`, `lengthCircular` se debe definir un predicado más que servirá de *helper* del predicado `lenC`.

Definición de los predicados `isN`, `isE`, `isEmpty`, `lenC`, `lengthCircular` del tipo `cycList`. A continuación, se dará una explicación de la lógica definida para cada uno de los predicados definidos. Todos los predicados reciben como parámetro un puntero `c` de tipo `cycList`, y el predicado *helper* `isPrec` recibe también dos punteros `n` y `t` de tipo `listNode`.

- `isN`
    - Este predicado verifica si una lista cíclica es nula. Si el puntero `c` es nulo (`\null`), devuelve `1`, indicando que la lista cíclica es nula. De lo contrario, devuelve `0`, indicando que la lista cíclica no es nula.

```
1         @   logic integer isN(cycList *c) =
2         @       (c == \null) ? 1 : 0;
```

- `isE`
    - Este predicado verifica si una lista cíclica es vacía. Si la lista cíclica `c` es nula (`isN(c) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, devuelve el resultado de `isNull(c->elems)`, utilizando el predicado `isNull` definido para `listNode` para verificar si el primer nodo es nulo.

```
1         @   logic integer isE(cycList *c) =
2         @       (isN(c) == 1) ? NPE : isNull(c->elems);
```

- `isEmpty`
    - Este predicado verifica si una lista cíclica está vacía. Si la lista cíclica `c` es nula (`isN(c) == 1`), devuelve `0`, indicando que no está vacía. De lo contrario, devuelve el resultado de `isNull(c->elems)`, verificando si el primer nodo es nulo.

```
1         @   logic integer isEmpty(cycList *c) =
2         @       (isN(c) == 1) ? 0 : isNull(c->elems);
```

- `isPrec`
    - Este predicado verifica si un nodo `n` precede a un nodo `t` en una lista cíclica. Utiliza una variable auxiliar `p` y la cabeza de la lista `head`. Comprueba que

head y t no sean nulos y recorre la lista desde `p` hasta `t` para verificar si `n` precede a `t`.

```
1       @   // Helper predicate to check if a node is in the cycle before another node
2       @   //  checks if node n precedes node t in the circular list c.
3       @   logic integer isPrec(cycList *c, listNode *n, listNode *t) =
4       @       \exists listNode *p; \let head = c->elems; isNull(head) == 0 && isNull(t
            ) == 0 &&
5       @       (\forall listNode *p; (isNull(p) == 0 && p != t); p = p->next; p != head
             && p != n);
```

- `lenC`
  - Este predicado calcula la longitud de una lista cíclica. Verifica si la lista `c` es nula o vacía, de ser así retorna el código de error definido `NPE`. De lo contrario, utiliza variables auxiliares `t`, `n` y `head` para recorrer la lista y contar los nodos hasta volver al nodo `head`.

```
1       @   logic integer lenC(cycList *c) =
2       @       (isN(c) == 1 || isE(c) == 1) ? NPE :
3       @       \let t = c->elems, n = c->elems->next;
4       @       \let head = (\exists listNode *p; t != n && isNull(n) == 0 && isPrec(c,
            n, t) == 0; n = n->next, t = n);
5       @       (isNull(head) == 1) ? NPE :
6       @       (n = head->next, \let counter = 1; \forall listNode* p; p != head &&
            isNull(p) == 0; p = n->next; counter++);
7       @       counter;
```

- `lengthCircular`
  - Este predicado calcula la longitud de una lista cíclica. Verifica si la lista `c` es nula o vacía, de ser así retorna `0`. De lo contrario, utiliza variables auxiliares `h` y `n` para iniciar el recorrido de la lista contando los nodos hasta volver al nodo `h`, o devuelve -1 si encuentra un nodo nulo durante el recorrido.

```
1       @   logic integer lengthCircular(cycList *c) =
2       @       (isN(c) == 1 || isE(c) == 1) ? 0 :
3       @       \let h = c->elems, n = h->next;
4       @       \let counter = 1;
5       @       \let result = (n == h) ? counter :
6       @           \exists listNode *p; \let current = n;
7       @           \forall listNode *p; (current != h && isNull(current) == 0);
8       @               current = current->next;
9       @               counter++;
10      @           (isNull(current) == 1) ? -1 : counter;
11      @       result;
```

Las funciones `isPrec`, `lenC`, y `lengthCircular` en particular fueron las más complejas de definir, por lo que es importante analizar la estructura y el propósito de cada una, así como el uso de las funcionalidades de *ACSL* utilizadas para la definición. A continuación se explica a detalle cada una de ellas.

isPrec
La función `isPrec` verifica si el nodo `n` precede al nodo `t` en la lista circular `c`. Se utiliza el cuantificador existencial `\exists` para indica que existe un nodo `p` que cumple ciertas condiciones:

```
1       \exists listNode *p;
```

1. `\let` es un *local binding* en *ACSL*, en este caso se usa para simplificar la expresión al referirse al primer elemento de la lista circular `c`.

```
\let head = c->elems;
```

2. Se evalúa si este elemento no es nulo con `isNull(head)` el cual debe retornar `\false` o `0`

```
isNull(head) == 0
```

3. Se evalúa el elemento `t` recibido como parámetro es nulo con `isNull(t)` el cual debe retornar `\false` o `0`

```
isNull(t) == 0
```

4. Por último se utiliza el cuantificador `\forall` el cual indica que para todos los nodos `p` que no son nulos y que no son `t`, al avanzar desde `head`, `p` no debe ser igual a `head` ni a `n`.

```
\forall listNode *p; (isNull(p) == 0 && p != t); p = p->next; p != head && p != n;
```

La condición completa significa que, comenzando desde `head`, recorriendo la lista hasta encontrar `t`, no se encuentra `n`, lo que implica que `n` precede a `t`.

lenC
La función `lenC` calcula la longitud de una lista circular `c`. La lógica sigue estos pasos:

1. La condición inicial verifica si la lista es nula con `isN(c)` o está vacía con `isE(c)`, si alguna de las dos es verdadera retorna `NPE`.

```
(isN(c) == 1 || isE(c) == 1) ? NPE :
```

2. Se utilizan `\let` para definir los nodos `t` y `n` como los elementos actuales y siguientes de la lista, respectivamente.

```
\let t = c->elems, n = c->elems->next;
```

3. Declara el nodo `head` con `\let` y le asigna el inicio del ciclo en la lista tras una búsqueda. El cuantificador existencial `\exists` indica que la expresión busca la existencia de algún nodo `p` en la lista que cumpla ciertas condiciones para encontrar el inicio: el nodo `t` no debe ser igual al nodo `n`, el nodo `n` no debe ser nulo y la función `isPrec(c, n, t)` debe ser `0`, lo que significa que `n` no debe preceder a `t` en la lista circular `c`. Se avanza `n` al siguiente nodo en la lista y se actualiza `t` para que sea igual al nuevo valor de `n` recorriendo la lista desde el nodo `t` hacia adelante, actualizando ambos punteros en cada iteración hasta encontrar un nodo `n` que cumpla con las condiciones mencionadas.

```
\let head = (\exists listNode *p; t != n && isNull(n) == 0 && isPrec(c, n, t) == 0;
    n = n->next, t = n);
```

4. Si `head` es nulo, retorna `NPE`. De lo contrario, se inicia el conteo de los nodos hasta llegar nuevamente a `head`. Se inicializa `n` como el nodo siguiente a `head`, se define con `\let` una variable local `counter` con valor inicial `1`, utilizando un bucle implícito definido por el cuantificador universal `\forall`, se recorre la lista circular comenzando desde n, para cada nodo `p`, mientras `p` no sea igual a `head` y `p` no sea nulo, se hace lo siguiente: `p` se actualiza para que apunte al siguiente nodo después de `n` y `counter` se incrementa en `1`. De esta manera se cuenta el número total de nodos en el ciclo, incluyendo `head`.

```
1     (isNull(head) == 1) ? NPE :
2     (n = head->next, \let counter = 1; \forall listNode* p; p != head && isNull(p) == 0;
          p = n->next; counter++);
```

lengthCircular
La función `lengthCircular` también calcula la longitud de una lista circular `c`, pero con una lógica ligeramente diferente a `lenC`. La lógica sigue estos pasos:

1. Valida si la lista es nula con `isN(c)` o está vacía con `isE(c)`, si alguna de las dos es verdadera retorna 0.

```
1     (isN(c) == 1 || isE(c) == 1) ? 0 :
```

2. Define con `\let` el nodo `h` como el primer elemento y `n` como el siguiente. Se inicia un contador `counter` en `1`.

```
1     \let h = c->elems, n = h->next;
2     \let counter = 1;
```

3. Si `n` es igual a `h`, retorna el valor del contador `counter`. De lo contrario, usa un bucle con cuantificadores para recorrer la lista y contar los nodos hasta volver a `h`. Si encuentra un nodo nulo en el camino (`isNull(current) == 1`), retorna `-1`, indicando un error.

```
1     \let result = (n == h) ? counter :
2     \exists listNode *p; \let current = n;
3     \forall listNode *p; (current != h && isNull(current) == 0);
4         current = current->next;
5         counter++;
6     (isNull(current) == 1) ? -1 : counter;
7     result;
```

Continuamos con el último caso *setList*
Tal y como hemos identificado, el tipo `setList` cuenta con su definición y sus observadores. Se define el tipo `setList` en la librería de la siguiente manera:

```
1     @   // ----- Set Lists --------------------------
2     @   type setList = struct {
3     @       int capacity; // Maximum capacity of the list
4     @       int lsize; // Size of the list
5     @       listNode *elems; // Pointer to the first element of the list
6     @   };
```

Esta definición crea una estructura llamada `setList` que representa una lista de conjuntos con una capacidad máxima definida. Cuenta con los componentes `int capacity`,

un entero que almacena la capacidad máxima de la lista, `int lsize`, un entero que almacena el tamaño actual de la lista y `listNode *elems`, un puntero al primer elemento (nodo) de la lista.

Definición de los predicados `isNull`, `isEmpty`, `isFull`, `contains`, `length` del tipo `setList`. A continuación, se dará una explicación de la lógica definida para cada uno de los predicados definidos. Todos los predicados reciben como parámetro un puntero `s` de tipo `setList`, y el predicado `contains` recibe también un parámetro `x` de tipo `int`.

- `isNull`
  - Este predicado verifica si una lista de conjuntos es nula. Si el puntero `s` es nulo (`\null`), devuelve `1`, indicando que la lista de conjuntos es nula. De lo contrario, devuelve `0`, indicando que la lista de conjuntos no es nula.

```
1       @   logic integer isNull(setList *s) =
2       @       (s == \null) ? 1 : 0;
```

- `isEmpty`
  - Este predicado verifica si una lista de conjuntos está vacía. Si la lista de conjuntos `s` es nula (`isNull(s) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, devuelve el resultado de `isNull(s->elems)`, utilizando el predicado `isNull` definido para `listNode` para verificar si el primer nodo es nulo.

```
1       @   logic integer isEmpty(setList *s) =
2       @       (isNull(s) == 1) ? NPE : isNull(s->elems);
```

- `isFull`
  - Este predicado verifica si una lista de conjuntos está llena. Si la lista de conjuntos `s` es nula (`isNull(s) == 1`), devuelve el código de error definido `NPE`, indicando un error de puntero nulo. De lo contrario, si el tamaño actual de la lista `s->lsize` es mayor o igual a su capacidad `s->capacity`, devuelve `1`, indicando que la lista está llena. De lo contrario, devuelve `0`, indicando que la lista no está llena.

```
1       @   logic integer isFull(setList *s) =
2       @       (isNull(s) == 1) ? NPE :
3       @       (s->lsize >= s->capacity) ? 1 : 0;
```

- `contains`
  - Este predicado verifica si un valor x está contenido en una lista de conjuntos. Si la lista de conjuntos `s` es nula (`isNull(s) == 1`), devuelve `0`, indicando que el valor no se encuentra en la lista. De lo contrario, recorre la lista de nodos `s->elems`, verificando si algún nodo contiene el valor `x`. Si se encuentra un nodo con el valor `x`, devuelve `1`. De lo contrario, devuelve `0`, indicando que el valor no se encuentra en la lista.

```
1       @   logic integer contains(setList *s, int x) =
```

```
2         @       (isNull(s) == 1) ? 0 :
3         @       \exists listNode *n; n == s->elems &&
4         @       (\forall listNode *p; isNull(p) == 0 && p->value != x; p = p->next; p ==
                n) &&
5         @       (isNull(n) == 0 && n->value == x) ? 1 : 0;
```

- `length`
  - Este predicado calcula la longitud de una lista de conjuntos. Si la lista de conjuntos `s` es nula (`isNull(s) == 1`), devuelve `0`, indicando que la lista no tiene elementos. De lo contrario, recorre la lista de nodos `s->elems`, incrementando un contador por cada nodo encontrado. Devuelve el valor del contador, que representa la longitud de la lista.

```
1         @   logic integer length(setList *s) =
2         @       (isNull(s) == 1) ? 0 :
3         @       \let n = s->elems;
4         @       \let counter = 0;
5         @       \forall listNode *p; isNull(p) == 0;
6         @           p = p->next;
7         @           counter++;
8         @       counter;
```

Al igual que `cycList` las funciones `contains` y `length` en particular fueron las más complejas de definir A continuación se explica a detalle.

contains
La función `contains` verifica si el valor `x` está presente en la lista `s`. Aquí está el desglose completo de su funcionamiento. La lógica sigue estos pasos:

1. Si `s` es nulo, retorna `0`, indicando que `x` no está presente.

```
1     (isNull(s) == 1) ? 0 :
```

2. Busca el valor de `x`, buscando un nodo `n` que cumpla las siguientes condiciones:

```
1     \exists listNode *n;
```

   a) Que el nodo `n` sea el primer elemento de la lista

```
1     n == s->elems
```

   b) Se asegura que para todos los nodos `p` en la lista, si `p` no es nulo y `p` no contiene el valor `x`, `p` avanza al siguiente nodo hasta recorrer toda la lista.

```
1     (\forall listNode *p; isNull(p) == 0 && p->value != x; p = p->next; p == n)
```

   c) Finalmente, se verifica que `n` no es nulo y que `n` contiene el valor `x`. Si se encuentra tal nodo `n` que cumple las condiciones, la función retorna `1`, indicando que `x` está presente en la lista. Si no se encuentra, retorna `0`.

```
1     (isNull(n) == 0 && n->value == x) ? 1 : 0;
```

length
La función `length` calcula la longitud de la lista `s`. La lógica sigue estos pasos:

1. La verificación inicial valida si `s` es nulo, retorna `0`, indicando que la lista está vacía.

```
1     (isNull(s) == 1) ? 0 :
```

2. Se define `n` como el primer elemento de la lista.

```
1     \let n = s->elems;
```

3. Se inicializa un contador `counter` en `0`.

```
1     \let counter = 0;
```

4. Utilizando un bucle implícito definido por el cuantificador universal `\forall`, se recorre la lista desde el primer elemento `n`. Para cada nodo `p`, mientras `p` no sea nulo, se avanza al siguiente nodo (`p = p->next`) y se incrementa el contador (`counter++`). Finalmente, la función retorna el valor final de `counter`, que es la cantidad de nodos en la lista.

```
1     \forall listNode *p; isNull(p) == 0;
2         p = p->next;
3         counter++;
4     counter;
```

Todas las definiciones en la librería permiten describir y verificar propiedades y operaciones sobre listas en el lenguaje `C`. Los predicados ayudan a asegurar que las funciones que operan sobre estas listas se comporten de manera correcta, manejando adecuadamente los casos de nodos nulos y calculando propiedades como la longitud de la lista o los valores de los elementos de manera precisa. Esta librería permitirá utilizar estas evaluaciones en una herramienta de verificación como *FramaC*.

4.3. Consideraciones: Predicados con el mismo nombre

Como podemos observar en la librería están definidos algunos de los predicados con el mismo nombre, como `isNull`, lo único que cambia es el tipo del parámetro que recibe. Esto es algo que podría confundir y para mitigar esta confusión se consideró tener diferentes librerías para cada tipo definido, pero esta solución no sería universal para todos los programas que se utilizan y se debe saber que librería importar para cada contrato traducido. Como parte de la consideración se evaluó la posibilidad de tener varias librerías y solventar este inconveniente creando una librería general que contenga cada una de las librerías, pero en *ACSL*, los módulos no pueden anidarse directamente.

Afortunadamente, en *ACSL*, se pueden tener predicados con el mismo nombre siempre y cuando los parámetros que reciben sean diferentes. En otras palabras *ACSL* soporta la sobrecarga de predicados y funciones de lógica, similar a como se hace

en algunos lenguajes de programación como *C++* o *Java*. Esto permite definir múltiples versiones de un predicado o función lógica que operan sobre diferentes tipos de argumentos o diferentes cantidades de argumentos.

Debido a que tener predicados con el mismo nombre no es un inconveniente, se han organizado los predicados y tipos lógicamente dentro de un único módulo utilizando comentarios y agrupando predicados relacionados juntos.

Capítulo 5

Traducción de Contratos

Como se ha expuesto en el capítulo de preliminares, los contratos inferidos *KindSpec* tienen siempre la misma estructura (Cap. 2.2.1). Debemos definir cual será la estructura final de los contratos en el formato estándar de expresiones lógicas en *ACSL* basándonos en los componentes que ya tenemos en el contrato inferido. Debido a que el contrato inferido por *KindSpec* cuenta con múltiples comportamientos, el tipo de contrato de *ACSL* que utilizaremos será el de contratos con comportamientos definidos (Cap. 3.4).

Traducción del contrato inferido a un contrato ACSL
Los contratos inferidos por *KindSpec* cuentan con cuatro elementos principales: `PRECONDITION P`, `POSTCONDITION Q`, `LOCATIONS L`, y `CANDIDATE AXIOMS Q#`. Para realizar la traducción solo necesitaríamos tres, `POSTCONDITION Q`, `LOCATIONS L`, y `CANDIDATE AXIOMS Q#`, debido a que las precondiciones `PRECONDITION P` es una disyunción de las precondiciones de los axiomas ($\vee p \mid (p \implies q)$) y las podemos obtener en la parte p de los axiomas $p \implies q$ en las postcondiciones `POSTCONDITION P`. El elemento de `PRECONDITIONS P` se agregará como un `requires` global para los comportamientos que definiremos.

A continuación, se expondrá el proceso de transformación y agrupación de estos elementos a en formato de contrato formal en *ACSL*. Para este ejercicio utilizaremos el contrato inferido de la función `collapseC`, en el anexo .2, en el programa `cyclic_lists.c`, en el anexo .1.

5.1. Traducción del elemento *POSTCONDITION Q*

Para poder realizar la traducción de las postcondiciones debemos analizar la estructura del los axiomas lógicos e interpretarlos como contratos con comportamientos definidos en *ACSL*.

Utilizaremos el siguiente axioma lógico `A6` del contrato inferido de la función `collapseC`, en el anexo .2, para traducirlo manualmente en expresiones lógicas para este ejercicio.

```
A6: (lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1)
```

Podemos dividir el axioma en tres partes.

1. `A6:`
   Será la primera parte que representará el nombre del comportamiento definido
2. `(lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0) =>`
   Será la precondición `requires` del comportamiento definido
3. `(lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1)`
   Será la postcondición `ensures` del comportamiento definido

Con esta división ya tendríamos el nombre del comportamiento el cual sería el mismo que la etiqueta del axioma, `A6`.

```
1 /*@ behaviour A6:
2   @*/
```

Ahora podemos evaluar la precondición. Quitando los paréntesis externos y la flecha de implicación del final (=>) del axioma tendríamos la parte p de $p \Rightarrow q$ la cual sería `lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0`. Debemos convertir las conjunciones ^ a la operación booleana `AND`. Como las expresiones lógicas de *ACSL* corresponden a expresiones puras de *C*, la operación booleana `AND` se expresa como `&&`, por lo que nos quedaría `lenC(c)=3 && isN(c)=0 && isE(c)=0`. Debido a que el signo igual simple (=) en *ACSL* se utiliza para definir/asignar y lo que tenemos en cada elemento de la precondición es una igualdad, debemos actualizar este signo al de la operación booleana de igualdad ==. Ahora nuestra precondición luce de esta manera, `lenC(c) == 3 && isN(c) == 0 && isE(c) == 0`. Para que una herramienta como *FramaC* pueda entender los predicados que se utilizan en los axiomas como `lenC`, `isN` o `isE` se deben de definir en un módulo. Estos predicados han sido incluidos con su interpretación en expresiones lógicas de *ACSL* en la librería `definedPredicates.acsl` definida en el capítulo anterior. Como último paso hay que utilizar la notación para acceder a las definiciones del modulo agregando el nombre del módulo `Pred` seguido por cuatro puntos (`::`) antes de cada predicado (esta notación esta definida en la versión 1.2 de *ACSL* para los módulos de especificación que son parte de las características experimentales de *ACSL* [2]). Guardaremos cada una de las precondiciones obtenidas para agregarlas en disyunción en un `requires` global al concluir de traducir cada uno de los comportamientos.. Al final podemos agregar la precondición en la cláusula `requires` del comportamiento definido `A6` , y quedaría de la siguiente manera:

```
1 /*@ behaviour A6:
2   @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0);
3   @*/
```

Ahora debemos seguir un proceso similar para postcondición, la parte q de $p \Rightarrow q$ la cual es `(lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1)`. Debemos aplicar los mismos cambios que hemos realizado para la precondición, quitando los paréntesis externos, cambiando las conjunciones ^ por `&&` y las igualdades = por ==. Además, en la postcondición el último elemento del conjunto representa el resultado que retorna la función (`ret=1`) por lo que lo tenemos que interpretarlo con el "*built-in construct*"de resultado de *ACSL* , `\result`. Tras aplicar estos cambios podemos agregar la postcondición en la cláusula `ensures` del comportamiento definido `A6`, y quedaría de la siguiente manera:

```
/*@ behaviour A6:
  @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0);
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result == 1
  @*/
```

En *KindSpec* se han añadido las excepciones más comunes del lenguaje de programación *C*, en la Figura 4.1. Estos códigos se encuentran en los axiomas como `(NPE, {45})`, y también se han sido incluidos con su interpretación correspondiente de *ACSL* en la librería `definedPredicates.acsl`, en el anexo .5, definida en el capítulo anterior. Como podemos observar estos códigos tienen la forma `(errorCode, {pc})`, donde el contador de programa `pc` identifica la declaración precisa del fragmento de código en el programa que causó el error. En la traducción solo debemos extraer el código `NPE` de `(NPE, {45})`, descartando el número de línea del código ya que aunque puede ser útil para depuración no suele usarse en herramientas de verificación. Por lo que quedaríamos solo con `Pred::NPE`, de tal forma que en una precondición o postcondición que incluye un código de error quedaría de la siguiente forma:

```
Pred::lenC(c) == (NPE, {45}) && Pred::isN(c) == 0 && Pred::isE(c) == 0
```

Quedaría de la siguiente manera:

```
Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0
```

Si aplicamos estos pasos para cada axioma de las postcondiciones en `POSTCONDITIONS Q` del contrato inferido por *KindSpec*, y agregamos en disyunción todas las precondicionas en un `requires` global, obtendríamos el siguiente contrato en *ACSL*.

```
/*@ requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 ||
  @          Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE ||
  @          Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @ behavior A1:
  @   requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == Pred::NPE;
  @ behavior A2:
  @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
  @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
  @   ensures \result == 0;
  @ behavior A3:
  @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
  @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
  @   ensures \result == 0;
  @ behavior A4:
  @   requires Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
  @ behavior A5:
  @   requires Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
  @ behavior A6:
  @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
```

```
31   @*/
```

5.2. Traducción del elemento *CANDIDATE AXIOMS Q#*

La traducción de los axiomas excepcionales o axiomas candidatos es similar a la de las postcondiciones debido a que comparten la misma estructura. Puede ocurrir la situación de que el contrato inferido por *KindSpec* no tenga ningún axioma candidato, en este caso este paso se saltaría. Debido a que el contrato inferido de la función `collapseC`, en el anexo .2, si tiene axiomas candidato se aplicaría el proceso de las postcondiciones y se agregarían como comportamientos definidos al final del contrato. Por lo que tras concurrir este proceso el contrato en *ACSL* tendría el siguiente aspecto:

```
1  /*@ requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
2    @          Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 ||
3    @          Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE ||
4    @          Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
5    @          Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
6    @          Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
7    @ behavior A1:
8    @   requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
9    @   ensures Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
10   @   ensures \result == Pred::NPE;
11   @ behavior A2:
12   @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
13   @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
14   @   ensures \result == 0;
15   @ behavior A3:
16   @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
17   @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
18   @   ensures \result == 0;
19   @ behavior A4:
20   @   requires Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
21   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
22   @   ensures \result == 1;
23   @ behavior A5:
24   @   requires Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
25   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
26   @   ensures \result == 1;
27   @ behavior A6:
28   @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
29   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
30   @   ensures \result == 1;
31   @ behavior C1:
32   @   requires Pred::lenC(c) == ?10 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
33   @   ensures Pred::lenC(c) == ?10 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
34   @   ensures \result == 1;
35   @ behaviour C2:
36   @   requires Pred::lenC(c) == ?10 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
37   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0
38   @   ensures \result == 1;
39   @*/
```

5.3. Traducción del elemento *LOCATIONS L*

El elemento `LOCATIONS L` del contrato inferido representa el conjunto de referencias a ubicaciones de memoria (parámetros de función y punteros y campos de estructuras de datos) cuyo valor podría verse afectado por la ejecución de la función. Este

elemento del contrato es comparable a la cláusula `assigns` en *ACSL*.

Debido a que no sabemos concretamente el comportamiento específico que modifica estas ubicaciones o si todos los comportamientos modifican, especificaremos estas ubicaciones globalmente en el contrato *ACSL*. Para cada elemento en del conjunto de ubicaciones se agregará una clausula `assigns` antes de la primera declaración de comportamiento y después de la declaración del `requires` global. Por lo que el contrato en *ACSL* tendría el siguiente aspecto:

```
/*@ requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 ||
  @          Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE ||
  @          Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
  @          Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @ assigns t;
  @ assigns n;
  @ assigns head;
  @ assigns head->next;
  @ assigns c->lsize;
  @ behavior A1:
  @   requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == Pred::NPE;
  @ behavior A2:
  @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
  @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
  @   ensures \result == 0;
  @ behavior A3:
  @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
  @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
  @   ensures \result == 0;
  @ behavior A4:
  @   requires Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
  @ behavior A5:
  @   requires Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
  @ behavior A6:
  @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
  @   ensures \result == 1;
  @ behavior C1:
  @   requires Pred::lenC(c) == ?l0 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >= 2;
  @   ensures Pred::lenC(c) == ?l0 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >= 2;
  @   ensures \result == 1;
  @ behaviour C2:
  @   requires Pred::lenC(c) == ?l0 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >= 2;
  @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0
  @   ensures \result == 1;
  @*/
```

Podemos notar el uso del operador `->` en los elementos de memoria que pueden ser alterados, este operador se utiliza para acceder a los miembros de una estructura a través de un puntero. Algunos de estos elementos de memoria son punteros a una estructura, este operador facilita la referencia directa a los campos de esa estructura sin necesidad de desreferenciar manualmente el puntero y luego acceder al miembro. En la librería de predicados definida en el capítulo anterior, consideramos estas estructuras a las que hace referencia el contrato inferido por lo que las herramientas de verificación podrán interpretar estas referencias directas. Por ejemplo, `head` es un puntero a una estructura `listNode` la cual contiene un campo `next`, la expresión

`head->next` accede directamente al miembro `next` de la estructura. Debido a que la función inferida trabaja con estructuras anidadas o listas enlazadas no realizaremos ninguna modificación adicional a estos elementos y los dejaremos como están definidas en el contrato inferido para simplificar el código y mejorar la legibilidad.

5.4. Agregar el contrato ACSL al programa original

Una vez el contrato inferido es traducido exitosamente a un contrato formal de *ACSL* se puede agregar justo antes de la función a la que corresponde el contrato.

Podemos obtener el nombre de la función por medio de esta sección del contrato, ya que como se ha mencionado anteriormente, todos los contratos tienen la misma estructura.

```
*********************************************
*********************************************
Inference performed May 26 2020
Selected modifier function: collapseC
From file: examples/cyclic_lists.c
*********************************************
*********************************************
```

Al tener el nombre de la función, en este caso `collapseC`, podemos agregar el contrato justo arriba de su definición. Además, hay que importar la librería de módulos de especificación (`definedPredicates.acsl`, en el anexo .5) de predicados, que se almacenará en un fichero llamad `Pred.acsl`, justo al principio del programa (`//@ import Pred;`).

Al agregar esto al programa `cyclic_lists.c`, en el anexo.1, tendríamos lo siguiente:

```
1  //@ import Pred;
2  #include <stdlib.h>
3
4  struct clist {
5   int lsize;
6   struct lnode *elems;
7  };
8
9  struct lnode {
10  int value;
11  struct lnode *next;
12 };
13
14 int isPrec(struct clist *c, struct lnode *n, struct lnode *t) {
15  . . .
16 }
17
18 /*@ requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
19   @          Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 ||
20   @          Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE ||
21   @          Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
22   @          Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
23   @          Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
24   @ assigns t;
25   @ assigns n;
26   @ assigns head;
27   @ assigns head->next;
28   @ assigns c->lsize;
29   @ behavior A1:
30   @   requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
31   @   ensures Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
```

```
32   @   ensures \result == Pred::NPE;
33   @ behavior A2:
34   @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
35   @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
36   @   ensures \result == 0;
37   @ behavior A3:
38   @   requires Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
39   @   ensures Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
40   @   ensures \result == 0;
41   @ behavior A4:
42   @   requires Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
43   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
44   @   ensures \result == 1;
45   @ behavior A5:
46   @   requires Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
47   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
48   @   ensures \result == 1;
49   @ behavior A6:
50   @   requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
51   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
52   @   ensures \result == 1;
53   @ behavior C1:
54   @   requires Pred::lenC(c) == ?10 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
55   @   ensures Pred::lenC(c) == ?10 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
56   @   ensures \result == 1;
57   @ behaviour C2:
58   @   requires Pred::lenC(c) == ?10 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?10 >= 2;
59   @   ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0
60   @   ensures \result == 1;
61   @*/
62 int collapseC(struct clist *c) {
63  struct lnode *n;
64  struct lnode *t;
65  struct lnode *head;
66
67  if(c == NULL) return 0;
68  if(c->elems == NULL) return 0;
69
70  t = c->elems;
71  n = c->elems->next;
72
73  while (t != n && !(isPrec(c, n, t))) {
74   t = n;
75   n=n->next;
76  }
77
78  head = n;
79  n = head->next;
80  while(n != head) {
81   t = n;
82   n = n->next;
83   free(t);
84   c->lsize--;
85  }
86
87  head->next = head;
88
89  return 1;
90 }
91
92 int isN(struct clist *c) {
93  . . .
94 }
95
96 int isE(struct clist *c) {
97  . . .
98 }
99
100 int lenC(struct clist *c) {
101  . . .
```

```
}
```

5.5. Algoritmo de traducción

A continuación se expone el algoritmo en pseudocódigo que describe los pasos para transformar los contratos en formato axioma lógico a un contrato en formato de especificación estándar *ACSL*:

```
ALGORITMO transformarContrato

    // Paso 1: Identificar los elementos en las secciones del contrato inferido
    ELEMENTOS:
        POSTCONDITION Q
        LOCATIONS L
        CANDIDATE AXIOMS Q#
    // Paso 2: Traducir los axiomas de la seccion POSTCONDITION Q
    PARA CADA axioma EN POSTCONDITION Q HACER
        traducirAxiomaACSL(axioma)
    FIN PARA

    // Paso 3: Adaptar el formato al lenguaje ACSL cambiando notacion
    FUNCION traducirAxiomaACSL(axioma)
        // Realiza las conversiones de notacion necesarias
        axiomacs := convertirNotacionACSL(axioma)
        RETORNAR axiomacs
    FIN FUNCION

    // Paso 4: Construir un comportamiento definido para cada uno de los elementos de
        POSTCONDITION Q
    PARA CADA axiomaACSL EN POSTCONDITION Q TRADUCIDOS HACER
        comportamiento := construirComportamientoDefinido(axionaACSL)
    FIN PARA

    // Paso 5: Identificar las precondiciones de los axiomas y agregarlas al principio como
        una clausula global
    clausulaGlobal := agregarClausulaGlobal(Precondicion de POSTCONDITION Q TRADUCIDOS)

    // Paso 6: Traduccion de los axiomas de la seccion CANDIDATE AXIOMS Q#
    PARA CADA axiomaCandidato EN CANDIDATE AXIOMS Q# HACER
        axiomaCandidatoACSL := traducirAxiomaACSL(axiomaCandidato)
    FIN PARA

    // Paso 7: Traduccion de los elementos de la seccion LOCATIONS L
    PARA CADA elemento EN LOCATIONS L HACER
        elementoACSL := traducirElementoACSL(elemento)
    FIN PARA

    // Paso 8: Identificar la funcion a la que pertenece el contrato inferido
    funcionContrato := identificarFuncionContrato()

    // Paso 9: Identificar la funcion en el codigo fuente proporcionado
    funcionCodigoFuente := buscarFuncionCodigoFuente(funcionContrato)

    // Paso 10: Agregar el contrato ACSL antes de la declaracion de la funcion
    insertarContratoACSL(funcionCodigoFuente, clausulaGlobal, POSTCONDITION Q TRADUCIDOS,
        CANDIDATE AXIOMS Q# TRADUCIDOS, LOCATIONS L TRADUCIDOS)

FIN ALGORITMO

FUNCION convertirNotacionACSL(axioma)
    // Realiza las conversiones especificas de la notacion logica a ACSL
    RETORNAR axiomacsConvertido
FIN FUNCION

FUNCION construirComportamientoDefinido(axiomaACSL)
```

```
    // Construye el comportamiento definido para el axioma traducido
    comportamiento := "behavior " + nombreAxioma + ":"
    comportamiento += " requires " + axiomaACSL.precondicion + ";"
    comportamiento += " ensures " + axiomaACSL.postcondicion + ";"
    RETORNAR comportamiento
FIN FUNCION

FUNCION agregarClausulaGlobal(postCondicionesTraducidas)
    clausulaGlobal := "/*@ requires "
    PARA CADA preCondicion EN postCondicionesTraducidas HACER
        clausulaGlobal += preCondicion + " || " // OR disyuncion
    FIN PARA
    clausulaGlobal.reemplazarFinal(" || ", ";")
    RETORNAR clausulaGlobal
FIN FUNCION

FUNCION traducirElementoACSL(elemento)
    // Traduce un elemento de la seccion LOCATIONS L al formato ACSL
    RETORNAR elementoTraducido
FIN FUNCION

FUNCION identificarFuncionContrato()
    // Identifica la funcion a la que pertenece el contrato inferido
    RETORNAR nombreFuncion
FIN FUNCION

FUNCION buscarFuncionCodigoFuente(funcionContrato)
    // Busca la funcion correspondiente en el codigo fuente proporcionado
    RETORNAR funcionEncontrada
FIN FUNCION

FUNCION insertarContratoACSL(funcionCodigoFuente, clausulaGlobal, postCondiciones,
    axiomasCandidatos, locations)
    // Inserta el contrato ACSL antes de la declaracion de la funcion en el codigo fuente
    insertarAntesDeDeclaracion(funcionCodigoFuente, clausulaGlobal)
    insertarAntesDeDeclaracion(funcionCodigoFuente, locations)
    insertarAntesDeDeclaracion(funcionCodigoFuente, postCondiciones)
    insertarAntesDeDeclaracion(funcionCodigoFuente, axiomasCandidatos)
FIN FUNCION
```

Este pseudocódigo detalla el proceso de transformación cubriendo desde la identificación de elementos hasta la inserción del contrato transformado en el código fuente. Servirá como base para el desarrollo.

Capítulo 6

Desarrollo

Considerando que la herramienta *KindSpec* está desarrollada en *Java* [12], se ha optado por utilizar el mismo lenguaje de programación, de tal forma que el desarrollo de este trabajo pueda utilizarse como base para agregar esta funcionalidad a la herramienta en una siguiente versión. En base a esto se ha optado utilizar el entorno de desarrollo integrado *IntelliJ IDEA* [13] por su interfaz de usuario intuitiva y sus herramientas de desarrollo robustas.

Solución desarrollada
El objetivo de esta solución crear un servicio que realice la traducción de los contratos inferidos a la notación de *ACSL*, e integrar el contrato obtenido en código fuente *C* de donde se infirió el contrato. Por lo que la solución obtendrá dos archivos independientes, uno con el código fuente del programa *C* y otro con el contrato inferido, y retornará un único archivo con el contrato en formato *ACSL* integrado como anotaciones/comentarios en el código fuente del programa C.

Como parte de la solución, se desarrolló un servicio en *Java* que se encarga de cargar el archivo de contrato inferido y el archivo de código fuente, traducir el contrato y agregarlo al código fuente. Además, el servicio cuenta con la funcionalidad para generar un nuevo archivo de código fuente o sobrescribir el archivo de código fuente original. También, se creó una aplicación *Java* con una única ventana que permite seleccionar ambos archivos y utilizar el servicio, se puede ver en la Figura 6.2. Esta herramienta facilita la integración de contratos inferidos en el código fuente, logrando el objetivo propuesto. La arquitectura de esta solución se puede observar en la Figura 6.1.

Aspectos importantes a considerar
Para asegurar el correcto funcionamiento del servicio, es esencial tener en cuenta que únicamente es compatible con contratos generados en el formato estandarizado que proporciona *KindSpec*. Por lo tanto, solo se permiten archivos con las extensiones `.contract` o `.output` para los contratos, y archivos con la extensión `.c` para el código fuente en *C*. El código fuente debe incluir la función a partir de la cual se infirió el contrato; de lo contrario, se producirá un error.

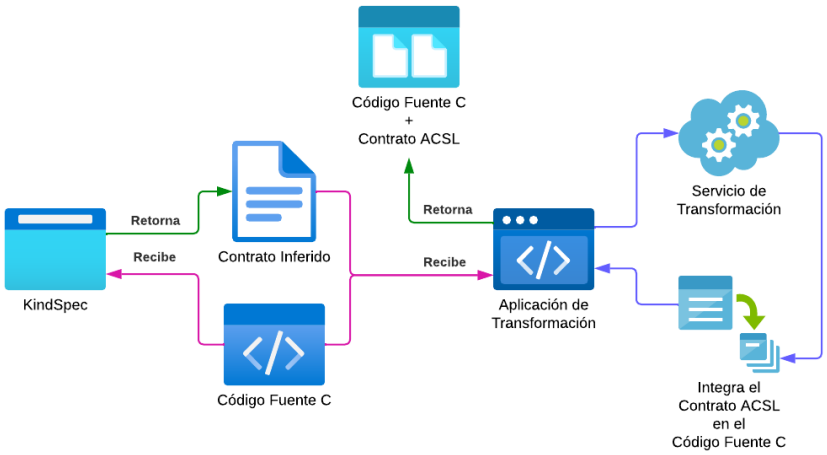

Figura 6.1: Arquitectura de la solución

6.1. Servicio de Traducción de Contratos Software

El servicio Java, `ACLSContractGeneratorService`, está diseñado para generar contratos en la notación *ACSL* a partir de un archivo de contrato inferido, e integrar el resultado en un archivo de código fuente del lenguaje *C*.

El código está estructurado en varias secciones.

- Variables públicas, que pueden ser utilizadas por cualquier programa que invoque el servicio.
- Variables privadas, donde se almacena la información que el servicio utiliza en el proceso de transformación.
- Función `main`, utilizada únicamente para realizar pruebas al servicio.
- Funciones públicas, que serán utilizadas por el programa que invoque el servicio.
- Funciones privadas, que se encargan de todos los pasos identificación y transformación de contratos.
- Funciones privadas de soporte, que ayudan a limpiar cadenas de texto entre otras funcionalidades.

A continuación, se describe cada parte del código y su funcionalidad.

6.1.1. Variables Públicas del Servicio

- `fileLoaded`: Indica si el archivo de contrato inferido ha sido cargado exitosamente.

```
1     // Flag to check if the inferred contract file has been loaded
2     public Boolean fileLoaded = false;
```

- `acslContractLines`: Almacena las líneas del contrato *ACSL* generadas.

```
1     // List to store the generated ACSL lines
2     public List<String> acslContractLines = new ArrayList<>();
```

- `modifierFunction`: Almacena el nombre de la función a la cual pertenece el contrato inferido.

```
1     // Name of the modifier function of the inferred contract file
2     public String modifierFunction = null;
```

6.1.2. Variables Privadas del Servicio

- `postconditionQLines`, `locationsLLines`, `candidateAxiomsQSharpLines`: Almacenan líneas específicas del archivo de contrato inferido relacionadas con las postcondiciones, ubicaciones modificadas y axiomas candidatos.

```
1     // List to store POSTCONDITION Q lines of the inferred contract file
2     private List<String> postconditionQLines = new ArrayList<>();
3     // List to store LOCATIONS L lines of the inferred contract file
4     private List <String> locationsLLines = new ArrayList<>();
5     // List to store CANDIDATE AXIOMS Q# lines of the inferred contract file
6     private List<String> candidateAxiomsQSharpLines = new ArrayList<>();
```

- `preconditionsP`, `postconditionsQ`: Almacenan las precondiciones y postcondiciones extraídas de las líneas de `postconditionQLines`.

```
1     // List to store preconditions (p in p=>q) of the POSTCONDITION Q lines
2     private List<String> preconditionsP = new ArrayList<>();
3     // List to store postconditions (q in p=>q) of the POSTCONDITION Q lines
4     private List<String> postconditionsQ = new ArrayList<>();
```

- `preconditionsPSharp`, `postconditionsQSharp`: Almacenan las precondiciones y postcondiciones extraídas de las líneas de `candidateAxiomsQSharpLines`.

```
1     // List to store the preconditions (p in p=>q) of the CANDIDATE AXIOMS Q# lines
2     private List<String> preconditionsPSharp = new ArrayList<>();
3     // List to store the postconditions (q in p=>q) of the CANDIDATE AXIOMS Q# lines
4     private List<String> postconditionsQSharp = new ArrayList<>();
```

- `equalityRegex`: Expresión regular para coincidir con patrones de igualdad.

```
1     // Regex to match the equality pattern
2     private String equalityRegex = "(?<![<>!])=";
```

6.1.3. Funciones Principales del Servicio

`main`
Función principal utilizada únicamente para probar el funcionamiento correcto del servicio. Realiza las siguientes tareas:

1. Define las rutas de los archivos de prueba, los cuales serán el contrato inferido y el código fuente en *C*.

```
String filePath = "/Users/lucarojo/Downloads";
String fileFolder = "/00_TestFiles";
String fileContractName = "/cyclic_lists.output";
String fileSourceCodeName = "/cyclic_lists.c";
```

2. Crea una instancia del servicio.

```
ACLSContractGeneratorService service = new ACLSContractGeneratorService();
```

3. Utilizando el servicio realiza la carga del archivo de contrato inferido.

```
service.fileLoaded = service.loadContractFile(filePath + fileFolder +
    fileContractName);
```

4. Verifica que se haya cargado el contrato correctamente.
5. Utilizando el servicio, genera las líneas del contrato en anotaciones *ACSL* y las almacena en el una variable del servicio.

```
service.acslContractLines = service.generateACSLContract(true);
// Generate ACSL lines from the inferred contract file
if(!service.acslContractLines.isEmpty()) {
    System.out.print("ACSL Lines generated: \n");
    for (int i = 0; i < service.acslContractLines.size(); i++) {
        System.out.print(service.acslContractLines.get(i));
    }
}
```

6. Utilizando el servicio, inserta las líneas generadas en el código fuente *ACSL*.

```
try {
    boolean success = service.insertACSLContractLines(service.acslContractLines,
        service.modifierFunction, filePath + fileFolder + fileSourceCodeName, false)
        ;
    if(success)
        System.out.print("ACSL Contract lines inserted successfully.\n");
    else
        System.out.print("ACSL Contract lines not inserted.\n");
} catch (IOException e) {
    e.printStackTrace();
}
```

`loadContractFile`

Esta función es crucial ya que se encarga de cargar el archivo de contrato inferido y extraer todos los componentes que se utilizarán para generar el contrato en ACSL. Cada uno de estos elementos los obtiene utilizando funciones que extraen específicamente los elementos de sección. A continuación se detalla lo que se extrae del contrato inferido.

1. La función a la cual pertenece el contrato inferido, se almacenará en la variable `modifierFunction`, y se utilizará más adelante para identificar la línea donde está declarada la función en el código fuente.

```
// Extract the modifier function from the inferred contract file
this.modifierFunction = extractModifierFunction(filePath);
```

Utiliza la función observadora `extractModifierFunction`, recibe como parámetro la ubicación proveída por el usuario donde se encuentra el contrato y se encarga de encontrar la función en el contrato.

2. Las líneas que contienen los axiomas ($p \implies q$) que se encuentran en la sección `POSTCONDITION Q`del contrato. Se almacenarán en la lista `postconditionQLines` y se utilizaran para generar los comportamientos definidos principales y la precondición global, debido a que estas líneas contienen los axiomas, cada axioma tiene una precondición p y postcondición q en forma de implicación $p \implies q$.

```
1     // Extract the POSTCONDITION Q lines from the inferred contract file
2     this.postconditionQLines = extractPostconditionQLines(filePath);
```

Utiliza la función observadora `extractPostconditionQLines`, que recibe como parámetro la ubicación del contrato y se encarga de encontrar los axiomas dentro de la sección `POSTCONDITION Q` del contrato inferido.

3. Una vez es que tenemos todas los axiomas, para cada uno de ellos se debe extraer la precondición, la parte p de $p \implies q$ y almacenarla en la lista `preconditionsP` y la postcondición, la parte q, y almacenarla en la lista `postconditionsQ`.

```
1     // Extract the preconditions (p in p=>q) from the POSTCONDITION Q lines
2     this.preconditionsP = extractPreconditionsP(this.postconditionQLines);
3     // Extract the postconditions (q in p=>q) from the POSTCONDITION Q lines
4     this.postconditionsQ = extractPostconditionsQ(this.postconditionQLines);
```

Utiliza las funciones observadoras `extractPreconditionsP` y `extractPostconditionsQ`, que reciben como parámetro la lista `postconditionQLines` con las postcondiciones. para extraer las precondiciones y postcondiciones de los axiomas. Las precondiciones almacenadas en `preconditionsP` se utilizarán en la cláusula `requires` de los comportamientos definidos y también se utilizarán para definir la precondición global de la función, la disyunción de precondiciones ($\vee p \mid p \implies q$), estas precondiciones también se pueden obtener de la sección `PRECONDITION P` del contrato inferido, pero por simplicidad se extraerán de las `POSTCONDITION Q`, almacenadas en la lista `postconditionQLines`. El formato que tendrán las precondiciones y postcondiciones en estas listas será el identificador del axioma unido con el axioma y separado por dos puntos `:`, por ejemplo `A1:isNull(c)= . . ..`

4. Las lista de ubicaciones en la sección `LOCATIONS L` del contrato y almacenarlas en la lista `locationsLLines`. Estas líneas se utilizarán en la cláusula `assigns` global del contrato *ACSL* de la función.

```
1     // Extract the LOCATIONS L lines from the inferred contract file
2     this.locationsLLines = extractLocationsLLines(filePath);
```

Utiliza la función observadora `extractLocationsLLines`, que recibe como parámetro la ubicación del contrato y se encarga de extraer la lista de referencias a ubicaciones de memoria que podrían verse afectadas por la ejecución de la función de la sección `LOCATIONS L` del contrato.

5. Las líneas de axiomas candidatos de la sección `CANDIDATE AXIOMS Q#` del contrato y las almacena en la lista `candidateAxiomsQSharpLines`.

```
1     // Extract the CANDIDATE AXIOMS Q# lines from the inferred contract file
2     this.candidateAxiomsQSharpLines = extractCandidateAxiomsQSharpLines(filePath);
```

   Utiliza la función observadora `extractCandidateAxiomsQSharpLines` que recibe como parámetro la ubicación del contrato y se encarga de extraer la lista de axiomas candidato de la sección `CANDIDATE AXIOMS Q#` del contrato inferido.

6. Las precondiciones y postcondiciones de los axiomas candidato extraídos y almacenados en `candidateAxiomsQSharpLines`, y almacenarlas en las listas `preconditionsPS` y `postconditionsQSharp` respectivamente.

```
1     // Extract the preconditions (p in p=>q) from the CANDIDATE AXIOMS Q# lines
2     this.preconditionsPSharp = extractPreconditionsP(this.candidateAxiomsQSharpLines);
3     // Extract the postconditions (q in p=>q) from the CANDIDATE AXIOMS Q# lines
4     this.postconditionsQSharp = extractPostconditionsQ(this.candidateAxiomsQSharpLines);
```

   Utilizando las funciones observadoras `extractPreconditionsP` y `extractPostconditionsQ` mencionadas anteriormente, para extraer estas partes del axioma.

7. Por último establece la variable booleana `fileLoaded` en `true` si todo se carga correctamente. Esta variable bandera es utilizada por el programa que utiliza el servicio para validar que el contrato inferido y sus elementos se hayan cargado correctamente.

`generateACSLContract`
Esta función se encarga de generar las líneas del contrato *ACSL* en comportamientos definidos a partir de los elementos extraídos del contrato inferido por medio de la función `loadContractFile`. Tiene la opción de considerar o no los axiomas candidatos, en caso de que el usuario no quiera que se incluyan dichos axiomas. Esta función solo se puede ejecutar si se ha ejecutado la función `loadContractFile` antes, si no se ha ejecutado dicha función o si al ejecutarla se produjo un error y no se cargaron los elementos correctamente, la función retornará `null`. Esta validación lo hace revisando la variable `fileLoaded`.

```
1     if(!this.fileLoaded) {
2         return null;
3     }
```

Si los elementos del contrato inferido se han cargado correctamente, se declara una lista `start` que contendrá todas las líneas del contrato en *ACSL*.

```
1     List<String> result = new ArrayList<>();
2     String start = "/*@";
```

Después utilizará esta lista para realizar las siguientes acciones:

1. Generará la cláusula `requires` global que contendrá la disyunción de precondiciones. Utilizará la lista `preconditionsP` para generar esta cláusula.

```
1     for (int i = 0; i < this.preconditionsP.size(); i++) {
```

```
2           String precondition = this.preconditionsP.get(i);
3           String[] preconditionArray = precondition.split(":");
4           String pre = preconditionArray[1].replace("^", "&&").replaceAll(equalityRegex, "
                == ");
5           pre = this.removeErrorPattern(pre);
6           pre = this.addModuleNotation(pre);
7           start = (i == 0 ? start + " requires " : "  @          ") + pre + " ||\n";
8           result.add(start);
9       }
```

Debido a que agrega al final de cada línea "`||\n`"se debe reemplazar este texto para la última línea por el texto "`;\n`".

```
1       result.set(result.size() - 1, this.removeTrailingStr(result.getLast(), " ||\n") + "
            ;\n");
```

2. Generará las líneas de las cláusulas `assigns` globales para cada elemento de la lista `locationsLLines`. Cabe destacar que habrá una cláusula `assigns` para cada uno de los elementos.

```
1       if(!this.locationsLLines.isEmpty()) {
2           for (int i = 0; i < this.locationsLLines.size(); i++) {
3               String location = this.locationsLLines.get(i);
4                start = "  @ assigns " + location + ";\n";
5                result.add(start);
6           }
7       }
```

3. Generará cada uno los comportamientos en una cláusula `behaviour` con sus respectivas precondiciones, en cláusulas `requires`, y postcondiciones, en clausulas `ensures`. Se utilizará cada línea de las listas `preconditionsP` y `postconditionsQ` para generar cada comportamiento utilizando la función observadora `getACSLNamedBehaviou` que recibe un elemento de cada lista como parámetro.

```
1       for (int i = 0; i < this.preconditionsP.size(); i++) {
2           String precondition = this.preconditionsP.get(i);
3           String postcondition = this.postconditionsQ.get(i);
4           String acsl = getACSLNamedBehaviourLine(precondition, postcondition);
5           result.add(acsl);
6       }
```

4. Si se especifica que se considerarán axiomas candidatos y la lista `preconditionsPSharp` no está vacía, se generarán comportamientos adicionales en cláusulas `behaviour` con cada uno de estos axiomas, de la misma forma que se generaron los comportamientos anteriores.

```
1       if(!preconditionsPSharp.isEmpty() && considerCandidateAxiomsQSharp) {
2           for (int i = 0; i < this.preconditionsPSharp.size(); i++) {
3               String precondition = this.preconditionsPSharp.get(i);
4               String postcondition = this.postconditionsQSharp.get(i);
5               String acsl = getACSLNamedBehaviourLine(precondition, postcondition);
6               result.add(acsl);
7           }
8       }
```

5. Por último agregará la terminación del contrato *ACSL*, el cierre de la anotación. y retornará la lista `start`.

```
1     String end = "  @*/\n";
2         result.add(end);
3
4         return result;
```

`insertACSLContractLines`
La última función principal se encarga de insertar el contrato *ACSL* generado en el archivo de código fuente *C* proporcionado agregando las líneas justamente antes de la declaración de la función identificada en el contrato inferido. Esta función recibe como parámetro la lista con las líneas del contrato *ACSL*, el nombre de la función, la ruta de la ubicación del archivo de código fuente *C* y una bandera si se modificará el archivo de código fuente o se generará uno nuevo. Esta función retorna `true` si se completan las acciones correctamente y retorna `false` si ocurre un error o si no se ha encontrado la función identificada en el contrato inferido. Los pasos que toma esta función para realizar esto son los siguientes:

1. Obtiene todas las líneas del archivo de código fuente *C* y las almacena en una lista `lines`.

```
1     List<String> lines = Files.readAllLines(Paths.get(pathToSourceCode));
```

2. Recorre esta lista y antes de la primera línea del código fuente agrega la importación de la librería de módulo de especificación definida en el capítulo 4. Luego busca la declaración de la función identificada en el contrato inferido y al encontrarla, establece la variable bandera `functionFound` a `true` e inserta las líneas del contrato *ACSL* antes de la línea encontrada.

```
1     // Find the declaration of the specified function
2     boolean functionFound = false;
3     for (int i = 0; i < lines.size(); i++) {
4         if(i == 0) {
5             // Insert library import for predicates specification module (//@ import
                  Pred;)
6             lines.add(i, "//@ import Pred;");
7         }
8         String line = lines.get(i).trim();
9         if (line.matches(".*\\b" + function + "\\b\\s*\\(.*\\).*")) {
10            functionFound = true;
11            // Insert the contract lines before the function declaration
12            for (int j = 0; j < contractLines.size(); j++) {
13                String stringToAdd = this.removeTrailingStr(contractLines.get(j), "\n");
14                lines.add(i + j, stringToAdd);
15            }
16            break;
17        }
18    }
```

3. Revisa se se encontró la función identificada en el contrato inferido por medio de la variable vandera `functionFound`, y de no ser así retorna `false`.

```
1     if (!functionFound) {
2         System.out.println("Function " + function + " not found in the source file.");
3         return false;
4     }
```

4. Dependiendo del parámetro bandera `overrideFile`, escribe el contenido modificado en el archivo fuente *C* original o genera uno nuevo, y si todo va bien retorna `true`

```
1     if(overrideFile) {
2         // Write the modified content back to the C source file
3         Files.write(Paths.get(pathToSourceCode), lines);
4     }
5     else {
6         // Write the modified content to a new C source file
7         Files.write(Paths.get(pathToSourceCode.replace(".c", "_contract.c")), lines);
8     }
9     return true;
```

6.1.4. Funciones Auxiliares Observadoras

Estas funciones ayudan a extraer y procesar las distintas secciones del archivo de contrato inferido. A continuación se explica de forma resumida lo que realiza cada una de ellas.

`extractModifierFunction`
Extrae la función a la cual pertenece el contrato inferido. Recibe como parámetro la ruta de la ubicación del contrato inferido y retorna el nombre de la función.

`extractPostconditionQLines`
Extrae las líneas de postcondición `POSTCONDITION Q` del contrato inferido. Recibe como parámetro la ruta de la ubicación del contrato inferido y retorna una lista con los axiomas extraídos.

`extractPreconditionsP` y `extractPostconditionsQ`
Extraen las precondiciones y postcondiciones de las líneas de postcondición `POSTCONDITION Q` extraídas. Recibe como parámetro la lista de postcondiciones extraídas, y retornan las precondiciones y postcondiciones junto con el identificador de axioma.

`extractLocationsLLines`
Extrae las líneas de ubicaciones `LOCATIONS L` del contrato inferido. Recibe como parámetro la ruta de la ubicación del contrato inferido y retorna la lista de referencias a ubicación.

`extractCandidateAxiomsQSharpLines`
Extrae las líneas de axiomas candidatos `CANDIDATE AXIOMS Q#` del contrato inferido. Recibe como parámetro la ruta de la ubicación del contrato inferido y retorna la lista de axiomas candidato.

`getACSLNamedBehaviourLine`
Genera una línea `behaviour` del comportamiento definido con las respectivas precondiciones `requires` y postcondiciones `ensures` del contrato *ACSL*. Recibe como parámetro una precondicion y una postcondicion, y retorna la línea `requires`.

`removeTrailingStr`, `removeLeadingStr`, `removeErrorPattern`, `addModuleNotation`
Eliminan o agregan partes específicas de las cadenas de texto para un procesamiento adecuado de los elementos.

Con el servicio `ACLSContractGeneratorService` se ha logrado cumplir el objetivo de obtener un único archivo con el contrato en formato ACSL integrado como ano-

taciones/comentarios en el código fuente del programa C. Este servicio cumple con todos los requerimientos que se han planteado en los objetivos, carga un archivo de contrato inferido, extrae los elementos importantes, genera un contrato *ACSL* a partir de estos elementos y lo inserta en un archivo de código fuente *C*. Este proceso automatiza la generación de contratos *ACSL* a partir de contratos inferidos por *KindSpec*. El código completo del servicio se puede consultar en el anexo .6.

6.2. Interfaz gráfica

Como soporte a la solución del servicio y para poder probarlo y utilizarlo de forma amigable, se creo una aplicación simple en *Java*. La aplicación desarrollada, mostrada en la Figura 6.2 cuenta con un formulario con dos campos para definir las rutas del contrato inferido y el código fuente del programa *C*. Además tiene la opción de especificar si se quieren considerar los axiomas candidato en el contrato *ACSL* generado y si se quiere crear un nuevo archivo de código fuente con el contrato integrado o sobrescribir el archivo otorgado.

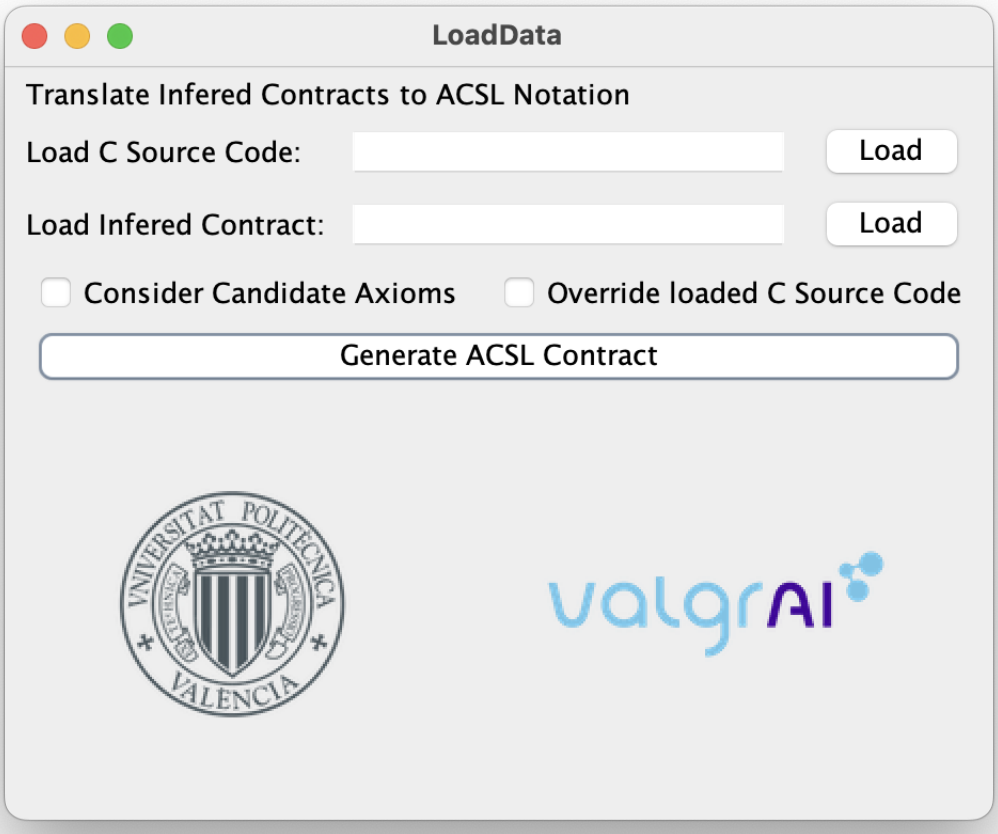

Figura 6.2: Ventana de la aplicación desarrollada

6.2.1. Funcionamiento

- Cuando se da clic en el botón `Load` del campo `Load C Source Code`, se abre una ventana para cargar el código fuente [Figura 6.3], solo acepta archivos con extensión `.c`.
- Cuando se da clic en el botón `Load` del campo `Load Inferred Contract`, se abre una ventana para cargar el contrato inferido [Figura 6.4], solo acepta archivos con extensión `.output` o `.contract`.
- Se pueden marcar los *checkBox* opcionales `Consider Candidate Axioms` para considerar los axiomas candidatos y agregarlos al contrato en notación `ACSL` y `Override loaded C Source Code` que especifica si se creará un archivo nuevo de código fuente [Figura 6.6] o se sobrescribirá el archivo cargado [Figura 6.5].
- Al dar clic en el botón `Generate ACSL Contract`, procederá a realizar la traducción del contrato inferido seleccionado a la notación de *ACSL* e integrarlo al código fuente del programa *C* cargado.
  - Si la opción `Consider Candidate Axioms` está seleccionada y existen axiomas candidatos en el contrato inferido, se agregarán al contrato en *ACSL*.
  - Si la opción `Override loaded C Source Code` está habilitada, se integrará el contrato ACSL en el código fuente del programa *C* cargado [Figura 6.5], de lo contrario creará un archivo nuevo en la misma ubicación del programa *C* cargado con el mismo nombre pero agregando `_contract` al final [Figura 6.6].

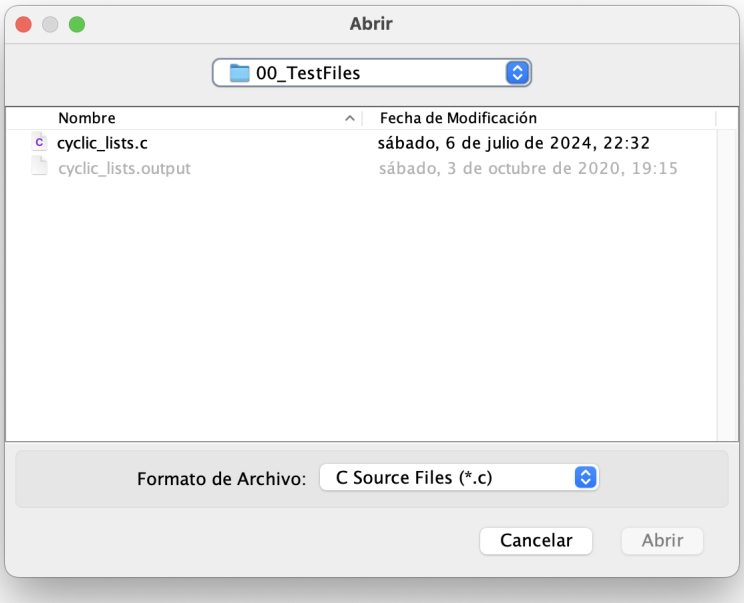

Figura 6.3: Ventana para cargar el código fuente del programa *C*

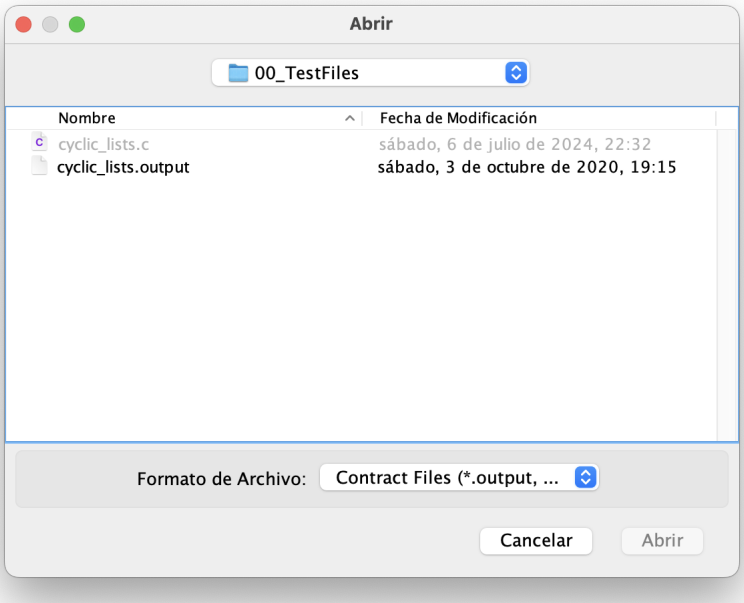

Figura 6.4: Ventana para cargar el contrato inferido

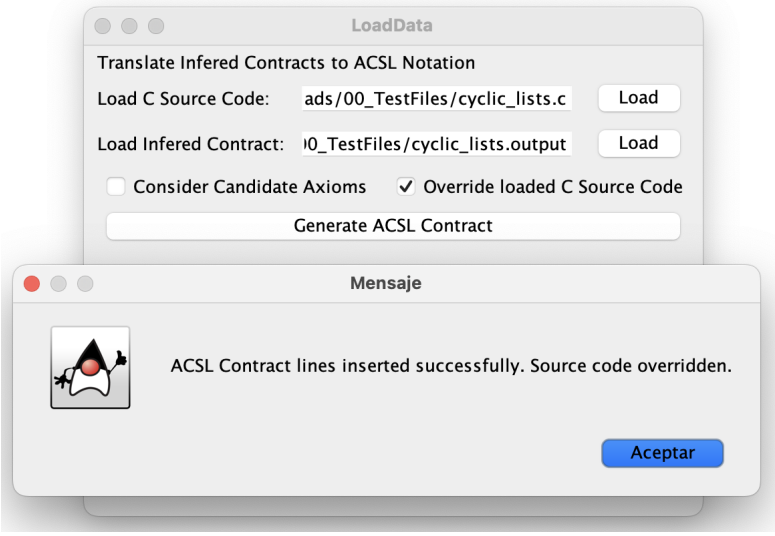

Figura 6.5: Mensaje código fuente sobrescrito

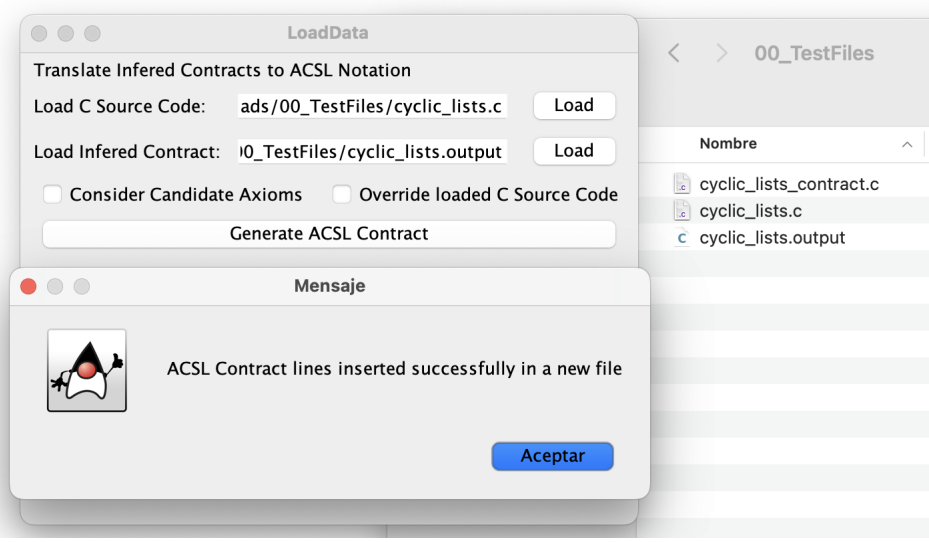

Figura 6.6: Mensaje nuevo archivo código fuente creado

Capítulo 7

Pruebas

Se han utilizado algunos de los contratos inferidos, expuestos en el anexo .4, con sus respectivos programas, expuestos en el anexo .3, obtenidos del catálogo de programas *C* de prueba de rendimiento predefinidos en *KindSpec* para probar la solución desarrollada.

7.1. Resultados

Se ejecutó el servicio utilizando los contratos inferidos y sus programas y se obtuvieron los siguientes resultados:

- Para el contrato de la función `collapseC`, en el anexo .2, del programa `cyclic_lists.c`, en el anexo .1. se obtuvo el siguiente resultado.

```
1  //@ import Pred;
2  #include <stdlib.h>
3
4  struct clist {
5   int lsize;
6   struct lnode *elems;
7  };
8
9  struct lnode {
10  int value;
11  struct lnode *next;
12 };
13
14 int isPrec(struct clist *c, struct lnode *n, struct lnode *t) {
15  . . .
16 }
17
18 /*@ requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
19   @          Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 ||
20   @          Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE ||
21   @          Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
22   @          Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 ||
23   @          Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
24   @ assigns t;
25   @ assigns n;
26   @ assigns head;
27   @ assigns head->next;
28   @ assigns c->lsize;
29   @ behavior A1:
30   @    requires Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0;
```

```
31   @    ensures Pred::lenC(c) == Pred::NPE && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result
         == Pred::NPE;
32   @ behavior A2:
33   @    requires Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1;
34   @    ensures Pred::lenC(c) == 0 && Pred::isN(c) == 0 && Pred::isE(c) == 1 && \result == 0;
35   @ behavior A3:
36   @    requires Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE;
37   @    ensures Pred::lenC(c) == 0 && Pred::isN(c) == 1 && Pred::isE(c) == Pred::NPE && \result
         == 0;
38   @ behavior A4:
39   @    requires Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
40   @    ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result == 1;
41   @ behavior A5:
42   @    requires Pred::lenC(c) == 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
43   @    ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result == 1;
44   @ behavior A6:
45   @    requires Pred::lenC(c) == 3 && Pred::isN(c) == 0 && Pred::isE(c) == 0;
46   @    ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result == 1;
47   @ behavior C1:
48   @    requires Pred::lenC(c) == ?l0 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >=
       2;
49   @    ensures Pred::lenC(c) == ?l0 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >= 2 && \
       result == 1;
50   @ behavior C2:
51   @    requires Pred::lenC(c) == ?l0 + 2 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && ?l0 >=
       2;
52   @    ensures Pred::lenC(c) == 1 && Pred::isN(c) == 0 && Pred::isE(c) == 0 && \result == 1;
53   @*/
54 int collapseC(struct clist *c) {
55  struct lnode *n;
56  struct lnode *t;
57  struct lnode *head;
58
59  if(c == NULL) return 0;
60  if(c->elems == NULL) return 0;
61
62  t = c->elems;
63  n = c->elems->next;
64
65  while (t != n && !(isPrec(c, n, t))) {
66   t = n;
67   n=n->next;
68  }
69
70  head = n;
71  n = head->next;
72  while(n != head) {
73   t = n;
74   n = n->next;
75   free(t);
76   c->lsize--;
77  }
78
79  head->next = head;
80
81  return 1;
82 }
83
84 int isN(struct clist *c) {
85  . . .
86 }
87
88 int isE(struct clist *c) {
89  . . .
90 }
91
92 int lenC(struct clist *c) {
93  . . .
94 }
```

- Para el contrato de la función `deallocate`, en el anexo .4.1, del programa `deallocate.c`, en el anexo .3.1, se obtuvo el siguiente resultado.

```
1  //@ import Pred;
2  // Copyright (c) 2014 K Team. All Rights Reserved.
3  /*
4   * Function that deallocates a singly linked list.
5   * In matchC, "." means "nothing"; technically, "." is the unit
6   * of all collection structures (lists, sets, bags, maps, etc.).
7   * By rewriting "list(x)(A)" into "." below, we mean that the list
8   * that x points to disappears after the function is applied.
9   */
10
11 #include <stdlib.h>
12
13 struct listNode
14 {
15   int val;
16   struct listNode *next;
17 };
18
19 /*@ requires Pred::length(x) == 0 && Pred::isNull(x) == 1 ||
20   @          Pred::length(x) == 3 && Pred::isNull(x) == 0 ||
21   @          Pred::length(x) == 2 && Pred::isNull(x) == 0 ||
22   @          Pred::length(x) == 1 && Pred::isNull(x) == 0;
23   @ assigns n;
24   @ assigns y;
25   @ assigns x;
26   @ behavior A1:
27   @    requires Pred::length(x) == 0 && Pred::isNull(x) == 1;
28   @    ensures Pred::length(x) == 0 && Pred::isNull(x) == 1 && \result == 0;
29   @ behavior A2:
30   @    requires Pred::length(x) == 3 && Pred::isNull(x) == 0;
31   @    ensures Pred::length(x) == 0 && Pred::isNull(x) == 1 && \result == 1;
32   @ behavior A3:
33   @    requires Pred::length(x) == 2 && Pred::isNull(x) == 0;
34   @    ensures Pred::length(x) == 0 && Pred::isNull(x) == 1 && \result == 1;
35   @ behavior A4:
36   @    requires Pred::length(x) == 1 && Pred::isNull(x) == 0;
37   @    ensures Pred::length(x) == 0 && Pred::isNull(x) == 1 && \result == 1;
38   @ behavior C1:
39   @    requires Pred::length(x) == ?l0 + 2 && Pred::isNull(x) == 0 && ?l0 >= 2;
40   @    ensures Pred::length(x) == 0 && Pred::isNull(x) == 1 && \result == 1;
41   @*/
42 int deallocate(struct listNode *x)
43 {
44   struct listNode *n;
45   if (x == NULL)
46     return 0;
47
48   n = x;
49   while (n != NULL)
50   {
51     n = n->next;
52   }
53
54   while (x->next != NULL)
55   {
56     struct listNode *y;
57
58     y = x->next;
59     free(x);
60     x = y;
61   }
62
63   free(x);
64   x = NULL;
65   return 1;
66 }
```

```
67
68 int isNull(struct listNode *x){
69   . . .
70 }
71
72 int length(struct listNode *x){
73   . . .
74 }
```

- Para el contrato de la función `delCircular`, en el anexo .4.2, del programa `delete_circular.c`, en el anexo .3.2, se obtuvo el siguiente resultado.

```
1  //@ import Pred;
2  /* deleteall() for circular lists */
3
4  #include <stdio.h>
5
6  struct lnode
7  {
8          int value;
9          struct lnode *next;
10 };
11
12 struct clist
13 {
14         int lsize;
15         struct lnode *elems;
16 };
17
18 /*@ requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == -1 && Pred::isNull(c) == 0 ||
19   @          Pred::isEmpty(c) == 1 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 0 ||
20   @          Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 1 ||
21   @          Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 3 && Pred::isNull(c) == 0 ||
22   @          Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 2 && Pred::isNull(c) == 0 ||
23   @          Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0;
24   @ assigns h;
25   @ assigns n;
26   @ assigns t;
27   @ assigns c->lsize;
28   @ assigns h->next;
29   @ behavior A1:
30   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == -1 && Pred::isNull(c) ==
         0;
31   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == -1 && Pred::isNull(c) == 0
        && \result == 0;
32   @ behavior A2:
33   @    requires Pred::isEmpty(c) == 1 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 0;
34   @    ensures Pred::isEmpty(c) == 1 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 0
        && \result == 0;
35   @ behavior A3:
36   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 1;
37   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 0 && Pred::isNull(c) == 1
        && \result == 0;
38   @ behavior A4:
39   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 3 && Pred::isNull(c) == 0;
40   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0
        && \result == 1;
41   @ behavior A5:
42   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 2 && Pred::isNull(c) == 0;
43   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0
        && \result == 1;
44   @ behavior A6:
45   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0;
46   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0
        && \result == 1;
47   @ behavior C1:
48   @    requires Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == ?l0 + 2 && Pred::isNull(c)
         == 0 && ?l0 >= 2;
```

```
49   @    ensures Pred::isEmpty(c) == 0 && Pred::lengthCircular(c) == 1 && Pred::isNull(c) == 0
         && \result == 1;
50   @*/
51 int delCircular(struct clist *c)
52 {
53         struct lnode *n;
54         struct lnode *h;
55         struct lnode *t;
56
57         if (c == NULL)
58                 return 0;
59         if (c->elems == NULL)
60                 return 0;
61
62         h = c->elems;
63         n = h->next;
64         while (n != h)
65         {
66                 if (n == NULL)
67                         return 0;
68                 n = n->next;
69         }
70
71         n = h->next;
72         while (n != h)
73         {
74                 t = n;
75                 n = n->next;
76                 free(t);
77                 c->lsize--;
78         }
79
80         h->next = h;
81         return 1;
82 }
83
84 int isNull(struct clist *c){
85         . . .
86 }
87
88 int isEmpty(struct clist *c){
89         . . .
90 }
91
92 int lengthCircular(struct clist *c){
93         . . .
94 }
```

Hemos logrado obtener en un único archivo el contrato en notación *ACSL* tal y como hemos especificado en la metodología de traducción [5].

Se intentó validar que la sintaxis para los contratos *ACSL* obtenidos utilizando el *plug-in E-ACSL* [4] de *FramaC* en la versión *29.0 (Copper)* [1] y la versión *1.2* de *ACSL* [3]. Esto no fue posible ya que se han utilizado características experimentales que aún no están implementadas en las versiones actuales, debido a que la sintaxis y semántica de estas características no están fijadas aún.

Las pruebas se ejecutaron en una MacBook Pro con procesador Apple M3 Pro de CPU de 12 núcleos y GPU de 18 núcleos, con 36 GB de RAM con la versión de Java 21.0.1. Todas las ejecuciones de traducción para todos los programas y contratos, en los anexos .3 y .4 respectivamente, se completaron en un tiempo promedio de 20 ms.

7.2. Aspectos Complejos y Desafíos

Estos aspectos subrayan la complejidad técnica y los desafíos que ha conllevado la traducción de contratos de software y su integración en un entorno de desarrollo formal, resaltando la importancia de un diseño cuidadoso y una implementación meticulosa.

1. Traducción de Formatos de Axiomas Lógicos a Lenguajes de Especificación Estándar.
   - Desafío: La traducción de contratos software desde un formato de axioma lógico a un lenguaje de especificación estándar como ACSL requiere una comprensión profunda tanto de la lógica subyacente en los axiomas como de la sintaxis y semántica del lenguaje de especificación.
   - Solución: Se deben desarrollar algoritmos que puedan interpretar correctamente los axiomas lógicos y mapearlos a construcciones equivalentes en ACSL, manteniendo la precisión y la semántica original del contrato.
2. Manejo de Comportamientos Múltiples:
   - Desafío: Los contratos inferidos por la herramienta *KindSpec* pueden contener múltiples comportamientos, lo que añade complejidad a su traducción y representación en *ACSL*.
   - Solución: Es necesario diseñar una estructura en *ACSL* que permita encapsular estos múltiples comportamientos de manera coherente, utilizando contratos con comportamientos definidos.
3. Validación y Verificación Formal:
   - Desafío: Una vez traducidos, los contratos deben ser validados y verificados formalmente para asegurar que cumplen con las especificaciones del software y que no introducen errores.
   - Solución: Desafortunadamente, debido a que se utilizan características experimentales no se ha podido realizar la validación y verificación formal de los contratos *ACSL* generados.
4. Desarrollo de la Interfaz de Usuario:
   - Desafío: Crear una interfaz de usuario intuitiva, simple y eficiente para seleccionar los archivos de contrato y código fuente, y para mostrar los resultados de la traducción y verificación.
   - Solución: Utilizar entornos de desarrollo integrados como *IntelliJ IDEA* y *frameworks* de interfaz gráfica de usuario como *Swing Ui* [14], asegurando una experiencia de usuario fluida y accesible.
5. Gestión de Errores y Excepciones:
   - Desafío: Manejar adecuadamente los posibles errores y excepciones que puedan surgir durante la carga de archivos, la traducción de contratos y la verificación del código para garantizar el funcionamiento correcto del servicio.

- Solución: Implementar mecanismos robustos de manejo de errores, incluyendo la validación de entradas, la captura de excepciones y la provisión de mensajes de error claros y útiles para el usuario.

Capítulo 8

Conclusiones y Trabajos Futuros

8.1. Conclusiones

Hay dos aspectos importantes los cuales han destacado a lo largo de este Trabajo Final de Máster (TFM). La importancia de los contratos de software ya que son fundamentales para garantizar que los programas cumplan con los requisitos especificados, mejorando la calidad y la confiabilidad del software. Los lenguajes de especificación estándar, como *ACSL*, que facilitan la documentación precisa y no ambigua de los requisitos, permitiendo un análisis formal efectivo.

En conclusión, este TFM se ha centrado en la traducción de los contratos inferidos por *KindSpec* desde un formato de axioma lógico al lenguaje de especificación estándar *ACSL*, con el objetivo de mejorar la verificación y validación del software, pasando de dos archivos que contienen el contrato inferido y el código fuente, a un solo archivo que contiene el código fuente y el contrato formal en forma de anotaciones. Este trabajo ha logrado crear una herramienta eficiente y práctica para traducir e integrar contratos de software en el código fuente de programas *C*, mejorando significativamente la fiabilidad del software y facilitando su análisis formal mediante el uso de herramientas estándares como *ACSL* y *FramaC*.

Este TFM ha demostrado con éxito la viabilidad de traducir contratos de software a la notación estándar *ACSL*. La herramienta desarrollada no solo mejora la verificación y validación del software, sino que también facilita la integración de contratos en el código fuente *C*, permitiendo un análisis formal más robusto y preciso mediante *FramaC*. Esto contribuye significativamente a la fiabilidad y seguridad del software, aspectos críticos en el desarrollo de sistemas confiables.

8.2. Trabajos Futuros

Dado que la herramienta *KindSpec* y la solución desarrollada en este TFM están el el mismo lenguaje de programación, *Java*, la solución puede utilizarse como base para una posible integración futura en *KindSpec*. Aunque se ha logrado el objetivo propuesto de traducir contratos de axioma lógico a una notación estándar, el servicio desarrollado es simplemente un cimiento para la generación de contratos completos y correcto es *ACSL*.

Este trabajo es un punto de partida que permitirá avanzar en la investigación de generación de contratos software. Aún queda trabajo por hacer para la robustez de la solución desarrollada como la optimización del proceso de traducción, mejorando los algoritmos de traducción para aumentar la precisión y eficiencia del servicio, contemplando todos los casos posibles en los contratos inferidos, reduciendo los posibles errores de traducción y generando contratos más detallados.

Debido a que obtenemos un contrato en *ACSL* el lenguaje de especificación de la herramienta *FramaC*, podríamos usar esta herramienta para una validación automatizada. Se podrían desarrollar funcionalidades adicionales para la validación automática de los contratos traducidos, reduciendo la necesidad de intervención manual y mejorando la exactitud de los contratos.

También existe la posibilidad de realizar estudios empíricos, combinando esta solución con *KindSpec* y utilizando casos de uso reales se podría evaluar el impacto de estas herramientas en proyectos de software, obteniendo retroalimentación valiosa para futuras mejoras y adaptaciones.

La solución desarrollada en este TFM depende de los contratos inferidos por *KindSpec*, es por esto que se deben enfocar los trabajos futuros en mejorar el manejo de hipótesis complejas en *KindSpec*, desarrollando métodos para su validación automática y efectiva utilizando esta solución para evitar, en la medida de lo posible, la intervención humana que es propensa a errores.

La continuación y mejora de este trabajo contribuirá significativamente a la evolución de las prácticas de verificación y validación de software, promoviendo el desarrollo de sistemas más seguros y fiables.

Bibliografía

[1] *Loïc Correnson, Pascal Cuoq, Florent Kirchner, André Maroneze, Virgile Prevosto, Armand Puccetti, Julien Signoles, Boris Yakobowski. Frama-C User Manual For Frama-C 29.0 (Copper), 2024*

[2] *Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto. ANSI/ISO C Specification Language Version 1.20, 2024*

[3] *Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto. ANSI/ISO C Specification Language Version 1.20 - Implementation in Frama-C 29.0, 2024*

[4] *Julien Signoles, Basile Desloges, Kostyantyn Vorobyov. E-ACSL Plug-in - For Frama-C 29.0 (Copper), 2024*

[5] *Virgile Prevosto. ACSL Mini-Tutorial*

[6] *Jens Gerlach. ACSL by Example, 2020*

[7] *Alpuente, María, Pardo, Daniel, and Villanueva, Alicia. Abstract Contract Synthesis and Verification in the Symbolic K Framework, 2020*

[8] *Alpuente, Maria, Villanueva, Alicia. Automated Synthesis of Software Contracts with KINDSPEC. En: Lopez-Garcia, P., Gallagher, J.P., Giacobazzi, R. (eds) Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems. Lecture Notes in Computer Science, vol 13160. Springer, Cham.*

[9] *KindSpec 2.2: A Rewriting-based, Abstract Symbolic Contract Synthesizer. https://kindspec.webs.upv.es/*

[10] *DSIC-VrAIn. KindSpec 2.2 Quick Start Guide, 2020*

[11] *Johannes Link. Unit Testing in Java, Chapter 4 - Test Ideas and Heuristics, Pages 65-90, 2004*

[12] *Java Programming Language. https://docs.oracle.com/javase/8/docs/technotes/guides/langu*

[13] *JetBrains IntelliJ IDEA. https://www.jetbrains.com/idea/*

[14] *Build UI using Swing. https://www.jetbrains.com/help/idea/design-gui-using-swing.html*

Anexos

.1. Anexo I: Código del programa cyclic_lists.c

Código del programa `cyclic_lists.c` que pertenece al conjunto de programas de prueba de rendimiento predefinidos en *KindSpec* [.3].

```
1  #include <stdlib.h>
2
3  struct clist {
4   int lsize;
5   struct lnode *elems;
6  };
7
8  struct lnode {
9   int value;
10  struct lnode *next;
11 };
12
13 int isPrec(struct clist *c, struct lnode *n, struct lnode *t) {
14  struct lnode *lPointer;
15
16  if(c == NULL || n == NULL || t == NULL) return 0;
17
18  lPointer = c->elems;
19
20  while(lPointer != t) {
21   if(lPointer == n) {
22    return 1;
23   }
24   lPointer = lPointer->next;
25  }
26
27  return 0;
28 }
29
30 int collapseC(struct clist *c) {
31  struct lnode *n;
32  struct lnode *t;
33  struct lnode *head;
34
35  if(c == NULL) return 0;
36  if(c->elems == NULL) return 0;
37
38  t = c->elems;
39  n = c->elems->next;
40
41  while (t != n && !(isPrec(c, n, t))) {
42   t = n;
43   n=n->next;
44  }
45
46  head = n;
47  n = head->next;
48  while(n != head) {
```

```
49    t = n;
50    n = n->next;
51    free(t);
52    c->lsize--;
53   }
54
55   head->next = head;
56
57   return 1;
58  }
59
60  int isN(struct clist *c) {
61   return c == NULL;
62  }
63
64  int isE(struct clist *c) {
65   return c->elems == NULL;
66  }
67
68  int lenC(struct clist *c) {
69   struct lnode *n;
70   struct lnode *t;
71   struct lnode *head;
72   int counter;
73
74   if(c == NULL) return 0;
75   if(c->elems == NULL) return 0;
76
77   t = c->elems;
78   n = c->elems->next;
79
80   while (t != n && !(isPrec(c, n, t))) {
81    t = n;
82    n=n->next;
83   }
84
85   head = n;
86   n = head->next;
87   counter = 1;
88
89   while(n != head) {
90    counter++;
91    n = n->next;
92   }
93
94   return counter;
95  }
```

.2. Anexo II: Contrato inferido de la función collapseC del programa cyclic_lists.c

Resultado del contrato inferido de la función `collapseC` del programa `cyclic_lists.c` [.1].

```
*********************************************
*********************************************
Inference performed May 26 2020
Selected modifier function: collapseC
From file: examples/cyclic_lists.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
-----------------------------------------------
PRECONDITION P:
(lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1) ||
(lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})) ||
(lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=2 ^ isN(c)=0 ^ isE(c)=0) ||
(lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=(NPE, {45}) ^ isN(c)=0 ^ isE(c)=0
    ^ ret=(NPE, {21})) ^
A2: (lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1) => (lenC(c)=0 ^ isN(c)=0 ^ isE(c)=1 ^ ret=0) ^
A3: (lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})) => (lenC(c)=0 ^ isN(c)=1 ^ isE(c)=(NPE, {32})
    ^ ret=0) ^
A4: (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1) ^
A5: (lenC(c)=2 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1) ^
A6: (lenC(c)=3 ^ isN(c)=0 ^ isE(c)=0) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^ ret=1)
-----------------------------------------------
-----------------------------------------------
LOCATIONS L:
t
n
head
head->next
c->lsize
-----------------------------------------------
-----------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (lenC(c)=?l0 + 2 ^ isN(c)=0 ^ isE(c)=0 ^ ?l0 >= 2) => (lenC(c)=?l0 ^ isN(c)=0 ^ isE(c)=0 ^
     ?l0 >= 2 ^ ret=1) ^
C2: (lenC(c)=?l0 + 2 ^ isN(c)=0 ^ isE(c)=0 ^ ?l0 >= 2) => (lenC(c)=1 ^ isN(c)=0 ^ isE(c)=0 ^
    ret=1)
-----------------------------------------------
```

.3. Anexo III: Catálogo de programas C

Conjunto de programas de prueba de rendimiento predefinidos en *KindSpec* utilizados para definir la librería de predicados y realizar pruebas al servicio de traducción de contratos.

.3.1. deallocate.c

```
1  // Copyright (c) 2014 K Team. All Rights Reserved.
2  /*
3   * Function that deallocates a singly linked list.
4   * In matchC, "." means "nothing"; technically, "." is the unit
5   * of all collection structures (lists, sets, bags, maps, etc.).
6   * By rewriting "list(x)(A)" into "." below, we mean that the list
7   * that x points to disappears after the function is applied.
8   */
9
10 #include <stdlib.h>
11
12 struct listNode
13 {
14   int val;
15   struct listNode *next;
16 };
17
18 int deallocate(struct listNode *x)
19 {
20   struct listNode *n;
21   if (x == NULL)
22     return 0;
23
24   n = x;
25   while (n != NULL)
26   {
27     n = n->next;
28   }
29
30   while (x->next != NULL)
31   {
32     struct listNode *y;
33
34     y = x->next;
35     free(x);
36     x = y;
37   }
38
39   free(x);
40   x = NULL;
41   return 1;
42 }
43
44 int isNull(struct listNode *x)
45 {
46   if (x == NULL)
47     return 1;
48   return 0;
49 }
50
51 int length(struct listNode *x)
52 {
53   int count;
54   struct listNode *y;
55   if (x == NULL)
56     return 0;
57
58   count = 0;
59   y = x;
```

```
60   while (y != NULL)
61   {
62     y = y->next;
63     count = count + 1;
64   }
65   return count;
66 }
```

.3.2. delete_circular.c

```
1  /* deleteall() for circular lists */
2
3  #include <stdio.h>
4
5  struct lnode
6  {
7          int value;
8          struct lnode *next;
9  };
10
11 struct clist
12 {
13         int lsize;
14         struct lnode *elems;
15 };
16
17 int delCircular(struct clist *c)
18 {
19         struct lnode *n;
20         struct lnode *h;
21         struct lnode *t;
22
23         if (c == NULL)
24                 return 0;
25         if (c->elems == NULL)
26                 return 0;
27
28         h = c->elems;
29         n = h->next;
30         while (n != h)
31         {
32                 if (n == NULL)
33                         return 0;
34                 n = n->next;
35         }
36
37         n = h->next;
38         while (n != h)
39         {
40                 t = n;
41                 n = n->next;
42                 free(t);
43                 c->lsize--;
44         }
45
46         h->next = h;
47         return 1;
48 }
49
50 int isNull(struct clist *c)
51 {
52         return c == NULL;
53 }
54
55 int isEmpty(struct clist *c)
56 {
57         return c != NULL && c->elems == NULL;
```

```
58 }
59
60 int lengthCircular(struct clist *c)
61 {
62         struct lnode *n;
63         struct lnode *h;
64         int counter;
65
66         if (c == NULL)
67                 return 0;
68         if (c->elems == NULL)
69                 return 0;
70
71         h = c->elems;
72         n = h->next;
73         counter = 1;
74
75         while (n != h)
76         {
77                 if (n == NULL)
78                         return -1;
79                 counter++;
80                 n = n->next;
81         }
82
83         return counter;
84 }
```

.3.3. insert.c

```
1  #include <stdlib.h>
2
3  struct lnode
4  {
5          int value;
6          struct lnode *next;
7  };
8
9  struct set
10 {
11         int capacity;
12         int lsize;
13         struct lnode *elems;
14 };
15
16 int insert(struct set *s, int x)
17 {
18         struct lnode *new_node;
19         struct lnode *n;
20         int found;
21
22         if (s == NULL)
23                 return 0; /* NULL set */
24
25         if (s->lsize >= s->capacity)
26                 return 0; /* no space left */
27
28         if (s->elems == NULL)
29         { /* empty set */
30                 new_node = (struct lnode *)malloc(sizeof(struct lnode));
31                 if (new_node == NULL)
32                         return 0; /* no memory left */
33                 new_node->value = x;
34                 new_node->next = NULL;
35
36                 s->elems = new_node;
37                 s->lsize = 1;
```

```
38
39                  return 1;
40          }
41
42          n = s->elems;
43          found = 0;
44          while (n != NULL)
45          {
46                  if (n->value == x)
47                  {
48                          found = 1;
49                  }
50                  n = n->next;
51          }
52
53          if (found)
54          {
55                  return 0; /* element already in the set */
56          }
57
58          /* Creation of new node */
59          new_node = (struct lnode *)malloc(sizeof(struct lnode));
60          if (new_node == NULL)
61                  return 0; /* no memory left */
62          new_node->value = x;
63          new_node->next = NULL;
64
65          n = s->elems;
66          new_node->next = n;
67          s->elems = new_node;
68          s->lsize = s->lsize + 1;
69
70          return 1; /* element added */
71  }
72
73  int isNull(struct set *s)
74  {
75          if (s == NULL)
76                  return 1;
77          return 0;
78  }
79
80  int isEmpty(struct set *s)
81  {
82          if (s == NULL)
83                  return 0;
84          if (s->elems == NULL)
85                  return 1; /* s is empty */
86          return 0;
87  }
88
89  int isFull(struct set *s)
90  {
91          if (s == NULL)
92                  return 0;
93          if (s->lsize >= s->capacity)
94                  return 1; /* s is full */
95          return 0;
96  }
97
98  int contains(struct set *s, int x)
99  {
100         struct lnode *n;
101
102         if (s == NULL)
103                 return 0; /* s is NULL */
104
105         n = s->elems;
106         while (n != NULL)
107         {
```

```
108                     if (n->value == x)
109                             return 1; /* element found */
110                     n = n->next;
111             }
112
113             return 0; /* element NOT found */
114     }
115
116     int length(struct set *s)
117     {
118             struct lnode *n;
119             int count;
120
121             if (s == NULL)
122                     return 0; /* s is NULL */
123
124             count = 0;
125             n = s->elems;
126             while (n != NULL)
127             {
128                     count = count + 1;
129                     n = n->next;
130             }
131
132             return count;
133     }
```

.3.4. insert_exceptions.c

```
1   #include <stdlib.h>
2
3   struct lnode
4   {
5           int value;
6           struct lnode *next;
7   };
8
9   struct set
10  {
11          int capacity;
12          int lsize;
13          struct lnode *elems;
14  };
15
16  int insert(struct set *s, int x)
17  {
18          struct lnode *new_node;
19          struct lnode *n;
20          int found;
21
22          if (s == NULL)
23                  return 0; /* NULL set */
24          if (s->elems == NULL)
25          { /* empty set */
26                  new_node->value = x;
27                  new_node->next = NULL;
28                  s->elems = new_node;
29                  s->lsize = 1;
30                  return 1;
31          }
32
33          n = s->elems;
34          found = 0;
35          while (n != NULL)
36          {
37                  if (n->value == x)
38                  {
```

```
39                          found = 1;
40                  }
41                  n = n->next;
42          }
43          if (found)
44          {
45                  return 0; /* element already in the set */
46          }
47
48          /* Creation of new node */
49          new_node = (struct lnode *)malloc(sizeof(struct lnode));
50          new_node->value = x;
51          new_node->next = NULL;
52          n = s->elems;
53          new_node->next = n;
54          s->elems = new_node;
55          s->lsize = s->lsize + 1;
56          return 1; /* element added */
57  }
58
59  int isNull(struct set *s)
60  {
61          return s == NULL;
62  }
63
64  int isEmpty(struct set *s)
65  {
66          return s->elems == NULL;
67  }
68
69  int isFull(struct set *s)
70  {
71          return s->lsize >= s->capacity;
72  }
73
74  int contains(struct set *s, int x)
75  {
76          struct lnode *n;
77          if (s == NULL)
78                  return 0; /* s is NULL */
79          n = s->elems;
80          while (n != NULL)
81          {
82                  if (n->value == x)
83                          return 1; /* element found */
84                  n = n->next;
85          }
86          return 0; /* element NOT found */
87  }
88
89  int length(struct set *s)
90  {
91          struct lnode *n;
92          int count;
93
94          if (s == NULL)
95                  return 0; /* s is NULL */
96
97          count = 0;
98          n = s->elems;
99          while (n != NULL)
100         {
101                 count = count + 1;
102                 n = n->next;
103         }
104
105         return count;
106 }
```

.3.5. reverse.c

```c
#include <stdlib.h>

struct listNode
{
        int val;
        struct listNode *next;
};

struct listElems
{
        struct listNode *elems;
};

int reverse(struct listElems *l)
{
        struct listNode *n;
        struct listNode *maux;
        struct listNode *p;

        if (l == NULL)
        {
                return 0; /* l is NULL */
        }

        if (l->elems == NULL)
        {
                return 0;
        }

        maux = l->elems;
        p = NULL;
        while (maux != NULL)
        {
                n = maux->next;
                maux->next = p;
                p = maux;
                maux = n;
        }

        l->elems = p;
        return 1; /* list reversed successfully */
}

struct listNode *firstE(struct listElems *l)
{
        return l->elems;
}

struct listNode *lastE(struct listElems *l)
{
        struct listNode *n;

        if (l == NULL)
        {
                return NULL;
        }
        if (l->elems == NULL)
                return NULL;

        n = l->elems;
        while (n->next != NULL)
        {
                n = n->next;
        }

        return n;
}

```

```
69 int isNull(struct listElems *l)
70 {
71         if (l == NULL)
72                 return 1;
73 
74         if (l->elems == NULL)
75                 return 1;
76         return 0;
77 }
78 
79 int length(struct listElems *l)
80 {
81         struct listNode *n;
82         int count;
83 
84         if (l == NULL)
85                 return 0; // s is NULL
86         if (l->elems == NULL)
87                 return 0;
88 
89         count = 0;
90         n = l->elems;
91         while (n != NULL)
92         {
93                 count = count + 1;
94                 n = n->next;
95         }
96 
97         return count;
98 }
```

.4. Anexo IV: Catálogo de los contratos inferidos de los programas C

Contratos inferidos con *KindSpec* para cada uno de los programas del catálogo de programas de prueba de rendimiento predefinidos [.3].

.4.1. deallocate

Contrato inferido para la función `deallocate` del programa `deallocate.c` [.3.1].

```
*********************************************
*********************************************
Inference performed May 26 2020
Selected modifier function: deallocate
From file: examples/deallocate.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
-------------------------------------------------
PRECONDITION P:
(length(x)=0 ^ isNull(x)=1) ||
(length(x)=3 ^ isNull(x)=0) ||
(length(x)=2 ^ isNull(x)=0) ||
(length(x)=1 ^ isNull(x)=0)
-------------------------------------------------
-------------------------------------------------
POSTCONDITION Q:
A1: (length(x)=0 ^ isNull(x)=1) => (length(x)=0 ^ isNull(x)=1 ^ ret=0) ^
A2: (length(x)=3 ^ isNull(x)=0) => (length(x)=0 ^ isNull(x)=1 ^ ret=1) ^
A3: (length(x)=2 ^ isNull(x)=0) => (length(x)=0 ^ isNull(x)=1 ^ ret=1) ^
A4: (length(x)=1 ^ isNull(x)=0) => (length(x)=0 ^ isNull(x)=1 ^ ret=1)
-------------------------------------------------
-------------------------------------------------
LOCATIONS L:
n
y
x
-------------------------------------------------
-------------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (length(x)=?l0 + 2 ^ isNull(x)=0 ^ ?l0 >= 2) => (length(x)=0 ^ isNull(x)=1 ^ ret=1)
-------------------------------------------------
```

.4.2. delCircular

Contrato inferido para la función `delCircular` del programa `delete_circular.c` [.3.2].

```
*********************************************
*********************************************
Inference performed April 17 2020
Selected modifier function: delCircular
From file: /Users/alicia/OtherRepos/safe-tools/synthetic/tools/KindSpec2Sources/
    LocalLimitadaKernelC/KindSpecKernelC/build/examples/delete_circular.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
```

```
-----------------------------------------------
PRECONDITION P:
(isEmpty(c)=0 ^ lengthCircular(c)=-1 ^ isNull(c)=0) ||
(isEmpty(c)=1 ^ lengthCircular(c)=0 ^ isNull(c)=0) ||
(isEmpty(c)=0 ^ lengthCircular(c)=0 ^ isNull(c)=1) ||
(isEmpty(c)=0 ^ lengthCircular(c)=3 ^ isNull(c)=0) ||
(isEmpty(c)=0 ^ lengthCircular(c)=2 ^ isNull(c)=0) ||
(isEmpty(c)=0 ^ lengthCircular(c)=1 ^ isNull(c)=0)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (isEmpty(c)=0 ^ lengthCircular(c)=-1 ^ isNull(c)=0) => (isEmpty(c)=0 ^ lengthCircular(c)
    =-1 ^ isNull(c)=0 ^ ret=0) ^
A2: (isEmpty(c)=1 ^ lengthCircular(c)=0 ^ isNull(c)=0) => (isEmpty(c)=1 ^ lengthCircular(c)=0
    ^ isNull(c)=0 ^ ret=0) ^
A3: (isEmpty(c)=0 ^ lengthCircular(c)=0 ^ isNull(c)=1) => (isEmpty(c)=0 ^ lengthCircular(c)=0
    ^ isNull(c)=1 ^ ret=0) ^
A4: (isEmpty(c)=0 ^ lengthCircular(c)=3 ^ isNull(c)=0) => (isEmpty(c)=0 ^ lengthCircular(c)=1
    ^ isNull(c)=0 ^ ret=1) ^
A5: (isEmpty(c)=0 ^ lengthCircular(c)=2 ^ isNull(c)=0) => (isEmpty(c)=0 ^ lengthCircular(c)=1
    ^ isNull(c)=0 ^ ret=1) ^
A6: (isEmpty(c)=0 ^ lengthCircular(c)=1 ^ isNull(c)=0) => (isEmpty(c)=0 ^ lengthCircular(c)=1
    ^ isNull(c)=0 ^ ret=1)
-----------------------------------------------
-----------------------------------------------
LOCATIONS L:
h
n
t
c->lsize
h->next
-----------------------------------------------
-----------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (isEmpty(c)=0 ^ lengthCircular(c)=?l0 + 2 ^ isNull(c)=0 ^ ?l0 >= 2) => (isEmpty(c)=0 ^
    lengthCircular(c)=1 ^ isNull(c)=0 ^ ret=1)
-----------------------------------------------
```

.4.3. insert

Contrato inferido para la función `insert` del programa `insert.c` [.3.3].

```
*********************************************
*********************************************
Inference performed May 26 2020
Selected modifier function: insert
From file: examples/insert_length_counting.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
-----------------------------------------------
PRECONDITION P:
(isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isFull(s)=0 ^ isNull(s)=0) ||
(isFull(s)=1 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=1)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=0 ^ isNull(s)=0) => (isEmpty(s)
    =0 ^ length(s)=1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
```

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

.4.4. insert

Contrato inferido para la funcieon `insert` del programa `insert_exceptions.c` [.3.4].

```
*********************************************
*********************************************
Inference performed June 1 2020
Selected modifier function: insert
From file: examples/insert_exceptions.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
-----------------------------------------------
PRECONDITION P:
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isNull(s)=0) ||
(isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isNull(s)=0) ||
(isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isNull(s)=0) ||
(isEmpty(s)=(NPE, {29}) ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=(NPE, {30}) ^ isNull(s)=1)
-----------------------------------------------
-----------------------------------------------
POSTCONDITION Q:
A1: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)
    =2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A2: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=0 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)
    =3 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=1) ^
A3: (isEmpty(s)=0 ^ length(s)=2 ^ contains(s,x)=1 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)
    =2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=0) ^
```

```
A4: (isEmpty(s)=0 ^ length(s)=1 ^ contains(s,x)=1 ^ isNull(s)=0) => (isEmpty(s)=0 ^ length(s)
    =1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ret=0) ^
A5: (isEmpty(s)=1 ^ length(s)=0 ^ contains(s,x)=0 ^ isNull(s)=0) => (isEmpty(s)=1 ^ length(s)
    =0 ^ contains(s,x)=0 ^ isNull(s)=0 ^ ret=(UMR, {7})) ^
A6: (isEmpty(s)=(NPE, {29}) ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=(NPE, {30}) ^ isNull(s
    )=1) => (isEmpty(s)=(NPE, {29}) ^ length(s)=0 ^ contains(s,x)=0 ^ isFull(s)=(NPE, {30}) ^
    isNull(s)=1 ^ ret=0)
-------------------------------------------------
-------------------------------------------------
LOCATIONS L:
n
found
new_node
new_node->value
new_node->next
s->elems
s->lsize
-------------------------------------------------
-------------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=0 ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty(
    s)=0 ^ length(s)=?l0 + 2 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=1) ^
C2: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty(
    s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=1 ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0) ^
C3: (isEmpty(s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^ isNull(s)=0 ^ ?l0 >= 2) => (isEmpty
    (s)=0 ^ length(s)=?l0 + 1 ^ contains(s,x)=?c ^ isNull(s)=0 ^ ?l0 >= 2 ^ ret=0)
-------------------------------------------------
```

.4.5. reverse

Contrato inferido para la función `reverse` del programa `reverse.c` [.3.5].

```
*********************************************
*********************************************
Inference performed May 12 2020
Selected modifier function: reverse
From file: examples/reverse.c
*********************************************
*********************************************

*********************************************
RESULTING INFERRED CONTRACT
*********************************************
-------------------------------------------------
PRECONDITION P:
(lastE(l)=null ^ length(l)=0 ^ firstE(l)=null ^ isNull(l)=1) ||
(lastE(l)=null ^ length(l)=0 ^ firstE(l)=(NPE, {17}) ^ isNull(l)=1) ||
(lastE(l)=pointer((((((* l).elems).next).next).next)) ^ length(l)=4 ^ firstE(l)=pointer(((* l)
    .elems)) ^ isNull(l)=0) ||
(lastE(l)=pointer(((((* l).elems).next).next)) ^ length(l)=3 ^ firstE(l)=pointer(((* l).elems)
    ) ^ isNull(l)=0) ||
(lastE(l)=pointer((((* l).elems).next)) ^ length(l)=2 ^ firstE(l)=pointer(((* l).elems)) ^
    isNull(l)=0) ||
(lastE(l)=pointer(((* l).elems)) ^ length(l)=1 ^ firstE(l)=pointer(((* l).elems)) ^ isNull(l)
    =0)
-------------------------------------------------
-------------------------------------------------
POSTCONDITION Q:
A1: (lastE(l)=null ^ length(l)=0 ^ firstE(l)=null ^ isNull(l)=1) => (lastE(l)=null ^ length(l)
    =0 ^ firstE(l)=null ^ isNull(l)=1 ^ ret=0) ^
A2: (lastE(l)=null ^ length(l)=0 ^ firstE(l)=(NPE, {17}) ^ isNull(l)=1) => (lastE(l)=null ^
    length(l)=0 ^ firstE(l)=(NPE, {17}) ^ isNull(l)=1 ^ ret=0) ^
A3: (lastE(l)=pointer((((((* l).elems).next).next).next)) ^ length(l)=4 ^ firstE(l)=pointer
    (((* l).elems)) ^ isNull(l)=0) => (lastE(l)=pointer(((* l).elems)) ^ length(l)=4 ^ firstE(
    l)=pointer((((((* l).elems).next).next).next)) ^ isNull(l)=0 ^ ret=1) ^
A4: (lastE(l)=pointer(((((* l).elems).next).next)) ^ length(l)=3 ^ firstE(l)=pointer(((* l).
    elems)) ^ isNull(l)=0) => (lastE(l)=pointer(((* l).elems)) ^ length(l)=3 ^ firstE(l)=
    pointer(((((* l).elems).next).next)) ^ isNull(l)=0 ^ ret=1) ^
```

```
A5: (lastE(l)=pointer((((* l).elems).next)) ^ length(l)=2 ^ firstE(l)=pointer(((* l).elems)) ^
     isNull(l)=0) => (lastE(l)=pointer(((* l).elems)) ^ length(l)=2 ^ firstE(l)=pointer((((* l
    ).elems).next)) ^ isNull(l)=0 ^ ret=1) ^
A6: (lastE(l)=pointer(((* l).elems)) ^ length(l)=1 ^ firstE(l)=pointer(((* l).elems)) ^ isNull
    (l)=0) => (lastE(l)=pointer(((* l).elems)) ^ length(l)=1 ^ firstE(l)=pointer(((* l).elems)
    ) ^ isNull(l)=0 ^ ret=1)
-------------------------------------------------
-------------------------------------------------
LOCATIONS L:
maux
p
n
maux->next
l->elems
-------------------------------------------------
-------------------------------------------------
CANDIDATE AXIOMS Q#:
C1: (lastE(l)=pointer(((((((* l).elems).next).next).next).next)) ^ length(l)=?l0 + 3 ^ firstE(
    l)=pointer(((* l).elems)) ^ isNull(l)=0 ^ ?l0 >= 2) => (lastE(l)=pointer(((* l).elems)) ^
    length(l)=?l0 + 3 ^ firstE(l)=pointer(((((((* l).elems).next).next).next).next)) ^ isNull(
    l)=0 ^ ?l0 >= 2 ^ ret=1)
-------------------------------------------------
```

.5. Anexo V: Librería de predicados en notación ACSL completa

A continuación, la librería completa con todos los elementos antes definidos, en un módulo de especificación de *ACSL* (`definedPredicates.acsl`). Esta es la librería que se utilizará para validar los predicados obtenidos en la traducción de los contratos inferidos.

```
/*  @ // Author: Luis Carlo Rodriguez Jose
    @ // Created: 2024-07
    @
    @ module Pred {
    @ // Error codes for known exceptions
    @   // ----- Error codes ------------------------
    @   type errorCode = int;
    @   logic errorCode NPE = -1; // Null Pointer Error
    @   logic errorCode DBZ = -2; // Division By Zero
    @   logic errorCode VVA = -3; // Void Value Access
    @   logic errorCode NMS = -4; // Non-valid Malloc Size
    @   logic errorCode NOD = -5; // Null Object Destruction
    @   logic errorCode UMA = -6; // Undefined Memory Access
    @   logic errorCode OOS = -7; // Out Of Scope
    @   logic errorCode IRT = -8; // Incorrect Return Type
    @   logic errorCode IAT = -9; // Incompatible Assign Types
    @   logic errorCode NEF = -10; // Non-Existing Function
    @   logic errorCode UAC = -11; // Unsuitable Call Arguments
    @
    @
    @ // Predicates for list nodes operations
    @   // ----- List Nodes -------------------------
    @   type listNode = struct {
    @      int val; // Value of the node
    @      struct listNode *next; // Pointer to the next node
    @   };
    @
    @   logic integer isNull(listNode *lst) =
    @       (lst == \null) ? 1 : 0;
    @
    @   logic integer length(listNode *lst) =
    @       (lst == \null) ? 0 : 1 + length(lst->next);
    @
    @   logic integer firstElem(listNode *lst) =
    @       (isNull(lst) == 1) ? NPE : lst->val;
    @
    @   logic integer lastElem(listNode *lst) =
    @       (isNull(lst) == 1) ? NPE : (isNull(lst->next) == 1) ? lst->val : lastElem(lst->
        next);
    @
    @   // Predicates for list operations
    @   // ----- Lists ------------------------------
    @   type list = struct {
    @       listNode *elems; // Pointer to the first element of the list
    @   };
    @
    @   logic integer isNull(list *lst) =
    @       (lst == \null || isNull(lst->elems) == 1) ? 1 : 0;
    @
    @   logic integer length(list *lst) =
    @       (isNull(lst) == 1) ? 0 : length(lst->elems);
    @
    @   logic integer firstE(list *lst) =
    @       (isNull(lst) == 1) ? NPE : firstElem(lst->elems);
    @
    @   logic integer lastE(list *lst) =
    @       (isNull(lst) == 1) ? NPE : lastElem(lst->elems);
    @
    @ // Predicates for cyclic lists operations
```

```
59      @   // ----- Cyclic Lists ------------------------
60      @   type cycList = struct {
61      @       int lsize; // Size of the list
62      @       listNode *elems; // Pointer to the first element of the list
63      @   };
64      @
65      @   logic integer isN(cycList *c) =
66      @       (c == \null) ? 1 : 0;
67      @
68      @   logic integer isE(cycList *c) =
69      @       (isN(c) == 1) ? NPE : isNull(c->elems);
70      @
71      @   logic integer isEmpty(cycList *c) =
72      @       (isN(c) == 1) ? 0 : isNull(c->elems);
73      @
74      @   // Helper predicate to check if a node is in the cycle before another node
75      @   //  checks if node n precedes node t in the circular list c.
76      @   logic integer isPrec(cycList *c, listNode *n, listNode *t) =
77      @       \exists listNode *p; \let head = c->elems; isNull(head) == 0 && isNull(t) == 0 &&
78      @       (\forall listNode *p; (isNull(p) == 0 && p != t); p = p->next; p != head && p != n
        );
79      @
80      @   logic integer lenC(cycList *c) =
81      @       (isN(c) == 1 || isE(c) == 1) ? NPE :
82      @       \let t = c->elems, n = c->elems->next;
83      @       \let head = (\exists listNode *p; t != n && isNull(n) == 0 && isPrec(c, n, t) ==
        0; n = n->next, t = n);
84      @       (isNull(head) == 1) ? NPE :
85      @       (n = head->next, \let counter = 1; \forall listNode* p; p != head && isNull(p) ==
        0; p = n->next; counter++);
86      @       counter;
87      @
88      @   logic integer lengthCircular(cycList *c) =
89      @       (isN(c) == 1 || isE(c) == 1) ? 0 :
90      @       \let h = c->elems, n = h->next;
91      @       \let counter = 1;
92      @       \let result = (n == h) ? counter :
93      @           \exists listNode *p; \let current = n;
94      @           \forall listNode *p; (current != h && isNull(current) == 0);
95      @               current = current->next;
96      @               counter++;
97      @           (isNull(current) == 1) ? -1 : counter;
98      @       result;
99      @
100     @ // Predicates for lists sets operations
101     @   // ----- Set Lists --------------------------
102     @   type setList = struct {
103     @       int capacity; // Maximum capacity of the list
104     @       int lsize; // Size of the list
105     @       listNode *elems; // Pointer to the first element of the list
106     @   };
107     @
108     @   logic integer isNull(setList *s) =
109     @       (s == \null) ? 1 : 0;
110     @
111     @   logic integer isEmpty(setList *s) =
112     @       (isNull(s) == 1) ? NPE : isNull(s->elems);
113     @
114     @   logic integer isFull(setList *s) =
115     @       (isNull(s) == 1) ? NPE :
116     @       (s->lsize >= s->capacity) ? 1 : 0;
117     @
118     @   logic integer contains(setList *s, int x) =
119     @       (isNull(s) == 1) ? 0 :
120     @       \exists listNode *n; n == s->elems &&
121     @       (\forall listNode *p; isNull(p) == 0 && p->value != x; p = p->next; p == n) &&
122     @       (isNull(n) == 0 && n->value == x) ? 1 : 0;
123     @
124     @   logic integer length(setList *s) =
125     @       (isNull(s) == 1) ? 0 :
```

```
126     @       \let n = s->elems;
127     @       \let counter = 0;
128     @       \forall listNode *p; isNull(p) == 0;
129     @           p = p->next;
130     @           counter++;
131     @       counter;
132     @ }
133     @*/
```

.6. Anexo VI: Código ACLSContractGeneratorService

Código completo del servicio de que genera contratos con comportamientos definidos en formato *ACSL* a partir de un contrato inferido con formato de axiomas lógicos obtenido de la herramienta *KindSpec*

```
1  import java.io.BufferedReader;
2  import java.io.FileReader;
3  import java.io.IOException;
4  import java.nio.file.Files;
5  import java.nio.file.Paths;
6  import java.util.ArrayList;
7  import java.util.List;
8  import java.util.regex.Matcher;
9  import java.util.regex.Pattern;
10
11 /*
12  * ACLS Named Behaviour Contract Generator Service
13  * Description: Service to generate ACSL named behaviour contract lines from an inferred
      contract file
14  *
15  * Author: Luis Carlo Rodriguez Jose
16  * Date: 2024-07
17  */
18
19 public class ACLSContractGeneratorService {
20 /* ------------------------------------------------------------------
21  * ------------------- Public Service Variables --------------------
22  * ------------------------------------------------------------------
23  */
24     // Flag to check if the inferred contract file has been loaded
25     public Boolean fileLoaded = false;
26     // List to store the generated ACSL lines
27     public List<String> acslContractLines = new ArrayList<>();
28     // Name of the modifier function of the inferred contract file
29     public String modifierFunction = null;
30
31 /* ------------------------------------------------------------------
32  * ------------------- Private Service Variables -------------------
33  * ------------------------------------------------------------------
34  */
35     // List to store POSTCONDITION Q lines of the inferred contract file
36     private List<String> postconditionQLines = new ArrayList<>();
37     // List to store LOCATIONS L lines of the inferred contract file
38     private List <String> locationsLLines = new ArrayList<>();
39     // List to store CANDIDATE AXIOMS Q# lines of the inferred contract file
40     private List<String> candidateAxiomsQSharpLines = new ArrayList<>();
41     // List to store preconditions (p in p=>q) of the POSTCONDITION Q lines
42     private List<String> preconditionsP = new ArrayList<>();
43     // List to store postconditions (q in p=>q) of the POSTCONDITION Q lines
44     private List<String> postconditionsQ = new ArrayList<>();
45     // List to store the preconditions (p in p=>q) of the CANDIDATE AXIOMS Q# lines
46     private List<String> preconditionsPSharp = new ArrayList<>();
47     // List to store the postconditions (q in p=>q) of the CANDIDATE AXIOMS Q# lines
48     private List<String> postconditionsQSharp = new ArrayList<>();
49     // Regex to match the equality pattern
```

```
50      private String equalityRegex = "(?<![<>!])=";
51
52  /* ------------------------------------------------------------------------
53   * ------------------- Private Service Testing Functions -------------------
54   * ------------------------------------------------------------------------
55   */
56      /*
57       * Main Function (Test)
58       * Description: Main function to test the service
59       * */
60      public static void main(String[] args) {
61          String filePath = "/Users/lucarojo/Downloads/00_TestFiles";
62          String fileFolder = "/00";
63          String fileContractName = "/cyclic_lists.output";
64          String fileSourceCodeName = "/cyclic_lists.c";
65          ACLSContractGeneratorService service = new ACLSContractGeneratorService();
66          service.fileLoaded = service.loadContractFile(filePath + fileFolder + fileContractName
                );
67          if(service.fileLoaded) {
68              service.acslContractLines = service.generateACSLContract(true);
69              // Generate ACSL lines from the inferred contract file
70              if(!service.acslContractLines.isEmpty()) {
71                  System.out.print("ACSL Lines generated: \n");
72                  for (int i = 0; i < service.acslContractLines.size(); i++) {
73                      System.out.print(service.acslContractLines.get(i));
74                  }
75              }
76              // Insert ACSL contract lines into the source code
77              try {
78                  boolean success = service.insertACSLContractLines(service.acslContractLines,
                        service.modifierFunction, filePath + fileFolder + fileSourceCodeName,
                        false);
79                  if(success)
80                      System.out.print("ACSL Contract lines inserted successfully.\n");
81                  else
82                      System.out.print("ACSL Contract lines not inserted.\n");
83              } catch (IOException e) {
84                  e.printStackTrace();
85              }
86          }
87
88      }
89
90  /* ------------------------------------------------------------------------
91   * ------------------- Public Service Functions -------------------
92   * ------------------------------------------------------------------------
93   */
94      /*
95       * Load Contract File
96       * Description: Function to load the inferred contract file
97       * Parameters:
98       *  - filePath: String with the path to the inferred contract file
99       * Return:
100      * - Boolean with the result of the operation (file loaded or not)
101      * */
102     public boolean loadContractFile(String filePath) {
103         try {
104             // Extract the modifier function from the inferred contract file
105             this.modifierFunction = extractModifierFunction(filePath);
106             // Extract the POSTCONDITION Q lines from the inferred contract file
107             this.postconditionQLines = extractPostconditionQLines(filePath);
108             // Extract the preconditions (p in p=>q) from the POSTCONDITION Q lines
109             this.preconditionsP = extractPreconditionsP(this.postconditionQLines);
110             // Extract the postconditions (q in p=>q) from the POSTCONDITION Q lines
111             this.postconditionsQ = extractPostconditionsQ(this.postconditionQLines);
112             // Extract the LOCATIONS L lines from the inferred contract file
113             this.locationsLLines = extractLocationsLLines(filePath);
114             // Extract the CANDIDATE AXIOMS Q# lines from the inferred contract file
115             this.candidateAxiomsQSharpLines = extractCandidateAxiomsQSharpLines(filePath);
116             // Extract the preconditions (p in p=>q) from the CANDIDATE AXIOMS Q# lines
```

```
117             this.preconditionsPSharp = extractPreconditionsP(this.candidateAxiomsQSharpLines);
118             // Extract the postconditions (q in p=>q) from the CANDIDATE AXIOMS Q# lines
119             this.postconditionsQSharp = extractPostconditionsQ(this.candidateAxiomsQSharpLines
                    );
120             // Print the extracted information
121             System.out.print("Modifier function loaded: " + this.modifierFunction + "\n");
122             // Print the extracted POSTCONDITION Q lines
123             if(!this.postconditionQLines.isEmpty()) {
124                 System.out.print("POSTCONDITION Q Lines loaded: \n");
125                 for (String postconditionQLine : this.postconditionQLines) {
126                     System.out.print(postconditionQLine + "\n");
127                 }
128             }
129             // Print the extracted LOCATIONS L lines
130             if(!this.locationsLLines.isEmpty()) {
131                 System.out.print("LOCATIONS L Lines loaded: \n");
132                 for (String locationsLLine : this.locationsLLines) {
133                     System.out.print(locationsLLine + "\n");
134                 }
135             }
136             // Print the extracted CANDIDATE AXIOMS Q# lines
137             if(!this.candidateAxiomsQSharpLines.isEmpty()) {
138                 System.out.print("CANDIDATE AXIOMS Q# Lines loaded: \n");
139                 for (String candidateAxiomsQSharpLine : this.candidateAxiomsQSharpLines) {
140                     System.out.print(candidateAxiomsQSharpLine + "\n");
141                 }
142             }
143             // Print the extracted preconditions (p in p=>q)
144             if(!this.preconditionsP.isEmpty()) {
145                 System.out.print("PRECONDITION P loaded: \n");
146                 for (String s : this.preconditionsP) {
147                     System.out.print(s + "\n");
148                 }
149             }
150             // Print the extracted postconditions (q in p=>q)
151             if(!this.postconditionsQ.isEmpty()) {
152                 System.out.print("POSTCONDITION Q loaded: \n");
153                 for (String s : this.postconditionsQ) {
154                     System.out.print(s + "\n");
155                 }
156             }
157             this.fileLoaded = true;
158         }
159         catch (Exception e) {
160             e.printStackTrace();
161             this.fileLoaded = false;
162         }
163         return this.fileLoaded;
164     }
165 
166     /*
167      * Generate ACSL Contract
168      * Description: Function to generate ACSL lines from the inferred contract file
169      * Parameters:
170      * - considerCandidateAxiomsQSharp: Boolean to consider the CANDIDATE AXIOMS Q# lines
171      * Return:
172      * - List of Strings with the generated ACSL lines
173      * */
174     public List<String> generateACSLContract(boolean considerCandidateAxiomsQSharp) {
175         if(!this.fileLoaded) {
176             return null;
177         }
178         List<String> result = new ArrayList<>();
179         String start = "/*@";
180         for (int i = 0; i < this.preconditionsP.size(); i++) {
181             String precondition = this.preconditionsP.get(i);
182             String[] preconditionArray = precondition.split(":");
183             String pre = preconditionArray[1].replace("^", "&&").replaceAll(equalityRegex, "
                    == ");
184             pre = this.removeErrorPattern(pre);
```

```
185             pre = this.addModuleNotation(pre);
186             start = (i == 0 ? start + " requires " : "  @          ") + pre + " ||\n";
187             result.add(start);
188         }
189         result.set(result.size() - 1, this.removeTrailingStr(result.getLast(), " ||\n") + ";\n
                ");
190         if(!this.locationsLLines.isEmpty()) {
191             for (int i = 0; i < this.locationsLLines.size(); i++) {
192                 String location = this.locationsLLines.get(i);
193                  start = "  @ assigns " + location + ";\n";
194                  result.add(start);
195             }
196         }
197         for (int i = 0; i < this.preconditionsP.size(); i++) {
198             String precondition = this.preconditionsP.get(i);
199             String postcondition = this.postconditionsQ.get(i);
200             String acsl = getACSLNamedBehaviourLine(precondition, postcondition);
201             result.add(acsl);
202         }
203         if(!preconditionsPSharp.isEmpty() && considerCandidateAxiomsQSharp) {
204             for (int i = 0; i < this.preconditionsPSharp.size(); i++) {
205                 String precondition = this.preconditionsPSharp.get(i);
206                 String postcondition = this.postconditionsQSharp.get(i);
207                 String acsl = getACSLNamedBehaviourLine(precondition, postcondition);
208                 result.add(acsl);
209             }
210         }
211         String end = "  @*/\n";
212         result.add(end);
213 
214         return result;
215     }
216 
217     /*
218      * Insert ACSL Contract Lines
219      * Description: Function to insert the generated ACSL contract lines into the C source
                code
220      * Parameters:
221      * - contractLines: List of Strings with the generated ACSL contract lines
222      * - function: String with the name of the function to insert the contract lines
223      * - pathToSourceCode: String with the path to the C source code file
224      * - overrideFile: Boolean to override the source code file or create a new one
225      * Return:
226      * - Boolean with the result of the operation (lines inserted or not)
227      * */
228     public boolean insertACSLContractLines(List<String> contractLines, String function, String
            pathToSourceCode, boolean overrideFile) throws IOException {
229         // Read the content of the C source file
230         List<String> lines = Files.readAllLines(Paths.get(pathToSourceCode));
231 
232         // Find the declaration of the specified function
233         boolean functionFound = false;
234         for (int i = 0; i < lines.size(); i++) {
235             if(i == 0) {
236                 // Insert library import for predicates specification module (//@ import Pred
                        ;)
237                 lines.add(i, "//@ import Pred;");
238             }
239             String line = lines.get(i).trim();
240             if (line.matches(".*\\b" + function + "\\b\\s*\\(.*\\).*")) {
241                 functionFound = true;
242                 // Insert the contract lines before the function declaration
243                 for (int j = 0; j < contractLines.size(); j++) {
244                     String stringToAdd = this.removeTrailingStr(contractLines.get(j), "\n");
245                     lines.add(i + j, stringToAdd);
246                 }
247                 break;
248             }
249         }
250 
```

```
251         if (!functionFound) {
252             System.out.println("Function " + function + " not found in the source file.");
253             return false;
254         }
255 
256         // Print the modified content
257         /*for (String line : lines) {
258             System.out.println(line);
259         }*/
260 
261         if(overrideFile) {
262             // Write the modified content back to the C source file
263             Files.write(Paths.get(pathToSourceCode), lines);
264         }
265         else {
266             // Write the modified content to a new C source file
267             Files.write(Paths.get(pathToSourceCode.replace(".c", "_contract.c")), lines);
268         }
269         return true;
270     }
271 
272 /* ------------------------------------------------------------------
273  * ------------------- Private Service Functions -------------------
274  * ------------------------------------------------------------------
275  */
276     /*
277      * Extract Modifier Function
278      * Description: Function to extract the modifier function from the inferred contract file
279      * Parameters:
280      * - filePath: String with the path to the inferred contract file
281      * Return:
282      * - String with the modifier function found
283      * */
284     private String extractModifierFunction(String filePath) {
285         String functionFound = null;
286         try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
287             String line;
288             while ((line = br.readLine()) != null) {
289                 if (line.contains("Selected modifier function:")) {
290                     functionFound = line.split(":")[1].trim();
291                     break;
292                 }
293             }
294         } catch (IOException e) {
295             e.printStackTrace();
296         }
297 
298         return functionFound;
299     }
300 
301     /*
302      * Extract Postcondition Q Lines
303      * Description: Function to extract the POSTCONDITION Q lines from the inferred contract
           file
304      * Parameters:
305      * - filePath: String with the path to the inferred contract file
306      * Return:
307      * - List of Strings with the POSTCONDITION Q lines
308      * */
309     private List<String> extractPostconditionQLines(String filePath) {
310         List<String> linesArray = new ArrayList<>();
311         boolean isPostconditionSection = false;
312 
313         try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
314             String line;
315             while ((line = br.readLine()) != null) {
316                 if (line.contains("POSTCONDITION Q:")) {
317                     isPostconditionSection = true;
318                     continue;
319                 }
```

```
320
321                 if (isPostconditionSection) {
322                     if (line.startsWith("-------------------------------------------------"))
                            {
323                         break;
324                     }
325                     linesArray.add(line);
326                 }
327             }
328         } catch (IOException e) {
329             e.printStackTrace();
330         }
331
332         return linesArray;
333     }
334
335     /*
336      * Extract Preconditions P
337      * Description: Function to extract the preconditions (p in p=>q) of the POSTCONDITION Q
               lines
338      * Parameters:
339      * - postconditionQLinesArray: List of Strings with the POSTCONDITION Q lines
340      * Return:
341      * - List of Strings with the preconditions (p in p=>q) of the POSTCONDITION Q lines
342      * */
343     private List<String> extractPreconditionsP(List<String> postconditionQLinesArray) {
344         List<String> preconditionsPfound = new ArrayList<>();
345         for (String line : postconditionQLinesArray) {
346             String result = this.removeTrailingStr(line.split("=>")[0].trim(), ")");
347             String behavior = result.split(":")[0].trim();
348             result = this.removeLeadingStr(result.split(":")[1].trim(), "(");
349             result = behavior + ":" + result;
350             preconditionsPfound.add(result);
351         }
352
353         return preconditionsPfound;
354     }
355
356     /*
357      * Extract Postconditions Q
358      * Description: Function to extract the postconditions (q in p=>q) of the POSTCONDITION Q
               lines
359      * Parameters:
360      * - postconditionQLinesArray: List of Strings with the POSTCONDITION Q lines
361      * Return:
362      * - List of Strings with the postconditions (q in p=>q) of the POSTCONDITION Q lines
363      * */
364     private List<String> extractPostconditionsQ(List<String> postconditionQLinesArray) {
365         List<String> postconditionsQfound = new ArrayList<>();
366         for (String line : postconditionQLinesArray) {
367             String behaviour = line.split("=>")[0].trim().split(":")[0].trim();
368             String result = this.removeTrailingStr(line.split("=>")[1].trim(), " ^");
369             result = this.removeTrailingStr(result, ")");
370             result = this.removeLeadingStr(result, "(");
371             postconditionsQfound.add(behaviour + ":" + result);
372         }
373
374         return postconditionsQfound;
375     }
376
377     /*
378      * Extract Locations L Lines
379      * Description: Function to extract the LOCATIONS L lines from the inferred contract file
380      * Parameters:
381      * - filePath: String with the path to the inferred contract file
382      * Return:
383      * - List of Strings with the LOCATIONS L lines
384      * */
385     private List<String> extractLocationsLLines(String filePath) {
386         List<String> linesArray = new ArrayList<>();
```

```
387         boolean isLocationSection = false;
388 
389         try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
390             String line;
391             while ((line = br.readLine()) != null) {
392                 if (line.contains("LOCATIONS L:")) {
393                     isLocationSection = true;
394                     continue;
395                 }
396 
397                 if (isLocationSection) {
398                     if (line.startsWith("-------------------------------------------------"))
                            {
399                         break;
400                     }
401                     linesArray.add(line);
402                 }
403             }
404         } catch (IOException e) {
405             e.printStackTrace();
406         }
407 
408         return linesArray;
409     }
410 
411     /*
412      * Extract Candidate Axioms Q# Lines
413      * Description: Function to extract the CANDIDATE AXIOMS Q# lines from the inferred
           contract file
414      * Parameters:
415      * - filePath: String with the path to the inferred contract file
416      * Return:
417      * - List of Strings with the CANDIDATE AXIOMS Q# lines
418      * */
419     private List<String> extractCandidateAxiomsQSharpLines(String filePath) {
420         List<String> linesArray = new ArrayList<>();
421         boolean isCandidateAxiomSection = false;
422 
423         try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
424             String line;
425             while ((line = br.readLine()) != null) {
426                 if (line.contains("CANDIDATE AXIOMS Q#:")) {
427                     isCandidateAxiomSection = true;
428                     continue;
429                 }
430 
431                 if (isCandidateAxiomSection) {
432                     if (line.startsWith("-------------------------------------------------"))
                            {
433                         break;
434                     }
435                     linesArray.add(line);
436                 }
437             }
438         } catch (IOException e) {
439             e.printStackTrace();
440         }
441 
442         return linesArray;
443     }
444 
445     /*
446      * Get ACSL Named Behaviour Line
447      * Description: Function to generate an ACSL named behaviour contract line
448      * Parameters:
449      * - precondition: String with the precondition (p in p=>q)
450      * - postcondition: String with the postcondition (q in p=>q)
451      * - i: Integer with the index of the ACSL named behaviour
452      * Return:
453      * - String with the ACSL named behaviour contract line
```

```
454      * */
455     private String getACSLNamedBehaviourLine(String precondition, String postcondition) {
456         String[] preconditionArray = precondition.split(":");
457         String[] postconditionArray = postcondition.split(":");
458         String behaviour = preconditionArray[0];
459         // Preconditions (requires) ACSL lines
460         String pre = preconditionArray[1].replace("^", "&&").replaceAll(equalityRegex, " == ")
                ;
461         pre = this.removeErrorPattern(pre);
462         pre = this.addModuleNotation(pre);
463         // Postconditions (ensures) ACSL lines
464         String post = postconditionArray[1].replace("^", "&&");
465         post = post.split("&& ret=")[0] + "&& \\result=" + post.split("&& ret=")[1];
466         post = post.replaceAll(equalityRegex, " == ");
467         post = this.removeErrorPattern(post);
468         post = this.addModuleNotation(post);
469         // Generate ACSL contract named behaviour
470         String acsl =
471                 "  @ behavior " + behaviour + ":\n" +
472                 "  @    requires " + pre + ";\n" +
473                 "  @    ensures " + post + ";\n";
474         return acsl;
475     }
476
477 /* ------------------------------------------------------------------
478  * ------------------- Private Supporting Functions -------------------
479  * ------------------------------------------------------------------
480  */
481     /*
482      * Remove Trailing String
483      * Description: Function to remove a trailing string from a string
484      * Parameters:
485      * - str: String to remove the trailing string from
486      * - trailingStr: String to remove from the end of the string
487      * Return:
488      * - String with the trailing string removed
489      * */
490     private String removeTrailingStr(String str, String trailingStr) {
491         if (str != null && str.endsWith(trailingStr)) {
492             return str.substring(0, str.length() - trailingStr.length());
493         }
494         return str;
495     }
496
497     /*
498      * Remove Leading String
499      * Description: Function to remove a leading string from a string
500      * Parameters:
501      * - str: String to remove the leading string from
502      * - leadingStr: String to remove from the beginning of the string
503      * Return:
504      * - String with the leading string removed
505      * */
506     private String removeLeadingStr(String str, String leadingStr) {
507         if (str != null && str.startsWith(leadingStr)) {
508             return str.substring(leadingStr.length());
509         }
510         return str;
511     }
512
513     /*
514      * Remove Error Pattern
515      * Description: Function to remove the error pattern (== (string, {number})) from a string
516      * Parameters:
517      * - input: String to remove the error pattern from
518      * Return:
519      * - String with the error pattern removed
520      * */
521     private String removeErrorPattern(String input) {
522         // Define the regex pattern to match '== (string, {number})'
```

```
523         String regex = "== \\(([^,]+), \\{\\d+\\}\\)";
524         Pattern pattern = Pattern.compile(regex);
525         Matcher matcher = pattern.matcher(input);
526 
527         // Replace the matched pattern with '== string'
528         StringBuffer sb = new StringBuffer();
529         while (matcher.find()) {
530             matcher.appendReplacement(sb, "== Pred::" + matcher.group(1));
531         }
532         matcher.appendTail(sb);
533 
534         return sb.toString();
535     }
536 
537     /*
538      * Add Specification Module Notation
539      * Description: Function to add the specification module notation 'Pred::' to the module
           functions
540      * Parameters:
541      * - input: String to add the module notation to
542      * Return:
543      * - String with the module notation added
544      * */
545     private String addModuleNotation(String input) {
546         String[] words = {
547                 "isNull",
548                 "length",
549                 "firstElem",
550                 "lastElem",
551                 "firstE",
552                 "lastE",
553                 "isN",
554                 "isE",
555                 "isEmpty",
556                 "lenC",
557                 "lengthCircular",
558                 "isFull",
559                 "contains"
560         };
561         for (String word : words) {
562             input = input.replaceAll("\\b" + word + "\\b", "Pred::" + word);
563         }
564         return input;
565     }
566 }
```