Pontificia Universidad Javeriana Cali
Facultad de Ingeniería.
Ingeniería de Sistemas y Computación.
Proyecto de Grado.

# Traducción de modelos en Event-B a programas correctos en C++

Enrique José París Cárdenas

Director: Dr. Carlos Alberto Ramirez Restrepo

Julio de 2022

Santiago de Cali, Julio de 2022.

Señores
**Pontificia Universidad Javeriana Cali.**
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Por medio de la presente hago constar que he revisado el proyecto de grado "Traducción de modelos en Event-B a programas correctos en C++", del estudiante de Ingeniería de Sistemas y Computación Enrique José París Cárdenas (cod: 8912138) y lo considero apto para ser presentado y sometido a consideración del jurado.

Atentamente,

Carlos Ramírez

Dr. Carlos Alberto Ramirez Restrepo

Santiago de Cali, Julio de 2022.

Señores
**Pontificia Universidad Javeriana Cali.**
Dr. Gerardo Mauricio Sarria
Director Carrera de Ingeniería de Sistemas y Computación.
Cali.

Cordial Saludo.

Me permito presentar a su consideración el proyecto de grado titulado "Traducción de modelos en Event-B a programas correctos en C++" con el fin de cumplir con los requisitos exigidos por la Universidad para llevar a cabo el proyecto de grado y posteriormente optar al título de Ingeniero de Sistemas y Computación.

Al firmar aquí, doy fe que entiendo y conozco las directrices para la presentación de trabajos de grado de la Facultad de Ingeniería aprobadas el 26 de Noviembre de 2009, donde se establecen los plazos y normas para el desarrollo del anteproyecto y del trabajo de grado.

Atentamente,


Enrique José París Cárdenas
Código: 8912138

# Agradecimientos

Agradezco a Dios, por siempre guiar mis pasos en el desarrollo de toda mi carrera profesional; a mi director de tesis, Carlos Alberto Ramirez Restrepo, por su atención, dedicación, paciencia y apoyo a lo largo del desarrollo de este trabajo; a los profesores Camilo Rueda y Camilo Rocha, cuyas clases sobre el desarrollo formal de software, fueron la inspiración de este proyecto. Mi gratitud a la Pontificia Universidad Javeriana Cali, por ser motor de mi carrera profesional; a Juan Fernando Escobar por compartir sus conocimientos y experiencias esenciales en la etapa inicial del trabajo. Por último, pero no menos importante, a mis padres por acompañarme en éste camino haciendo posible llegar a éste momento, a mi familia entera, amigos y colegas que colaboraron directa o indirectamente en la finalización exitosa de este proyecto.

# Abstract

Software has become an essential part of many aspectos of society, from communication, to transport, finances, medicine, etc. Given how valuable software is, it's more important than ever that it can function correctly as intended. The problem is that, like all technology, it is not perfect and it is known to exhibit failures. In response to this, engineers in the field of software development have created practices that help to prevent and mitigate these errors. One of these techniques is the use of formal methods.

Formal methods are techniques based on mathematics to be used in the specification, development and verification of software systems. With these techniques, it's possible to prove the correctness of a system and therefore guarantee its proper functioning once it is deployed in the real world. However, these methods aren't popular in the industry. In part, this is because it is difficult to take the mathematical notation of a formal model and implement a software system that follows its specifications.

This document develops a tool to help with this transition from model to implementation: a translator that interprets a formal model defined in Event-B y automatically generates a program in C++ that functions according to the model. This translator was developed as a plugin for the formal development Eclipse platform Rodin.

This document includes a study of the theoretical foundations behind Event-B modelling, the identification of the challenges of translating a formal model to C++, the design of the translator EB2Cpp, the definition of equivalencias between Event-B and C++, the implementation of the EB2Cpp tool alongside an analysis of its computational complexity, and the test of the generated code's functionality by way of three case studies.

**Keywords**: Event-B, C++, Rodin, Formal methods, Translation, Code generation, Formal software development, Formal models.

# Resumen

El software se ha vuelto una parte esencial de diversos aspectos de la sociedad, como la comunicación, transporte, finanzas, medicina, entre otros. Dado lo valioso que es el software, su buen funcionamiento es más importante que nunca. El problema es que, como toda tecnología, no es perfecta y es propensa a presentar fallas. En respuesta a esto, los ingenieros del campo del software han desarrollado prácticas que ayudan a prevenir y mitigar estos errores. Una de estas técnicas es el uso de métodos formales.

Los métodos formales son técnicas basadas en las matemáticas para la especificación, desarrollo y verificación de sistemas de software. Con estas técnicas, es posible demostrar la correctitud de un sistema y poder así garantizar su buen comportamiento una vez se aplique al mundo real. Sin embargo, estos métodos no son populares en la industria. En parte, esto se debe a que es difícil tomar la notación matemática del modelo formal e implementar un sistema de software que siga sus especificaciones.

El presente trabajo desarrolla una herramienta para ayudar en esta transición de modelo a implementación: un traductor que interpreta un modelo formal definido en Event-B y genera automáticamente un programa en C++ que hace lo descrito en el modelo. Este traductor se crea como un plugin de la plataforma de desarrollo formal Rodin ambientada en Eclipse.

Este trabajo incluye un estudio de las bases teóricas del modelaje en Event-B, la identificación de los desafíos de traducir un modelo formal a C++, el diseño del traductor EB2Cpp, la definición de equivalencias entre Event-B y C++, la implementación de la herramienta EB2Cpp con un análisis de su complejidad computacional, y la prueba de la funcionalidad del código generado mediante tres modelos de estudio.

**Palabras Clave**: Event-B, C++, Rodin, Modelo formal, Traducción, Generación de Código, Desarrollo formal de software, Métodos formales.

# Índice general

Capítulo 1

# Introducción

La ingeniería de software se refiere al estudio y práctica de la ingeniería para construir, diseñar, desarrollar, mantener y retirar software; con el propósito de prevenir y/o solucionar problemas de sistemas de software [4]. En la actualidad, el software en nuestra sociedad es esencial, ya que se utiliza en múltiples áreas tales como los automóviles, plantas de energía, redes telefónicas, bancos, restaurantes, medios de comunicación, internet, entre otros. Para negocios como estos, el desarrollo del software es muy importante pues sus aplicaciones pueden ayudarlos a distinguirse de sus competidores de las siguientes formas [12]:

- El desarrollo de software mejora la experiencia del cliente y ayuda a traer productos más innovadores al mercado.
- La digitalización de la información y su almacenamiento en línea no solo ayuda a ahorrar espacio físico, sino también centraliza la información en un lugar, facilitando el acceso a ella para empleados y clientes.
- A largo plazo, cuando el negocio crece, el volumen de información aumenta, y almacenar esta información eficientemente se vuelve importante para la compañía. Para esto, los negocios necesitan desarrollar y mejorar su organización digital con tal de seguir brindando un buen desempeño. Por ejemplo, en un banco, se espera que los clientes puedan realizar sus transacciones en línea exitosamente y en un tiempo de espera razonable, sin importar que tantos clientes y cuentas el banco tenga que administrar.

La prevalencia del software trae como consecuencia que su correcto funcionamiento sea un asunto de crítica importancia. Desafortunadamente, la tecnología no es infalible ya que a lo largo de los años, millones de personas han sido perjudicadas debido a la falla de sistemas de software. En respuesta a problemas como estos, en el proceso de la ingeniería de software se han desarrollado diversas prácticas para asegurar el buen funcionamiento a largo plazo de los sistemas informáticos, como la programación extrema, DevOps, pruebas de caja blanca, etc. Una de estas técnicas, y la que es de interés para este proyecto, es la de los métodos formales.

Aplicados al desarrollo de sistemas de computación, los métodos formales son técnicas basadas en las matemáticas para la especificación, desarrollo y verificación de sistemas de software y hardware [16]. Dichos métodos formales brindan estructuras dentro de las cuales se puede especificar, desarrollar y verificar sistemas en una manera sistemática. La base matemática de estos métodos

provee herramientas que permiten definir con precisión nociones como la consistencia, completitud, especificación, implementación y correctitud de un sistema. Con un método formal, se puede demostrar (i) que una especificación es realizable, (ii) que un sistema ha sido implementado correctamente, y (iii) que se puede demostrar las propiedades de un sistema sin necesariamente ejecutarlo para determinar su comportamiento.

Los métodos formales siguen siendo una práctica poco popular en la industria del software, a pesar del aparente entusiasmo demostrado por la comunidad de software en los años 90 y el hecho de que han habido aplicaciones industriales exitosas [10]. Una de las razones presentadas por detractores es la dificultad de tomar la notación matemática empleada por los métodos formales en el diseño de un sistema, y traducirlo a un lenguaje de programación a la hora de la implementación, particularmente para desarrolladores de software comunes que no son especialistas en matemáticas o en métodos formales.

El propósito de este trabajo es entonces solventar este problema, con la creación de una herramienta de traducción automática en la forma de un plugin para la plataforma Rodin, que tomará un modelo formal de software definido en Event-B (un lenguaje de modelaje formal), y lo traducirá automáticamente a un programa en C++, que cumplirá con lo definido en dicho modelo. A lo largo de este documento se presentarán las bases teóricas que soportan esta herramienta, las decisiones de diseño e implementación que se realizaron en su construcción, y muestras de las traducciones generadas.

Este documento está organizado de la siguiente manera. En la sección 2, se plantea en mayor detalle el problema que este trabajo busca resolver, los objetivos del proyecto, la utilidad que brindará a los interesados y la delimitación de lo que no incluirá. En la sección 3, se exponen los conocimientos técnicos necesarios para entender el trabajo, incluyendo los trabajos previos que se han hecho en el campo de interés. En la sección 4, se presentan las decisiones de diseño que llevan a la creación del traductor y del programa que genera, entrando en detalle sobre la arquitectura de ambos y la estrategia de traducción. En la sección 5, se analiza la complejidad computacional del proceso de traducción y del programa que se obtiene. Finalmente, en la sección 7, se presentan las conclusiones sobre los aportes realizados por este trabajo y algunas recomendaciones sobre los futuros trabajos que se pueden realizar.

Capítulo 2

# Análisis

## 2.1. Planteamiento del Problema

El software se ha vuelto una parte integral de la sociedad, ya que se utiliza en los medios de comunicación, transporte, medicina, educación, servicios, finanzas, entre muchos más. Por esto, el mal funcionamiento de sistemas de software es un problema cuyas consecuencias pueden ser severas. A lo largo de los años, han ocurrido diversos casos que evidencian el impacto que tiene este problema. A continuación se presentan algunos ejemplos de estas fallas [14]:

- En agosto del 2015, alrededor de 275.000 pagos fallaron en ser procesados por HSBC (un banco británico), lo cual dejó a muchos sin su pago antes de sus días feriados. La causa de esta falla fue un problema con su sistema de pago electrónico para sus empleados bancarios, lo cual afectó el pago de salarios.

- Entre los años 2018 y 2020, la empresa fabricante de automóviles Nissan solicitó de vuelta las bolsas de aire de más de un millón de carros, debido a un mal funcionamiento que los sensores de detección estaban presentando. Se sabe que se presentaron por lo menos dos accidentes a causa de esta falla.

- En abril del 2019, los servicios de emergencia en siete estados de los EE.UU. cayeron fuera de línea. El incidente afectó 81 centros de llamada, lo cual resultó en alrededor de 6000 llamadas al 911 que no pudieron ser atendidas. Un estudio de la Comisión de Comunicaciones Federales encontró una falla de software evitable que generó la caída del servicio.

Los métodos formales prometen una solución a problemas como los anteriores. Dichos métodos utilizan matemáticas y lógica para la especificación, desarrollo y verificación de sistemas de software. Con un método formal, se puede demostrar (i) que una especificación de un software a crear es realizable, (ii) que un sistema ha sido implementado correctamente, y (iii) que se puede demostrar las propiedades de un sistema sin necesariamente ejecutarlo para determinar su comportamiento. Esta utilidad podría usarse para evitar muchos problemas de software y se tiene evidencia de esto. Los métodos formales aplicados al desarrollo de software han existido en la comunidad por más de treinta años, con diversos proyectos industriales exitosos y en la actualidad, se utilizan en sistemas críticos como la aviación, transporte y finanzas. Los sistemas críticos son un tipo de sistemas en los que una falla de software puede causar la pérdida de vidas, o de una gran cantidad de tiempo/dinero. A continuación se presentan tres ejemplos del uso industrial de los métodos formales en software [8]:

- Para la misión DeepSpace 1 en 1998, NASA utilizó métodos formales para verificar un componente del software de *Remote Agent*: el primer sistema de control de inteligencia artificial para dirigir una nave espacial sin supervisión humana. Las verificaciones facilitadas por el método formal detectaron cinco errores de concurrencia que los desarrolladores reconocieron no hubiesen sido detectados durante las pruebas convencionales. Es notable que luego, en 1999, un error ocurrió en otro componente de Remote Agent que *no* había sido sujeto a la verificación de métodos formales. Cuando los investigadores examinaron el problema, encontraron que se trataba de uno de los cinco errores que presentó el primer componente en 1997.

- El método formal *B* fue utilizado para desarrollar y validar las partes críticas de seguridad de la lanzadera del aeropuerto Roissy Charles de Gaulle en Paris, el cual a la fecha nunca ha presentado un error y se encuentra en completa operación desde el 2007.

- Métodos formales fueron aplicados en el desarrollo de *Mondex*, una cartera electrónica alojada en una tarjeta. Cada tarjeta almacena un valor financiero como información electrónica en un micro chip y provee operaciones para hacer transacciones financieras con otras tarjetas a través de un dispositivo de comunicación. Gracias a las verificaciones traídas por los métodos formales, en 1999 Mondex obtuvo la certificación de sistemas ITSEC E6 (el primer producto en los Reinos Unidos en lograr esto).

Como se mencionó antes, los métodos formales son una herramienta eficaz para reducir substancialmente las fallas de software. Pero, actualmente en la industria, no son populares y su uso se ve limitado a sistemas de seguridad crítica y a discusiones académicas. Esto se debe a que, a pesar de que muchos reconocen la existencia de los métodos formales y sus aplicaciones en la industria, la comunidad de ingeniería de software en general no está convencida de su utilidad [10]. Algunas de las razones que los detractores presentan en su contra son [7]:

- *Incrementan los costos de desarrollo.* Suele decirse que el uso de métodos formales es muy caro, pues implica una carga adicional de personal, tiempo y trabajo en las fases iniciales del desarrollo. Aunque se puede argumentar que a largo plazo significa menos costos en mantenimiento, es un punto de vista que no es popular con gerentes de proyectos preocupados por el presupuesto de desarrollo, y no de mantenimiento.

- *Son incomprensibles para clientes.* Una especificación formal está llena de símbolos matemáticos, lo cual hace que sea difícil de entender para cualquier persona que no esté familiarizada con la terminología. Por lo tanto, esto la hace inútil para clientes no-matemáticos que quieren tener un buen entendimiento del proyecto en sus etapas iniciales.

- *Involucran matemáticas complejas.* Los métodos formales se basan en las matemáticas y muchos creen que esto los hace muy difíciles para ingenieros de software tradicionales. Considerando que, al usar métodos formales en la especificación y diseño del sistema, el siguiente paso en el proceso de desarrollo consiste en implementar el sistema en un lenguaje de programación basado en el representación formal, se puede ver cómo esto es un problema.

En este trabajo estamos interesados en este último aspecto. Para ayudar a los desarrolladores de software que tengan dificultades con la implementación de un sistema a partir de un modelo formal, se quiere crear una herramienta que realice este proceso automáticamente. Para esto, se identifican distintos aspectos que este trabajo debe solucionar.

1. Existen diferentes lenguajes empleados en la especificación formal de sistemas, y por lo tanto el proceso de traducción es diferente dependiendo de cuál es el lenguaje origen. Algunos de los lenguajes en cuestión son: notación Z, B, VDM-SL, CSP, Petri Nets y TLA+ [15]. Intentar crear un traductor que sirva para todos estos métodos sería muy complejo. Con esto en mente, este trabajo se enfocará en traducir a partir de un lenguaje formal en específico: *Event-B* [1] (una extensión del método B anteriormente mencionado). Por razones similares, el único lenguaje de programación hacia el cual la herramienta podrá traducir será *C++*.

2. Se deben poder identificar equivalencias entre las expresiones de Event-B y C++. Este es probablemente el problema raíz al que los detractores de los métodos formales hacen referencia, cuando hablan de lo difícil que es implementar un sistema a base de un modelo formal: una evaluación superficial de Event-B revela expresiones y lógicas que no tienen un paralelo inmediato con el paradigma de programación en el que C++ opera [5].

3. Cómo lidiar con la naturaleza no-determinista de Event-B, ya que las semánticas de este lenguaje son tal que el siguiente evento a ejecutar se escoge de manera no-determinista del conjunto de eventos disponibles. Esto no se traduce directamente a C++, que es un lenguaje secuencial y determinista.

4. Como esta herramienta busca ayudar a desarrolladores en sus proyectos reales, y no solo estudios en el campo académico, se debe facilitar lo más posible la legibilidad de la traducción final, así como la posibilidad de que el desarrollador extienda el código generado con sus propios aportes personalizados.

### 2.1.1. Formulación

¿Cómo implementar un modelo de traducción del lenguaje de desarrollo formal, Event-B, a un programa correcto en C++, que cumpla la especificación definida por el modelo formal?

### 2.1.2. Sistematización

- ¿Cómo se puede interpretar la sintaxis interna de Event-B en la plataforma Rodin?
- ¿Qué funcionalidades de C++ son equivalentes a las distintas funciones de modelado de Event-B, que esta herramienta cubre?
- ¿Cómo implementar una herramienta de generación de código en C++, basado en lo especificado en un modelo determinado de Event-B?
- ¿Qué prácticas se pueden tener en cuenta a la hora de generar el código, con tal de que la traducción final sea legible y abierta a modificaciones?

## 2.2. Objetivos

### 2.2.1. Objetivo General

- Implementar un modelo de traducción del lenguaje formal Event-B a programas correctos en C++, que cumplan con la especificación definida por el modelo formal.

### 2.2.2. Objetivos Específicos

- Describir la sintaxis interna de las funcionalidades a traducir de Event-B en Rodin, el entorno donde trabaja.
- Identificar equivalencias entre el lenguaje de modelamiento formal y un código de C++.
- Diseñar e implementar una herramienta que tome un modelo formal y genere un código en C++ que funcione basado en lo descrito en el modelo.
- Verificar que el programa generado por la herramienta cumple con las especificaciones definidas por el modelo formal.
- Presentar la traducción generada de una forma legible para desarrolladores de software.

## 2.3. Justificación

A lo largo de las últimas décadas, se ha demostrado la utilidad de los métodos formales en la construcción de sistemas informáticos de alta confianza, a través de múltiples aplicaciones industriales exitosas [8]. Se cree que muchos otros campos de la comunidad se beneficiarían de la misma forma al adoptar esta metodología en sus proyectos. La importancia de este trabajo es que es un aporte para la integración de los métodos formales al proceso de desarrollo de software. En el caso de un proyecto real, esta herramienta brinda el puente necesario entre la especificación formal realizada por los especialistas en métodos formales y los ingenieros de software que deben implementar el sistema.

Este trabajo también realiza aportes al campo de generadores de código para Event-B. Una observación relevante es que, en el momento de la elaboración de este trabajo, los traductores a C++ para la plataforma de Rodin que fueron desarrollados en el pasado ya no funcionan para las versiones actuales de Rodin. Por lo tanto, uno de los aportes de este trabajo es el de agregar de nuevo C++ a los lenguajes para los cuales hay un generador de código en Rodin.

Otro aspecto es que los traductores existentes para C++ funcionan bajo la condición de que el modelo de Event-B haya sido refinado al punto de ya ser un programa; es decir, que consiste de un específico subconjunto de expresiones simples, para las cuales hay una equivalencia directa a C++. En contraste, el trabajo presente es capaz de traducir modelos que no son efectivamente

programas, siendo así de más utilidad para el desarrollador.

Este trabajo puede beneficiar a equipos de desarrollo que quieran en el futuro utilizar métodos formales en la especificación y diseño de sus proyectos, pero que se preocupan por cómo su actual equipo de ingenieros sin experiencia previa con métodos formales pueden implementar un sistema informático a partir de un modelo formal. También puede ser de beneficio para docentes en cualquier campo educativo, ya sea en universidades o en procesos de capacitación en el trabajo, ya que se puede usar para demostrarles a los estudiantes cómo traducir los distintos componentes de un modelo en Event-B a un programa.

## 2.4. Delimitaciones y Alcances

Se tiene definido que la herramienta no será capaz de traducir todas las funcionalidades de Event-B: solo las fundamentales usadas frecuentemente en el diseño de sistemas de software (la gramática soportada por esta herramienta será presentada en un futuro capítulo). La herramienta tampoco podrá hacer una traducción en reversa. Es decir, tomar un programa compilado en C++, y desplegar un modelo en Event-B que representa dicho programa.

Capítulo 3

# Marco de Referencia

## 3.1. Áreas Temáticas

- Computer Science - Software Engineering - Formal Methods
- Computer Applications - Programming Tool

## 3.2. Marco Teórico

### 3.2.1. Métodos formales

Como se mencionó anteriormente, este trabajo tiene como objetivo facilitar el uso de métodos formales para desarrollar programas de software correctos. Los métodos formales se basan en matemáticas y lógica para diseñar modelos de sistemas y verificar mediante pruebas que los sistemas sean correctos. Los métodos formales desempeñan un rol similar a los planos de un proyecto arquitectónico. Los ingenieros en estos proyectos arquitectónicos usan planos para razonar acerca de cómo va a funcionar la estructura a construir. Con esta herramienta, el ingeniero podrá definir restricciones y extraer conclusiones acerca de su comportamiento una vez abandone la página del plano, a través de los cuales puede evitar posibles fallas de funcionamiento. Se puede decir entonces, que los métodos formales son técnicas utilizadas para construir planos adaptadas a nuestra disciplina (ingeniería de software). Dichos planos se llaman *modelos formales* [1]. Estos modelos serán escritos usando los lenguajes de lógica y teoría de conjuntos. Estos lenguajes permiten expresar de manera clara y sin ambigüedades las características del sistema a construir. El uso de dichos lenguajes permite hacer razonamientos en la forma de pruebas matemáticas [1].

### 3.2.2. Sistemas complejos y discretos

Los métodos formales pueden aplicarse a varios campos de las ciencias de la computación pero para efectos de este trabajo, nos interesa su aplicación en el desarrollo de *sistemas complejos y discretos*. Un sistema es (i) *complejo* cuando está hecho de varios agentes que se ejecutan concurrentemente e interactúan en un ambiente volátil (un ambiente cuyas variables cambian a veces en formas difíciles de predecir); y es (ii) *discreto* cuando se puede abstraer en una sucesión de *estados* finitos o infinitos enumerables. Un estado es una descripción de las propiedades de un sistema (pueden ser en la forma de constantes y variables, como lo hace el lenguaje Event-B). Los cambios

de un estado a otro reciben el nombre de *transiciones*. A los sistemas discretos también se les llama *sistemas de transiciones discretos*, y los planos de estos sistemas se llaman *modelos discretos* [1].

Un modelo discreto es una representación de los elementos y características más importantes de un sistema (*abstracción*), que en cualquier momento dado se encuentra en exactamente *un* estado, esperando por la ejecución de alguna transición, que recibe el nombre de *evento*. Cada evento está hecho de una *guarda*, que es una condición construida utilizando la información contenida en el estado. Una guarda representa las condiciones necesarias para que el evento ocurra. Cada evento también contiene *acciones* que describen modificaciones sobre las propiedades del estado como una consecuencia del evento que ocurre. Son entonces estos cambios a las propiedades del sistema que modifican el estado del mismo, y por lo tanto es lo que hace que las transiciones ocurran [1].

Para entender cómo funciona un modelo discreto en Event-B, se deben hacer ciertas observaciones. La ejecución de un evento se realiza inmediatamente y dos eventos no pueden ocurrir simultáneamente. Al contemplar todos los eventos posibles a ocurrir, en cualquier momento ocurren una de las siguientes cosas:

- Cuando ninguna guarda de evento se cumple, entonces la ejecución del modelo se detiene. Si esto pasa, el sistema se considera *deadlocked*.
- Cuando alguna guarda de evento se cumple, entonces alguno de los eventos correspondientes (aquellos cuyas guardas se cumplen) necesariamente ocurre y el estado es modificado de acuerdo a las acciones del evento. Subsecuentemente, las guardas son revisadas nuevamente, y así hasta que ninguna guarda se cumpla.

Con este comportamiento, es claro que estos sistemas de transición en Event-B pueden ser no deterministas, ya que varias guardas pueden cumplirse simultáneamente, y por lo tanto más de una transición es posible en un estado dado.

Estos modelos suelen poseer algunas *propiedades invariantes* (o solo *invariantes*). Una invariante es una condición de las propiedades del estado que debe cumplirse en todo momento durante la ejecución del sistema [1].

Los modelos a construir no solo describen el comportamiento del futuro sistema, sino también contienen una representación del ambiente dentro del cual el sistema a construir va a comportarse. Por ejemplo, si el sistema a construir es el controlador (el sistema de software que controla sus operaciones) de unos semáforos en una intersección, un modelo de este sistema no solo contiene los comandos de cambio de luz de los semáforos, sino también el número de carros en cada calle, y otros factores ambientales que el controlador de semáforos observa para incorporar en su funcionamiento. Por esto, se puede decir que los modelos desarrollados con este método son *modelos cerrados*, los cuales son capaces de exhibir las acciones y reacciones tomando lugar entre cierto ambiente y un controlador correspondiente [1].

Considerando la escala que pueden tener los ambientes sobre los que trabaja el modelo y la complejidad del sistema, el número de transiciones y variables a estudiar puede ser muy grande; para lidiar con esta complejidad se usa el proceso de *refinamiento*. En este contexto, *refinar* es crear un nuevo modelo que toma como base un modelo anteriormente creado, y agrega nuevas características (constantes, variables, eventos, etc.) que complementan el modelo antiguo. Un refinamiento nos permite construir un modelo gradualmente al hacerlo cada vez más y más preciso, para que así sea más cercano a la realidad. En otras palabras, no se construye un modelo representando toda la realidad del sistema y más bien, se construirá una secuencia ordenada de modelos, donde cada uno de ellos se supone es un refinamiento del que le precede en la secuencia. Esto significa que un modelo refinado tendrá más variables que su abstracción (el modelo antiguo previo a ser refinado); dichas nuevas variables son una visible consecuencia de analizar el sistema desde más cerca y con más detalle [1]. Es así que la estrategia de refinamiento es adecuada para el modelado de sistemas muy complejos, que harían muy difícil crear un primer modelo que ya represente todos los aspectos del sistema.

Los refinamientos también se usan para modificar el estado del sistema con tal de que pueda ser implementado en una computadora mediante algún lenguaje de programación. Este segundo uso del refinamiento se llama *refinamiento de datos*. Se usa como una segunda técnica, una vez todas las propiedades importantes han sido modeladas [1].

Es posible unir los conceptos presentados hasta ahora para articular cómo desarrollar formalmente un programa de software. El proceso consiste en identificar el ambiente en donde el software a desarrollar va a trabajar, y cuáles son los datos de ese ambiente que el software va a usar. Esto es necesario para realizar la abstracción del ambiente real y del controlador de nuestro sistema en el modelo formal, en la forma de conjuntos, constantes, axiomas, variables, etc. y después, definir las restricciones del sistema, en forma de invariantes; condiciones que durante el funcionamiento del sistema deben cumplirse para garantizar que el programa funcione correctamente.

Generalmente, este primer modelo es una abstracción general de todo el sistema, sin todos los detalles. Luego, se deben realizar pruebas que demuestren como las restricciones (principalmente invariantes) del sistema siempre se cumplirán, utilizando demostraciones lógicas y matemáticas. Finalmente, se comienza a refinar el modelo anterior, agregando más detalles sobre el funcionamiento del sistema a través de variables y eventos, y se repiten los pasos expuestos anteriormente para cada refinamiento.

### 3.2.3. Event-B

Event-B es un lenguaje para modelaje formal basado en la notación de máquinas abstractas [9]. Entre sus funcionalidades principales está el uso de teoría de conjuntos como notación de modelaje, el uso de refinamientos para representar sistemas en diferentes niveles de abstracción, y el uso de pruebas matemáticas para verificar consistencia entre los niveles de refinamiento [1].

El concepto primario en el desarrollo formal en Event-B es el de un *modelo*. Un modelo contiene el desarrollo matemático completo de un sistema de transición discreto, y está hecho de varios componentes de dos tipos: máquinas y contextos. Las máquinas contienen los elementos dinámicos de un modelo, como por ejemplo, variables, invariantes y eventos; mientras que los contextos contienen los elementos estáticos de un modelo, como por ejemplo conjuntos, constantes y axiomas. Los distintos ítems pertenecientes a máquinas o contextos (variables, invariantes, etc.) son llamados elementos de modelaje [1].

Las máquinas y los contextos tienen varias relaciones: una máquina puede ser *refinada* por otra, y un contexto puede ser *extendido* por otro. Al hablar de contextos, extender un contexto significa refinar un contexto: el mismo concepto de refinamiento anteriormente explicado. En otras palabras, un contexto C2 extiende al contexto C1, cuando el contexto C2 incorpora los elementos del contexto C1 y agrega detalles que acercan el modelo más a la realidad. También, una máquina puede acceder y utilizar los elementos de uno o más contextos; y en este caso se dice que la máquina *observa* dichos contextos. Las relaciones entre máquinas y contextos se definen por ciertas reglas de visibilidad [1]:

- Una máquina tiene acceso explícitamente a uno, varios o ningún contexto.
- Un contexto puede extender explícitamente uno, varios o ningún contexto.
- La noción de extensión de contexto es transitiva: un contexto C1 explícitamente extendiendo un contexto C2, implícitamente extiende todos los contextos extendidos por C2.
- Cuando un contexto C1 extiende un contexto C2, los conjuntos y constantes de C2 pueden ser usados en C1.
- Una máquina implícitamente ve todos los contextos extendidos por un contexto explícitamente visto.
- Cuando una máquina M ve un contexto C, significa que todos los conjuntos y constantes de C pueden usar en M.
- Las relaciones de “refina” y “extiende” (“refines”, “extends”) definidas no deben llevar a ningún ciclo.
- Una máquina solo refina a lo mucho una máquina.
- El conjunto de contextos vistos explícita o implícitamente por una máquina debe ser tan grande como el de la abstracción de esta máquina.

### 3.2.4. Gramática de lenguaje Event-B a traducir

En esta sección se presenta la sintaxis concreta de Event-B que este proyecto será capaz de traducir a C++ en notación BNF. Primero, a continuación se puede ver la estructura sintáctica

de un contexto en Event-B. Los elementos que conforman esta estructura (los cuales no son todos obligatorios) son:

- En ⟨context_identifier⟩ se ubica el nombre del contexto, el cual debe ser distinto al nombre de cualquier otro componente (contexto o máquina) del modelo. Este es el único elemento del componente que es obligatorio.
- En la cláusula extends, se listan todos los contextos que este contexto extiende.
- En la cláusula sets, se listan los conjuntos definidos por el usuario (también se llaman carrier sets).
- En la cláusula constants, se listan las constantes definidas en este contexto. El nombre (identificador) de cada constante debe ser único.
- En la cláusula axioms se listan los predicados que las constantes deben obedecer. Por cada axioma, se le asigna una etiqueta que lo identifica, junto al predicado a evaluar.

| | |
|---|---|
| ⟨context⟩ | ::= 'context'⟨context_identifier⟩ |
| | {⟨extend_clause⟩} |
| | {⟨sets_clause⟩} |
| | {⟨constants_clause⟩} |
| | {⟨axioms_clause⟩} |
| | 'end' |
| ⟨extend_clause⟩ | ::= 'extends' [⟨context_identifier⟩] |
| ⟨sets_clause⟩ | ::= 'sets' [⟨set_identifier⟩] |
| ⟨constants_clause⟩ | ::= 'constants' [⟨constant_identifier⟩] |
| ⟨axioms_clause⟩ | ::= 'axioms' [⟨label_identifier⟩ ⟨predicate⟩] |

Segundo, se presenta la estructura sintáctica de una máquina. Los elementos que conforman esta estructura (los cuales no son todos obligatorios) son:

- En ⟨machine_identifier⟩ se ubica el nombre de la máquina, el cual debe ser distinto al nombre de cualquier otro componente (contexto o máquina) del modelo. Este es el único elemento del componente que es obligatorio.
- En la cláusula refines, se listan todos las máquinas que esta máquina refina.
- En la cláusula sees, se listan todos los contextos que esta máquina referencia para el uso de conjuntos y constantes.

- En la cláusula variables, se listan las variables definidas en esta máquina. El nombre (identificador) de cada variable debe ser único.
- En la cláusula invariants se listan los predicados que las variables deben obedecer, incluyendo variables de las máquinas refinadas. Por cada invariante, se le asigna una etiqueta que lo identifica, junto al predicado a evaluar.
- En la cláusula events se listan los eventos de la máquina.

⟨machine⟩ ::= 'machine'⟨machine_identifier⟩
{⟨refines_clause⟩}
{⟨sees_clause⟩}
{⟨variables_clause⟩}
{⟨invariants_clause⟩}
{⟨events_clause⟩}
'end'

⟨refines_clause⟩ ::= 'refines' ⟨machine_identifier⟩

⟨sees_clause⟩ ::= 'sees' [⟨context_identifier⟩]

⟨variables_clause⟩ ::= 'variables' [⟨variable_identifier⟩]

⟨invariants_clause⟩ ::= 'invariants' [⟨label_identifier⟩ ⟨predicate⟩]

⟨events_clause⟩ ::= 'events' [⟨event⟩]

⟨event⟩ ::= 'event' ⟨event_identifier⟩
{'extends' ⟨event_identifier⟩}
{'refines' ⟨event_identifier⟩}
{'any' [⟨identifier⟩]}
{'where' [⟨label_identifier⟩ ⟨predicate⟩]}
{'then' [⟨label_identifier⟩ ⟨variable_identifier⟩':='⟨expression⟩]}
'end'

Ahora se presenta la gramática de Event-B usada para construir predicados y expresiones, ilustrando solamente las que esta herramienta podrá traducir. Se excluye la gramática correspondiente a cuantificadores, comprensión de conjuntos, identidad, proyección, abstracción lambda, asignación por conjunto y asignación por predicado.

⟨predicate⟩ ::= ⟨unquantified_predicate⟩

⟨unquantified_predicate⟩ ::= ⟨simple_predicate⟩[‘ ⇒ ’⟨simple_predicate⟩] |
⟨simple_predicate⟩[‘ ⇔ ’⟨simple_predicate⟩]

⟨simple_predicate⟩ ::= ⟨literal_predicate⟩{‘ ∨ ’⟨literal_predicate⟩} |
⟨literal_predicate⟩{‘ ∧ ’⟨literal_predicate⟩}

⟨literal_predicate⟩ ::= {‘¬’}⟨atomic_predicate⟩

⟨atomic_predicate⟩ ::= ‘⊥’ | ‘⊤’ | ‘$finite$’(⟨expression⟩) | ⟨pair_exp⟩⟨relop⟩⟨pair_exp⟩ | (⟨predicate⟩)

⟨relop⟩ ::= ‘ = ’ | ‘ ≠ ’ | ‘ ∈ ’ | ‘ ∉ ’ | ‘ ⊂ ’ | ‘ ⊄ ’ | ‘ ⊆ ’ | ‘ ⊈ ’ |
‘ < ’ | ‘ ≤ ’ | ‘ > ’ | ‘ ≥ ’

⟨expression⟩ ::=⟨pair_exp⟩

⟨pair_exp⟩ ::=⟨rel_set_expr⟩{‘ ↦ ’⟨rel_set_expr⟩}

⟨rel_set_expr⟩ ::=⟨set_expr⟩{⟨rel_set_op⟩⟨set_expr⟩}

⟨rel_set_op⟩ ::= ‘ ↔ ’ | ‘ ⇻ ’ | ‘ ↚ ’ | ‘ ⇼ ’ | ‘ ⇸ ’ | ‘ → ’ | ‘ ⤔ ’ |
‘ ⤖ ’ | ‘ ⇾ ’ | ‘ ↠ ’ | ‘ ⤀ ’ | ‘ ⤖ ’

⟨set_expr⟩ ::=⟨interval_expr⟩{‘ ∪ ’⟨interval_expr⟩} |
⟨interval_expr⟩{‘ × ’⟨interval_expr⟩} |
⟨interval_expr⟩{‘ ◁ −’⟨interval_expr⟩} |
⟨interval_expr⟩{‘ ∘ ’⟨interval_expr⟩} |
⟨interval_expr⟩‘ || ’⟨interval_expr⟩ |
[⟨domain_mod⟩]⟨rel_expr⟩

⟨domain_mod⟩ ::=⟨interval_expr⟩(‘ ◁ ’ | ‘ ⩤ ’)

⟨rel_expr⟩ ::=⟨interval_expr⟩‘ ⊗ ’⟨interval_expr⟩ |
⟨interval_expr⟩{‘ ; ’⟨interval_expr⟩}[⟨range_mod⟩] |
⟨interval_expr⟩{‘ ∩ ’⟨interval_expr⟩}[‘ \ ’⟨interval_expr⟩ |
⟨range_mod⟩]

⟨range_mod⟩ ::=(‘ ▷ ’ | ‘ ⩥ ’)⟨interval_expr⟩

⟨interval_expr⟩ ::=⟨arith_expr⟩[‘..’⟨arith_expr⟩]

⟨arith_expr⟩ ::=[‘ − ’] ⟨term⟩{(‘ + ’ | ‘ − ’)⟨term⟩}

⟨term⟩ ::=⟨factor⟩{(‘ ∗ ’ | ‘ div ’ | ‘$mod$’)⟨factor⟩}

⟨factor⟩ ::=⟨image⟩[‘ ∧ ’⟨image⟩]

⟨image⟩ ::=⟨primary⟩{‘[’⟨expression⟩‘] ’ | ‘(’⟨expression⟩‘)’}

⟨primary⟩ ::=⟨simple_expr⟩{$^{-1}$}

⟨simple_expr⟩ ::=‘$bool$’‘(’⟨predicate⟩‘)’ |
⟨unary_op⟩‘(’⟨expression⟩‘)’ |
‘(’⟨expression⟩‘)’ |

‘$\mathbb{Z}$’ | ‘$\mathbb{N}$’ | ‘$\mathbb{N}_1$ ’ | ‘BOOL’ | ‘TRUE’ | ‘FALSE’ | ‘$\varnothing$’ |
⟨ident⟩ | ⟨integer_literal⟩

⟨unary_op⟩ ::=‘card’ | ‘$\mathbb{P}$ ’ | ‘$\mathbb{P}_1$ ’ | ‘union’ | ‘inter’ | ‘dom’ | ‘ran’ | ‘min’ | ‘max’

La gramática consiste de dos categorías sintácticas principales: predicados (⟨predicate⟩) y expresiones (⟨expression⟩). Un predicado puede ser una construcción de múltiples predicados conectados mediante símbolos lógicos (como $\Rightarrow$ y $\vee$). Sean o no múltiples predicados, cada uno de éstos consiste de evaluar si una expresión es finita o en realizar una comparación relacional (con operaciones como $<$, $=$, $\subseteq$...). Estas comparaciones se hacen entre expresiones, que pueden ser conjuntos, parejas y unidades atómicas como valores booleanos y números. Cada una de estas puede construirse con operaciones más complejas, como por ejemplo usar un producto cartesiano (la operación $\times$) entre dos conjuntos para crear un conjunto de parejas.

### 3.2.5. Árbol de sintaxis abstracta

En el campo de las ciencias de la computación, un *árbol de sintaxis abstracta* (abreviado a árbol sintáctico, o AST), es una estructura de datos de árbol usada para representar la estructura sintáctica jerárquica de un programa. Al representar una expresión, cada nodo interno es un operador; y los hijos de ese nodo son sus operandos. Más generalmente, toda construcción en programación puede representarse al crear un operando para su expresión raíz, y tratando las expresiones componentes de esa raíz como sus operandos. Por ejemplo, si se quiere representar la expresión matemática ”$9 - 5 + 2$ ”, se haría como se ilustra en el árbol sintáctico de la Figura 3.1. El nodo raíz representa el operador $+$, y los subárboles representan las sub-expresiones $9 - 5$ y $2$ [2]. Los árboles de sintaxis abstracta serán usados en este trabajo para representar el modelo a traducir.

Finalmente, el propósito de representar un código en un árbol sintáctico es poder después navegar el árbol para realizar operaciones como verificaciones sintácticas y la generación de nuevo código. Una forma de diseñar esta lectura del árbol, y que es la que Rodin emplea, es usando el patrón de diseño *Visitor*. Este diseño consiste en separar un algoritmo de la estructura de objeto sobre la cual opera. La utilidad de esta separación es que permite agregar nuevas operaciones a estructuras de objetos existentes sin modificarlas [6].

De forma simplificada, el patrón Visitor tiene el siguiente comportamiento (aplicado a árboles sintácticos): se tiene un objeto, llamado *visitor*, que contiene las operaciones relacionadas a cada clase que compone el árbol sintáctico. Mientras se navega el árbol, el visitor se pasa a cada elemento en el recorrido. Cuando un elemento *acepta* el visitor, le envía una petición al visitor que codifica la clase del elemento, incluyendo también el elemento en sí como un argumento. El visitor entonces ejecuta la operación para ese elemento [6].

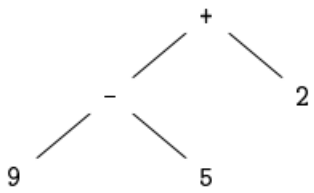


Figura 3.1: Árbol sintáctico para expresión $9 - 5 + 2$

### 3.2.6. Trabajos Relacionados

En esta sección se presentarán trabajos previos que abordaron el tema de traducciones de Event-B a programas en lenguajes de programación. Estos trabajos servirán como guía y fuente de conocimientos y herramientas para el cumplimiento de los objetivos de este trabajo. La intención es señalar los logros y en ciertos casos limitaciones de los diferentes estudios.

#### 3.2.6.1. Traducción de B a JML

Este trabajo fue realizado por Néstor Cataño, Tim Wahls, Camilo Rueda, Víctor Rivera y Danni Yu en marzo del 2014 [13]. Su objetivo es presentar una traducción de máquinas B a especificaciones en JML, que es un lenguaje basado en modelos para especificar el comportamiento de clases en Java. Para hacer esto implementaron una herramienta, B2Jml, que automatiza la traducción. Su decisión de traducir a JML les permite a los desarrolladores aprovecharse de las herramientas que JML posee para verificar invariantes y otras propiedades de correctitud.

El equipo también logró validar la implementación de su traducción al aplicarlo a un modelo en B de un sitio de redes sociales, y luego al ejecutar las especificaciones de JML generadas con varios casos de prueba desarrollados para una traducción manual del modelo B. Adicionalmente, la herramienta B2Jml soporta toda la sintaxis de B excepto características del modelo en B que hayan sido especificados incrementalmente a través de refinamiento.

#### 3.2.6.2. EB2ALL: Generador de código de Event-B a C, C++, C# y Java

Desarrollado por Dominique Mery y Neeraj Kumar Singh, *EB2ALL* [11] es un conjunto de herramientas de traducción que automáticamente genera código a un lenguaje específico (C, C++, C# y Java) a partir de una especificación formal en Event-B. El proyecto se encuentra en estado beta, y en consecuencia solo ofrece traducciones para pocas expresiones de Event-B. El aporte relevante para motivos de este proyecto presente es la traducción que EB2ALL usa para C y C++, como se puede ver en la Figura 3.2. A pesar de que la traducción es limitada pues necesita que el modelo en Event-B a traducir solo use expresiones que se encuentran en la primera columna de la tabla, es

| Event-B | 'C' & 'C++' Language | Comment |
|---|---|---|
| n..m | int | Interger type |
| x ∈ Y | Y x; | Scaler declaration |
| x ∈ tl_int16 | int x; | 'C' & 'C++' Context declaration |
| x ∈ n..m → Y | Y x [m+1]; | Array declaration |
| x :∈ Y | /* No Action */ | Indeterminate initialization |
| x : \| Y | /* No Action */ | Indeterminate initialization |
| x = y | if(x==y) { | Conditional |
| x ≠ y | if(x!=y) { | Conditional |
| x < y | if(x<y) { | Conditional |
| x ≤ y | if(x<=y) { | Conditional |
| x > y | if(x>y) { | Conditional |
| x ≥ y | if(x>=y) { | Conditional |
| (x>y) ∧ (x≥z) | if ((x>y) && (x>=z) { | Conditional |
| (x>y) ∨ (x≥z) | if ((x>y) ‖ (x>=z) { | Conditional |
| x := y + z | x = y + z; | Arithmetic assignment |
| x := y - z | x = y - z; | Arithmetic assignment |
| x := y * z | x = y * z; | Arithmetic assignment |
| x := y ÷ z | x = y / z; | Arithmetic assignment |
| x := F(y) | x = F(y); | Function assignment |
| a := F(x↦y) | a = F(x, y); | Function assignment |
| x := a(y) | x = a(y); | Array assignment |
| x := y | x = y; | Scalar action |
| a := a ⩤ {x↦y} | a(x) = y; | Array action |
| a := a ⩤ {x↦y} ⩤ {i↦j} | a(x)=y; a(i)=j; | Array action |
| X⇒Y | if(!X ‖ Y){ | Logical Implication |
| X⇔Y | if((!X ‖ Y) && (!Y ‖ X)){ | Logical Equivalence |
| ¬x<y | if(!(x<y)){ | Logical not |
| x ∈ ℕ | unsigned long int x | Natural numbers |
| x ∈ ℤ | signed long int x | Integer numbers |
| ∀ | /* No Action */ | Quantifier |
| ∃ | /* No Action */ | Quantifier |
| Sets | Supported by C++ | Using STL library based Sets operations |
| Set1 | set <*data type*> Set1 | STL library |
| ∪ | set_union(...) | STL library |
| ∩ | set_intersection(...) | STL library |
| − | set_difference(...) | STL library |
| ⊂ | if (includes(...)){ | STL library |
| ⊆ | if (includes(...) \|\| equal (...)){ | STL library |
| ⊄ | if (!(includes(...))){ | STL library |
| ⊈ | if (!(includes(...) \|\| equal (...))){ | STL library |
| fun ∈ ℕ × ℕ → ℕ | long int fun(unsigned long int $arg1$, unsigned long int $arg2$) { //TODO: Add your Code return; } | Function Definition |

Figura 3.2: Sintaxis de traducción Event-B a C y C++ para *EB2ALL* [11]

un buen punto de partida para establecer equivalencias para las expresiones más simples de Event-B.

La herramienta EB2ALL actualmente no funciona para las versiones más recientes de la plataforma de Rodin, como se mencionó en la sección de justificación en el capítulo anterior. Se desconoce si los desarrolladores planean actualizar la herramienta a la fecha de este escrito, sin embargo, aún así su documentación es un aporte valioso para propósitos del trabajo presente.

#### 3.2.6.3. Aproximación a generación de código para Event-B

En el 2014, Andreas Furst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato y Kunihiko Miyazaki presentaron una aproximación a la generación de código correcto a partir de modelos de Event-B [5]. El proceso de generación propuesto consiste de:

1. Restringir el modelo a traducir a través de refinamiento para asegurar que las variables sean de tipos apropiados y las operaciones sobre ellas están bien definidas.
2. Usar un lenguaje especial de planificación para especificar un cronograma que describe el orden de ejecución deseado para los eventos traducidos.
3. Ejecutar el generador de código con el cronograma como entrada. Basado en el cronograma, el generador de código traduce el modelo restringido a un programa secuencial.
4. El traductor también genera un modelo de Event-B planificado representando las semánticas del programa secuencial, y finalmente se demuestra que este modelo planificado refina el modelo de Event-B restringido.

Esta aproximación habla de cómo el modelo de Event-B debe restringirse de acuerdo a las limitaciones del lenguaje al cual se va a traducir. Para demostrar esto, se concentran solamente en C como el lenguaje objetivo. El trabajo incorpora consideraciones como el límite de bits de los tipos de datos en C, y restricciones a las operaciones aritméticas del modelo en Event-B para evitar overflows en el momento de ejecución. También, el trabajo identifica las obligaciones de pruebas que se pueden generar del cronograma para demostrar la correctitud del programa generado.

#### 3.2.6.4. Traducción automática de Event-B a Python

Este trabajo es una herramienta para la traducción automática de un modelo en Event-B a un código en Python, que fue realizado por Juan Fernando Escobar en agosto del 2020 [3]. Esta herramienta es actualmente la única capaz de traducir de Event-B hacia Python. A diferencia de muchos de los otros traductores, impone muy pocas restricciones sobre el modelo de Event-B, ya que es capaz de traducir expresiones complejas de Event-B como lo son cuantificadores y conjuntos infinitos.

La traducción también agrega varios comentarios para ayudar al usuario a entender qué partes del código corresponden a qué componentes del modelo, logrando así una legibilidad poco vista en

otros traductores. Hablando de accesibilidad, la herramienta genera métodos que el usuario puede ejecutar para monitorear el estado actual del sistema traducido. El desarrollador logró validar las traducciones generadas con varios ejemplos académicos, aunque admite que no se emplearon métodos formales para verificar la correctitud del código generado.

Capítulo 4

# Diseño

Este trabajo consiste de dos grandes fases. Primero, la implementación de distintas herramientas en C++ para poder representar apropiadamente modelos de Event-B; y segundo, la creación del plugin de Rodin que lee modelos de Event-B y los traduce a C++. A continuación, se va a exponer el diseño del plugin de Rodin que realiza la traducción.

## 4.1. Diseño de plugin

El plugin será programado en Java siguiendo el paradigma de Programación Orientada a Objetos, dado que Rodin está construido en ese lenguaje. Para determinar cómo modelar el programa, se dividió el proceso de traducción en tres etapas:

1. Extraer/leer el modelo a traducir de Rodin.
2. Leer cada variable, predicado y expresión en dicho modelo; determinando lo que expresan para saber así a qué traducirlo después (saber que una variable es un conjunto de enteros, que una acción es asignarle a una variable la unión entre dos relaciones, que un predicado es un menor-o-igual-que, etc.).
3. Generar los archivos en lenguaje C++: se debe elegir qué texto generar basada en la información recopilada en la segunda fase, y luego escribir dicho texto en archivos.

Con las etapas anteriores en mente, se diseñó un programa con distintos componentes, cada uno siendo únicamente responsable por las tareas asociadas a cada etapa. Se buscó separar lo más posible cada módulo con tal de evitar redundancias, y facilitar la lectura del código al momento de rastrear por errores.

En la Figura 4.1 se muestra la arquitectura de nuestro plugin, el cuál es llamado EB2CPP. El programa consiste de cuatro componentes, que se comunican entre sí a través de una interfaz central llamada MainHandler. Estos cuatro componentes son:

1. SampleHandler, una clase que recibe la instrucción del usuario para empezar la traducción, especificando qué modelo traducir.
2. EB2CppRodinHandler, una clase que se encarga de la primera etapa de traducción: extraer el modelo a traducir de la base de datos de Rodin.

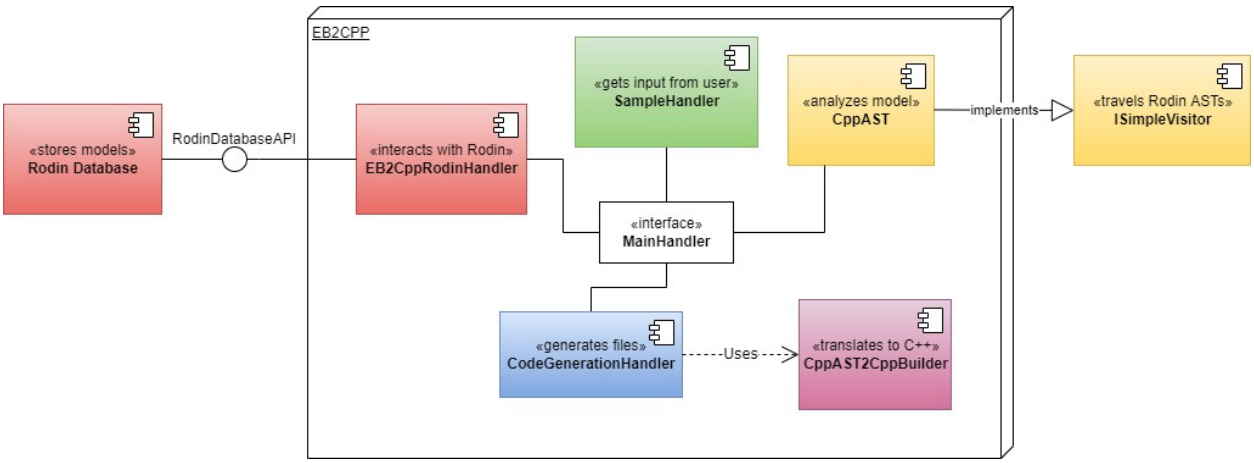


Figura 4.1: Diagrama de Componentes

3. CppAST, una clase que se encarga de la segunda etapa: hacer el análisis sintáctico del modelo para saber las propiedades del sistema que describe, utilizando el Visitor de Rodin y organizando la información en un árbol sintáctico. El árbol consiste de un número de clases diseñadas para representar cada tipo de expresión en Event-B que la herramienta puede traducir.

4. CodeGenerationHandler, que se encarga de la tercera y última etapa de traducción: la escritura del código generado en archivos. Para esto también se usa otro componente robusto llamado CppAST2CppBuilder para hacer las equivalencias de modelo Event-B a código C++.

Dos de los componentes: el EB2CppRodinHandler y el CppAST hacen uso de elementos en la plataforma de Rodin para llevar a cabo sus tareas. También, se puede observar en la Figura 4.1 el uso del MainHandler anteriormente mencionado cuyo trabajo es servir de centro para la comunicación entre los componentes.

El comportamiento del programa para llevar a cabo el proceso de traducción es el siguiente (ver Figura 4.2):

1. Empieza con el usuario, tras haber modelado su sistema en Rodin, solicitando la traducción automática del plugin, y especificando cuál modelo quiere traducir.

2. Esta solicitud es gestionada por SampleHandler, que se encarga de verificar que el modelo a traducir existe. Si no existe tiene que notificarle al usuario este error. Si existe, le informa a la interfaz central, MainHandler, que empiece a traducir el modelo.

3. MainHandler crea el EB2CppRodinHandler, y le instruye que extraiga de la base de datos de Rodin el modelo.

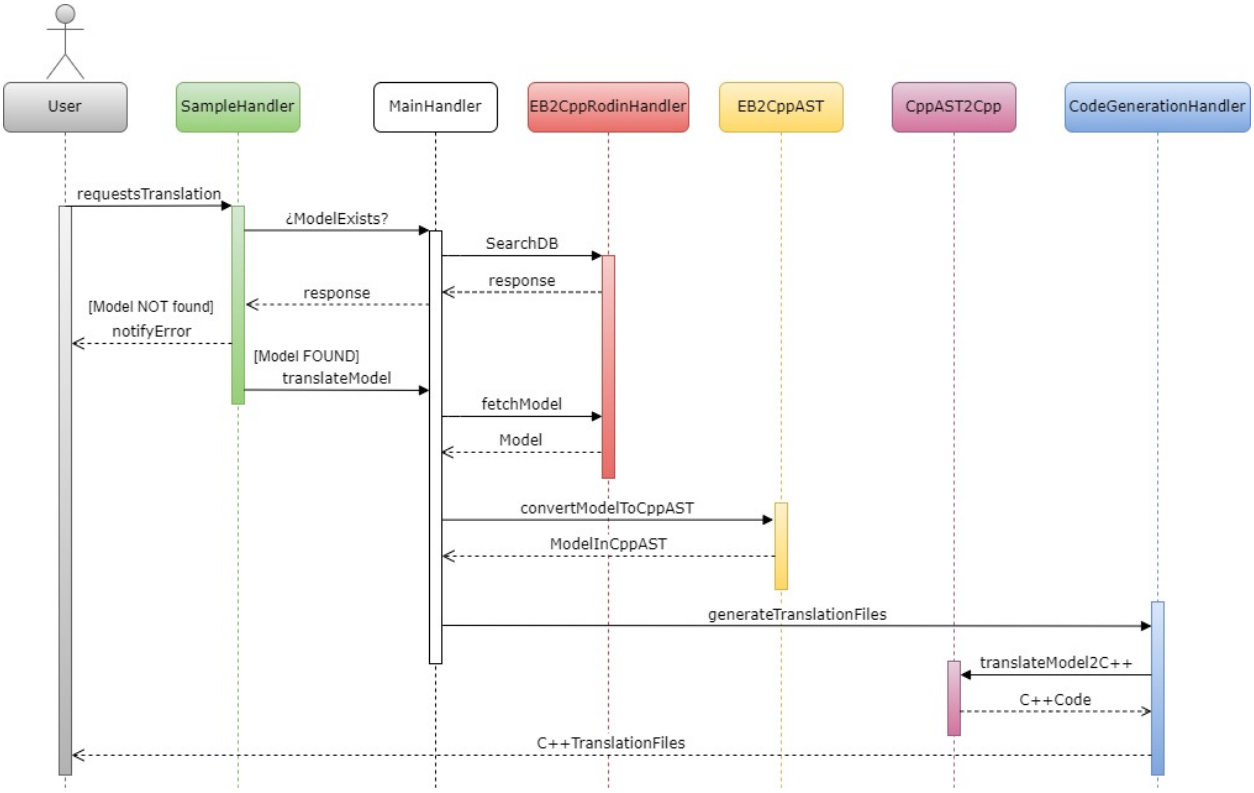


Figura 4.2: Diagrama de Secuencia

4. MainHandler recibe ese modelo, crea el CppAST y comienza a transferirle como entrada los distintos elementos del modelo obtenido por el EB2CppRodinHandler, construyendo en este proceso un AST del modelo a traducir (es una reorganización de la información que provee el AST de Rodin).

5. Cuando el árbol sintáctico está completo, el MainHandler crea el CodeGenerationHandler, y le entrega una referencia del CppAST construido con la cual generar el código de C++.

6. El usuario recibe las traducciones en su directorio de archivos.

En las siguientes secciones, se entrará en detalle acerca del diseño de cada componente, articulando las distintas decisiones que surgieron en su concepción.

### 4.1.1. EB2CppRodinHandler

Al inicio del proceso de diseño, se planteó un plugin que leería directamente el texto de los archivos que describen cada componente de Event-B. Por lo tanto, parte del plugin luego tendría un analizador sintáctico (parser) que leería el texto para entender el modelo descrito en él para ser

traducido después. Sin embargo, en el proceso de investigación acerca de las herramientas de la plataforma de Rodin, y particularmente el estudio de los trabajos previos en este campo, se encontró que esta estrategia no era la mejor idea, y que había un mejor plan para la lectura del modelo. Para explicar por qué la estrategia inicialmente planteada no sería ideal, es necesario explicar primero cómo Rodin almacena en su base de datos los modelos de Event-B.

Internamente en Rodin, la información de los modelos se guarda en un grupo de objetos pertenecientes a clases que representan cada tipo de elemento, y toda esta información es almacenada en una base de datos interna. Por ejemplo, si se tiene un modelo con un contexto, dicho contexto es representado como un objeto de clase SCContextRoot, que dentro contiene por cada constante un objeto tipo SCConstant, y por cada axioma un objeto tipo SCAxiom. Lo más importante es que cada uno de estos elementos (constantes y axiomas, en el anterior ejemplo) tiene un árbol de sintaxis abstracta que los describe.

Se decidió utilizar estos ASTs internos de Rodin, en vez de llevar a cabo una lectura directa sobre el texto de los archivos, debido a los beneficios que trae a la hora de traducir a C++. El principal beneficio es que simplifica mucho el trabajo a llevar a cabo. En el método de lectura directa de archivos, sería necesario programar un parser para convertir las cadenas de texto en información entendible para que el traductor use. Hacer el análisis sintáctico desde cero de dicha entrada pura de texto es muy complejo. En comparación, la programación de la navegación del AST de Rodin es algo más simple. Esta lectura de los ASTs es el método utilizado por las herramientas de traducción previas a este trabajo. Por lo tanto, usar el mismo método facilita el estudio y análisis de como esas herramientas resuelven distintos problemas que surjan durante el desarrollo.

El componente EB2CppRodinHandler es una clase que se encarga de la primera fase de traducción. El trabajo de esta clase es el de leer la base de datos de Rodin para poder obtener el modelo a traducir, en forma de objetos que representan cada componente (contextos y máquinas), y las relaciones de extensión-visibilidad-refinamiento entre ellos. Estos objetos contienen información que representa cada elemento del modelo, desde variables hasta predicados como invariantes. Estos elementos pueden ser suministrados al algoritmo de navegación del árbol sintáctico abstracto de Rodin (dicho algoritmo es el Visitor del AST de Rodin) para así poder descifrar qué describen y a qué serán traducidos después. También se puede ver a EB2CppRodinHandler como aquel responsable de manejar toda interacción entre el plugin y la base de datos de Rodin. Esto incluye cosas como conocer la dirección de la carpeta donde se encuentra el modelo siendo traducido, ver si un determinado proyecto existe, etc.

### 4.1.2. EB2CppAST

Este componente se encarga de suministrar los elementos obtenidos por el EB2CppRodinHandler al Visitor del AST de Rodin, y utilizar la navegación para construir un AST propio del plugin, que será llamado CppAST. Este almacena toda la información en un árbol de sintaxis abstracta, orga-

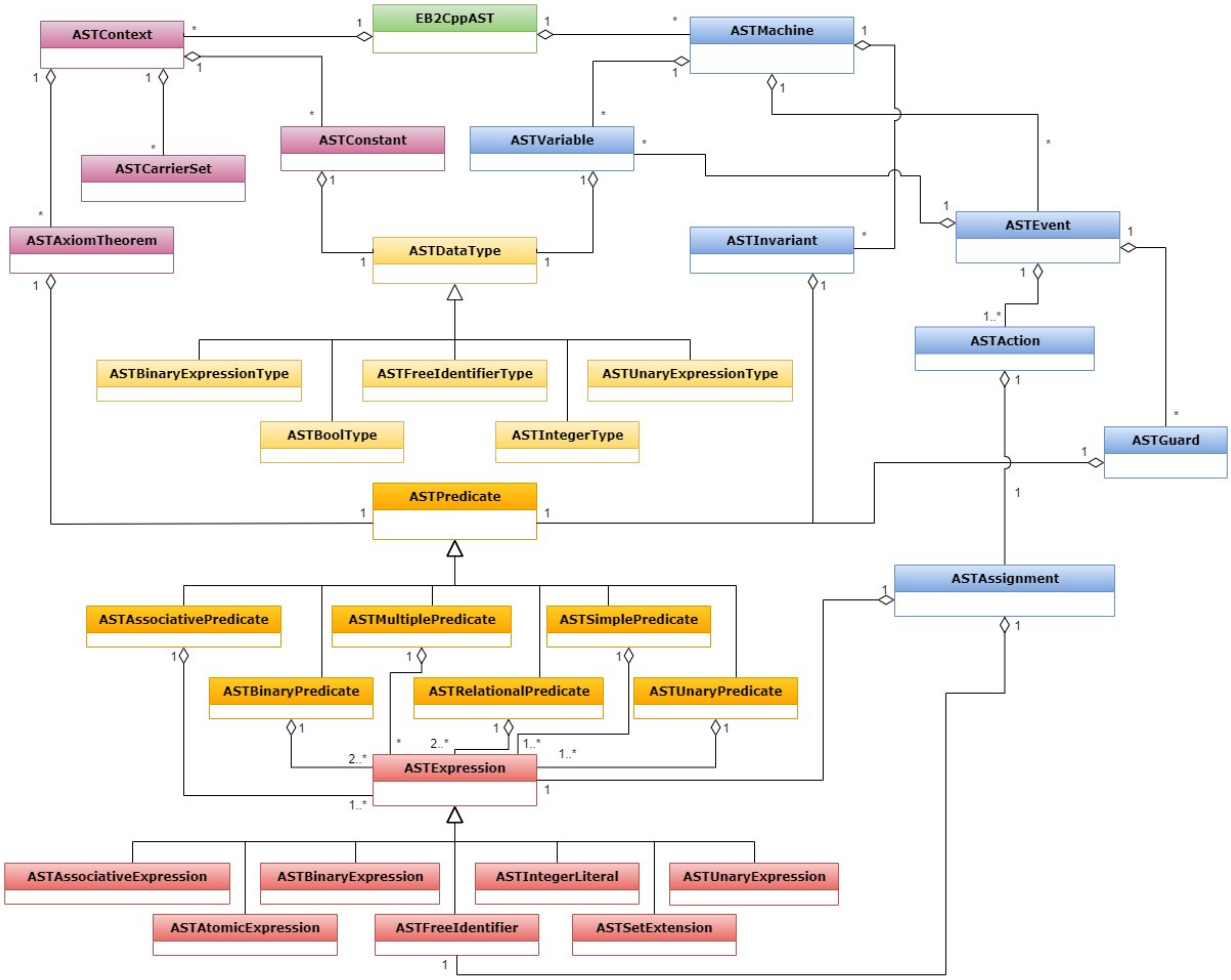


Figura 4.3: Diagrama de Clases conceptual del árbol de sintaxis abstracta CppAST.

nizado óptimamente para propósitos de la traducción, que luego puede ser leída por componentes en la tercera fase de traducción para generar código de C++. Este componente se encarga de la segunda fase del proceso de traducción. En la Figura 4.3 se puede ver la estructura de todos los elementos que componen el árbol de sintaxis abstracta CppAST. Esta estructura está basada en el AST de Rodin, asi que la Figura 4.3 sirve también como una aproximación a la estructura del AST de Rodin. Véase de esta forma: tras la extracción del EB2CppRodinHandler, tenemos un objeto que representa una invariante, con un atributo encriptado que describe el predicado que la invariante expresa, solamente descifrable por el Visitor de Rodin. Con su conversión al EB2CppAST, tenemos ahora un objeto que representa una invariante, con un atributo de la clase Relacional de tipo "Menor Que", y dos hijos que son el número 3 y el número 7. Aquí se puede entender que el invariante en cuestión es un predicado de la forma "3 es menor que 7".

El CppAST va a contener como atributos, objetos que representan los contextos y máquinas del modelo que se esté traduciendo. Éstos a su vez contendrán elementos como constantes, invariantes, eventos, etc. Para almacenar cada uno de estos, se crean clases para cada uno: ASTVariable para variables, ASTInvariant para invariantes, y así para el resto. Para representar los tipos de datos, expresiones y predicados en que consisten los elementos anteriores, se usó como guía en la creación y nombramiento de clases la forma en la que el Visitor del AST de Rodin clasifica las distintas expresiones (sumas, uniones de conjuntos, igual-que, etc).

Por ejemplo, el Visitor anteriormente mencionado clasifica la suma entre enteros bajo una categoría de expresiones asociativas. La multiplicación entre enteros, y la intersección entre conjuntos también caen en esta categoría. En consecuencia, dentro del CppAST se tiene una clase llamada ASTAssociativeExpression. Cuando se esté recorriendo un predicado u expresión, y se encuentra una suma, se creará un objeto de esta clase ASTAssociativeExpression en la cual se almacenan todos los operandos de la suma, al igual que un identificador que indique que la operación de esta expresión asociativa es de "suma entre enteros".

Las distintas categorías de expresiones y predicados (asociativas, binarias, relacionales, etc) van a heredar de dos clases padres: ASTExpression y ASTPredicate, respectivamente. De esta forma, se puede usar el principio de sustitución de Liskov para definir los distintos atributos de los nodos (los hijos del nodo de una suma, por ejemplo) para que sean del tipo de estas clases padre, y poder así englobar toda posible construcción y ensamblaje de expresiones.

### 4.1.3. CodeGenerationHandler

El último componente del plugin, que se encarga de la tercera y última etapa del proceso de traducción, es una clase llamada CodeGenerationHandler. Esta clase crea archivos y escribe en ellos textos que, en su totalidad, generan un programa de C++ que hace una labor equivalente a la del modelo siendo traducido.

Inicialmente, se decidió que la misma clase CodeGenerationHandler iba a encargarse de seleccionar el código de C++ a generar. Pero, al analizar la clase se pudo observar que se estaba encargando de dos tareas que no necesariamente deben realizarse en la misma clase: la conversión de la información del CppAST a código de C++, y la escritura de dicho código en archivos. De acuerdo a buenas prácticas en la programación orientada a objetos, preferiblemente se quieren separar responsabilidades entre distintas clases, en vez de crear una clase que se encarga de muchas tareas a la vez. Con esto en mente, se agregó al diseño la clase CppAST2CppBuilder, que se encarga de construir expresiones en C++ que sean la traducción de expresiones encontradas en el modelo a traducir en el CppAST. Luego, CodeGenerationHandler se encarga de la escritura de dichas expresiones a archivos, manejando también aspectos como la indentación del texto, líneas en blanco, comentarios, etc.

## 4.2. Diseño de código C++ generado

Tras la anterior definición del plugin que convierte un modelo en Rodin a información manipulable en la forma del CppAST, es necesario ahora definir cómo representar un modelo de Event-B en C++, para que así el CodeGenerationHandler y el CppAST2CppBuilder sepan qué texto generar. Para hacer esto, hay que establecer las equivalencias entre los elementos de modelos de Event-B y expresiones en C++, e identificar si hay características de Event-B que van a requerir de herramientas adicionales para poder replicarse en C++.

Temprano en el proceso de diseño, y en la investigación que se hizo a la par, se pudo discernir que había dos grandes direcciones que el proyecto podía tomar. El plugin podía realizar traducciones: (1) solamente sobre modelos que habían sido refinados al punto de ya ser programas (consisten solo de guardas y acciones simples y determinísticas), o (2) sobre cualquier modelo, incluso si tienen guardas y acciones complejas y no determinísticas. A estas dos opciones de diseño se les referirá de ahora en adelante como la primera y la segunda dirección de diseño, respectivamente.

En la primera dirección, las traducciones resultantes son programas que compiten en eficiencia con programas que hubiesen sido programados directamente sin usar Event-B. En la segunda dirección, la traducción resultante es un simulador. Esta salida desempeña una tarea similar a la que hace ProB: la herramienta oficial de Rodin usada para crear una animación de los modelos y que permite hacer pruebas concretas sobre su comportamiento. Es importante aclarar que, al usar un plugin que adopte la segunda dirección de diseño, sobre un modelo que ya es un programa, la traducción resultante también sería algo muy cercano a un programa desarrollado normalmente sin Event-B.

Al investigar los otros traductores automáticos que se han desarrollado para Rodin, se pueden ver ambas versiones de diseño. Hay traductores que solamente funcionan sobre modelos que ya son programas (como la traducción a C [5] y el traductor de C++ de EB2ALL [11]), y también hay traductores que sirven para todo modelo, generando programas que son efectivamente simuladores en el caso de traducir modelos complejos (el traductor a Python [3] y el traductor a JML [13]).

Escoger uno sobre otro es un asunto de entender y asumir la carga adicional de trabajo que cada una implica, y también decidir cuál es la utilidad que se quiere que este plugin brinde.

Elegir que el plugin solo pueda traducir modelos que ya son programas claramente reduce mucho la utilidad de la herramienta. También, esta restricción demanda del usuario saber cómo se refina un modelo hasta el punto de volverlo un programa. Por otra parte, esta dirección podría hacer más fácil la programación del traductor. No habría que programar la traducción de expresiones y guardas complejas, ya que éstas no están presentes en modelos que ya son programas. Lo opuesto es cierto si el plugin va a traducir cualquier modelo. También, el resultado final de un plugin que solo traduzca modelos simples es más directamente útil para el cliente. Es decir, como el código generado ya es un programa (competitivo con uno que fuese desarrollado sin usar Event-B), y no un simulador, es más viable el acoplarlo directamente a un sistema de software real que el usuario está construyendo a base del modelo. En el caso del traductor que genera un simulador, la utilidad del programa a la hora de ser usado en sistemas de software reales es tal vez diferente a la imaginada originalmente. Este código generado sirve más la labor de ser una aproximación a la implementación real del modelo.

Finalmente, tras considerar esta balanza de pros y contras, se decidió adoptar la segunda dirección de diseño descrita anteriormente. Es decir, el plugin va a traducir cualquier modelo, incluso uno con guardas y acciones complejas; en el caso del cual genera un programa de C++ que es efectivamente un simulador, como lo que hace el animador ProB. Con esta decisión de diseño tomada, se prosigue a hablar del diseño específico que tendrán los archivos de C++ generados por el plugin.

Como se ha abordado anteriormente, un modelo en Event-B está hecho de componentes: contextos y máquinas. El plan entonces es que, para cada componente del modelo, se genere un archivo de C++ que implementará una clase que representa el componente. Si un contexto a traducir se llama baseCtx, se generará un archivo llamado baseCtx.cpp, en el cual se define una clase llamada baseCtx.

Las relaciones de refinamiento y acceso entre componentes se manejarán utilizando herencia de clases. Así que, si un contexto B es una extensión de un contexto A, B será una subclase de A. Lo mismo entre refinamientos de máquinas y máquinas accediendo a algún número de contextos. De esta forma, toda referencia a constantes y variables que no pertenecen al componente que los está utilizando, y que por lo tanto están en otro ámbito, serán accesibles a través de la herencia. En el ámbito de los archivos que representan contextos, tenemos que definir cómo se representan los carrier sets (conjuntos definidos por el usuario), constantes y axiomas:

- Los carrier sets serán representados usando una enumeración. Este es un tipo de dato creado por el usuario en C++, que mapea un número finito de expresiones a valores numéricos únicos y diferentes. Esta representación imita lo que hace ProB. En esa herramienta, los carrier sets se inicializan con unos elementos finitos que se manipulan en eventos. Esta representación elegida de carrier sets en C++ es evidencia de que los componentes de C++ generados por este plugin

son últimamente una aproximación a la implementación de los modelos de Event-B. En un software real hecho a base de un modelo en Event-B, los carrier sets probablemente representan clases que los desarrolladores completarán con información en forma de atributos. Por ejemplo, si se modela en Event-B el sistema de un banco con personas, sus cuentas, y el dinero en cada cuenta; probablemente se creará un carrier set de "Personas" para representar a los clientes. Al implementar un sistema de software real basado en este modelo, estas entidades de "Personas" probablemente serán objetos de una clase "Persona" con atributos como el nombre completo de la persona y un identificador numérico único. Como el plugin de traducción automática solo puede leer lo descrito en el modelo de Event-B a la hora de generar código, no se puede traducir carrier sets a clases con atributos detallados.

- Las constantes serán variables que son atributos de la clase del contexto. La declaración del tipo de dato de la variable se hace de acuerdo al tipo de dato descrito en el CppAST para esa constante. Por ejemplo, si un contexto tiene una constante "capacidad" que es un entero, se traduce a:

  ```
  int capacidad;
  ```

  El valor inicial de la constante será definido por el usuario. Para guiar al usuario en la inicialización de la constante, se genera un comentario demostrando como hacerlo, siendo una guía específica al tipo de dato de la constante.

- Los axiomas serán funciones que retornan un booleano. Dichas funciones retornan el resultado de evaluar el predicado descrito en el axioma. Por cada axioma se genera una función como las anteriormente descritas. Por ejemplo: si se tiene un axioma con la etiqueta @ax0:

  ```
  @ax0: capacidad <10,
  ```

  se traduce a:

  ```
  bool checkAxiom_ax0() { return capacidad<10;  }
  ```

  Luego de esto, se genera una función que verifica todos los axiomas en el contexto y declara si todos se cumplen. La idea es hacer un llamado a esta función después de la inicialización de las constantes, para poder así verificar que los valores introducidos por el usuario satisfacen los axiomas.

Ahora, en el ámbito de los archivos que representan máquinas, se define la representación de variables, invariantes y eventos de la siguiente manera:

- Las variables de la máquina serán variables de C++ que son atributos de la clase de la máquina. La declaración del tipo de dato de la variable se hace de acuerdo al tipo de dato descrito en el CppAST para esa variable. Por ejemplo, si se tiene una variable admin_id que es un

entero, se traduce a:

```
int admin_id;
```

El valor inicial de la variable se establece en la traducción del evento de inicialización, el cual se convoca en el método constructor de la máquina.

- Los invariantes serán funciones que retornan un booleano, que es el resultado de evaluar el predicado descrito en el invariante. Por cada invariante se genera una función como las anteriormente descritas. Por ejemplo, si se tiene un invariante con la etiqueta @inv0:

  ```
  @inv0: ids_usadas = {},
  ```

  se traduce a:

  ```
  bool checkInvariant_inv0() { return ids_usadas.esVacio(); }
  ```

  Luego de esto, se genera una función que verifica todos los invariantes en la máquina y declara si se cumplen todos. Esta verificación de todos los invariantes debería convocarse después de cada llamado a un evento.

- Los eventos serán dos funciones: una que verifica las guardas del evento y retorna si se cumplen o no; y otra que ejecuta las acciones del evento. La primera función retorna un booleano, y la segunda no retorna nada (es una función void). Ambas reciben como parámetros de entrada los parámetros any del evento. El tipado de estos parámetros se deduce a partir del CppAST, ya que de la misma forma en la que se puede usar el Visitor del AST de Rodin para encontrar el tipado de constantes y variables, también se puede encontrar el tipo de dato de los parámetros de un evento.

### 4.2.1. Ejemplo de modelo en Event-B

Para articular mejor en la práctica los conceptos explicados en la sección anterior, se presentará un ejemplo de un modelo diseñado en Event-B (ver Figura 4.4), y se va a señalar cómo se traduce cada uno de sus elementos (ver Figura 4.5). El modelo en cuestión será el de un sistema de control para un semáforo, que controla cuando se le da permiso a los carros y a los peatones para moverse.

### 4.2.2. Herramientas adicionales para traducción C++

Un problema que destaca al pensar en cómo traducir modelos de Event-B a C++, es que hay varias funcionalidades de modelos formales que no tienen una representación directa y simple en un lenguaje de programación como C++. Se puede imaginar fácilmente como representar números enteros, por ejemplo, pero no es el mismo caso para algo como conjuntos, relaciones; y las operaciones que se pueden efectuar con estos, tales como la intersección, cálculo de dominio, composición,

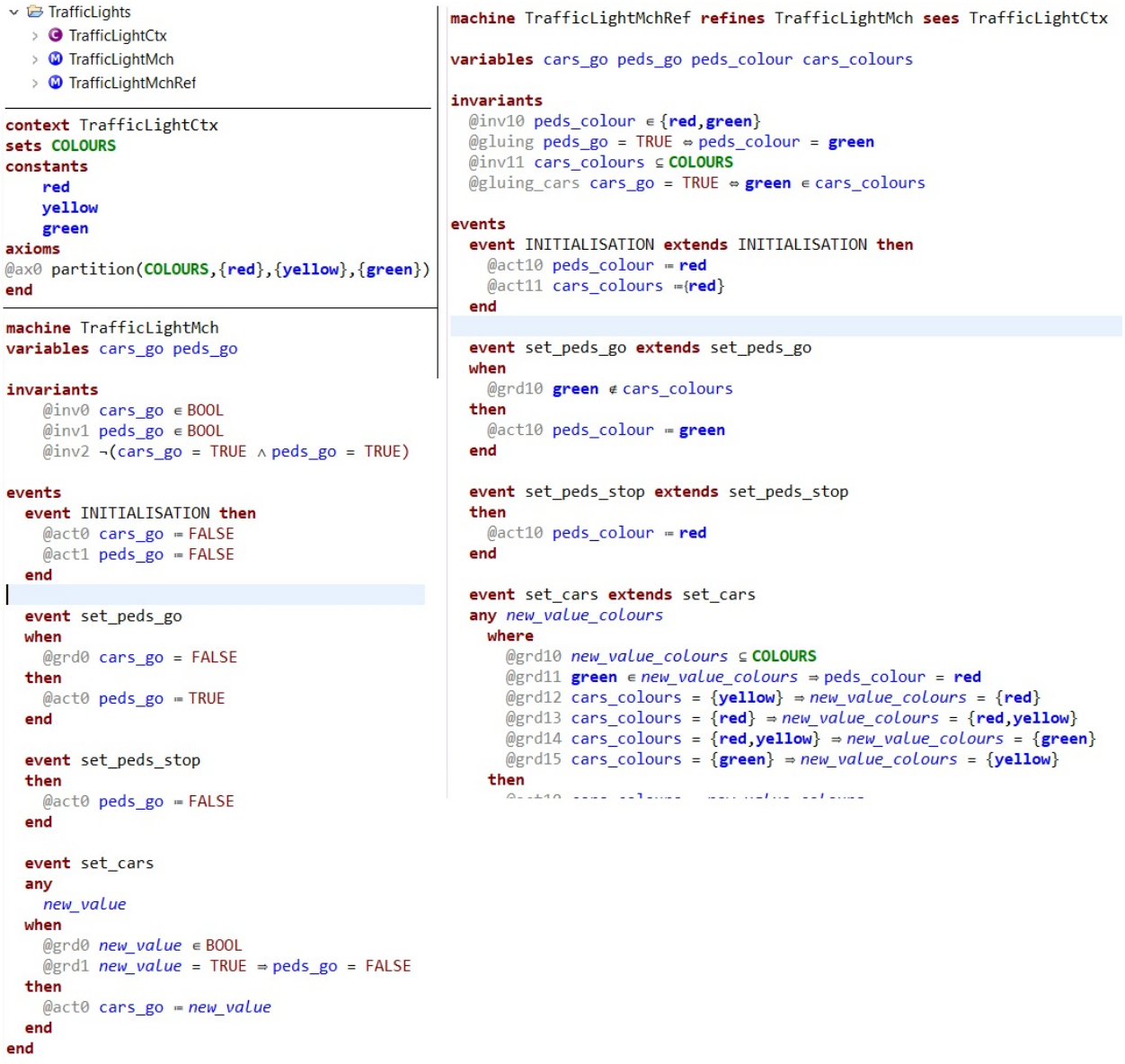


Figura 4.4: Modelo de Event-B: TrafficLights (solo se ilustra parte de la máquina refinada)

| Modelo | Traducción C++ |
|---|---|
| Modelo contiene tres componentes: TrafficLightCtx (contexto), TrafficLightMch y TrafficLightMchRef (máquinas). | Genera seis archivos en directorio de archivos (un .cpp y un .h por cada componente), llamados TrafficLightCtx.cpp/.h, TrafficLightMch.cpp/.h y TrafficLightMchRef.cpp/.h. Cada par de archivos .cpp y .h declara e implementa una clase. |
| La máquina TrafficLightMchRef tiene acceso al contexto TrafficLightCtx. | class TrafficLightMchRef: public TrafficLightCtx { ... } |
| La máquina TrafficLightMchRef refina la máquina TrafficLightMch. | class TrafficLightMchRef: public TrafficLightMch { ... } |
| El contexto define un carrier set llamado COLOURS. | enum COLOURS {colours1, colours2...} |
| El contexto tiene tres constantes: red, yellow, green. Pertenecen al carrier set COLOURS. | COLOURS red; COLOURS yellow; ... |
| El contexto tiene un axioma @ax0. El predicado descrito es partition(COLOURS,{red},{yellow},{green}) | bool checkAxiom_ax0 { return COLOURS.IsPartition(red, yellow, green); } |
| La máquina TrafficLightMch tiene las variables cars_go y peds_go. Son booleanos. | bool cars_go; bool peds_go; |
| La máquina TrafficLightMch tiene las invariantes @inv0, @inv1, @inv2. | bool checkInvariant_inv0 { ... } bool checkInvariant_inv1 { ... } ... |
| La máquina TrafficLightMch tiene un evento set_cars. Este evento consiste de un parámetro de entrada booleano new_value, una guarda @grd0, y una acción @act0. | bool set_cars_checkGuards(bool new_value) { return is_grd0_True(); }<br><br>void set_cars(bool new_value) { (traducción de acción @act0) } |

Figura 4.5: Tabla de equivalencia entre modelo de Event-B TrafficLights (ver Figura 4.4) y su traducción a C++.

obtención de imagen relacional, etc. Esto indica que será necesario ubicar o programar un complemento, una librería, que implemente todas estas funcionalidades en C++, que será utilizado por las traducciones.

Los detalles minuciosos de cómo se logra la representación de estas distintas herramientas se abordará en el capítulo describiendo la implementación. Sin embargo, se quiere que en el proceso de diseño se establezca una estrategia correcta para solucionar el problema.

El punto clave del problema tiene que ver con los tipos de datos, estructuras y operaciones que se usan en Event-B. En particular, los conjuntos en Event-B son un tipo de dato esencial para muchas de las operaciones de los modelos, y es necesario identificar la mejor forma de representarlos en C++. Al contemplar las estructuras de datos que se tienen en C++ para representar esto, a primera vista se tienen los arreglos. Un conjunto, visto de una manera simple, es una colección de datos del mismo tipo. Los arreglos hacen esto, sin embargo, en los detalles se demuestran sus limitaciones. Los arreglos son estructuras estáticas ya que cuando se define un arreglo y sus contenidos, no se puede luego durante la ejecución del código agregar un elemento al arreglo (para lograr esto sería necesario la manipulación explícita de memoria lo cual no es una operación eficiente). Preferiblemente se quiere tener más libertad con una estructura de datos más robusta que sea dinámica.

Al realizar una búsqueda entre distintas estructuras de datos contenedoras en C++ como vectores, arreglos y listas; se seleccionó la estructura `set` como la más adecuada por las siguientes razones (de aquí en adelante se referirá a `set` como conjuntos en C++):

- Los conjuntos en C++ permiten coleccionar datos dinámicamente, lo cual descarta a los arreglos como opción. Los vectores y listas también son contenedores de tamaño dinámico.
- Los conjuntos en C++ no permiten duplicados, que es una restricción que aplica para los conjuntos en Event-B. Esta característica de no permitir duplicados descarta la posibilidad de usar vectores, arreglos y listas, ya que estos no aplican esta restricción. Es posible implementar esta restricción en vectores y listas a través de un código personalizado, pero esto no es óptimo en cuanto a complejidad computacional y las necesidades de las traducciones.
- Al hablar de la complejidad computacional de la estructura de datos, hay que considerar el costo de agregar elementos, encontrar elementos, y las operaciones de unión/intersección/diferencia de conjuntos (solo estas operaciones serán usadas en la implementación C++ de las operaciones de Event-B).
    - *Agregar elemento*: Los vectores y listas agregan un elemento en tiempo constante. Los conjuntos en C++ agregan un elemento en tiempo logarítmico en función del tamaño del conjunto (la estructura utiliza su implementación interna de árbol binario para optimizar el costo computacional). Pero, debe resaltarse que el conjunto en C++ agrega en tiempo logarítmico porque es una estructura ordenada, y debe tomarse el tiempo para insertar el elemento en el lugar correcto. Más importante, es en este proceso que determina

si el elemento a insertar ya se encuentra en el conjunto, evitando así duplicados. Si implementamos una versión de inserción de elementos con vectores o listas para evitar duplicados, agregar en un vector/lista se vuelve lineal. Es por esto que al tomar en cuenta la restricción de evitar duplicados, los conjuntos en C++ agrega elementos más óptimamente.

- *Encontrar elemento*: Para realizar operaciones de pertenencia de conjuntos, se necesita saber si un elemento dado está en el contenedor de datos. Los vectores y listas deben hacer un recorrido lineal a través de los datos para encontrar un elemento. Los conjuntos en C++ encuentran un elemento en tiempo logarítmico en función al tamaño del conjunto.
- *Unión/intersección/diferencia de conjuntos*: C++ tiene una biblioteca llamada algorithm que implementa la unión, intersección y diferencia de conjuntos (set_union, set_intersection, set_difference). Estas operaciones son lineales en función del tamaño de ambos conjuntos. Pero un aspecto de estas operaciones de algorithm es que requieren que los rangos de datos a operar estén ordenados. Esto hace que se puedan usar con los conjuntos en C++, que automáticamente ordenan los datos agregados, pero no inherentemente con vectores y listas.

Con tal de representar todas las operaciones que se pueden realizar sobre conjuntos y relaciones de Event-B, se deciden crear clases personalizadas para estas dos estructuras de datos, que tendrán como atributo un conjunto en C++ para coleccionar los elementos, y contendrán métodos para implementar cada operación. Estas clases serán plantillas: reciben tipos de datos como parámetros que son usados luego por miembros de la clase. De esta forma, solo se define una especificación genérica para la clase "conjunto" que luego se puede instanciar en objetos que son "conjuntos de enteros", o "conjuntos de conjuntos de enteros", o "conjuntos de parejas de enteros", entre otros.

Capítulo 5

# Implementación

En este capítulo se hablará de la complejidad computacional del proceso de traducción y del programa que se obtiene como resultado.

## 5.1. Proceso de traducción

Primero hablaremos de la implementación del proceso de traducción explicado en el capítulo anterior. Recordemos que la traducción se divide en tres fases: extraer el modelo a traducir de la base de datos de Rodin (EB2CppRodinHandler), realizar análisis sintáctico (EB2CppAST), y generar archivos en C++ (CodeGenerationHandler).

### 5.1.1. EB2CppRodinHandler

Para extraer el modelo a traducir, es necesario primero encontrar el modelo que el usuario nos pide traducir. Esto se hace iterando por todos los proyectos que el usuario tiene en su espacio de trabajo en Rodin, hasta encontrar el proyecto con el nombre solicitado (ver Figura 5.1). Este recorrido es lineal en función del número de proyectos en el espacio de trabajo del usuario.

Una vez se ubica el proyecto a traducir, hay que extraer los contextos y máquinas que contiene. Para hacer esto se navega por cada archivo en el proyecto y cada vez que encuentre un componente se detiene para extraerlo. Este recorrido es lineal en función del número de archivos en el proyecto.

Un aspecto importante de la extracción es que una vez que se obtengan todos los contextos y máquinas, éstos deben organizarse en el orden en el que deben ser traducidos, basándose en las relaciones de extensiones y refinamientos entre ellas. Por ejemplo, si un contexto B refina un contexto A, debe establecerse que A debe traducirse antes que B. Esta reorganización se realiza con un ordenamiento topológico, calculando previamente los grados de incidencia de cada componente utilizando las relaciones de refinamiento entre ellos y organizándolos en tiempo lineal.

### 5.1.2. EB2CppAST

El análisis sintáctico del modelo se divide en el procesamiento de los contextos y de las máquinas. Primero se analizará la complejidad del procesamiento de contextos y máquinas, y después se hablará del Visitor de Rodin que ambos procesos usan para obtener la información necesaria de los axiomas, invariantes, eventos, etc.

```java
//Extract contexts of target project from RodinDB
public void fetchContexts() throws RodinDBException {

    for (IRodinElement projectChild : rodinProject.getChildren()) { // O( files in project )
        IInternalElement elementRoot = ((IRodinFile)projectChild).getRoot();

        if (elementRoot instanceof SCContextRoot) {
            SCContextRoot contextRoot = (SCContextRoot) elementRoot;
            String contextName = elementRoot.getElementName();
            contexts.put(contextName, contextRoot);

            //See which contexts it extends
            ArrayList<String> emptyListExtensions = new ArrayList<String>();
            contextsExtensions.put(contextName, emptyListExtensions);
            for (ISCExtendsContext extendClause : contextRoot.getSCExtendsClauses()) {
                try {
                    String extendedContextName = extendClause.getAbstractSCContext().getComponentName();
                    contextsExtensions.get(contextName).add(extendedContextName);
                }
                catch (CoreException e) {
                    e.printStackTrace();
                }
            }

        }
    }
```

Figura 5.1: Parte del código de RodinHandler, donde se ilustra el proceso de navegar los archivos de un proyecto para extraer los contextos.

#### 5.1.2.1. Procesamiento de contextos

Se navega por cada contexto de acuerdo al orden de traducción establecido en el paso anterior (ver Figura 5.2). Para cada uno, se itera sobre los otros contextos que extiende, carrier sets, constantes y axiomas que especifica para obtener la información necesaria y organizarla en el CppAST. Por lo tanto, el costo computacional de este proceso es lineal en función del número de contextos a traducir. La complejidad del procesamiento de cada contexto individual depende de qué tantos elementos tenga (como constantes y axiomas) y qué tan complejas sean las fórmulas que describen cada elemento (cómo se explicará en la sección 5.1.2.3).

#### 5.1.2.2. Procesamiento de máquinas

Se navega por cada máquina de acuerdo al orden de traducción establecido en el paso anterior (ver Figura 5.3). Para cada uno, se itera sobre los contextos que lee, máquinas que refina, variables, invariantes y eventos que especifica para obtener la información necesaria y organizarla en el CppAST. En consecuencia, el costo computacional de este proceso es lineal en función del número de maquinas a traducir. La complejidad del procesamiento de cada máquina individual depende de qué tantos elementos tenga (variables, invariantes, eventos) y qué tan complejas sean las fórmulas que describen cada elemento (cómo se explicará en la sección 5.1.2.3).

```
//Using the sorted translation order, obtain each name to store each into an AST
for (String contextName2Translate : rodinHandler.getContextNamesOrdered()) {
    context2Translate = rodinHandler.getContextRoot(contextName2Translate);

    extendedContextsNames = contextExtensions.get(contextName2Translate);


    System.out.println("CREATING EMPTY CONTEXT INTO AST...");
    ASTContext contextBeingBuilt = CppAST.addContext(contextName2Translate, extendedContextsNames);

    System.out.println("TURNING CARRIER SETS INTO AST...");

    //Turn carrier sets to CPPAST.
    for (ISCCarrierSet carrierSet : context2Translate.getSCCarrierSets()) {
        try {
            CppAST.addCarrierSetToContext(carrierSet, contextBeingBuilt);
        }
        catch (CoreException  e) {
            e.printStackTrace();
        }
    }
```

Figura 5.2: Parte del código donde se ilustra la iteración por cada contexto y la organización de los carrier sets en ellos.

```
//Using the sorted translation order, obtain each name to store each into an AST
for (String machineName2Translate : rodinHandler.getMachinesOrdered()) {
    machine2Translate = rodinHandler.getMachineRoot(machineName2Translate);

    refinedMachinesNames = machineRefinements.get(machineName2Translate);

    if (machineSeenContexts.containsKey(machineName2Translate))
        seenContexts = machineSeenContexts.get(machineName2Translate);

    System.out.println("CREATING EMPTY MACHINE IN CPPAST...");
    ASTMachine machineBeingBuilt = CppAST.addMachine(machineName2Translate, refinedMachinesNames, seenContexts);

    System.out.println("TURNING VARIABLES INTO CPPAST...");
    //Turn variables to CPPAST
    for (ISCVariable variable : machine2Translate.getSCVariables()) {
        try {
            CppAST.addVariableToMachine(variable, machineBeingBuilt);
        }
        catch (CoreException e) {
            e.printStackTrace();
        }
    }
```

Figura 5.3: Parte del código donde se ilustra la iteración por cada máquina y el procesamiento de las variables en ellos.

```
switch(predicateTag) {
case Formula.LIMP: //251
    if (isGettingPredicateOrExpression)
        newBinaryPredicate = new ASTBinaryPredicate("LogicalImplication");
    break;
case Formula.LEQV: //252
    if (isGettingPredicateOrExpression)
        newBinaryPredicate = new ASTBinaryPredicate("LogicalEquivalence");
    break;
default:
    hasDefaultError = true;
    if (isGettingPredicateOrExpression)
        predicateBeingFound.setAsError();
}

if (!hasDefaultError) {
    if (isGettingPredicateOrExpression) {
        predicate.getLeft().accept(this);
        newBinaryPredicate.setLeftSide(predicateBeingFound);

        predicate.getRight().accept(this);
        newBinaryPredicate.setRightSide(predicateBeingFound);

        predicateBeingFound = newBinaryPredicate;
```

Figura 5.4: Parte del código donde se ilustra la lectura por profundidad de un predicado binario, como una implicación lógica.

#### 5.1.2.3. EB2CppVisitor

La parte más compleja del EB2CppAST es el análisis de las distintas expresiones de Event-B: constantes, axiomas, variables, invariantes y eventos. El traductor debe leer las fórmulas de estos elementos y clasificar su contenido de tal forma que pueda después automáticamente generar el código de C++ correspondiente.

A continuación, se presenta una explicación simplificada del algoritmo de EB2CppAST para la lectura de expresiones y predicados de Event-B, las cuales se llamarán fórmulas de ahora en adelante. Rodin ofrece una interfaz llamada ISimpleVisitor que puede implementarse por un desarrollador para explorar los árboles sintácticos de cualquier fórmula. Como su nombre lo indica se basa en el patrón de diseño Visitor, pues en lugar de implementar la navegación de este árbol en el objeto de la fórmula en sí, se implementa en esta entidad separada. Nuestra implementación de esta interfaz se llama EB2CppVisitor. Mientras se navega el árbol sintáctico de una fórmula, dependiendo del tipo de nodo en el que se encuentre (si es una suma, equivalencia lógica, número entero...), el Visitor ejecutará la función de visita correspondiente y de esta forma continua navegando el árbol, recopilando la información necesaria para poder decirle al traductor todos los elementos que componen la fórmula (ver Figura 5.4 para un ejemplo de código al visitar un predicado binario como lo es una implicación lógica).

Ya que el algoritmo es básicamente una navegación por profundidad, la complejidad del análisis de una fórmula es lineal en función del número de elementos y construcciones que contiene (es decir, el número de vértices en el árbol). Por ejemplo, analizar el predicado $3 == 2$ es simple porque solo contiene tres elementos: un igual que y los dos números enteros. Por otra parte, el predicado $(1 \mapsto 4) \in (\{1,2,3\} \nrightarrow (\mathbb{N}_1 \cup \{-1,-2,-3\}))$ es más complejo, requiriendo navegar una pertenencia de conjuntos, una pareja de enteros, los dos enteros individuales en la pareja, una unión de conjuntos, y así en adelante hasta llegar a todos los elementos terminales.

### 5.1.3. CodeGenerationHandler

Cuando toca generar el código de C++, la complejidad del algoritmo es muy similar al de la segunda fase de traducción ya que ambos implican navegar un AST. A lo largo de este proceso, se utiliza StringBuilder para componer las líneas de texto a ser generadas, ya que esta clase permite crear cadenas de texto mutables que anexan diferentes porciones de texto más eficientemente que al usar concatenación de cadenas de texto directamente.

El traductor itera linealmente por cada componente del modelo y en cada uno navega por los elementos que contiene, usando la información recopilada y reorganizada de la fase anterior para rellenar los campos vacíos en una secuencia de instrucciones de C++ predefinidas (ver Figura 5.5). Cuando se llega al momento en la generación del archivo donde se necesita traducir una fórmula, se solicita a CppAST2CppBuilder que construya la cadena de texto a través de una navegación recursiva del AST de esa fórmula (ver Figura 5.6). Debido a esta navegación recursiva, la generación de la traducción de una fórmula es lineal en función del número de nodos en su AST (o en otras palabras, el números de construcciones sintácticas en la fórmula).

Para finalizar esta sección, recordemos que se deben generar dos archivos por componente: uno de cabecera (header) y uno de implementación. Por lo tanto, la mayoría de estos procesos de generación deben ejecutarse dos veces por cada componente.

## 5.2. Traducción C++

En esta sección hablaremos de la complejidad de ejecutar el programa generado por el traductor. Como se mencionó en el capítulo de diseño, el traductor genera un archivo EB2CppTools que implementa en C++ las diversas funciones matemáticas de Event-B, y dos archivos por cada componente del modelo (un archivo de cabecera y uno de implementación). Los métodos de los componentes traducidos son tan complejos computacionalmente como las operaciones de Event-B que aparecen en ellos. Por ejemplo, el método C++ que es la traducción de un axioma $card(clientes)! = 0$ es tan complejo computacionalmente como lo es la operación de *card* y desigualdad en C++. Por lo tanto, para saber que tan compleja es la ejecución de los distintos métodos es más útil hablar de la implementación de las distintas operaciones de Event-B en C++.

```
for (ASTAxiomTheorem axiom : axioms.values()) {
    String axiomLabel = axiom.getLabel();

    writeLine(indentTier,"// AXIOM: " + axiomLabel);

    //Declaration of axiom function
    functionLine = new StringBuilder();
    functionLine.append("bool ");
    if (!isHeaderFile) {
        functionLine.append(context.getContextName());
        functionLine.append("::");
    }
    functionLine.append("checkAxiom_");
    functionLine.append(axiomLabel);
    functionLine.append("()");

    if (isHeaderFile)
        functionLine.append(";");
    else
        functionLine.append(" {");

    writeLine(indentTier,functionLine.toString());

    if (!isHeaderFile) {
        //Axiom return line
        String axiomLine = AST2Cpp.generatePredicate(axiom.getPredicate());

        functionLine = new StringBuilder();
        functionLine.append("return ");
        functionLine.append(axiomLine);
        functionLine.append(";");

        writeLine(indentTier+1, functionLine.toString());

        writeLine(indentTier,"}");
    }
}
```

Figura 5.5: Parte del código donde se ilustra el proceso de generar la sección de axiomas para un contexto, y la composición de una línea de texto a escribir.

```
604  public String generateExpression(ASTExpression expression) {
605      String result = "";
606
607      StringBuilder builtResult = new StringBuilder();
608
609      // A expression will always be surrounded by parenthesis
610      builtResult.append("(");
611
612      String expressionType = expression.getType();
613
614      switch(expressionType) {
615      case "AssociativeExpression":
616          ASTAssociativeExpression associativeExp = (ASTAssociativeExpression) expression;
617
618          int childIndex = 0;
619
620          switch(associativeExp.getAssociativeType()) {
621          case "Addition":
622              for (ASTExpression child : associativeExp.getChildExpressions()) {
623                  if (childIndex != 0)
624                      builtResult.append(" + ");
625                  builtResult.append(generateExpression(child));
626
627                  childIndex += 1;
628              }
629              break;
```

Figura 5.6: Parte del código donde se ilustra la composición de la generación de una suma aritmética. Nótese el llamado recursivo a la misma función generateExpression en la línea 625.

### 5.2.1. EB2CppTools

A continuación se presenta una tabla que lista cada operación de Event-B para la cual hay una traducción, y su complejidad algorítmica correspondiente usando notación Big O. Cabe resaltar que en esta tabla no se incluyen ciertas traducciones que tienen una equivalencia directa y obvia en C++, como las sumas, restas, menor que, igual que, etc.

| Tabla de complejidad algorítmica de traducción | | |
|---|---|---|
| Fórmula Event-B | Traducción C++ | Complejidad algorítmica |
| $x \in Y$ | Y.contains(x); | O(log n) n: tamaño de conjunto |
| $x \notin Y$ | Y.notContains(x); | O(log n) n: tamaño de conjunto |
| $X \subset Y$ | Y.hasSubset(X); | O(n+m) n,m: tamaño de conjuntos |
| $X \not\subset Y$ | Y.hasNotSubset(X); | O(n+m) n,m: tamaño de conjuntos |
| $X \subseteq Y$ | Y.hasSubsetOrEqual(X); | O(n+m) n,m: tamaño de conjuntos |
| $X \not\subseteq Y$ | Y.hasNotSubsetOrEqual(X); | O(n+m) n,m: tamaño de conjuntos |
| $X \cup Y$ | X.cppUnion(Y); | O(n+m) n,m: tamaño de conjuntos |
| $X \cap Y$ | X.cppIntersection(Y); | O(n+m) n,m: tamaño de conjuntos |
| $X \setminus Y$ | X.cppDifference(Y); | O(n+m) n,m: tamaño de conjuntos |
| $\mathbb{P}(X)$ | X.powerSet(); | O($2^n * n$) n: tamaño de conjunto |
| $\mathbb{P}_1(X)$ | X.powerSet1(); | O($2^n * n$) n,m: tamaño de conjunto |
| $card(X)$ | X.cardinality(); | O(1) |
| $X \times Y$ | X.cartesianProduct(Y); | O(n*m) n,m: tamaño de conjuntos |
| $partition(S, x, y)$ | S.partition(x,y); | O(n+m) n,m: tamaño de conjuntos |
| $min(X)$ | X.min(); | O(1) |
| $max(X)$ | X.max(); | O(1) |
| $x..y$ | numbersRange(x,y); | O(n) n: diferencia entre x-y |
| $domain(X)$ | X.domain(); | O(n) n: tamaño de conjunto |

| Tabla de complejidad algorítmica de traducción | | |
|---|---|---|
| Fórmula Event-B | Traducción C++ | Complejidad algorítmica |
| $range(X)$ | X.range(); | O(n) n: tamaño de conjunto |
| $r[S]$ | r.relationalImage(S); | O(n) n: tamaño de conjunto |
| $S \lhd r$ | r.domainRestriction(S); | O(n) n: tamaño de conjunto |
| $S \lhd\!\!\!- r$ | r.domainSubtraction(S); | O(n) n: tamaño de conjunto |
| $r \rhd S$ | r.rangeRestriction(S); | O(n) n: tamaño de conjunto |
| $r \rhd\!\!\!- S$ | r.rangeSubtraction(S); | O(n) n: tamaño de conjunto |
| $p \ ; \ q$ | p.forwardComposition(q); | O(n*m) n,m: tamaño de relaciones |
| $p \circ q$ | r.backwardComposition(q); | O(n*m) n,m: tamaño de relaciones |
| $r \lhd\!\!\!- s$ | r.relationalOverride(s); | O(n+m) n,m: tamaño de relaciones |
| $p \mid\mid q$ | p.parallelProduct(q); | O(n*m) n,m: tamaño de relaciones |
| $p \otimes q$ | p.directProduct(q); | O(n*m) n,m: tamaño de relaciones |
| $r^{-1}$ | p.inverse(); | O(n) n: tamaño de relación |
| $r(x)$ | r.functionImage(x); | O(n) n: tamaño de relación |

## 5.3. Análisis de eficiencia y escalabilidad

Basado en las observaciones hechas a lo largo de este capítulo, se pueden destacar los siguientes puntos acerca del traductor y las características de su implementación:

- La parte más compleja del traductor es la navegación por profundidad de los árboles sintácticos de cada fórmula matemática en el modelo. Debido a esto, mientras más complejas sean las fórmulas describiendo predicados y expresiones en el modelo, más tiempo consume la traducción.

- Refinar máquinas incrementa notablemente la cantidad de operaciones a realizar en la traducción. Esto se debe a que Rodin guarda una refinación como una copia del modelo abstracto agregando las modificaciones traídas por la refinación, en vez de solo guardar las nuevas características y hacer referencias a la máquina abstracta donde sea necesario. Por lo tanto, al traducir un modelo con una máquina abstracta y una máquina refinada, traduce toda la máquina abstracta y al moverse a la máquina refinada, traduce de nuevo la máquina abstracta más lo agregado por la refinación.

- La mayoría de algoritmos escalan linealmente, lo cual permite que el traductor funcione independientemente del tamaño del modelo. El único aspecto de la implementación que puede causar problemas en cuanto a escalabilidad es la enumeración usada para representar los carrier sets, la cual genera un número finito (5) de elementos a utilizar en el modelo.
- La decisión de traducir solo el modelo individual que el usuario solicite permite que la traducción se ejecute en el tiempo esperado sin importar cuantos proyectos el usuario tenga en su espacio de trabajo en Rodin.

Capítulo 6

# Resultados

En este capítulo se mostrará la aplicación en uso, presentando modelos de prueba que serán traducidos, y analizando los programas generados para identificar los éxitos y limitaciones de la herramienta desarrollada.

Antes de usar la herramienta con modelos, se probó exitosamente cada funcionalidad de Event-B para la cual creamos una traducción conforme se iba agregando al traductor. En otras palabras, todas las construcciones de Event-B que se pueden ver en la gramática de la Sección 3.2.4 se pudieron generar automáticamente y funcionaron al ser ejecutados.

Se traducirán los siguientes tres modelos creados en Rodin, los cuales provienen de libros de Event-B [1] [9] y cursos académicos:

1. SearchArray: un programa para encontrar el índice de un número específico en un arreglo.
2. Celebrity: un programa para resolver el "problema de la celebridad", que consiste en encontrar a una persona en un conjunto de personas que no conoce a ninguno otro y todos lo conocen.
3. TrafficLights: el sistema de control de un semáforo peatonal y de tránsito en una calle.

## 6.1. Traducción de búsqueda en arreglo

El primer modelo a traducir describe un algoritmo para encontrar el índice de un número dado dentro de un arreglo. Vale la pena destacar que se detendrá mucho más en la presentación y análisis

```
context SearchArrayCtx
constants
    n
    f
    s
axioms
@ax0 n∈ℕ1
@ax1 s ⊆ ℕ
@ax2 f∈1‥n⇸ s
end
```

Figura 6.1: Contexto de modelo SearchArray.

```
machine SearchArrayMch sees SearchArrayCtx

variables i x d k

invariants
@inv0 i∈dom(f)
@inv1 x∈ran(f)
@inv2 d∈{0,1}
@inv3 k∈0‥n−1
@inv4 x∉f[1‥k]


events
  event INITIALISATION then
    @act0 x ≔ max(ran(f) )
    @act1 d≔1
    @act2 i≔1
    @act3 k≔0
  end

  event aprog
  when
    @grd0 d≠0
    @grd1 f(k+1)=x
  then
    @act0 i ≔ k+1
    @act1 d ≔ 0
  end

  event progress
  when
    @grd0 d≠0 ∧ f(k+1)≠x
  then
    @act0 k≔k+1
  end

end
```

Figura 6.2: Máquina de modelo SearchArray.

de esta primera traducción ya que es el primer código generado que se está presentando en este documento. Tiene dos componentes: un contexto SearchArrayCtx y una máquina SearchArrayMch. El algoritmo consiste simplemente de un recorrido por cada posición del arreglo, deteniéndose cuando ubique el número deseado.

El contexto del modelo (ver Figura 6.1) tiene tres constantes y tres axiomas. El arreglo a explorar está representando por la función total f, que relaciona n posiciones del arreglo con n números, los cuales pertenecen a un conjunto de números naturales s.

La máquina del modelo (ver Figura 6.2) tiene cuatro variables y cuatro invariantes. El número a encontrar es x y el índice del arreglo donde se encuentra se guarda en i tras ser encontrado. La variable d es una bandera que indica si el algoritmo debe seguir buscando y la variable k se usa para iterar en cada posición del arreglo en búsqueda del número. Los eventos aprog y progress describen el funcionamiento del algoritmo: cuando el número en la posición k no es el número que se busca, el único evento que se puede ejecutar es progress, el cual avanza el iterador a la siguiente posición.

```
class SearchArrayCtx {
    protected:

        //// CARRIER SETS

        //// CONSTANTS

        // CONSTANT: s
        Set<int> s;

        // CONSTANT: f
        Relation<int,int> f;

        // CONSTANT: n
        int n;


    public:
        ////// CONTEXT CONSTRUCTOR METHOD
        SearchArrayCtx();

        //// GET FUNCTIONS
        // GET CONSTANT: s
        Set<int> get_s();

        // GET CONSTANT: f
        Relation<int,int> get_f();

        // GET CONSTANT: n
        int get_n();
```

Figura 6.3: Archivo de cabecera C++ de contexto SearchArrayCtx, ilustrando la declaración de la clase del contexto, las constantes y funciones de obtención para ellas.

Esto se repite hasta que encuentre el número objetivo x, en donde las guardas del evento progress no se cumplen y es posible ejecutar aprog, el cual guarda la posición indicada por k en la variable i y detiene el algoritmo cambiando d a 0.

A continuación se inspecciona la traducción generada por la herramienta, la cual se generó exitosamente. Antes de empezar, recordemos que por cada componente se genera un archivo de cabecera y un archivo de implementación. Habiendo dicho eso, en la Figura 6.3 se puede ver la generación del archivo de cabecera para el contexto del modelo (SearchArrayCtx), específicamente la declaración de la clase, constantes y de funciones para obtener sus valores. Al prestar atención al tipado de las constantes, se puede evidenciar como el traductor es capaz de componer en C++ el tipo de una constante/variable basado en como es usado en fórmulas. En este caso, se identifica que s es un conjunto de enteros y se traduce a Set<int> s; al igual que se identifica que f es una

```
////// CONTEXT CONSTRUCTOR METHOD
SearchArrayCtx::SearchArrayCtx() {
    // MODIFY THE INITIAL VALUE OF THE CONSTANT(S) IN THE LINE(S) BELOW
    s = Set<int>({ 1,2,7,9,14 });
    f = Relation<int,int>({ Tuple<int,int>(1,2), Tuple<int,int>(2,1), Tuple<int,int>(3,14), Tuple<int,int>(4,9) });
    n = 4;

    assert(checkAllAxioms_SearchArrayCtx());
}
```

Figura 6.4: Archivo de implementación C++ de contexto SearchArrayCtx, ilustrando la inicialización del valor de las constantes hecha por el usuario.

relación de enteros y se traduce a Relation<int,int> f. El resto de este archivo de cabecera declara funciones para inicializar constantes y verificar axiomas que luego son implementados en el archivo de implementación.

En la Figura 6.4, se ve el método constructor generado automáticamente para el contexto, donde se inicializan las constantes. Como se dijo previamente en el documento, esto debe hacerlo el usuario basado en su entendimiento del modelo para saber que constantes satisfacen los axiomas y siguiendo las instrucciones en la documentación de esta herramienta para usar la sintaxis correcta. Antes de la intervención del usuario, se genera una plantilla/ejemplo de valores que las constantes pueden tomar. En este contexto, s define los cinco números que pueden estar en el arreglo, n define que el arreglo tiene cuatro elementos y f define que el arreglo a recorrer es [ 2 , 1 , 14 , 9 ].

En la Figura 6.5 se puede observar la traducción de los axiomas en este contexto. En esta porción del código resalta el axioma ax2, donde se crea una instancia de relación tipo función total, se definen su rango como s y su dominio como un rango de números de 1 a n usando una función del EB2CppTools, y luego se convoca un método para consultar si la constante f pertenece a él: efectivamente preguntando si f es una función total. Tras definir e implementar los axiomas, también se genera una función que verifica la veracidad de todos ellos. En la Figura 6.4 se puede ver como esta función se usa para verificar si los valores de constantes definidos por el usuario cumplen con estos axiomas.

Luego, se tiene la traducción de la máquina SearchArrayMch. En el archivo de cabecera, las variables se definen con el mismo estilo que las constantes en el contexto (ver Figura 6.3). En las Figuras 6.6 y 6.7 se ven las declaraciones de las invariantes y eventos. Las invariantes se traducen de una forma similar a los axiomas vistos anteriormente. En la traducción de los eventos se puede ver como por cada evento, se generan tres métodos: (1) la verificación de sus guardas, (2) la ejecución de sus acciones que cambian el estado de la máquina y (3) el método para intentar ejecutar el evento que consulta si las guardas se cumplen, y de ser el caso ejecuta las acciones.

En la Figura 6.8 se puede ver la traducción de las invariantes en el archivo de implementación.

```
//// AXIOMS
// AXIOM: ax0
bool SearchArrayCtx::checkAxiom_ax0() {
    return ((NAT1_SET()).contains((n)));
}
// AXIOM: ax2
bool SearchArrayCtx::checkAxiom_ax2() {
    return ((RelationType<Set<int>,Set<int>>("TotalFunction", (numbersRange((1),(n))), (s))).contains((f)));
}
// AXIOM: ax1
bool SearchArrayCtx::checkAxiom_ax1() {
    return ((NAT_SET()).hasSubsetOrEqual((s)));
}

// BOOL FUNCTION TO CHECK ALL AXIOMS
bool SearchArrayCtx::checkAllAxioms_SearchArrayCtx() {
    bool ax0 = checkAxiom_ax0();
    bool ax2 = checkAxiom_ax2();
    bool ax1 = checkAxiom_ax1();

    return ax0 && ax2 && ax1;
}
```

Figura 6.5: Archivo de implementación C++ de contexto SearchArrayCtx, ilustrando la traducción de los predicados de axiomas.

```
//// INVARIANTS
// INVARIANT: inv2
bool checkInvariant_inv2();

// INVARIANT: inv1
bool checkInvariant_inv1();

// INVARIANT: inv4
bool checkInvariant_inv4();

// INVARIANT: inv3
bool checkInvariant_inv3();

// INVARIANT: inv0
bool checkInvariant_inv0();


// BOOL FUNCTION TO CHECK ALL INVARIANTS
bool checkAllInvariants_SearchArrayMch();
```

Figura 6.6: Archivo de implementación C++ de máquina SearchArrayMch, ilustrando la declaración de invariantes.

```
////// EVENTS

//// EVENT: aprog

// Event Guards Function
bool aprog_checkGuards();

// Event Actions Function
void aprog_actions();

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void aprog();

//// EVENT: progress

// Event Guards Function
bool progress_checkGuards();

// Event Actions Function
void progress_actions();

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void progress();
```

Figura 6.7: Archivo de implementación C++ de máquina SearchArrayMch, ilustrando la declaración de un evento, donde se separa la verificación de la guardas con las acciones, y el evento en sí que emplea ambas.

En general se evidencia la habilidad del traductor de anidar distintas expresiones para crear predicados complejos, y también generar automáticamente paréntesis para evitar ambigüedades (algo importante en C++). La herramienta también genera una función checkAllInvariants que verifica todas estas invariantes con un solo comando.

Finalmente en cuanto a la traducción, observemos la generación de un evento. En la Figura 6.9 se muestra uno de estos eventos: progress, donde se evidencia la implementación de las declaraciones ilustradas en la Figura 6.7. La verificación de guardas en checkGuards y las acciones en actions traducen fórmulas de manera similar a las invariantes, y se puede ver como en el método progress se usa la función checkGuards en un condicional para ver si se ejecutan las acciones del evento. En caso de no cumplirse las guardas, el programa imprime un mensaje notificando al usuario que se intentó ejecutar un evento que no está disponible. También, si se ejecutan las acciones, el programa verifica la veracidad de las invariantes en el nuevo estado del sistema, y de no cumplirse, alza una excepción y aborta la ejecución.

Ahora se presentarán los resultados de ejecutar esta traducción, observando si logra su objetivo de encontrar la posición de un número en un arreglo. En la Figura 6.10 se puede ver un archivo main que se elaboró para probar la traducción. El modelo de SearchArray básicamente consiste en ejecutar el evento progress hasta poder ejecutar el evento aprog, el cual guarda la posición buscada

```
//// INVARIANTS
// INVARIANT: inv2
bool SearchArrayMch::checkInvariant_inv2() {
    return ((Set<int>({{0),(1)})).contains((d)));
}

// INVARIANT: inv1
bool SearchArrayMch::checkInvariant_inv1() {
    return (((f).range()).contains((x)));
}

// INVARIANT: inv4
bool SearchArrayMch::checkInvariant_inv4() {
    return (((f).relationalImage((numbersRange((1),(k))))).notContains((x)));
}

// INVARIANT: inv3
bool SearchArrayMch::checkInvariant_inv3() {
    return ((numbersRange((0),((n) - (1)))).contains((k)));
}

// INVARIANT: inv0
bool SearchArrayMch::checkInvariant_inv0() {
    return (((f).domain()).contains((i)));
}
```

Figura 6.8: Archivo de implementación C++ de máquina SearchArrayMch, ilustrando la traducción de los predicados de invariantes.

```
//// EVENT: progress

// Event Guards Function
bool SearchArrayMch::progress_checkGuards() {
    bool grd0 = (((d) != (0)) && (((f).functionImage(((k) + (1)))) != (x)));

    return grd0;
}

// Event Actions Function
void SearchArrayMch::progress_actions() {
    // Action: act0
    (k) = ((k) + (1));

}

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void SearchArrayMch::progress() {
    if (progress_checkGuards()) {
        progress_actions();
        assert( checkAllInvariants_SearchArrayMch() );
    }
    else { cout << "Can't execute progress event. Guards are false. " << endl; }
}
```

Figura 6.9: Archivo de implementación C++ de máquina SearchArrayMch, ilustrando la traducción del evento progress en métodos separados que verifican las guardas y ejecutan las acciones.

```cpp
#include <iostream>
#include "EB2CppTools.h"

#include "SearchArrayCtx.h"
#include "SearchArrayMch.h"

using namespace std;

int main() {
    SearchArrayMch mch;

    int done = mch.get_d();
    int MAX_TRIES = mch.get_n() + 5;

    while (done != 0 && MAX_TRIES > 0) {
        mch.showAllVariables_SearchArrayMch();
        cout << endl;
        mch.aprog();
        mch.progress();
        done = mch.get_d();
        MAX_TRIES -= 1;
    }

    int i = mch.get_i();

    cout << "Index with target number found! | i: " << i << endl;

    return 0;
```

Figura 6.10: Archivo main para probar la traducción de SearchArray.

en la variable i y cambia la variable d a 0, haciendo imposible la ejecución de algún otro evento. Para ahorrar líneas en este archivo main, se repetirá la acción de mostrar el valor actual de todas las variables, el intento de hacer aprog y el intento de hacer progress dentro de un ciclo que se repetirá hasta que la variable d sea cero (lo cual significa que ningún evento puede ejecutarse). También, por motivos de prueba, agregaremos una variable MAX_TRIES para establecer un número máximo de iteraciones que este ciclo puede hacer, en caso de que d nunca se vuelva cero y el ciclo se volviese infinito. Tras terminar el ciclo, el programa imprimirá en un mensaje la posición donde se encuentra el número buscado.

Es importante aclarar que hay otras formas de poner en funcionamiento la traducción en este main: por ejemplo, se podría definir que el ciclo se repita mientras las guardas de aprog *o* progress se cumplan, en vez de consultar el valor de d. La forma de usar las traducciones depende de la aplicación que quiera darle el usuario a la traducción y de sus preferencias. Lo mismo podrá decirse de los archivos main en los otros modelos a traducir en este capítulo.

Tras compilar el main y todos los archivos de implementación generados por el traductor, se ejecuta el programa y se ven los resultados, ilustrados en la Figura 6.11. Se puede ver que el programa funcionó correctamente de acuerdo al modelo, basado en la última línea que dice que el número

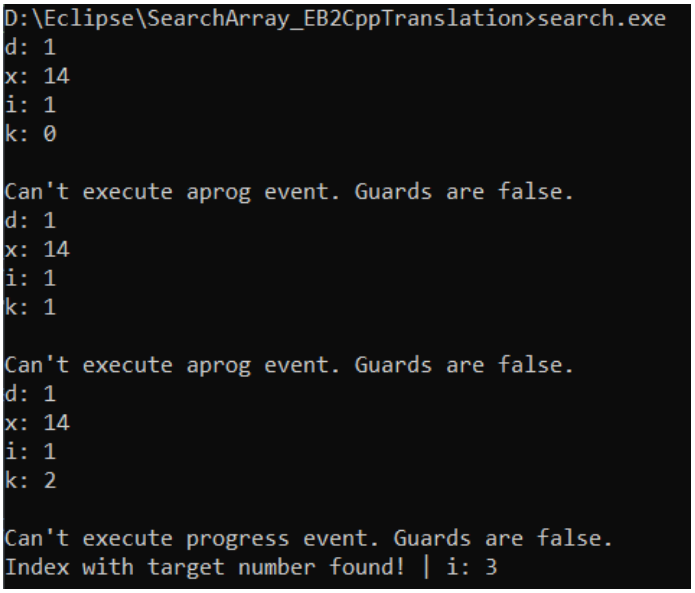


Figura 6.11: Resultado de ejecutar traducción SearchArray.

buscado (14) se encuentra en la posición 3 del arreglo (recordemos que el arreglo se inicializó en la Figura 6.4 como [ 2 , 1 , 14 , 9 ]). Recordemos que en el modelo la relación representando el arreglo empieza a marcar posiciones en el número 1, y no en el número 0 que típicamente es el índice inicial de un arreglo. También se evidencia como al intentar llamar al evento aprog al iterar en la primera y segunda posición del arreglo, las guardas del evento no se cumplían, porque ciertamente el número 14 no estaba en esas posiciones, y que más bien se ejecutó progress lo cual cambió el valor del iterador k. En la última iteración se ve como es el evento progress el que ahora no se pudo ejecutar, señalando que aprog se había ejecutado. La compilación tardó unos seis segundos y su ejecución fue instantánea. La ejecución tampoco generó problemas al ampliar el arreglo para contener veinte o cincuenta números.

## 6.2. Traducción de problema de celebridad

El siguiente modelo es un programa diseñado para resolver el *problema de la celebridad*. El problema consiste de lo siguiente: en un conjunto de personas, hay una celebridad, y una relación entre personas sobre quién conoce a quién. Esta relación está definida de tal forma que:

- Nadie se conoce a sí mismo.
- La celebridad no conoce a nadie.
- Todos los demás conocen a la celebridad.

```
context Celebrity_c0

constants k c P

axioms
  @axm1 P ⊆ N
  @axm2 c ∈ P
  @axm3 k ∈ (P ∖ {c}) ↔ P
  @axm4 k~ [{c}] = P ∖ {c}
end
```

Figura 6.12: Contexto Celebrity_c0 de modelo Celebrity.

```
context Celebrity_c1 extends Celebrity_c0

constants n

axioms
  @axm1 n ∈ N
  @axm2 n > 0
  @axm3 P = 0‥n
end
```

Figura 6.13: Contexto Celebrity_c1 de modelo Celebrity.

El objetivo del problema es encontrar quién es la celebridad en el conjunto. El modelo describe un algoritmo para encontrar una solución. Esta traducción nos permitirá ver como el traductor maneja varias refinaciones y extensiones en el caso de un modelo que ha sido refinado hasta el punto de ser un programa.

El contexto visto en la Figura 6.12 describe:

- La constante P que es el conjunto de personas en donde buscar a la celebridad representados como números naturales.
- La constante c que es la celebridad dentro de esas personas.
- La constante k que son las relaciones de 'conoce a' entre las personas de P. Son de notar como en los axiomas 3 y 4 se define como la celebridad no puede estar en el dominio de la relación (pues él no conoce a nadie) y como al buscar en la inversa de la relación la celebridad debe estar relacionado a todos los demás (pues todos los demás lo conocen a él).

El contexto visto en la Figura 6.13 extiende el contexto Celebrity_c0, solo agregando una constante n que define el número de personas en P.

En la máquina abstracta Celebrity_0 (ver Figura 6.14) se define el evento celebrity que guarda el resultado del algoritmo (la persona que es la celebridad) en la variable r y señala el fin del proceso.

```
machine Celebrity_0 sees Celebrity_c0

variables r

invariants
  @inv1 r ∈ P

events
  event INITIALISATION
    then
      @act1 r ≔ min(P)
  end

  event celebrity
    then
      @act1 r ≔ c
  end
end
```

Figura 6.14: Máquina abstracta Celebrity_0 de modelo Celebrity que solo define el evento para guardar el resultado del algoritmo.

Por el momento, no se especifica en qué estado del sistema se ejecuta el evento, y claramente no se está evidenciando ningún algoritmo funcionando porque guarda quién es la celebridad solo consultando directamente el valor de quién es la celebridad. La idea es refinar este evento conforme se construya el algoritmo en futuras máquinas.

También se debe señalar que la variable r se inicializa con el mínimo de P, reemplazando la asignación por pertenencia que el modelo tenía originalmente. Esto es un ejemplo del alcance delimitado del proyecto, que no podrá traducir asignaciones no deterministas como la asignación por pertenencia. Aunque vale la pena resaltar, en el caso de este modelo, que incluso si se dejara con la asignación por pertenencia, este componente no es el programa que el usuario va a querer ejecutar, pues no hace nada. El usuario va a querer ejecutar como programa la máquina resultante al final de todas las refinaciones y llegados a ese punto idealmente toda indeterminación se ha eliminado (como es el caso de este modelo).

En la máquina Celebrity_1 (ver Figura 6.15) se puede ver la versión más básica del algoritmo. Se tiene un conjunto Q que encapsula a todas las personas del problema. A partir de esto, los eventos remove_1 y remove_2 sustraen a personas que conocen a alguien y aquellos no conocidos por nadie respectivamente. El evento de finalización celebrity es refinado para especificar que se llama cuando solo queda *una* persona en el conjunto, que será alguien que no conoce a nadie y que era conocido por alguien: la celebridad.

La máquina Celebrity_2 en la Figura 6.16 refina la máquina Celebrity_1, cambiando el diseño del algoritmo. En vez de tener solo un conjunto Q del cual se van eliminando personas que no son la celebridad, se tiene un candidato a celebridad b y el resto de personas R. Los eventos refinados remove_1 y remove_2 ahora sustraen de R aquellos que conocen al candidato y cambian el candidato

```
machine Celebrity_1 refines Celebrity_0  sees Celebrity_c0

variables r Q

invariants
  @inv11 Q ⊆ P
  @inv12 c ∈ Q

events
  event INITIALISATION
    then
      @act11 r ≔ min(P)
      @act21 Q ≔ P
  end

  event celebrity refines celebrity
    any x
    where
      @grd11 x ∈ Q
      @grd21 Q = {x}
    then
      @act11 r ≔ x
  end

  event remove_1
    any x y
    where
      @grd1 x ∈ Q
      @grd2 y ∈ Q
      @grd3 x ↦ y ∈ k
    then
      @act11 Q ≔ Q ∖ {x}
  end

  event remove_2
    any x y
    where
      @grd1 x ∈ Q
      @grd2 y ∈ Q
      @grd3 x ↦ y ∉ k
      @grd4 x ≠ y
    then
      @act11 Q ≔ Q ∖ {y}
  end
```

Figura 6.15: Máquina Celebrity_1 de modelo Celebrity.

```
machine Celebrity_2 refines Celebrity_1  sees Celebrity_c0

variables r R b

invariants
  @inv21 R ⊆ P
  @inv22 b ∈ P
  @inv23 b ∉ R
  @inv24 Q = R ∪ {b}

events
  event INITIALISATION
    then
      @act1 r ≔ min(P)
      @act2 b ≔ min(P)
      @act3 R ≔ P ∖ { min(P) }
  end

  event celebrity refines celebrity
    where
      @grd1 R = ∅
    with
        @x x = b
    then
      @act1 r ≔ b
  end

  event remove_1 refines remove_1
    any x
    where
      @grd1 x ∈ R
      @grd2 x ↦ b ∈ k
    with
        @y y = b
    then
      @act1 R ≔ R ∖ {x}
  end

  event remove_2 refines remove_2
    any x
    where
      @grd1 x ∈ R
      @grd2 x ↦ b ∉ k
    with
        @y y = b
    then
      @act2 b ≔ x
      @act1 R ≔ R ∖ {x}
```

Figura 6.16: Máquina Celebrity_2 de modelo Celebrity.

si hay alguien que no lo conoce. Este proceso continúa hasta que solo quede el candidato en el conjunto de personas. Este candidato será la celebridad.

El último componente es la máquina Celebrity_3 (ver Figura 6.17) que refina Celebrity_2, y esta es la máquina que ejecutaremos en las pruebas. A diferencia de las máquinas anteriores, Celebrity_3 lee el contexto Celebrity_c1. También refina una última vez las operaciones del algoritmo, agregando un iterador a que empieza en la primera persona del conjunto (se mantiene el candidato a celebridad b). Los eventos refinados funcionan así:

- Si esta persona a conoce al candidato, se avanza a la siguiente persona del conjunto.
- Si esta persona a no conoce al candidato, el valor actual de a se vuelve el candidato, y avanza a la siguiente persona del conjunto.
- Cuando el iterador a haya recorrido todo el conjunto, se fija como resultado del algoritmo el candidato b.

Tras haber explicado el modelo, se solicita a la herramienta que genere la traducción C++. No se hablará en detalle acerca de los aspectos de la traducción que ya se mostraron en el modelo de búsqueda en arreglo (como se generan los tipados para las constantes, la declaración de

```
machine Celebrity_3 refines Celebrity_2  sees Celebrity_c1

variables r a b

invariants
  @inv31 a ∈ 1‥n+1
  @inv32 R = a‥n

events
  event INITIALISATION
    then
      @act1 r ≔ 0
      @act2 a ≔ 1
      @act3 b ≔ 0
  end

  event remove_1 refines remove_1
    where
      @grd11 a ≤ n
      @grd12 a ↦ b ∈ k
    with
      @x x = a
    then
      @act11 a ≔ a + 1
  end

  event remove_2 refines remove_2
    where
      @grd1 a ≤ n
      @grd2 a ↦ b ∉ k
    with
      @x x = a
    then
      @act1 b ≔ a
      @act2 a ≔ a + 1
  end

  event celebrity refines celebrity
    where
      @grd1 a = n + 1
    then
      @act1 r ≔ b
  end
end
```

Figura 6.17: Máquina Celebrity_3 de modelo Celebrity.

```
class Celebrity_c1: virtual public Celebrity_c0 {
    protected:
```

Figura 6.18: Archivo de cabecera C++ de contexto Celebrity_c1 ilustrando cómo hereda del contexto al que extiende.

```
//// EVENT: INITIALISATION

// Event Guards Function
bool Celebrity_3::INITIALISATION_checkGuards() {

    return true;
}

// Event Actions Function
void Celebrity_3::INITIALISATION_actions() {
    // Action: act31
    (r) = (0);

    // Action: act32
    (a) = (1);

    // Action: act33
    (b) = (0);

}
```

Figura 6.19: Archivo de implementación C++ de máquina Celebrity_3 inicializa solamente las variables definidas en esta máquina (ver Figura 6.17) y ninguna de máquinas abstractas pasadas.

invariantes, etc), y nos enfocaremos en lo novedoso que este modelo tiene en comparación al anterior.

La traducción se completó exitosamente y en menos de un segundo. Las distintas fórmulas matemáticas se tradujeron correctamente. Lo primero novedoso a señalar es cómo se traduce la relación de extensión entre contextos, ilustrado en la Figura 6.18. Gracias a esta relación, cuando se crea una instancia de este contexto Celebrity_c1, también crea las constantes/axiomas del contexto al que extiende.

Cuando se inspecciona la traducción de la máquina refinada final, se observa un resultado inesperado. Los eventos traducidos de Celebrity_3 no incorporan las acciones y guardas de los eventos previos en Celebrity_1 y Celebrity_2 (ver Figura 6.19), lo cual de por sí no es un problema. Pero a la vez, el traductor genera la declaración de las variables e invariantes de esas máquinas previas (ver Figura 6.20). Se puede identificar que esto se debe a que los eventos en cada máquina después de la primera solo 'refinan' más no 'extienden' su evento abstracto (véase la palabra clave 'refines' en

```
class Celebrity_3: virtual public C
    protected:
        //// VARIABLES

        // VARIABLE: Q
        Set<int> Q;


        // VARIABLE: a
        int a;


        // VARIABLE: R
        Set<int> R;


        // VARIABLE: b
        int b;


        // VARIABLE: r
        int r;
```

Figura 6.20: Archivo de cabecera C++ de máquina Celebrity_3 ilustrando todas las variables generadas en esta máquina, incluyendo las de máquinas abstractas que son refinadas.

los eventos de remove en la Figura 6.17). Para efectos de la traducción, estos eventos reemplazan a los que le preceden. El problema es que como sí se generan las invariantes de esas máquinas abstractas, ocurrirá un error cuando intente validar esos predicados porque no se están inicializando ni modificando esas variables abstractas en los eventos.

Esto no es un error por parte del modelo, pues es lo que debes hacer cuando estás refinando un modelo hasta el punto de ser un programa (reemplazas variables más abstractas con nuevas más concretas). Sin embargo, el traductor no es capaz de identificar cuáles variables e invariantes pertenecen a las máquinas abstractas para poder así no generarlas y de igual forma no puede incorporar las acciones y guardas de los eventos abstractos sin cambiar el funcionamiento del algoritmo.

Como consecuencia de esta observación, se modificará el traductor para incluir en la clase generada un atributo que determine si se debe validar o no las invariantes durante la ejecución del código C++. Este atributo tendrá el modo 'no verificar invariantes' por defecto y el usuario podrá activar la verificación si no está haciendo la refinación de eventos hasta ser programas donde se reemplazan variables de las máquinas abstractas. Se cree que esto no es un gran problema para

```
int main() {
    Celebrity_3 mch;
    int tries = 0;
    while ((mch.remove_1_checkGuards() or
        mch.remove_2_checkGuards()) and
        tries < 15) {
            mch.showAllVariables_Celebrity_3();
            cout << "Running remove_1 event..." << endl;
            mch.remove_1();
            mch.showAllVariables_Celebrity_3();
            cout << "Running remove_2 event..." << endl;
            mch.remove_2();
            tries += 1;
    }
    mch.showAllVariables_Celebrity_3();
    cout << "Running celebrity event..." << endl;
    mch.celebrity();
    cout << "Result is: " << mch.get_r() << endl;
```

Figura 6.21: Archivo main para probar la traducción de Celebrity.

el buen funcionamiento de la traducción, ya que como se dijo en el Capítulo 2, se asume que el usuario ha hecho las pruebas del modelo en Rodin antes de solicitar su traducción. Si se demostró formalmente que el modelo es correcto, se puede contar con que la traducción va a seguir el comportamiento del modelo y no incumplirá las invariantes (eso se busca mostrar en estas pruebas).

Con tal de poner en ejecución esta traducción, deshabilitaremos la verificación de invariantes. Por lo tanto, el enfoque estará en si la traducción soluciona correctamente el problema de la celebridad. El programa generado debe identificar la celebridad 3 en el conjunto de personas [ 0 , 1 , 2 , 3 , 4 ] donde:

- 0 conoce a 3.
- 1 conoce a 3.
- 2 conoce a 1.
- 2 conoce a 3.
- 4 conoce a 1.
- 4 conoce a 3.

En la Figura 6.21 se puede ver el archivo que ejecutará la traducción. Intentaremos ejecutar los eventos remove_1 y remove_2 hasta que ninguno de los dos pueda ejecutarse más (o excedamos 15 iteraciones de este ciclo usando una variable tries, para evitar un ciclo infinito en caso de un error). Tras esto, ejecutaremos el evento celebrity que guarda el resultado en la variable r. El archivo

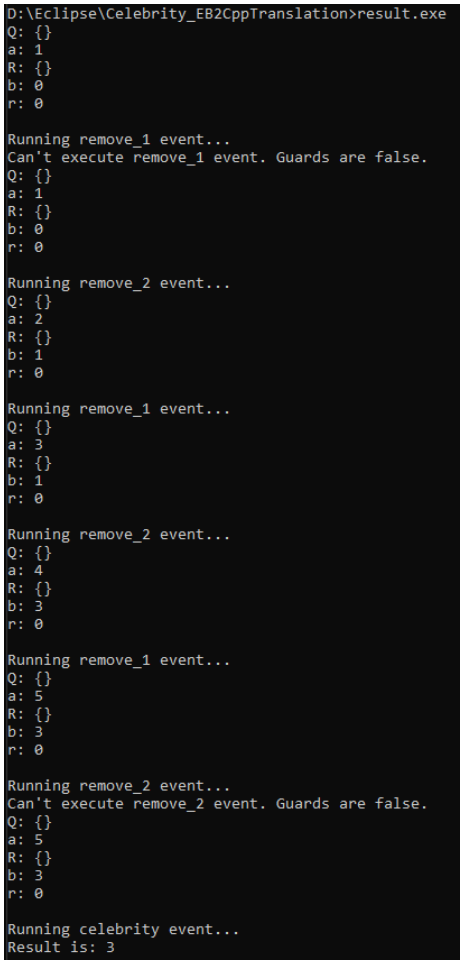


Figura 6.22: Resultado de ejecutar traducción de Celebrity, donde se ve la celebridad 3 correctamente almacenada en la variable r tras finalizar los eventos.

también incluye algunas impresiones a consola que no alteran el funcionamiento de la traducción y existen solo para hacer los resultados más legibles.

Los resultados de ejecutar la traducción se pueden ver en la Figura 6.22 y se evidencia que el algoritmo funcionó correctamente, ya que la persona 3 se identificó exitosamente como la celebridad y se guardó en la variable r en el último paso. Si observamos el resto de los resultados, se puede ver el flujo del algoritmo:

1. El candidato a celebridad es la persona 0 y la primera persona a inspeccionar (variable a) es 1. Como 1 no conoce a 0, no se puede ejecutar el evento remove_1 y recibimos un mensaje notificando esto. Esto significa que remove_2 se puede ejecutar: 1 se vuelve el candidato a celebridad y avanzamos a la persona 2.

```
context TrafficLightCtx
sets COLOURS
constants
    red
    yellow
    green
axioms
@ax0 partition(COLOURS,{red},{yellow},{green})
end
```

Figura 6.23: Contexto de modelo TrafficLights.

2. El candidato a celebridad es la persona 1. Como 2 conoce a 1, se puede ejecutar el evento remove_1: la persona 1 se mantiene como el candidato y avanzamos a la persona 3.

3. El candidato a celebridad es la persona 1. Como 3 no conoce a 1, el evento remove_2 se puede ejecutar: 3 se vuelve el candidato a celebridad y avanzamos a la persona 4.

4. El candidato a celebridad es la persona 3. Como 4 conoce a 3, se ejecuta el evento remove_1: la persona 3 sigue siendo el candidato y avanzamos a la persona 5. Se intenta ejecutar el evento remove_2 pero como la variable a se ha salido del conjunto de personas del problema (0 a 4), no se puede ejecutar y recibimos un mensaje notificando esto.

La traducción también sirve si cambiamos el número de personas en el problema, las relaciones entre ellas y la identidad de la celebridad.

## 6.3. Traducción de sistema de semáforo

El tercer modelo a traducir describe el sistema controlador de un semáforo. A diferencia de los primeros dos modelos, este sistema no ha sido refinado hasta el punto de ser un programa, sino que continúa siendo un modelo más abstracto. Como se mencionó anteriormente en este documento, este traductor está diseñado para también ser capaz de generar código C++ para esta clase de modelo y eso se busca evidenciar con esta prueba.

El modelo consiste de tres componentes: un contexto TrafficLightCtx, una máquina TrafficLightMch y su refinación TrafficLightMchRef. Visto de una manera simple, el modelo controla cuando los carros/peatones tienen permiso para pasar, y sincroniza esa configuración con los colores del semáforo vehicular y peatonal (delimitando mediante invariantes que el semáforo peatonal no puede estar en verde si el semáforo de los carros está en verde, etc).

El contexto TrafficLightCtx (ver Figura 6.23) define un carrier set COLOURS para representar el conjunto de colores que los semáforos pueden adoptar, y establece tres constantes para que sean

```
machine TrafficLightMch
variables cars_go peds_go

invariants
    @inv0 cars_go ∈ BOOL
    @inv1 peds_go ∈ BOOL
    @inv2 ¬(cars_go = TRUE ∧ peds_go = TRUE)

events
  event INITIALISATION then
    @act0 cars_go ≔ FALSE
    @act1 peds_go ≔ FALSE
  end

  event set_peds_go
  when
    @grd0 cars_go = FALSE
  then
    @act0 peds_go ≔ TRUE
  end

  event set_peds_stop
  then
    @act0 peds_go ≔ FALSE
  end

  event set_cars
  any
    new_value
  when
    @grd0 new_value ∈ BOOL
    @grd1 new_value = TRUE ⇒ peds_go = FALSE
  then
    @act0 cars_go ≔ new_value
  end
end
```

Figura 6.24: Máquina TrafficLightMch de modelo TrafficLights.

```
machine TrafficLightMchRef refines TrafficLightMch sees TrafficLightCtx

variables cars_go peds_go peds_colour cars_colours

invariants
  @inv10 peds_colour ∈ {red,green}
  @gluing peds_go = TRUE ⇒ peds_colour = green
  @inv11 cars_colours ⊆ COLOURS
  @gluing_cars cars_go = TRUE ⇒ green ∈ cars_colours

events
  event INITIALISATION extends INITIALISATION then
    @act10 peds_colour ≔ red
    @act11 cars_colours ≔ {red}
  end

  event set_peds_go extends set_peds_go
  when
    @grd10 green ∉ cars_colours
  then
    @act10 peds_colour ≔ green
  end

  event set_peds_stop extends set_peds_stop
  then
    @act10 peds_colour ≔ red
  end

  event set_cars extends set_cars
  any new_value new_value_colours
    where
      @grd10 new_value_colours ⊆ COLOURS
      @grd11 green ∈ new_value_colours ⇒ peds_colour = red
      @grd12 cars_colours = {yellow} ⇒ new_value_colours = {red}
      @grd13 cars_colours = {red} ⇒ new_value_colours = {red,yellow}
      @grd14 cars_colours = {red,yellow} ⇒ new_value_colours = {green}
      @grd15 cars_colours = {green} ⇒ new_value_colours = {yellow}
      @grd16 new_value = TRUE ⇔ new_value_colours = {green}
    then
      @act10 cars_colours ≔ new_value_colours
  end
end
```

Figura 6.25: Máquina TrafficLightMchRef de modelo TrafficLights.

los elementos de ese conjunto (red, yellow, green; que son rojo, amarillo y verde).

La máquina TrafficLightMch (ver Figura 6.24) introduce dos variables booleanas cars_go y peds_go que indican si se les da paso a los carros y peatones respectivamente. Estas variables son acompañadas de una invariante declarando que ambas variables no pueden ser verdaderas al mismo tiempo (los peatones y carros no pueden tener el paso a la vez). También contiene tres eventos:

- set_peds_go: si los carros no tienen el paso, le da paso a los peatones.
- set_peds_stop: le niega el paso a los peatones.
- set_cars: le niega o da paso a los carros. Si quiere darle paso a los carros, los peatones no deben tener paso.

```
//// CARRIER SETS ENUMERATIONS
//// CARRIER SET: COLOURS
enum COLOURS_CS {
    // Constants that belong to this carrier set:
    red,
    yellow,
    green
    // End of constants
};


class TrafficLightCtx {
    protected:

        //// CARRIER SETS
        // SET THAT CONTAINS ALL ELEMENTS OF COLOURS
        Set<COLOURS_CS> COLOURS = Set<COLOURS_CS>({yellow, red, green});
```

Figura 6.26: Parte de traducción contexto TrafficLightCtx donde se ilustra la generación del carrier set COLOURS en C++.

La máquina TrafficLightsMchRef (ver Figura 6.25) refina la máquina TrafficLightsMch. En este componente se agregan las variables que representan la luz actual del semáforo vehicular y del semáforo peatonal (peds_colour y cars_colours). Dos de las nuevas invariantes conectan estas variables con las anteriores, asegurando en el sistema que cuando los peatones tengan el paso el semáforo peatonal debe estar en luz verde, y vice versa. La misma lógica aplica para el semáforo vehicular y el paso de los carros. Cabe destacar que la variable para las luces del semáforo vehicular es un conjunto de colores, lo cual se hace para contemplar el momento cuando hay luces rojas y amarillas iluminadas en el semáforo antes de cambiar solo a verde. El evento refinado set_cars también establece un orden en el que las luces del semáforo vehicular debe cambiar: rojo ->rojo-amarillo ->verde ->amarillo ->rojo. No se le puede pedir al semáforo que cambie a una luz que no sea la siguiente en este orden.

Los eventos de esta máquina refinada *extienden* los eventos de la máquina anterior (nótese en la Figura 6.25 el uso de la palabra extends en los eventos). Esto contrasta con las refinaciones vistas en el modelo de la celebridad que no extendían los eventos abstractos.

Habiendo explicado el modelo, se solicita la traducción a la herramienta y se completa exitosamente en el mismo tiempo de ejecución visto previamente. Lo primero novedoso a señalar de esta traducción es como se genera en C++ el carrier set COLOURS. Como se ve en la Figura 6.26, el traductor identifica el carrier set y como está compuesto de las tres constantes red, yellow y green (esta partición se ve en el axioma de la Figura 6.23). Con esta información el traductor es capaz de generar una enumeración que incluye estos elementos para usar como un tipo de dato y también un atributo en la clase del contexto que se puede usar en operaciones de conjuntos.

```
//// EVENT: set_cars

// Event Guards Function
bool set_cars_checkGuards(bool new_value);

// Event Actions Function
void set_cars_actions(bool new_value);

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void set_cars(bool new_value);
```

Figura 6.27: Parte de traducción máquina TrafficLightMch que ilustra la declaración del evento set_cars incorporando el parámetro new_value.

```
//// EVENT: set_cars

// Event Guards Function
bool TrafficLightMch::set_cars_checkGuards(bool new_value) {
    bool grd0 = ((BOOL_SET()).contains((new_value)));
    bool grd1 = (!((new_value) == (true)) || ((peds_go) == (false)));

    return grd0 && grd1;
}

// Event Actions Function
void TrafficLightMch::set_cars_actions(bool new_value) {
    // Action: act0
    (cars_go) = (new_value);

}

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void TrafficLightMch::set_cars(bool new_value) {
    if (set_cars_checkGuards(new_value)) {
        set_cars_actions(new_value);
        assert( checkAllInvariants_TrafficLightMch() );
    }
    else { cout << "Can't execute set_cars event. Guards are false. " << endl; }
}
```

Figura 6.28: Parte de traducción máquina TrafficLightMch que ilustra la implementación del evento set_cars incorporando el parámetro new_value.

```
//// EVENT: INITIALISATION

// Event Guards Function
bool TrafficLightMchRef::INITIALISATION_checkGuards() {

    return true;
}

// Event Actions Function
void TrafficLightMchRef::INITIALISATION_actions() {
    // Action: act0
    (cars_go) = (false);

    // Action: act1
    (peds_go) = (false);

    // Action: act10
    (peds_colour) = (red);

    // Action: act11
    (cars_colours) = (Set<COLOURS_CS>({(red)}));

}

// Event Function. CALL THIS EVENT TO CHECK GUARDS AND IF TRUE DO ACTIONS, AND CHECK FOR INVARIANTS
void TrafficLightMchRef::INITIALISATION() {
    if (INITIALISATION_checkGuards()) {
        INITIALISATION_actions();
        assert( checkAllInvariants_TrafficLightMchRef() );
    }
    else { cout << "Can't execute INITIALISATION event. Guards are false. " << endl; }
}
```

Figura 6.29: Parte de traducción máquina TrafficLightMchRef que ilustra la implementación del evento de inicialización, donde se integran también las inicializaciones de la máquina que refina (TrafficLightMch).

Al inspeccionar la traducción de la máquina TrafficLightMch vemos otra funcionalidad del traductor no vista antes en efecto, que es como incorpora parámetros en los eventos. El evento set_cars debe recibir como parámetro el paso aprobado o negado a los carros y en el evento refinado también recibe los nuevos colores del semáforo. En las Figuras 6.27 y 6.28 vemos como se genera new_value como un argumento de los métodos del evento. Cuando se quiera convocar este evento, se deberá pasar un valor booleano.

La máquina TrafficLightMchRef nos permite ver la generación de código para una máquina con eventos que extienden eventos en máquinas previas. Si vemos la traducción del evento de inicialización (ver Figura 6.29), se puede ver como contiene las acciones de tanto la inicialización de TrafficLightMch como la de TrafficLightMchRef (ver eventos de inicialización en Figuras 6.24 y 6.25

```
//// EVENT: set_cars

// Event Guards Function
bool TrafficLightMchRef::set_cars_checkGuards(bool new_value, Set<COLOURS_CS> new_value_colours) {
    bool grd0 = ((BOOL_SET()).contains((new_value)));
    bool grd1 = (!((new_value) == (true)) || ((peds_go) == (false)));
    bool grd10 = ((COLOURS).hasSubsetOrEqual((new_value_colours)));
    bool grd11 = (!((new_value_colours).contains((green))) || ((peds_colour) == (red)));
    bool grd12 = (!((cars_colours) == (Set<COLOURS_CS>({{yellow}}))) || ((new_value_colours) == (Set<COLOURS_CS>({{red}}))));
    bool grd13 = (!((cars_colours) == (Set<COLOURS_CS>({{red}}))) || ((new_value_colours) == (Set<COLOURS_CS>({{red},{yellow}}))));
    bool grd14 = (!((cars_colours) == (Set<COLOURS_CS>({{red},{yellow}}))) || ((new_value_colours) == (Set<COLOURS_CS>({{green}}))));
    bool grd15 = (!((cars_colours) == (Set<COLOURS_CS>({{green}}))) || ((new_value_colours) == (Set<COLOURS_CS>({{yellow}}))));
    bool grd16 = (((new_value) == (true)) == ((new_value_colours) == (Set<COLOURS_CS>({{green}}))));

    return grd0 && grd1 && grd10 && grd11 && grd12 && grd13 && grd14 && grd15 && grd16;
}

// Event Actions Function
void TrafficLightMchRef::set_cars_actions(bool new_value, Set<COLOURS_CS> new_value_colours) {
    // Action: act0
    (cars_go) = (new_value);

    // Action: act10
    (cars_colours) = (new_value_colours);

}
```

Figura 6.30: Parte de traducción máquina TrafficLightMchRef que ilustra la implementación del evento set_cars, donde se integran las guardas, acción y parámetro de la máquina que refina (TrafficLightMch).

respectivamente). Esto contrasta con la máquina refinada del modelo Celebrity, donde los eventos no extendían sino que solo refinaban los que les precedían, y por lo tanto el traductor solo incluía las acciones introducidas en el evento refinado. Otro ejemplo de la relación de extensión es la traducción correcta del evento refinado set_cars (ver Figura 6.30), donde se agrega el nuevo parámetro new_value_colours junto al previo parámetro new_value (también esta traducción es otro buen ejemplo de la habilidad del traductor de generar varios predicados complejos como se tienen en este evento).

Finalmente, crearemos un pequeño programa con la traducción para saber si funciona como se espera. Este modelo no es un algoritmo con un final determinado como el modelo para encontrar en un arreglo y el modelo de la celebridad. Este programa servirá más como un simulador del sistema de un semáforo. El objetivo es ejecutar distintos eventos para ver si el estado del sistema cambia de la manera que se espera y ver si al intentar realizar acciones ilógicas, las guardas de los eventos impiden la ejecución y las invariantes nunca se incumplen (negando acciones como por ejemplo darle luz verde a los carros cuando los peatones tienen el paso).

El programa a ejecutar se puede ver en la Figura 6.31 (en esta imagen del código se borraron los comandos para imprimir el estado actual del sistema con tal de salvar espacio). Los pasos a intentar ejecutar son:

1. Dar luz verde a los peatones para que crucen la calle.

2. Dar luz roja-amarilla a los carros, y luego luz verde para que puedan avanzar (son dos eventos

```
TrafficLightMchRef mch;

cout << "--Luz verde a peatones" << endl;
mch.set_peds_go();

cout << "--Luz roja-amarilla y luego verde a carros con peatones en verde. Debe ser error." << endl;
mch.set_cars(false, Set<COLOURS_CS>({red,yellow}) );
mch.set_cars(true, Set<COLOURS_CS>({green}) );

cout << "--Luz roja a peatones" << endl;
mch.set_peds_stop();

cout << "--Luz verde a carros" << endl;
mch.set_cars(true, Set<COLOURS_CS>({green}) );

cout << "--Luz verde a peatones con carros en verde. Debe ser error." << endl;
mch.set_peds_go();

cout << "--Luz roja a carros sin antes dar amarilla. Debe ser error." << endl;
mch.set_cars(false, Set<COLOURS_CS>({red}) );

cout << "--Luz amarilla y luego roja a carros" << endl;
mch.set_cars(false, Set<COLOURS_CS>({yellow}) );
mch.set_cars(false, Set<COLOURS_CS>({red}) );
```

Figura 6.31: Archivo main para probar la traducción del modelo TrafficLight. No se visualizan las líneas necesarias para imprimir los resultados en consola.

separados). Esto no debe ser permitido por el sistema, ya que los peatones tienen luz verde y están cruzando la calle. Si el segundo evento para dar luz verde ciertamente falla, el semáforo de los carros se habrá quedado en luz roja-amarilla.

3. Dar luz roja a los peatones.

4. Pasar de luz roja-amarilla a luz verde para los carros, para que puedan proseguir.

5. Dar luz verde a los peatones. Esto no debe ser permitido, ya que los carros tienen la luz verde.

6. Dar luz roja a los carros. Esto no debe ser permitido, ya que actualmente los carros tienen luz verde, y se les debe dar luz amarilla antes de luz roja.

7. Dar luz amarilla y luego luz roja a los carros para que se detengan.

Los resultados de ejecutar este código se ven en la Figura 6.32. El programa compiló y funcionó como se esperaba, en los mismos tiempos de ejecución que en casos previos. Es importante notar que para las variables colours en esta ilustración: 0, 1 y 2 son rojo, amarillo y verde respectivamente. Se ve como el sistema empieza con peatones y carros con luz roja y de ese punto prosigue de la siguiente manera:

1. Al realizar el evento para dar paso a los peatones, el semáforo peatonal pasa a tener luz verde y los peatones tienen el paso en peds_go.

2. Al intentar darle luz verde a los carros mientras los peatones tienen el paso, ciertamente recibimos el mensaje de que las guardas no se cumplen para ejecutar el evento. En este momento, el estado del sistema se detuvo en la transición de luz roja-amarilla a verde.

3. Se instrucciona para dar luz roja a peatones, y esto cambia los valores peds_go a falso y peds_colour a ser luz roja.

4. El siguiente evento le da luz verde a los carros, por lo que, como se ve, cars_go pasa a verdadero y cars_colours pasa de mostrar rojo y amarillo a iluminar verde.

5. Al intentar dar luz verde a peatones mientras los carros tienen el paso, se rechaza la acción y recibimos una notificación informando que las guardas del evento no se cumplen.

6. Al intentar dar luz roja a los carros sin antes dar luz amarilla, se rechaza la acción y recibimos una notificación informando que las guardas del evento no se cumplen. En este evento y en el anterior el estado del sistema no cambia.

7. Finalmente, en dos eventos separados se les da luz amarilla y luego luz roja a los carros. Como se ve en las últimas líneas de la figura, cars_colours regresa a solo contener luz roja y cars_go es falso pues los carros no tienen el paso.

```
cars_colours: {0}
cars_go: 0
peds_go: 0
peds_colour: 0

--Luz verde a peatones
cars_colours: {0}
cars_go: 0
peds_go: 1
peds_colour: 2

--Luz roja-amarilla y luego verde a carros con peatones en verde. Debe ser error.
Can't execute set_cars event. Guards are false.

cars_colours: {0,1}
cars_go: 0
peds_go: 1
peds_colour: 2

--Luz roja a peatones
cars_colours: {0,1}
cars_go: 0
peds_go: 0
peds_colour: 0

--Luz verde a carros
cars_colours: {2}
cars_go: 1
peds_go: 0
peds_colour: 0

--Luz verde a peatones con carros en verde. Debe ser error.
Can't execute set_peds_go event. Guards are false.

--Luz roja a carros sin antes dar amarilla. Debe ser error.
Can't execute set_cars event. Guards are false.

--Luz amarilla y luego roja a carros
cars_colours: {0}
cars_go: 0
peds_go: 0
peds_colour: 0
```

Figura 6.32: Resultado de ejecutar traducción de TrafficLights, donde se ven los cambios en el sistema de semáforo y la traducción enforzando las guardas descritas en el modelo para impedir errores de funcionamiento.

Los resultados muestran que la traducción del modelo de semáforo fue un éxito. Además del programa específico ilustrado en la Figura 6.31 se hicieron pruebas con otras combinaciones de eventos y otro par de intentos a forzar errores; y todos resultan en el mismo comportamiento que vimos en la prueba anterior. Esto demuestra que el traductor también funciona y ofrece utilidad incluso cuando el modelo no ha sido refinado hasta el punto de ser un programa y es más bien un sistema aún abstracto.

En resumen, el traductor presentó buenos resultados, con el código generado siguiendo el comportamiento definido en los modelos. Vale la pena resaltar que también se hicieron pruebas con otros modelos de Rodin (incluyendo el sistema controlador de visitas en un hospital, un sistema controlador de estacionamiento, un programa para encontrar un número en una matriz y un programa para encontrar el elemento común en dos conjuntos de números). Sin embargo, los resultados y conclusiones que se pueden obtener de ellos son muy similares a los tres modelos presentados anteriormente, por lo que se considera redundante presentar y analizar esas traducciones.

El único resultado mixto que presentó el traductor es la forma en la que maneja eventos que refinan, pero no extienden, eventos abstractos. En estos casos, la máquina refinada sigue incorporando las variables e invariantes de estas máquinas abstractas, pero sin los cambios a esas variables a lo largo de los eventos, lo cual causa que la verificación de las invariantes en C++ cause un error o simplemente sean falsas inmediatamente. Aún con este problema, al desactivar por defecto la verificación de invariantes estas traducciones pueden ser ejecutadas, y las pruebas demuestran que su comportamiento sigue siendo el esperado.

Capítulo 7

# Conclusiones

## 7.1. Conclusiones

En este trabajo se logró desarrollar exitosamente un traductor automático de modelos correctos en Rodin a C++, bajo las delimitaciones y alcances establecidos al principio del proyecto. Esta herramienta ofrece una ayuda para el desarrollo formal de software al cerrar la brecha identificada en el planteamiento del problema entre el modelaje formal del sistema a crear y su implementación. Los desarrolladores de software pueden usar el programa generado por el traductor o como una base de inspiración a partir de la cual implementar su propio sistema, o como una librería que pueden integrar a un sistema existente.

La creación exitosa de este traductor significa que ahora se tiene un plugin de traducción automática para Rodin a C++, el cual es un lenguaje para el que no existía un traductor en funcionamiento en el tiempo presente. Adicionalmente, a diferencia de la mayoría de otros traductores en el campo que solo pueden traducir modelos que han sido refinados hasta el punto de ser programas, este traductor es capaz de traducir casi cualquier modelo en diferentes niveles de abstracción y refinamiento.

En el proceso de creación, se describió la forma en la que los modelos de Event-B están representados internamente en la base de datos de Rodin, que sería a través de árboles sintácticos que pueden recorrerse con la implementación de una interfaz visitor. Este AST de Rodin se utilizó para la lectura de modelos, lo cual garantiza que sin importar las idiosincrasias sintácticas del escritor del modelo (cuántos espacios, paréntesis y plugins de entorno use), siempre y cuando el modelo sea sintácticamente correcto el traductor será capaz de leerlo.

Se establecieron equivalencias entre distintas fórmulas matemáticas en Event-B e implementaciones en C++. Antes de este trabajo, se sabía sin mucho esfuerzo cómo traducir una desigualdad entre enteros a C++; pero no se puede decir lo mismo acerca de cómo traducir una relación entre cualquier tipo de dato y de cómo saber si es una función total. Este trabajo ofrece tales traducciones para la mayoría de operaciones en Event-B en la forma del archivo generado EB2CppTools.h, el cual puede ser usado como un punto de partida para futuros trabajos en este campo de traducción.

La implementación C++ de las funciones matemáticas de Event-B hizo uso de programación por plantilla para minimizar la cantidad de código y flexibilizar la capacidad expresiva del traductor. Esto permite que el traductor establezca una sola implementación para un tipo de dato y sus operaciones (como por ejemplo, conjuntos) que luego puede usarse para distintos tipos de datos

y operaciones (conjuntos de enteros, conjuntos de parejas de booleanos, conjuntos de conjuntos de enteros). En consecuencia, el traductor no impone restricciones al modelador sobre qué tipos complejos de datos puede o puede no utilizar.

Relacionado a lo anterior, la implementación C++ de estas funcionalidades de Event-B optimizó el costo computacional de varias de estas operaciones donde fuese posible, como por ejemplo el uso del conjunto ordenado en C++ para poder encontrar elementos en tiempo logarítmico en vez de tiempo lineal. Se pudieron implementar abstracciones de conjuntos infinitos como los enteros y naturales de tal forma que al hacer ciertas operaciones con ellos, es posible realizarlas sin tener que hacer algo como iterar por todos los números enteros representables en C++.

El programa generado por el traductor también demostró una gran legibilidad para el usuario, gracias a la incorporación de distintas buenas prácticas de programación en la generación automática de código. El traductor automáticamente agrega varios espacios en blanco, tabulaciones, comentarios y nombrado de métodos apropiado. Esta legibilidad facilita la habilidad del usuario de comprender y personalizar el código de acuerdo a sus necesidades.

El traductor logró traducir modelos básicos sin error y al hacer un análisis empírico, se encontró que los programas generados pueden ejecutarse y que funcionan bajo los mismos parámetros que los modelos en los que están basados. El único aspecto negativo es que no es posible verificar las invariantes en el código C++ cuando el modelo tiene eventos que refinan y no extienden eventos pasados.

## 7.2. Trabajo Futuro

Considerando las delimitaciones, dificultades y resultados de este traductor, se recomienda como trabajo futuro los siguientes puntos:

- Implementar la traducción de cuantificadores. Se conoce de por lo menos un traductor existente que logró una aproximación de esto.
- Inicializar el valor de las constantes en contextos utilizando alguna herramienta automática, en vez de dejarlo en manos del usuario.
- Optimizar computacionalmente algunas partes del proceso de traducción. Es posible unir algunos procesos que actualmente están modularizados por separado en un solo algoritmo que resultaría en una traducción más eficiente.
- Agregar más herramientas de pruebas al usuario. Se podrían generar más métodos a las traducciones que le permitan al usuario hacer pruebas personalizadas de manera automática y con más facilidad. Por ejemplo, un método que toma un evento que necesita un parámetro, e intenta ejecutarlo con un rango aleatorio de valores apropiados.

- Demostrar formalmente la correctitud del programa generado. Las pruebas de este trabajo se realizaron empíricamente, pero lo más apto sería realizar pruebas lógicas que demuestren que el programa generado cumple las especificaciones del modelo.
- Restringir datos que no puedan ser representados en C++ debido a límites de bits.
- Cambiar la interfaz para solicitar la traducción a algo más intuitivo.
- Agregar la traducción a otras funcionalidades menores de Event-B como lambda, constant relations, comprensión de conjunto y asignación por predicado/conjunto.

# Bibliografía


[1] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010. DOI: 10.1017/CBO9781139195881.

[2] Alfred V. Aho y col. *Compilers: Principles, Techniques and Tools*. English. Addison Wesley Pearson, 2007.

[3] Juan Fernando Escobar Barona. *Traducción automática a Python de modelos correctos de Event-B en la plataforma Rodin*. Ago. de 2020.

[4] CAST. *Importance of Software Engineering*. https://www.castsoftware.com/glossary/importance-of-software-engineering-code-of-ethics-future. Último acceso: 10-10-2020.

[5] Andreas Fürst y col. "Code Generation for Event-B". En: sep. de 2014. ISBN: 978-3-319-10180-4. DOI: 10.1007/978-3-319-10181-1_20.

[6] Erich Gamma y col. *Design Patterns: Elements of Reusable Object-Oriented Software*. English. Addison Wesley, 1994.

[7] A. Hall. "Seven myths of formal methods". En: *IEEE Software* 7.5 (1990), págs. 11-19. DOI: 10.1109/52.57887.

[8] Anne Elisabeth Haxthausen. *An Introduction to Formal Methods for the Development of Safety-critical Applications*. English. This report is a delivery to The Danish Government's railway authority, Trafikstyrelsen, as a part of the Public Sector Consultancy service offered by the Technical University of Denmark. 2010.

[9] Michael Jastram. *Rodin User's Handbook*. CreateSpace Independent Publishing Platform, 2014.

[10] Michael G. Hinchey Jonathan P. Bowen. "Ten Commandments of Formal Methods ...Ten Years Later". En: *Computer* 39.1 (2006), págs. 40-48. DOI: 10.1109/MC.2006.35.

[11] Dominique Mery y Neeraj Singh. "Automatic Code Generation from Event-B Models". En: oct. de 2011, págs. 179-188. DOI: 10.1145/2069216.2069252.

[12] Alexey Reshko. *5 Reasons Why Software Development is Important*. https://fortyseven47.com/blog/5-reasons-why-software-development-is-important/. Último acceso: 28-02-2022.

[13] Victor Rivera y Néstor Cataño. "Translating Event-B to JML-Specified Java Programs". En: *ACM Symposium on Applied Computing, Software Verification and Testing track (SAC-SVT)*. Mar. de 2014. DOI: 10.1145/2554850.2554897.

[14] Computerworld UK staff. *The biggest software failures in recent history*. https://www.computerworld.com/article/3412197/top-software-failures-in-recent-history.html. Feb. de 2020.

[15] Ian Summerville. *Formal Specification*. https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/WebChapters/PDF/Ch_27_Formal_spec.pdf. Último acceso: 02-03-2022. 2009.

[16] J.M. Wing. "A specifier's introduction to formal methods". En: *Computer* 23.9 (1990), págs. 8-22. DOI: 10.1109/2.58215.