

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

## Traducción automática a Python de modelos correctos de Event-B en la plataforma Rodin

Juan Fernando Escobar Barona

Director: Dr. Camilo Rocha  
Codirector: Dr. Camilo Rueda

Agosto 2020





Santiago de Cali, Agosto 2020.

Señores

**Pontificia Universidad Javeriana Cali.**

Dr. Gerardo Mauricio Sarria

Director de Carrera de Ingeniería de Sistemas y Computación.  
Cali.

Cordial Saludo.

Por medio de la presente hacemos constar que hemos revisado el proyecto de grado “Traducción automática a Python de modelos correctos de Event-B en la plataforma Rodin”, del estudiante de Ingeniería de Sistemas y Computación Juan Fernando Escobar Barona (cod: 2111898), del cual somos directores, y lo consideramos apto para ser presentado y sometido a consideración del jurado.

Atentamente,



---

Dr. Camilo Rocha



---

Dr. Camilo Rueda

Santiago de Cali, Agosto 2020.

Señores

**Pontificia Universidad Javeriana Cali.**

Dr. Gerardo Mauricio Sarria

Director de Carrera de Ingeniería de Sistemas y Computación.  
Cali.

Cordial Saludo.

Tengo el placer de presentar ante usted el proyecto de grado titulado como “Traducción automática a Python de modelos correctos de Event-B en la plataforma Rodin”, para ser sometido a consideración del jurado.

Espero que este proyecto de grado reúna los requisitos académicos necesarios para su aprobación.

Atentamente,

Juan Fernando Escobar Barona

Juan Fernando Escobar Barona

Código: 2111898

# Agradecimientos

Quiero expresar un especial agradecimiento a mis directores de tesis, Camilo Rocha y Camilo Rueda, quienes me apoyaron durante todo el proceso: dándome sugerencias, ayudándome en el diseño del software, ofreciéndome material de referencia, revisando mis avances, entre otras cosas. Quisiera agradecer a Sebastián Lopez por ofrecerme sus conocimientos y material relacionado a este trabajo en las etapas iniciales de este proyecto. Y en general, agradecer a todos los amigos, familiares, profesores, y demás personas, que directa o indirectamente contribuyeron a que este proyecto de grado pudiera finalizar de manera exitosa.



# Abstract

Formal methods offer a lot of value to software development, especially if we want to develop a trustworthy and bug-free software. A software where generally we can find complex topics involved like security, privacy, automatization, among others. The formal methods achieve their goals by the construction of correct models that are mainly based in mathematics and logic. However, the amount of usage that the formal methods get in the industry do not seem to reflect all the advantages that they are meant to provide, and one of the reasons behind that is the huge gap that we find when we try to jump from the correct models to a software that manages to represent and execute such a formal model.

A solution for the problematic that we just mentioned are the translators from formal languages to traditional programming languages. Those translators are meant to automatize and facilitate the code generation of a software that manages to successfully represent and execute the systems that are modeled with formal methods. In this work we effectively built a translator of that nature, and to be more specific, we are talking about the construction of a translator that goes from correct models written in Event-B (formal language for systems modelling) to a program in Python (traditional programming language).

In order to reach that goal, we will face different challenges that go from the theory to technical matters. But the biggest obstacle that we will face in this project is the problem of finding a way to conciliate two languages that are different by nature. For example, in Event-B we can have infinite sets and non-deterministic situations, something that we cannot directly translate in a deterministic and sequential language by nature like Python. Those, and many other challenges, are obstacles that other translator in this context have tried to deal with. It is worth to mention that, until today, there is no translator capable of a complete translation of any kind of model that we can find in Event-B, without having some limitations.

Based on all that we have mentioned up until this point, what we will find in this project is a study of the whole process that we had when building such a translator. This will include: (i) An analysis of the context and theoretical framework of this work. (ii) The design and all the implementation details that we had when building this translator, which were oriented towards finding a solution to the biggest obstacles that we can face when working in this area. (iii) A description of the translation that is carried for every single possible operation, component, and any other characteristic of Event-B that is supported by our code generator. (iv) The analysis of the results that we had when running our translator with different models, of different kind and different sources, as input.

**Keywords:** Code Generation, Event-B, Python, Design by Contract, Programming Languages, Formal Methods.



# Resumen

Los métodos formales traen muchas ventajas en el desarrollo de software, especialmente si lo que se quiere es desarrollar un sistema confiable e infalible, y donde generalmente se tratan temas complejos que requieren de un software robusto como la seguridad, la privacidad, la automatización, entre otros. Los métodos formales logran su cometido a través de la construcción de modelos correctos que se basan principalmente en las matemáticas y la lógica. Sin embargo, el uso de los métodos formales en la industria no parece reflejar todas las ventajas que trae esta manera de desarrollar de software, y una de las razones por las que ocurre esto es que en los métodos formales hay una gran brecha que se debe superar cuando se trata de pasar de los modelos correctos a un software que represente dichos modelos y los ejecute en el mundo real.

Una solución a lo anterior son los traductores de lenguajes formales a lenguajes de programación tradicionales, los cuales buscan automatizar y facilitar la generación de código de un software que permita representar y ejecutar los sistemas que se modelen con métodos formales. En este trabajo buscaremos justamente construir un traductor de esa índole, específicamente, se hablará de la construcción de un traductor de modelos correctos escritos en Event-B (lenguaje formal de modelado) a un código en Python (lenguaje tradicional de programación).

Lograr esta tarea implica varios retos, desde lo teórico hasta lo técnico, pero el obstáculo más grande con el que nos enfrentaremos en este proyecto es el de conciliar dos lenguajes que tienen rasgos muy diferentes. Por ejemplo, en Event-B se pueden tener conjuntos infinitos y puede haber indeterminaciones, algo que no se puede traducir de manera directa en un lenguaje determinístico y secuencial por naturaleza como Python. Éste, y muchos retos más, son obstáculos con los que otros traductores en este campo han tratado de lidiar, y no sobra mencionar que, hasta el momento, no existe un traductor que permita una traducción completa de cualquier modelo que se pueda construir en Event-B, sin incurrir en algunas limitaciones.

Con base en todo lo anterior, lo que encontraremos en este trabajo será un estudio de todo el proceso que llevó la construcción del traductor mencionado, el cual, entre otras cosas, incluirá: (i) El análisis de todo el contexto y marco teórico de este trabajo. (ii) El diseño y detalles de implementación del traductor, los cuales estarán orientados a buscar soluciones a los obstáculos más grandes que hay al trabajar en esta área. (iii) Una descripción de la traducción que se lleva a cabo para todas las posibles operaciones, componentes, y demás características de Event-B soportadas por nuestro generador de código. (iv) El análisis de los resultados que se obtuvieron al enfrentar nuestro traductor con modelos formales de características variadas y provenientes de diversas fuentes.

**Palabras Clave:** Generado de código, Event-B, Python, Diseño por contrato, Lenguajes de programación, Métodos formales.



# Índice general

|  |           |
|--|-----------|
| <b>1. Análisis</b>   | <b>15</b> |
| 1.1. Planteamiento del Problema . . . . .                          | 15        |
| 1.1.1. Formulación . . . . .                                       | 16        |
| 1.1.2. Sistematización . . . . .                                   | 16        |
| 1.2. Objetivos . . . . .   | 17        |
| 1.2.1. Objetivo General . . . . .                                  | 17        |
| 1.2.2. Objetivos Específicos . . . . .                             | 17        |
| 1.2.3. Delimitaciones y Alcance . . . . .                          | 17        |
| 1.3. Justificación . . . . .                                       | 18        |
| 1.4. Marco de referencia . . . . .                                 | 19        |
| 1.4.1. Áreas temáticas . . . . .                                   | 19        |
| 1.4.2. Marco teórico . . . . .                                     | 19        |
| 1.4.3. Trabajos relacionados . . . . .                             | 26        |
| <b>2. Diseño</b>   | <b>33</b> |
| 2.1. Diseño por Contrato . . . . .                                 | 33        |
| 2.2. Tipos de Datos . . . . .                                      | 34        |
| 2.2.1. Mypy . . . . .  | 34        |
| 2.2.2. Manejo de diferentes tipos de datos de Event-B . . . . .    | 36        |
| 2.3. Arquitectura . . . . .  | 38        |
| 2.3.1. Diseño y Arquitectura General del Traductor . . . . .       | 38        |
| 2.3.2. Diseño y Arquitectura General del Código Generado . . . . . | 42        |
| <b>3. Traducción</b>   | <b>47</b> |
| 3.1. Contextos . . . . .   | 47        |
| 3.1.1. Dependencias . . . . .                                      | 47        |
| 3.1.2. Constructor . . . . .                                       | 49        |
| 3.1.3. Métodos . . . . .   | 50        |
| 3.2. Máquinas . . . . .  | 51        |
| 3.2.1. Dependencias . . . . .                                      | 53        |
| 3.2.2. Constructor y Métodos Básicos . . . . .                     | 53        |
| 3.2.3. Eventos . . . . .   | 54        |
| 3.2.4. Métodos para el Usuario . . . . .                           | 57        |
| 3.3. Reglas de Traducción . . . . .                                | 59        |
| 3.3.1. Elementos Especiales de Event-B . . . . .                   | 60        |
| 3.3.2. Operaciones sobre Predicados . . . . .                      | 61        |
| 3.3.3. Operaciones sobre Conjuntos . . . . .                       | 61        |

|  |            |
|--|------------|
| 3.3.4. Predicados sobre Conjuntos . . . . .                          | 61         |
| 3.3.5. Operaciones sobre Enteros . . . . .                           | 61         |
| 3.3.6. Predicados sobre Enteros . . . . .                            | 64         |
| 3.3.7. Relaciones . . . . .  | 64         |
| 3.3.8. Funciones . . . . .   | 64         |
| 3.3.9. Operaciones sobre Relaciones . . . . .                        | 66         |
| 3.3.10. Operaciones sobre Funciones . . . . .                        | 66         |
| 3.3.11. Asignaciones . . . . .                                       | 66         |
| <b>4. No Determinismo y el Infinito</b>                              | <b>69</b>  |
| 4.1. Métodos no Determinísticos . . . . .                            | 69         |
| 4.1.1. PyAutoExecute . . . . .                                       | 69         |
| 4.1.2. PyRandValGen . . . . .  | 70         |
| 4.1.3. PyChoice y PyBecomesSuchThat . . . . .                        | 71         |
| 4.2. Conjuntos Infinitos . . . . .                                   | 72         |
| 4.2.1. Conjuntos Infinitos y por Comprensión . . . . .               | 72         |
| 4.2.2. Conjuntos Infinitos Extendidos . . . . .                      | 73         |
| 4.3. Relaciones y Funciones Infinitas . . . . .                      | 74         |
| 4.3.1. Relaciones Infinitas . . . . .                                | 74         |
| 4.3.2. Funciones Infinitas . . . . .                                 | 78         |
| <b>5. Detalles de Implementación y Uso del Traductor</b>             | <b>85</b>  |
| 5.1. Buscando Legibilidad del Código Generado . . . . .              | 85         |
| 5.2. Buenas Prácticas . . . . .                                      | 87         |
| 5.3. Retos de la Implementación . . . . .                            | 87         |
| 5.4. Instalación del Traductor . . . . .                             | 88         |
| 5.5. Uso del traductor . . . . .                                     | 88         |
| <b>6. Resultados</b>   | <b>93</b>  |
| 6.1. Traducción de Programas Completos . . . . .                     | 94         |
| 6.2. Traducción de modelos de Libros de Event-B . . . . .            | 100        |
| 6.3. Traducción de modelos usados en Cursos Universitarios . . . . . | 107        |
| 6.4. Traducción de modelos construidos para este Traductor . . . . . | 108        |
| <b>7. Conclusiones</b>   | <b>115</b> |
| 7.1. Conclusiones Finales . . . . .                                  | 115        |
| 7.2. Trabajo Futuro . . . . .  | 117        |
| <b>Bibliografía</b>  | <b>119</b> |

# Introducción

Hoy en día, el poder gozar de la proliferación que la tecnología ha tenido en prácticamente todos los ámbitos se puede considerar uno de los mayores beneficios de la humanidad, sin embargo, también hemos oído constantemente de las fallas que estos productos tecnológicos presentan. En un estudio realizado en 2015 por Standish Group, sólo el 29 % de 50000 proyectos de software estudiados fueron exitosos [SH15]. Esto revela efectivamente que, a pesar de todos los grandes productos que la industria del software le ha ofrecido a la sociedad, todavía existen muchos obstáculos cuando se habla de desarrollo de software. En el momento en que se despliega un software lo suficientemente complejo, es muy difícil asegurar que este funcionará perfectamente bajo cualquier circunstancia sin requerir ningún tipo de arreglo o servicio de mantenimiento, pues las estrategias de “testing” son muy buenas para encontrar errores, pero no para demostrar su ausencia [SG14].

El desarrollo formal de software podría ser una excepción a lo anterior, ya que permite un camino a la construcción y análisis de software correcto, y por lo tanto es una solución para una industria en busca de sistemas informáticos que no presenten fallas. ¿Pero cómo sucede esto? Los métodos formales hacen uso de las matemáticas y la lógica como herramienta para modelar sistemas de software (o hardware), gracias a ello se asegura su objetividad y precisión a la hora de demostrar la correctitud del producto final. De manera sencilla, el proceso consiste en modelar un sistema correcto, y usar ese modelo para obtener un programa que pueda representarlo y ejecutarlo.

Un ejemplo exitoso de la aplicación de métodos formales ocurre en París. En esta ciudad se hizo uso del desarrollo formal de software para construir un sistema automático de protección de trenes que funciona desde 1989, y que con el paso de los años no se le han detectado bugs (fallas informáticas). Este sistema controla y regula constantemente la velocidad de los trenes, lo cual en 2010 aseguraba la protección de 1800.000 pasajeros diariamente! [Hax10].

Naturalmente, el presentar los métodos formales de esta manera haría verlos como una herramienta perfecta sin ningún tipo de desventaja. Alguien se preguntará: Si el producto de software que se obtiene empleando métodos formales no tiene fallas, ¿por qué la industria del software no se limita al uso de esta metodología? La respuesta está en que el desarrollar software bajo este paradigma formal no es una tarea sencilla e implica diversas problemáticas durante su desarrollo.

En este trabajo nos enfocaremos principalmente en las problemáticas que se presentan cuando ya se ha modelado un sistema correcto de software usando un lenguaje de modelado (Event-B), y sólo queda pendiente el obtener un programa en un lenguaje tradicional de programación (Python) que represente y execute lo que se modeló. La idea general será estudiar la naturaleza de esta problemática y el proceso de construcción de una herramienta que busque solventarla, esta última, hará parte de las herramientas que se suelen construir para combatir este problema: los traductores automáticos de lenguajes de modelado a lenguajes de programación tradicionales.



# Análisis

---

## 1.1. Planteamiento del Problema

Tal y como lo expresa Hillel Wayne, quien se especializa en el empleo de métodos formales en los negocios, el desarrollo formal de software promete productos de software correctos, pero para ello, hay que superar varios obstáculos durante su desarrollo que para la industria pueden representar varios inconvenientes [Way19]. En este trabajo nos afectan los siguientes problemas relacionados a ello: (i) El primer obstáculo es el tiempo que toma desarrollar un sistema tan robusto usando métodos formales. Aunque esto sea un tema discutible, no se puede obviar el hecho que la industria tiende a ver los métodos formales como un proceso que ralentiza la producción, lo cual es problemático en un mercado que tiende a priorizar las soluciones rápidas [Wol12]. (ii) El segundo inconveniente viene siendo la necesidad de tener expertos en matemáticas y lógica que se encarguen de crear los modelos y demostrar su correctitud. Conseguir este grupo de expertos es complicado en comparación a conseguir programadores que desarrollen software con métodos más tradicionales. (iii) Finalmente, los métodos formales llegan a una etapa en que se debe migrar del modelo correcto obtenido a un programa que lo represente y ejecute [FHB<sup>+</sup>14]. Esta última es la problemática que más nos interesará.

Afortunadamente, los métodos formales proveen diferentes soluciones para tratar estos problemas, siendo una de ellas los traductores automáticos de lenguajes de modelado a lenguajes de programación tradicionales, también conocidos como “Generadores de Código” (usaremos este término como sinónimo de traductores automáticos de lenguajes a lo largo de este documento). Para delimitar de una vez el enfoque de este trabajo, nos enfocaremos en la traducción que va desde Event-B (un lenguaje formal de modelado muy reconocido en este ámbito) a Python (un lenguaje de programación tradicional muy famoso en todo el mundo).

Es importante entender cómo la generación automática de código aporta a la solución de las problemáticas que se mencionaron previamente: (i) La traducción automática de un modelo correcto a un lenguaje de programación tradicional reduce el tiempo que requiere el proceso de construir software haciendo uso de métodos formales. (ii) Se reduce la posibilidad de errores que puede cometer un humano a la hora de realizar la traducción manualmente. Incluso, estos traductores pueden funcionar bajo reglas de traducción formales, sobre las cuales se pueden realizar pruebas de correctitud que demuestran que la traducción es exacta [EBI<sup>+</sup>12]. (iii) Paralelamente, estos traductores pueden ayudar a cerrar la brecha que hay entre el modelo formal y el producto final,

puesto que demuestran que existe una relación entre estos dos ámbitos que se puede automatizar en una traducción.

Ahora podemos proceder con mencionar algunas de las problemáticas inherentes a la traducción de lenguajes (particularmente de Event-B a Python), que son con las que lidiaremos primordialmente en este trabajo. El principal problema de traducir un modelo correcto a un lenguaje de programación tradicional es claramente el hecho de que ambos están basados en diferentes paradigmas: Mientras Event-B hace uso de expresiones matemáticas, teoría de conjuntos, y la lógica, Python es un lenguaje multiparadigma que soporta la programación orientado a objetos y que se ejecuta bajo instrucciones tradicionales de los lenguajes de programación (while, if-else, for, etc.). Es más, al ser Python dinámicamente tipado, la traducción se hace menos evidente cuando se hace contraste con la rigurosidad en el tipado de Event-B. Otro gran obstáculo (el más grande de todos, y el cual se estudiará eventualmente en este trabajo), es que mientras Python es un lenguaje secuencial y determinístico, Event-B permite diferentes tipos de indeterminaciones (ej. Elegir un elemento cualquiera de un conjunto infinito). Los obstáculos que se han mencionado hasta el momento son problemas con los que otros traductores han tenido que lidiar muchas veces de maneras poco ideales, o incluso, estos generadores de código renuncian a traducir ciertas características de los modelos formales, lo cual se estudiará eventualmente en la sección de trabajos relacionados.

Finalmente, está el reto adicional de buscar que la traducción final sea lo más natural y entendible posible, por ejemplo, ofreciendo herramientas que permitan hacer relucir la relación que hay entre el modelo formal y el código generado. De este modo, al usuario le será más fácil hacer uso de estos traductores, entender su funcionamiento, y hacerle seguimiento a su ejecución. Esto último con el fin de ayudar a cerrar la brecha que existe entre el mundo formal y los programas que buscan representar y ejecutar dichos modelos en el mundo real.

### 1.1.1. Formulación

¿Cómo lograr una traducción automática de modelos correctos de sistemas de software en Event-B (en la plataforma Rodin) a un programa en Python que pueda representarlo y ejecutarlo?

### 1.1.2. Sistematización

- ¿Cómo definir un proceso de traducción desde un modelo formal escrito en Event-B a un programa ejecutable en Python?
- ¿Cómo superar los retos que implica la traducción de dos modelos que funcionan bajo paradigmas distintos? Específicamente, considerando que Event-B se basa en operaciones/elementos matemáticos, hace uso constante de predicados lógicos, permite situaciones no determinísticas, entre otras características que Python no posee por defecto.

- ¿Cómo diseñar el código que se generará en Python de manera que se busque facilitar su entendimiento y funcionamiento, y posiblemente, ofreciendo métodos que permitan hacer seguimiento a la ejecución del código?
- ¿Cómo vincular un Plug-in a Rodin que sea capaz de obtener los modelos que se definan en dicha plataforma para posteriormente generar código en Python?

## 1.2. Objetivos

### 1.2.1. Objetivo General

Diseñar e implementar un Plug-in en Rodin que permita la traducción automática de modelos correctos en Event-B al lenguaje de programación Python.

### 1.2.2. Objetivos Específicos

- Implementar en Rodin un Plug-in que extraiga los modelos que se definan en la plataforma, los procese, y genere código en Python que represente dicho modelo y permita ejecutarlo.
- Definir e implementar una relación entre la manera en que se especifican modelos formales en Event-B (axiomas, invariantes, contextos, máquinas, eventos, refinamientos, etc.), y un código en Python que lo represente y sea ejecutable.
- Paralelo al objetivo anterior, buscar reducir las limitaciones del traductor (como ocurre con otros generadores de código) que se hacen necesarias para lidiar con las diferencias fundamentales entre los modelos formales y los lenguajes de programación (por ejemplo, el no determinismo en Event-B contrastado con la ejecución secuencial de programas en Python).
- Diseñar el traductor de modo que el código de salida busque también facilitar su uso, entendimiento y funcionamiento, por ejemplo, a través de métodos que permitan hacer seguimiento a la ejecución del programa (observando su estado, evaluando el valor de sus axiomas o invariantes, etc.). Esto con el fin de resaltar y evidenciar la relación entre el modelo y el código generado.

### 1.2.3. Delimitaciones y Alcance

- El traductor se construye asumiendo que se trabaja con modelos escritos en la versión 3.4 de Rodin, y el código generado se debería poder ejecutar como mínimo en Python 3.7 (específicamente, Python 3.7.2).
- Debido a que el traductor será un Plug-in en Rodin, que a la vez hace parte de la plataforma *Eclipse*, se hace obligatorio programarlo en Java.
- El traductor soportará la traducción de modelos abstractos (“cuasi-programas”, donde pueden haber indeterminaciones) y de modelos concretos.

- El traductor asumirá que el modelo a traducir no tiene errores de sintaxis, más aún, Rodin debe indicar que el modelo a traducir no tiene errores de ninguna clase (ej. Errores que indiquen que a una variable de tipo entero se le está asignando un valor de otro tipo de dato).
- El traductor estará principalmente diseñado para traducir modelos correctos, de lo contrario, a pesar de que genere código ejecutable, lo más probable es que eventualmente se llegue a un punto en que el código viole un teorema, axioma, variante, o invariante, y por lo tanto, se genere una excepción que detenga la ejecución del código.
- El traductor, como se ha ido explicando, está pensado para generar código ejecutable, incluyendo la traducción de predicados y expresiones que aseguren su correctitud (como los teoremas, variantes, invariantes, axiomas, y demás características como el tipado de variables y constantes). Pero es importante aclarar que el código que se genera no está pensado para hacer pruebas formales sobre la correctitud del modelo, por dos razones: (i) Justamente esa es la tarea que se supone se hace en Rodin, el cual ya posee de muchas herramientas para realizar dicho trabajo. (ii) Como se ha ido justificando, nos estamos centrando en la problemática de obtener código ejecutable que represente y ejecute un modelo formal, asumiendo que a dicho modelo ya se le ha demostrado su correctitud.
- Para estudiar los resultados de la traducción, se usará el Plug-in frente a las siguientes situaciones: (i) Modelos correctos de libros oficiales y reconocidos en el ámbito de Event-B (ej. *Rodin User's Handbook*). (ii) Modelos correctos usados en universidades para la enseñanza de métodos formales. (iii) Modelos correctos de un artículo científico de Abrial (El creador de Event-B) que simulan programas tradicionales, y por lo tanto son modelos refinados a tal punto en que no hay indeterminismos (los cuales son ejemplos muy interesantes de estudiar, ya que la brecha entre el modelo y el código que lo ejecuta se encuentra reducida en gran medida). (iv) Modelos que se construyeron específicamente para demostrar ciertas características del traductor que se construyó en este trabajo.
- Debido a que la traducción de un modelo formal a un lenguaje de programación como Python implica dos ámbitos con bastantes diferencias que son difíciles de lidiar (de hecho, otros traductores de renombre en este mismo ámbito tienden a omitir la traducción de algunas características de los modelos formales, como se estudiará en la sección de trabajos relacionados), es muy probable que haya ciertas características que no puedan traducirse completamente debido a diferentes circunstancias (ej. Limitaciones técnicas), las cuales deberán ser expuestas en este trabajo, pues es importante saber que limitaciones tendrá el traductor del que se hablará en este proyecto.

### 1.3. Justificación

Este trabajo busca en general resolver dos grandes problemáticas.

La primera (y la más importante), es la de ofrecer los beneficios que traen los generadores de código en el mundo de los métodos formales, los cuales se pueden resumir en: (i) Agilizar el proceso de desarrollo formal que es criticado por tomar mucho tiempo. (ii) Descartar la posibilidad de cometer errores en traducciones manuales. (iii) La traducción se realiza a Python, un lenguaje que hoy en día es muy famoso alrededor del mundo en la industria del software (lo cual es muy conveniente), además de que no hay traductores oficiales a este lenguaje. (iv) Y finalmente, considerando lo que se mencionó en la sección de “Planteamiento del Problema”, hay un problema a la hora de conciliar la relación entre un modelo formal y un código que lo represente y ejecute. Los ingenieros tradicionales tienden a ser escépticos respecto a este tema, o simplemente no están acostumbrados a relacionar las matemáticas, la lógica, y las especificaciones y constructos formales, con un programa ejecutable. ¡Sin embargo!, dependiendo del diseño del traductor, la relación que hay entre el modelo y el código generado puede hacerse más evidente y acercar así al mundo formal con el “real”. El traductor que se presentará en este trabajo tendrá esto último en cuenta en su diseño e implementación.

Queremos agregar además que estos generadores de código también son usados en universidades para facilitar la enseñanza a los estudiantes de ingeniería de software sobre esta manera de desarrollar código, y que a través de los traductores se puede ser testigo de que no hay mucha lejanía entre desarrollar formalmente a hacerlo tradicionalmente [CR09]. Dependiendo del diseño, se puede aportar indirectamente a esta causa con el traductor que se construyó en este proyecto.

La segunda problemática en la que se busca trabajar es la de mejorar en lo posible las limitaciones que tienen los traductores ya existentes en este campo. Veremos en la sección de trabajos relacionados que los generadores de código tienen problemas para lidiar con algunas características de los modelos formales tales como: el indeterminismo, el refinamiento y extensión de modelos, y la interpretación de por sí de un lenguaje.

## 1.4. Marco de referencia

### 1.4.1. Áreas temáticas

- Computer Science – Software Engineering – Formal Methods
- Computer Science – Software – Programming Languages - Processors
- Computer Applications – Programming Tool

### 1.4.2. Marco teórico

Ya que Event-B (en la plataforma Rodin) es el lenguaje formal que buscamos traducir, lo primero que sería pertinente aclarar es la definición de Event-B y sus características generales. De esta manera entenderemos el contexto que envuelve a todos los componentes que vamos a traducir.

Posteriormente, a medida que vayamos explicando la traducción, iremos conociendo los detalles de dichos componentes, pues debemos tener mucho cuidado con plasmar todas las cualidades esenciales de un modelo en Event-B si lo que se quiere realmente es generar un código que lo pueda representar y ejecutar.

Sin más dilación, veamos las características generales de Event-B:

- Event-B es un lenguaje para el modelamiento formal de sistemas reactivos en el que se pueden resolver problemas de alta complejidad que involucren software y hardware. Este lenguaje está pensado para permitir realizar un análisis riguroso y objetivo de un sistema, y para ello se basa en las matemáticas, la lógica, la teoría de conjuntos, y la lógica de predicados [CR16].
- Rodin es la plataforma sobre la que se puede usar Event-B, en la cual: se generan automáticamente las “Proof Obligations” (obligaciones de prueba) que permiten demostrar la correctitud del sistema, además, se ofrecen distintas herramientas de edición para crear y construir los modelos los modelos, y se le pueden integrar Plug-ins con facilidad (como el que se construye en este proyecto) [Jas12].
- Algunos de los elementos que Event-B soporta son: el uso de predicados, expresiones (con un tipo de dato fijo afiliado a las mismas), cuantificación sobre variables, conjuntos definidos por comprensión, entre muchos otros elementos y operaciones (que veremos en detalle a lo largo del documento) [Jas12].

En Rodin, uno separa el espacio de trabajo por proyectos, siendo cada uno independiente del otro. En cada proyecto se puede tener cualquier cantidad de modelos (independientes uno del otro). Siendo los modelos el componente de Event-B que nos interesa traducir cada uno por separado, concentrémonos en estudiar sus características:

- En Event-B, un modelo está compuesto por contextos y máquinas.
- Los contextos declaran los elementos estáticos de un modelo: las constantes y “Carrier Sets” (que son conjuntos pensados para que el usuario pueda definir su propio tipo de dato). En los contextos se definen axiomas que se encargan de dar propiedades a dichas constantes y conjuntos (por ejemplo, asignarle el tipo de dato que almacena una constante o la cardinalidad de un conjunto).
- Mientras tanto las máquinas contienen la parte dinámica de un modelo. En ellas se definen variables, cuyos valores en un momento determinado de tiempo definen el estado de la máquina. Las máquinas contienen invariantes, que son predicados que dan características a las variables y que deben ser verdad en cualquier estado que la máquina pueda alcanzar. Luego están los eventos, los cuales asignan nuevos valores a las variables y por lo tanto permiten a la máquina alcanzar un nuevo estado. Un evento tiene guardas (precondiciones que deben cumplirse para que el evento pueda ejecutarse) y acciones (aquellas que les asignan nuevos

valores a las variables de la máquina). Entre los eventos hay uno especial que es el evento de inicialización, el cual se encarga de generar el estado inicial de la máquina. Los eventos también pueden tener parámetros, los cuales pueden tomar cualquier valor mientras hagan cumplir las precondiciones de un evento, y se usan para dar más flexibilidad a las acciones del mismo.

- Una máquina también puede tener una variante, la cual es una expresión que retorna un valor entero o un conjunto, y se usa para demostrar la terminación de un proceso que lo requiera. Se usa en conjunto con eventos convergentes (un evento convergente es aquel que para validar la correctitud del modelo debe disminuir el valor/cardinalidad de la variante luego de ejecutarse).
- Otro aspecto importante son las relaciones que se pueden definir entre los contextos y las máquinas. Una máquina puede hacer uso de las constantes y conjuntos de un contexto, a este fenómeno se le refiere uno como: la máquina  $m$  “ve” un contexto  $c$ . Pero el asunto no termina ahí, Event-B permite la extensión de contextos y el refinamiento de máquinas. La extensión de contextos es básicamente: Un contexto (que será el contexto concreto) puede extender otro contexto (al que nos referiremos como contexto abstracto), lo cual implica que el contexto concreto podrá hacer uso de todas las constantes y conjuntos del contexto abstracto, pero además de eso debe respetar los axiomas del contexto abstracto. Adicional a eso, el contexto concreto podrá definir nuevas constantes, axiomas, y conjuntos. El refinamiento de máquinas es un proceso similar, una máquina (concreta) puede refinar una máquina (abstracta), donde podrá agregar nuevos elementos mientras respete la correctitud de la máquina refinada y de ella misma. Es importante mencionar que el refinamiento tiene otras características, pero haremos referencia a ellas en su debido momento.
- En general, la ejecución de un modelo de Event-B es la siguiente (asumiendo que hay un contexto y una máquina que “ve” dicho contexto): El contexto asigna valores a sus constantes de modo que se cumplan los axiomas del contexto, dichos valores serán estáticos a lo largo de la ejecución. Luego, la máquina usa esas constantes para inicializar los valores de sus variables de modo que se cumplan las invariantes, generando el estado inicial de la máquina. Posteriormente, se puede ejecutar cualquier evento de la máquina mientras sus guardas estén habilitadas. Cada vez que se ejecuta un evento, la máquina alcanza un nuevo estado, pues al menos el valor de una variable debe de haber cambiado en el proceso.
- Todos los conceptos que se expusieron en los puntos anteriores le permiten a Event-B la siguiente manera de modelar sistemas (y que de hecho es la recomendada, pues los refinamientos son un concepto central en Event-B [Jas12]): Se empieza con contextos y máquinas simples y se demuestra su correctitud. Luego se extienden los contextos y se refinan las máquinas, agregando más elementos y niveles de detalle. El proceso se repite hasta que eventualmente se obtenga una máquina y contexto finales que recojan todas las características del sistema que se quiere modelar (ver Figura 1.1). Siendo lo que se acaba de describir la forma en que se recomienda modelar en Event-B, es de suma importancia permitir que el generador de

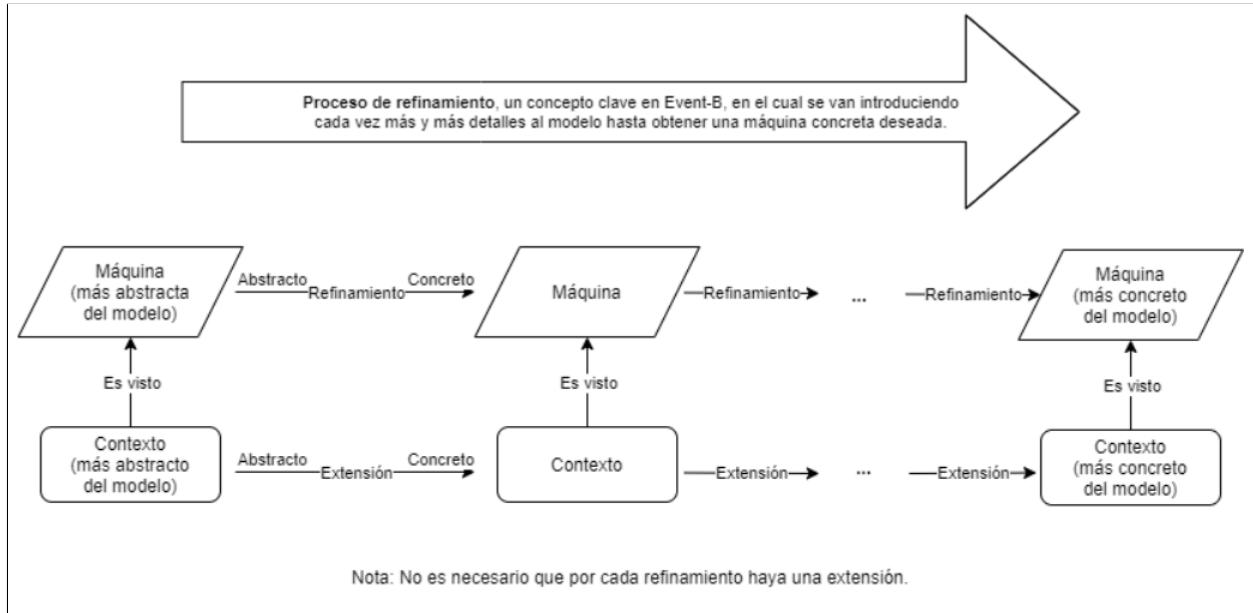


Figura 1.1: *Proceso (recomendado) de modelado en Event-B, en el cual, partiendo de componentes abstractos, se van agregando más detalles a los contextos y máquinas hasta obtener los componentes concretos deseados.*

código permita la traducción de modelos que contengan múltiples contextos y máquinas que se relacionan entre ellas mediante extensiones, refinamientos, y máquinas “viendo” contextos.

Hay dos características más sobre Event-B que son muy importantes para tener en cuenta a la hora de construir el generador de código:

- La primera, es tener en cuenta la documentación oficial sobre la manera en que Event-B (en Rodin) define e interpreta todos sus componentes. Para este trabajo se tienen en cuenta principalmente los siguientes documentos: *Rigorous Open Development Environment for Complex Systems* [MJRV05] y *A Concise Summary of the Event B mathematical toolkit* [Rob14]. El primer documento esboza la manera en que se definen expresiones, predicados, conjuntos, entre otros elementos (por ejemplo, ver Figura 1.2). También se encuentran detalles sobre la manera en que se infiere el tipo de dato de constantes o expresiones. Por ejemplo, si una variable representa los naturales, su tipo de dato serán todos los enteros. El segundo documento reúne de manera concisa todas las operaciones matemáticas que están permitidas en Event-B, y nos sirve como guía para saber cuáles operaciones y elementos matemáticos debemos poder traducir potencialmente.
- La segunda característica es el indeterminismo en Event-B, esta es la cualidad que los generadores de código en este ámbito tienen mayor dificultad de tratar. Los modelos en este lenguaje

pueden tener indeterminismos en situaciones de diferente índole tales como: (i) En cualquier estado en que se encuentre una máquina. Cualquier evento que pueda ejecutarse en un momento dado puede hacerlo, no existe una regla para elegir cuál debería ser el siguiente evento por ejecutarse. (ii) Varias situaciones en un modelo de Event-B pueden ejecutarse de muchas maneras, por ejemplo: a la hora de inicializar las constantes de un contexto, éstas pueden tener más de una manera (infinitas incluso) de inicializarse y cumplir con los axiomas. (iii) Los cuantificadores requieren verificar el cumplimiento de un predicado, el cual, si se quiere verificar de manera determinista para cada valor que lo cumpla, podría implicar situaciones (que de hecho son muy comunes) en las cuales toque verificar un conjunto de valores infinito (lo cual no es computable).

Antes de continuar, conviene detenernos para hablar un poco de ProB. ProB es la herramienta oficial en Rodin que permite animar máquinas, esto quiere decir, permite simular el funcionamiento de una máquina: Le asigna valores a las constantes y variables para inicializar el estado de las componentes del modelo. Ejecuta eventos (de manera no determinística) para que la máquina se mueva entre estados. Y revisa constantemente que se mantenga la correctitud del modelo; Aunque no usaremos ProB para la construcción de este traductor, nos sirve como referencia pues realiza algunas tareas que el código que se genere debe hacer. Pero hay algo más que queremos mencionar, ProB también tiene dificultad para animar cualquier tipo de máquinas: en algunos casos no es capaz de generar valores que cumplan las restricciones del modelo, también hay ciertas dificultades para operar con conjuntos infinitos, y las simulaciones tienden a hacerse en conjuntos de datos pequeños. Lo que mencionamos se usa para recalcar que tratar de generar una herramienta que ejecute un modelo en Event-B no es una tarea que se pueda lograr completamente sin algunas limitaciones.

Habiendo estudiado ya el ámbito del modelo formal con Event-B, pasemos al otro lado de la moneda, el código que se debe generar en Python. Lo primero es encontrar una manera de representar todas las restricciones que permiten la correctitud del modelo formal que se quiere traducir. Para ello, afortunadamente hay una rama del diseño de software que se encarga de ello: el diseño por contrato.

El diseño por contrato exige que se definan especificaciones verificables para cada componente del software o sistema a construir, lo cual se hace mediante el uso de axiomas, invariantes, precondiciones, postcondiciones, y tipos abstractos de datos. A estas especificaciones se les denomina *contratos* (ítem que usaremos constantemente en este documento!), y las cuales podemos relacionar directamente con los correspondientes elementos que aseguran la correctitud en un modelo formal en Event-B. Incluso, al ser una rama del diseño de software, existen ya librerías en Python que se hacen cargo de facilitar este tipo de diseños. En este trabajo veremos que, aunque no se encontró una librería que se adaptara a la construcción de nuestro traductor, es importante tener en cuenta que sí existe una manera de verificar correctitud en el software tradicional, y que se hace usando constructos similares a los de Event-B.

construct  $\mathbb{P}(s)$  is a set called the *power set of s*. Finally, given a list of variables  $x$  with distinct identifiers, a predicate  $P$ , and an expression  $E$ , the construct  $\{x \cdot P \mid E\}$  is called a *set defined in comprehension*. Here is our new syntax:

```

predicate ::=  $\perp$   

 $\top$   

 $\neg$  predicate  

predicate  $\wedge$  predicate  

predicate  $\vee$  predicate  

predicate  $\Rightarrow$  predicate  

predicate  $\Leftrightarrow$  predicate  

 $\forall$  var_list  $\cdot$  predicate  

 $\exists$  var_list  $\cdot$  predicate  

[var_list := exp_list] predicate  

expression = expression  

expression  $\in$  set

expression ::= variable  

[var_list := exp_list] expression  

expression  $\mapsto$  expression  

set

variable ::= identifier

set ::= set  $\times$  set  

 $\mathbb{P}(\textit{set})$   

{ var_list  $\cdot$  predicate | expression }  

variable

```

SY6

## 5.2 Non-freeness and Substitution Rules

Figura 1.2: Sintaxis oficial que se maneja en Event-B para los predicados, presentada a través de reglas recursivas. Tomado de “Rigorous Open Development Environment for Complex Systems” [MJRV05].

Ahora, hablemos del lenguaje en el que estará escrito el código generado: Python.

Python es un lenguaje tradicional de programación. En este marco teórico queremos resaltar dos características de este lenguaje: Python es dinámicamente tipado, la cual es una característica completamente contradictoria al tipado estático de Event-B. Además, Python se ejecuta por defecto de manera secuencial (como la gran mayoría de lenguajes de programación), y, por lo tanto, debe buscarse una manera de traducir los indeterminismos en modelos que se quieran traducir y contengan elementos/situaciones que no sean deterministas.

Hemos visto ya teoría sobre el lenguaje de los modelos que se quieren traducir, y el lenguaje al que lo traduciremos. Pero falta un aspecto muy importante por revisar en este marco teórico, el cual es la teoría que permitirá la traducción. Específicamente, revisaremos dos cosas: (i) La teoría sobre el proceso de compilación de los lenguajes, pues esta nos ayudará a entender cómo el traductor obtiene la información del modelo a traducir, y también nos permitirá entender eventualmente cómo será el proceso de traducción. (ii) Dos maneras en que se traducen lenguajes de software.

Empecemos por la compilación de lenguajes de programación. Este proceso busca traducir las instrucciones en un lenguaje y convertirlas en lenguaje de máquina. El proceso empieza con el análisis léxico, el cual consiste en leer el programa que se piensa compilar y agruparlo en *tokens*. Un *token* es un objeto que tiene un tipo y un valor. Los *tokens* sirven para representar operaciones, palabras clave, u otros componentes sintácticos del lenguaje que se está compilando. Otra tarea que se realiza en esta etapa es la de eliminar elementos que no tengan significado (y que generalmente solo se usan para facilitar la lectura de un programa), tales como, espacios en blanco extra, saltos de línea innecesarios, comentarios, etc. Adicional a esto se hacen análisis sintácticos y semánticos (bajo las reglas del lenguaje en que esté escrito, tales como la asociatividad y precedencia de las operaciones). Para poder hacer esto se lleva a cabo un proceso de *parsing* (análisis gramatical), el cual permite descifrar la estructura de los *tokens* obtenidos [Wik20].

Para poder hacer lo anterior realidad, se hacen uso de estructuras intermedias que puedan representar el programa que se está compilando, y la estructura más famosa para lograr esto, y la que realmente nos interesa de todo este proceso es: El árbol de sintaxis abstracta (AST). Este tipo de árboles son las estructuras que un *parser* genera para almacenar toda la información necesaria que permite representar el programa que se está compilando. Llegados a este punto, debemos detenernos aquí y concentrarnos en esta estructura. La razón de esto es que (como lo observaremos eventualmente), esta es la estructura a la que Rodin permite acceder para obtener la información de los modelos que estén definidos en la plataforma, y es de estas estructuras de donde obtendremos los datos suficientes que luego de un proceso de traducción permitirán generar el código que nuestro traductor quiere producir.

Un AST es un árbol que representa la estructura sintáctica de un programa de acuerdo con las reglas de gramática pertenecientes al lenguaje en que está escrito [Spi15]. Los AST almacenan la

información de un programa de manera muy compacta en comparación a otras estructuras que se usan en el proceso de compilación, como se puede ver en la Figura 1.3. Por ejemplo, y como se ve en esa Figura, en un AST se pueden omitir paréntesis, ya que la manera en que se recorren estos árboles permite implícitamente conocer la precedencia y asociatividad de todas las operaciones (pues son construidos siguiendo justamente las reglas gramaticales del lenguaje). Para recorrer un AST se usa el patrón de diseño *Visitor*. Afortunadamente, Rodin ofrece su propia interfaz *Visitor* para que los usuarios la implementen y puedan recorrer los AST de los modelos que se hayan definido en la plataforma.

El último tema pendiente por tratar es el de estudiar dos formas de traducir lenguajes de software.

La primera, es leer el código fuente del lenguaje a traducir como texto plano, interpretarlo, procesarlo, y aplicar un algoritmo de traducción para generar código. Este proceso es muy complejo y riesgoso ya que requiere hacer un proceso de *parsing* manual. Al ser manual, es muy propenso a tener errores, y por lo tanto, para asegurar su buen funcionamiento, se tiende a sacrificar algo de flexibilidad en el traductor a través de limitar la manera en que se escribe el código fuente a traducir (si se especifica un formato para los programas a traducir, se reduce la complejidad de los mismos y se hace más factible analizar su estructura, lo que permite que el generar una traducción se vuelva una tarea más sencilla).

La otra manera para traducir un software es interpretando directamente el AST que lo representa si se tiene acceso a dicho árbol, afortunadamente, este es el caso de Event-B en Rodin. Hacer uso de este árbol ahorra la tarea del proceso de *parsing*, y no sólo eso, dicho *parsing* no se podría hacer de mejor manera, puesto que es el AST “oficial” construido bajo las reglas gramaticales del lenguaje del que estemos hablando. Además, el usar el AST ayuda a que el formato en que se escriban los programas a traducir sea completamente flexible (pues no hay la necesidad de restringirlo con el fin de facilitar un *parsing* manual). Eventualmente veremos en el capítulo de *Diseño* que hacer uso de los AST es la estrategia que se toma para nuestro traductor, y además veremos que hacer esto de esta manera traerá más ventajas de las que hemos mencionado hasta el momento.

### 1.4.3. Trabajos relacionados

Existen muchos traductores vinculados a Event-B, pero no todos son de la misma naturaleza al que vamos a construir en este trabajo. En Event-B, además de traductores a lenguajes de programación tradicionales, podemos encontrar también hacia otro tipo de lenguajes: como UML, Dafny, SQL, o también traductores que hacen el proceso contrario (parten de un lenguaje de programación y se obtiene un modelo en Event-B). Claramente, a nosotros nos interesan son traductores de la misma naturaleza que el que vamos a construir. Afortunadamente, generadores de código desde Event-B a un lenguaje de programación tradicional hay bastantes. Tanto es así que solo nombraremos aquellos que son oficiales (que se encuentran en la página oficial de Event-B), en dicha página podemos encontrar traductores a: Java (+JML), C, C++, y C# [WEB15]. La ventaja de estudiar

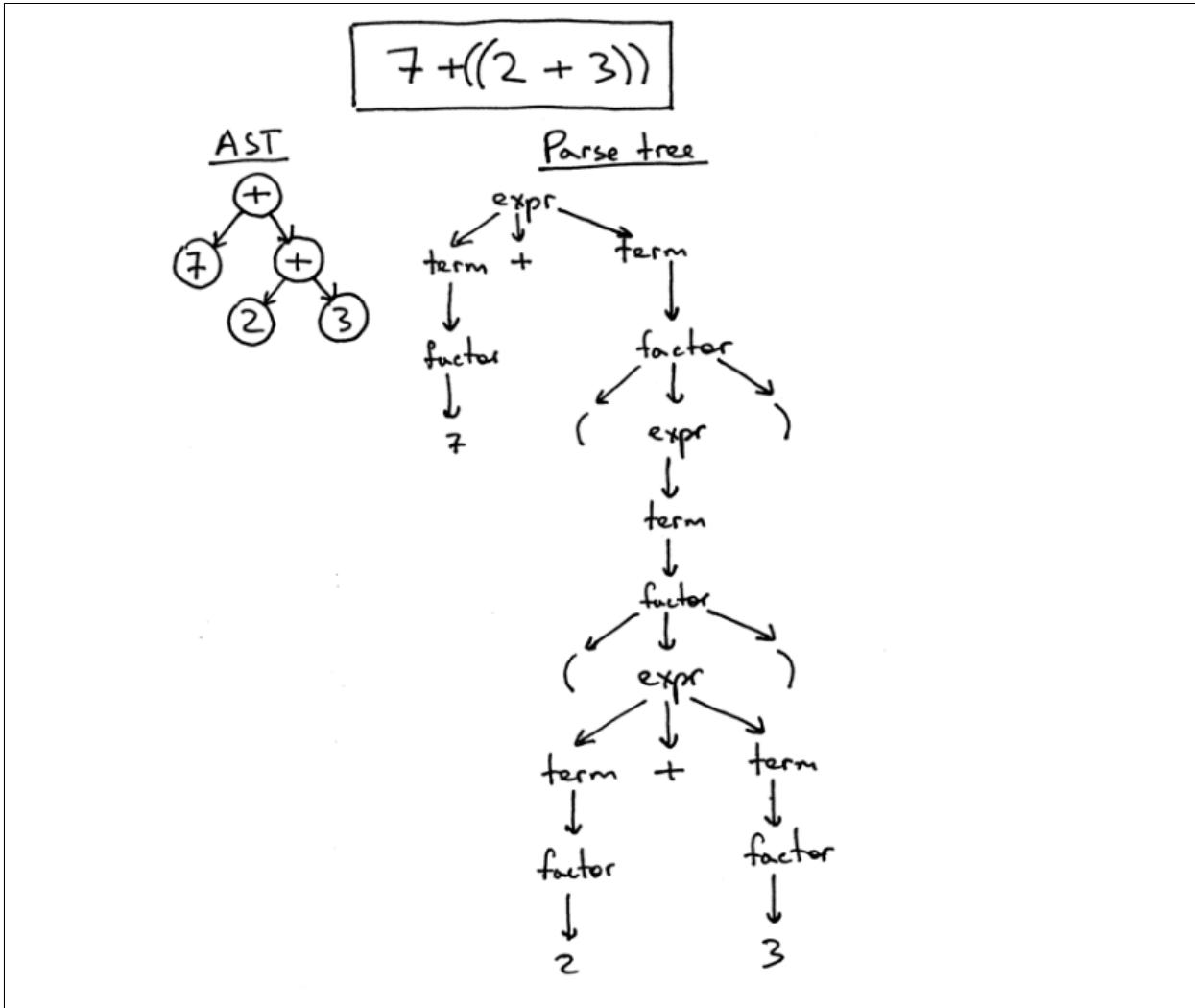


Figura 1.3: Ejemplo de un AST que se construye a partir de la expresión:  $7 + ((2+3))$ , en comparación a un árbol de parsing construido por la misma expresión. Tomado del blog “Let’s Build A Simple Interpreter” [Spi15].

trabajos relacionados es que nos permite ver que provechos o limitaciones se obtienen según la estrategia de traducción que tomen.

Lo primero que veremos es que todos los traductores tienen sus propias limitaciones, pues hasta el momento, no existe un generador de código que sea capaz de traducir cualquier modelo que se pueda definir en Event-B sin tener que exigirle al usuario que realice algunos ajustes manuales al código generado. Es más, los traductores que frecuentemente se usan en la industria son aquellos que exigen un sub-lenguaje específico del modelo, de modo que cuando se tiene un modelo concreto que se acerca mucho a lo que sería el programa que lo represente y ejecute, su traducción pueda hacerse de manera muy eficiente.

Empecemos por uno de los traductores oficiales a C/C++ (cuyo nombre oficial es *EB2C*). Este traductor tiene una cantidad limitada de operaciones que puede traducir, como se puede ver en la Figura 1.4. En la tabla podemos apreciar también algunos elementos en Event-B que no se traducen debido a que son operaciones indeterministas, específicamente, los cuantificadores y operaciones indeterministas como el “Becomes Such That” y el “Choice from set” (5ta y 6ta fila de la tabla, detalles sobre estas operaciones los veremos cuando las vayamos a traducir) [SM11]. Vemos efectivamente que, como lo mencionamos, los traductores oficiales tienen problemas al traducir algunas operaciones (especialmente cuando están relacionadas al infinito o a los indeterminismos).

Hay otro traductor que mencionaremos antes de pasar al más importante, y es otro traductor oficial de Event-B a C (cuyo nombre oficial es *B2C*). Este traductor también solo puede trabajar con un subconjunto de operaciones de Event-B, y, además, para poder uso de él, se deben tomar varios pasos intermedios en el que el usuario altere el código generado para poder ejecutarlo [WEB10]. Es importante mencionar aquí que esto será algo que nuestro traductor buscará evitar en lo máximo, pues se desea que el proceso sea lo más automático posible, minimizando la intervención del usuario (que como lo vimos, ayuda a que el proceso sea menos largo y menos propenso a errores).

El traductor de Event-B a Java (+JML) es seguramente el traductor más completo que hay en este ámbito, y, por esta razón, es la referencia más importante para el traductor que construiremos en este trabajo. Empecemos por el hecho de que este traductor hace uso del AST generado por Rodin, por lo tanto, esto le permite traducir cualquier elemento de un modelo en Event-B para el cual haya definido una regla de traducción. Adicional a eso, esto le permite disminuir la necesidad de limitar la manera en que se deben escribir los modelos en Rodin que se quieran traducir, pues toda la compilación y *parsing* del código será trabajo de Rodin.

El traductor de Event-B a Java tiene muchas otras ventajas: (i) Permite la traducción de la gran mayoría de operaciones de Event-B. (ii) Hace uso de JML (lenguaje de modelado oficial para el diseño por contrato en Java), lo que ayuda a que su correcto funcionamiento sea mucho más confiable. (iii) Se le han aplicado varios estudios formales para buscar demostrar la correctitud de la traducción (lo cual se ha logrado hacer para algunos subconjuntos de operaciones). (iv) Este

| Event-B  | 'C' & 'C++' Language  | Comment                                 |
|--|---|---|
| n..m   | int   | Intger type                             |
| $x \in Y$  | Y x;  | Scaler declaration                      |
| $x \in tl.int16$   | int x;  | 'C' & 'C++' Context declaration         |
| $x \in n..m \rightarrow Y$   | Y x [m+1];  | Array declaration                       |
| $x : \in Y$  | /* No Action */   | Indeterminate initialization            |
| $x :   Y$  | /* No Action */   | Indeterminate initialization            |
| $x = y$  | if(x==y) {  | Conditional                             |
| $x \neq y$   | if(x!=y) {  | Conditional                             |
| $x < y$  | if(x<y) {   | Conditional                             |
| $x \leq y$   | if(x<=y) {  | Conditional                             |
| $x > y$  | if(x>y) {   | Conditional                             |
| $x \geq y$   | if(x>=y) {  | Conditional                             |
| $(x>y) \wedge (x \geq z)$  | if ((x>y) && (x>=z) {   | Conditional                             |
| $(x>y) \vee (x \geq z)$  | 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                   |
| $x := y \div z$  | x = y / z;  | Arithmetic assignment                   |
| $x := F(y)$  | x = F(y);   | Function assignment                     |
| $a := F(x \rightarrow y)$  | a = F(x, y);  | Function assignment                     |
| $x := a(y)$  | x = a(y);   | Array assignment                        |
| $x := y$   | x = y;  | Scalar action                           |
| $a := a \Leftarrow \{x \rightarrow y\}$                                | a(x)=y;   | Array action                            |
| $a := a \Leftarrow \{x \rightarrow y\} \Leftarrow \{i \rightarrow j\}$ | a(x)=y; a(i)=j;   | Array action                            |
| $X \Rightarrow Y$  | if(!X    Y){  | Logical Implication                     |
| $X \Leftrightarrow Y$  | if((!X    Y) && (!Y    X)){   | Logical Equivalence                     |
| $\neg x < y$   | if(!(x<y)){   | Logical not                             |
| $x \in \mathbb{N}$   | unsigned long int x   | Natural numbers                         |
| $x \in \mathbb{Z}$   | signed long int x   | Integer numbers                         |
| $\forall$  | /* No Action */   | Quantifier                              |
| $\exists$  | /* No Action */   | Quantifier                              |
| Sets   | Supported by C++  | Using STL library based Sets operations |
| Set1   | set <data type> Set1  | STL library                             |
| $\cup$   | set_union(...)  | STL library                             |
| $\cap$   | set_intersection(...)   | STL library                             |
| $-$  | set_difference(...)   | STL library                             |
| $\subset$  | if (includes(...)){   | STL library                             |
| $\subseteq$  | if (includes(...))    equal (...)){   | STL library                             |
| $\not\subset$  | if (!(includes(...))){  | STL library                             |
| $\not\subseteq$  | if (!(includes(...))    equal (...))){  | STL library                             |
| $fun \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$          | long int fun(unsigned long int arg1,<br>unsigned long int arg2)<br>{<br>//TODO: Add your Code<br>return;<br>} | Function Definition                     |

Table 1: Event-B to C &amp; C++ translation syntax

Figura 1.4: *Operaciones soportadas por el traductor EB2C. Tomado de “EB2ALL - The Event-B to C, C++, Java and C# Code Generator” [SM11].*

traductor ofrece un método para que el usuario pueda ejecutar el código generado automáticamente, entre muchas otras ventajas.

El traductor a Java también se diferencia de los demás al ser el primero en permitir la traducción de máquinas refinadas, un aspecto que como vimos es supremamente importante a la hora de trabajar en Event-B, al ser esencial en la manera que se recomienda modelar sistemas (de software o hardware) en Rodin [CWR<sup>+</sup>11].

Este traductor también tiene algunas limitaciones que quedaron pendientes por solucionarse pues es un trabajo en proceso, pero de todos modos veamos cuáles son pues sería interesante ver si podemos lidiar con algunas de ellas en el traductor que construiremos en este trabajo: (i) Los “Carrier Sets” se traducen como conjuntos de enteros por defecto pues no pertenecen a ninguno de los tipos nativos de Event-B, sin embargo, sería más preciso traducirlo como un tipo genérico y nuevo. (ii) Los conjuntos por compresión no están soportados. La razón de lo anterior es que es difícil expresar el conjunto que representan (que por ejemplo puede ser muy grande o incluso de infinitos elementos). (iii) Los cuantificadores no se traducen a Java (sólo a JML), ya que la mayoría de los cuantificadores en Rodin exigen verificar predicados para valores pertenecientes a un conjunto infinito. (iv) Finalmente, en este traductor no se pueden traducir extensiones de contextos.

Para terminar, veamos brevemente un ejemplo del traductor a Java en funcionamiento en las Figuras 1.5 y 1.6, siendo la primera el código en Event-B a traducir, y la segunda una parte del código generado correspondiente. Podemos observar cómo una máquina llamada SOCIAL\_NETWORK se transforma en una clase abstracta del mismo nombre en Java. Se puede ver cómo las 5 invariantes del modelo se convierten en sola invariante en Java (que representa la conjunción de las 5 invariantes del modelo), y para su definición se usa JML que permite el diseño por contrato (comandos precedidos de /\*@ y que terminan en \*/). JML también permite hacer las verificaciones de los tipos de datos [CWR<sup>+</sup>11].

```

MACHINE SOCIAL_NETWORK
SETS
  PERSON; RAWCONTENT
VARIABLES
  person, rawcontent, content
INVARIANT
  person ⊆ PERSON ∧
  rawcontent ⊆ RAWCONTENT ∧
  content ∈ person ↔ rawcontent ∧
  dom(content) = person ∧
  ran(content) = rawcontent
INITIALISATION
  person := ∅ ||
  rawcontent := ∅ ||
  content := ∅
OPERATIONS
  transmit_rc(ow, rc, pe) ^=
PRE
  rc ∈ rawcontent ∧
  ow ∈ person ∧
  pe ∈ person ∧
  ow ≠ pe ∧ pe ↪ rc ∉ content
THEN
  ANY prs WHERE prs ⊆ person
  THEN
    content := content ∪ {pe ↪ rc} ∪ prs × {rc}
  END
  END
END

```

Figura 1.5: Código de muestra a traducir por el generador de código de Event-B a Java. Tomado de “Translating B machines to JML specifications” [CWR<sup>+</sup>11].

```

import org.jmlspecs.models.*;
public abstract class SOCIAL_NETWORK {
    //@ public final ghost JMLEqualsSet<Integer> PERSON;
    //@ public final ghost JMLEqualsSet<Integer> RAWCONTENT;
    //@ public model JMLEqualsSet<Integer> person;
    //@ public model JMLEqualsSet<Integer> rawcontent;
    //@ public model JMLEqualsToEqualsRelation<Integer, Integer>
        content;

    /*@ public invariant person.isSubset(PERSON)
       && rawcontent.isSubset(RAWCONTENT)
       && (new Relation<Integer, Integer>
           (person, rawcontent)).has(content)
       && content.domain().equals(person)
       && content.range().equals(rawcontent);*/

    /*@ public initially person.isEmpty() &&
       rawcontent.isEmpty() && content.isEmpty();*/

    /*@ public normal_behavior
       requires rawcontent.has(rc) && person.has(ow)
           && person.has(pe) && !ow.equals(pe)
           && !content.has(ModelUtils.maplet(pe,rc));
       assignable content;
       ensures (\exists JMLEqualsSet<Integer> prs;
           \old(prs.isSubset(person));
           content.equals(\old(ModelUtils.toRel(
               content.union(ModelUtils.toRel(
                   ModelUtils.maplet(pe,rc))))))
           .union(ModelUtils.cartesian(prs,
               ModelUtils.toSet(rc))));*/
    also public exceptional_behavior
    requires !(rawcontent.has(rc) && person.has(ow)
              && person.has(pe) && !ow.equals(pe)
              && !content.has(ModelUtils.maplet(pe,rc)));
    assignable \nothing; signals (Exception) true;*/
    public abstract void transmit_rc(Integer rc, Integer ow,
                                    Integer pe);
}

```

Figura 1.6: Código de muestra generado por el traductor de Event-B a Java a partir del modelo que se ve en la Figura 1.5. Tomado de “Translating B machines to JML specifications” [CWR<sup>+</sup>11].

# CAPÍTULO 2

# Diseño

---

En este capítulo se explicarán todas las decisiones generales referentes al diseño del traductor, y las razones detrás de cada decisión. En los capítulos posteriores veremos los detalles de implementación de cada componente del traductor, los cuales estarán guiados por las decisiones que se tomen en este capítulo.

## 2.1. Diseño por Contrato

El primer tema que vamos a tratar en este capítulo es el diseño por contrato. Como vimos en el marco teórico, el diseño por contrato es una rama oficial del diseño de software que se asemeja a los métodos formales que se emplean en Event-B (se utilizan invariantes, precondiciones, postcondiciones, etc.). Al ser una rama oficial, los lenguajes tradicionales tienden a tener librerías que facilitan programar bajo dicho paradigma. Por ejemplo, y como vimos, el traductor a Java usa JML, que es un lenguaje de modelado incluido en Java para el diseño por contrato.

Para nuestro traductor, lo que se hizo fue buscar distintas librerías o alternativas que ofreciera Python (oficialmente o de código abierto) para el diseño por contrato. Se tuvieron en cuenta varias alternativas:

- La primera alternativa era usar una de las siguientes librerías: *PyContracts* de Andrea Censi, PEP 316 o Python-contracts 0.1.4. Desafortunadamente ninguna de estas librerías tenía las suficientes herramientas para representar invariantes, axiomas, o precondiciones de Event-B, por ejemplo, porque estaban limitadas a una cantidad específica de operaciones a la hora de evaluar las invariantes, mientras que en Event-B se permite una cantidad muy variada de predicados.
- La segunda alternativa, la cual por mucho tiempo fue la que se pensaba usar, era la librería de *iContract*. Esta librería tiene soporte para invariantes, axiomas, precondiciones, postcondiciones, y particularmente, para herencia de invariantes a clases que fueran hijas de superclases que estuvieran haciendo uso de *iContract*, lo cual, se ajustaba perfecto para el refinamiento de máquinas (aspecto que estudiaremos posteriormente). Sin embargo, había un pequeño inconveniente, los refinamientos en Event-B generan un efecto que en inglés se conoce como “Guard Strengthening”, que básicamente ayuda a asegurar que las máquinas abstractas puedan simular el comportamiento de las máquinas concretas. Mientras tanto, la librería de *iContract* hace todo lo contrario, denominado “Guard Weakining”. Esto último implicaría

que las restricciones de las guardas en un refinamiento se cumplirían si las guardas de la máquina abstracta ¡o! las de la concreta se cumplen, algo contradictorio con el paradigma que se maneja en Event-B.

- La tercera y última alternativa es hacer el diseño por contrato manualmente usando instrucciones nativas de Python que permitan evaluar expresiones que representen predicados de modelos de Event-B (como invariantes o axiomas), y en caso de que ocurra una situación en que se viole alguno de esos predicados, se puede detener la ejecución del programa generando una excepción. Para ello se pueden usar las instrucciones: *assert* o *raise*. Esta es la alternativa por la cual se optó, pero es importante mencionar que las otras alternativas sirvieron de inspiración, pues su estudio permitió ver maneras en que se diseña por contrato en Python. No sobra mencionar tampoco que las 3 alternativas realizan un chequeo de *contratos* durante la ejecución del programa, no estáticamente como en Event-B.

Ya que el diseño por contrato se realizará manualmente, veremos al llegar a las reglas de traducción cómo se lidió con cada aspecto de los modelos de Event-B. Sin embargo, podemos agregar desde ya que se incluyó una característica muy importante del diseño por contrato: la posibilidad de desactivar todas las verificaciones de correctitud (en otras palabras, la opción de ejecutar el código generado sin revisar que se mantengan los axiomas, guardas, invariantes, etc.). La razón de esto es que, si uno va a ejecutar el programa que representa el modelo en el mundo real, si ya se demostró que el modelo es correcto, no es necesario que en producción se sigan evaluando los *contratos*. En producción se hace más importante que el código sea lo más eficiente posible, y los *contratos* (al menos aquellos que exijan muchos cálculos computacionales) se dejan únicamente para las etapas de desarrollo y depuración.

## 2.2. Tipos de Datos

### 2.2.1. Mypy

Como ya lo mencionamos en el marco teórico, Python es un lenguaje dinámicamente (y fuertemente) tipado. Un efecto de lo anterior es que muchos de los códigos construidos en Python son difíciles de leer cuando lo que se quiere es saber directamente qué tipo de dato tiene cada variable. Más aún, los tipos de datos de las variables pueden cambiar durante la ejecución, lo cual hace la situación mucho más difícil de digerir. Aunque claro, aquí conviene precisar que esto técnicamente no es un problema para el código generado, puesto que estamos asumiendo que los modelos que se traducen son correctos y que Rodin no identifica ningún tipo de error en el modelo (por ejemplo, de tipado), por lo tanto, si la traducción es correcta, el usuario no debería tener la necesidad de preocuparse por revisar el código para verificar que ninguna variable sea de un tipo de dato incorrecto en un momento dado de la ejecución y se genere una excepción.

Sin embargo, siendo Event-B un lenguaje más estricto con el tipado de datos, y siendo uno de los objetivos de este trabajo el resaltar la relación entre los lenguajes procesados y hacer la traducción

lo más entendible posible, podríamos pensar en alguna manera de mostrar la relación de los tipos de datos que se generan en Python en comparación a los de Event-B. ¿Pero, cómo? La primera opción que se le vendría a uno a la cabeza es hacer Python estáticamente tipado, pero bueno, el solo lograr eso sería un trabajo mucho más extenso que el construir este traductor. Afortunadamente, hay una segunda opción, que además sólo es posible gracias a que estamos trabajando con la versión de Python 3.7. Desde Python 3.5 se permite algo que se denomina *type hints* (que funciona con la librería oficial *Typing*) [Fou20]. Básicamente, esta librería permite indicar los tipos de datos que se supone debería representar cualquier elemento en Python al que se le pueda vincular un tipo de dato. Es importante aclarar que esta funcionalidad de Python no obliga a que la variable sea del tipo de dato que se le indique, ya que a la hora de ejecutar el código, todos los *type hints* son ignorados y se permiten cosas como asignarle una cadena a una variable que ya tenía un valor entero asignado, sin ningún problema. En otras palabras, los *type hints* no quitan el hecho de que Python sea dinámicamente tipado. Sin embargo, esta nueva funcionalidad permite documentar el código, y de hecho nació justamente por algo que se le criticaba mucho a Python en códigos de software a gran escala (como los que se suelen construir con métodos formales): “Un lenguaje muy fácil de escribir, pero muy difícil de leer y verificar”.

Podríamos conformarnos únicamente con aprovechar los “*type hints*” para documentar el código y hacer evidenciar la relación entre el tipado de datos del modelo con el código generado, pero no nos detendremos ahí, ya que también podemos hacer uso de softwares terceros para hacer un chequeo de tipado estático (*static type checking*) del código generado, lo cual permite revisar que el tipado a cualquier componente se cumpla (de manera estática). Lo anterior quiere decir que a pesar de que el código en el momento de ejecutarse será dinámicamente tipado, se puede ser prudente y revisar que los tipos de datos se respeten antes de ejecutar el código al hacer un chequeo de tipado estático del código. Lo anterior, además, es particularmente beneficioso cuando la idea del usuario es extender el código generado directamente en Python, donde si no hacíamos algo al respecto, no habría manera de garantizar de que se respete el tipado estático de Event-B con el código adicionado. Para lograr esta gran tarea existe *Mypy*, la cual es la herramienta oficial de Python (todavía en desarrollo, pero que ya es lo suficientemente robusta) relativa al chequeo de tipado estático de datos.

Para resumir lo dicho hasta el momento: El traductor estará pensado para hacer uso de los *type hints* de Python, lo cual ayudará a hacer el código generado mucho más legible al lector, y ayudará a resaltar la relación que hay entre el código generado y el modelo formal que representa, buscando así superar la barrera de que Python sea dinámicamente tipado. Adicionalmente, se buscará en lo posible que la implementación del traductor permita que el código generado sea correcto frente a un chequeo de tipado estático que realice *Mypy*, lo cual, también será útil para aquel usuario que quiera extender el código generado y desee seguir teniendo una manera de asegurar un tipado de datos similar al de Event-B.

La razón por la que no daremos un soporte completo al chequeo estático del código generado se divide en 3 ideas: (i) *Mypy*, a pesar de ser la herramienta oficial, todavía está en desarrollo y tiene algunos bugs [Lic20]. (ii) Algunas funcionalidades de *Mypy* requieren de bastante alteración del código, lo cual afecta el código generado (por ejemplo, lo hace más *Verbose*), y no queremos sacrificar nuestro objetivo principal por cumplir completamente objetivos opcionales. (iii) *Mypy* sigue algunos principios (como el Principio de Liskov) que generan retos técnicos para la traducción de algunos elementos de Event-B considerando algunas características particulares de nuestro generador de código (como veremos posteriormente); Afortunadamente, a lo largo de este documento veremos que son muy pocas situaciones en las que habrá limitaciones con el análisis de tipado estático que se hace con esta herramienta a los códigos generados por este traductor.

Finalmente, no sobra mencionar que el traductor generará los archivos necesarios, importará librerías, y demás aspectos técnicos necesarios que permiten el correcto funcionamiento de *Mypy*. Además, hay que recordar que el chequeo de tipado estático no afectará el tiempo de ejecución de los códigos generados (a diferencia de los *contratos* que genera nuestro traductor) pues los *type hints* son ignorados por el compilador durante la ejecución (como se dijo, se comportan como comentarios en ejecución), ya que los *type hints* solo se tienen en cuenta externamente cuando se usan herramientas como *Mypy*.

### 2.2.2. Manejo de diferentes tipos de datos de Event-B

Event-B tiene los siguientes tipos de datos: Enteros, Booleanos, Tuplas, Conjuntos, Relaciones, y los generados por los “Carrier Sets” (ja los cuales nos referiremos ocasionalmente como *enumeraciones* de ahora en adelante!, ya veremos por qué).

El traducir de Event-B a Python implica el poder ejecutar las operaciones que se asocian a cada tipo de dato que existe en Event-B. Para realizar esta tarea también debemos tener en cuenta que nuestro objetivo es hacer el código generado lo más entendible posible, y para lograr esto, lo que se buscó al construir el traductor es que si hay un tipo de dato que se comporte exactamente igual en Python y en Event-B, mantendríamos la traducción lo más transparente posible a través del uso directo de los tipos nativos de Python. Afortunadamente, este será el caso de los enteros, booleanos, y las tuplas. Por ejemplo, si el tipo de dato de una constante en Event-B es un entero, en el código generado su tipo será también el tipo nativo de Python de los enteros (*int*).

Respecto a las *enumeraciones*, sabemos por el marco teórico que las *enumeraciones* son conjuntos diferidos que podrían verse básicamente como tipos de datos no nativos de Event-B (son creados a libertad por la persona que haya hecho el modelo en Rodin), de hecho, esto se puede confirmar cuando uno visita el AST de Rodin, donde cualquier variable o constante que tenga valores pertenecientes a dichas *enumeraciones* tendrá como dato afiliado la *enumeración* a la que pertenece. Afortunadamente, Mypy permite una solución sencilla a lo anterior, cuando uno define una clase en Python, el tipo de dato asociado a cualquier objeto perteneciente a esa clase será esa misma clase

(lo que generaría un efecto similar a lo que ocurre en Event-B con las *enumeraciones*, pues crear una clase es como crear un nuevo tipo de dato). Por lo tanto, si por cada *enumeración* creamos una clase, estaremos indirectamente creando un tipo de dato único asociado a dicha *enumeración*.

Ahora, la razón por la que nos referimos a los “Carrier Sets” como *enumeraciones* es porque en Event-B a los tipos de datos creados por los “Carrier Sets” solo le podemos definir sus valores directamente (no sus operaciones), siendo así, y asumiendo que dichos conjuntos son enumerables, las enumeraciones de los lenguajes de programación tradicionales sirven perfectamente para representarlos, ya que a fin de cuentas, las enumeraciones son también un conjunto de valores diferentes sin ninguna operación asociada a esos valores (por defecto). Para nuestro traductor, usaremos la librería *Enum*. Hay otras cualidades relativas a la traducción de *enumeraciones* que se explicarán posteriormente.

Los conjuntos y las relaciones (conjuntos de tuplas) son aspectos un poco más complicados de traducir, ya que en Event-B tienen bastantes operaciones que Python no tiene en sus conjuntos nativos. Una forma de traducir estos tipos de datos sería crear una clase que herede de los conjuntos nativos de Python y agregarle las operaciones que le falten (este es el método que se usa en el traductor a Java). Sin embargo, *Mypy* no se ajusta a esta decisión, puesto que varias funciones al heredarlas generan problemas de tipado que violan el Principio de Liskov (este principio exige que la firma de un método de la clase padre sea la misma en sus hijos). Específicamente, muchos métodos nativos de Python reciben como parámetro objetos de tipo *Object* (no sobra mencionar que prácticamente todas las clases en Python heredan de la clase *Object*), el cual es muy genérico para operaciones traducidas de Event-B. Por ejemplo, si hubiéramos procedido de esa manera, no hubiera sido posible especificar la función de pertenencia de un conjunto usando el método nativo de Python de pertenencia. La razón es que no es posible sobrescribir la firma de dicha función para especificar el tipo de dato del elemento que se manda como parámetro (es obligatorio dejarlo como *Object* para cumplir con el Principio de Liskov), y por lo tanto, *Mypy* permitiría verificar pertenencia de elementos de prácticamente cualquier tipo (independientemente del tipo de dato del conjunto con el que se está verificando la pertenencia), algo que no está permitido en Rodin.

Afortunadamente hay otra solución (la que se implementa en este traductor) que en el mundo del software se conoce como *Wrapping*. Entendamos lo que se hace con esta estrategia a través de la explicación de cómo se implementó la misma en los conjuntos y las relaciones. Para los conjuntos se crea una clase *PySet*, la cual como atributo privado tiene un conjunto nativo de Python. Básicamente lo que se hizo es construir un método por cada operación posible en Event-B, y la implementación de cada método lo que hará es hacer cambios al atributo privado que guarda el conjunto nativo. Por ejemplo, ver la Figura 2.1. Para las relaciones la situación es similar, se crea una clase llamada *PyRel*, la cual internamente tiene un conjunto de tuplas (pues en Event-B, una relación es un conjunto de tuplas, sólo que se le pueden aplicar más operaciones adicional a todas las operaciones que tienen los conjuntos, como obtener el dominio). Todos los métodos que representen operaciones en Event-B harán cambios internos a dicho atributo. *Mypy* no tiene ningún

```

379
380 |     def PyContains(self,element : T) -> bool:
381 |         #O(1)
382 |         #print('Test_PySet.PyContains')
383 |         return self._FiniteInclusion.__contains__(element)
384

```

Figura 2.1: *Ejemplo de Wrapping implementada en el Preludio, en el cual la operación de pertenencia de los conjuntos se logra usando las operaciones nativas de Python sobre atributos internos de la clase.*

inconveniente con esta manera de proceder, pues al no haber herencia sino envoltura, el Principio de Liskov no será un obstáculo.

## 2.3. Arquitectura

### 2.3.1. Diseño y Arquitectura General del Traductor

Lo primero que hay que mencionar es que nuestro traductor hará uso del AST al que permite acceder Rodin. En el marco teórico vimos que proceder de esta manera trae muchas ventajas, pero vamos a mencionar algunos beneficios adicionales que trae el leer el AST de Rodin (a diferencia de interpretar directamente el texto plano de los modelos):

- El primero, y uno de los mayores beneficios, es que el AST de Rodin deja obtener el tipo de dato inferido para cualquier variable, constante, o parámetro de evento en cualquier modelo que haya. Ahorrarnos esta tarea es de suma importancia, ya que el tipo inferido se calcula siguiendo las leyes gramaticales de Rodin, para ver un ejemplo recordemos nuevamente la Figura 2.2. En dicha imagen se puede ver por ejemplo una regla que indica que, si una variable pertenece a los naturales, su tipo serán los enteros, algo que nos tocaría procesar por nosotros mismos si no tuviéramos acceso al AST. Más aún, los axiomas o invariantes generalmente indican el tipo de dato de las constantes o variables de manera indirecta (por ejemplo, puede que haya un solo axioma que se refiera a una constante “x”, y que diga únicamente que  $x > 5$ , solo con esa información deberíamos inferir que su tipo será los enteros), situaciones así o más complicadas implicaría un proceso de interpretación bastante complicado. Afortunadamente, no tenemos que reinventar la rueda, y podemos hacer uso del AST de Rodin.
- Cuando se tienen operaciones muy complicadas, el AST tendrá la información de dichas expresiones o predicados completamente organizados en el árbol, ya se habrán compilado todas las leyes gramaticales de asociatividad y precedencia (se habrán procesado los paréntesis y orden de las operaciones).
- Hay operaciones que en texto plano se escriben de una manera pero se interpretan de otra,

|                            |                           |
|----------------------------|---------------------------|
| $E.expr-lit$               | $\tau$                    |
| integer, natural, natural1 | $\mathbb{P}(\mathbb{Z})$  |
| bool                       | $\mathbb{P}(\text{BOOL})$ |
| true, false                | BOOL                      |
| emptyset                   | $\mathbb{P}(\alpha)$      |

Figura 2.2: Tabla de cómo se infiere oficialmente el tipo de dato de algunas expresiones en Event-B. Por ejemplo, se puede apreciar que si tenemos el conjunto de los naturales, el tipo de dato serán los enteros. Tomado de “Rigorous Open Development Environment for Complex Systems” [MJRV05].

por ejemplo, si yo tengo  $f(3) = 5$ , lo que Rodin realmente interpreta es una operación de *Overriding* a la relación-función  $f$ . Con el acceso al AST podemos ahorrarnos este cálculo, y también nos ahorraremos la confusión de si tenemos  $f(3)$  a la derecha de una asignación o en un predicado, pues en esas situaciones nos estamos refiriendo es al valor de la función evaluada en 3, y no a una operación de *Overriding*.

- Podremos acceder a cualquier información que necesitemos del modelo, por ejemplo, referente a las máquinas, podemos saber cuáles son sus eventos, cuáles son las características de dichos eventos, cuáles son las variables y sus tipos, saber si la máquina está refinando una máquina abstracta (y obtener la referencia a dicha máquina), etc.

Ahora veamos de manera general cómo es que funciona nuestro traductor, ver Figura 2.3. Básicamente, lo primero que hacemos es obtener los contextos y las máquinas de los modelos con una clase que llamamos RodinHandler, la cual hace uso de métodos que proveen las librerías de Rodin. Luego usamos la clase *Visitor* que también provee Rodin para recorrer el AST que ofrece Rodin para acceder a las componentes de los modelos organizadamente. Con la información que obtenemos del AST de Rodin generamos nuestra propia versión de AST (a la que llamamos PyAST), donde almacenamos y organizamos la información que nos interesa, y que usaremos para generar nuestro código Python. Cuando se tiene el PyAST armado, el OutputHandler lee los componentes del PyAST para generar código Python. El código generado va a la carpeta de salida (que se ubicará

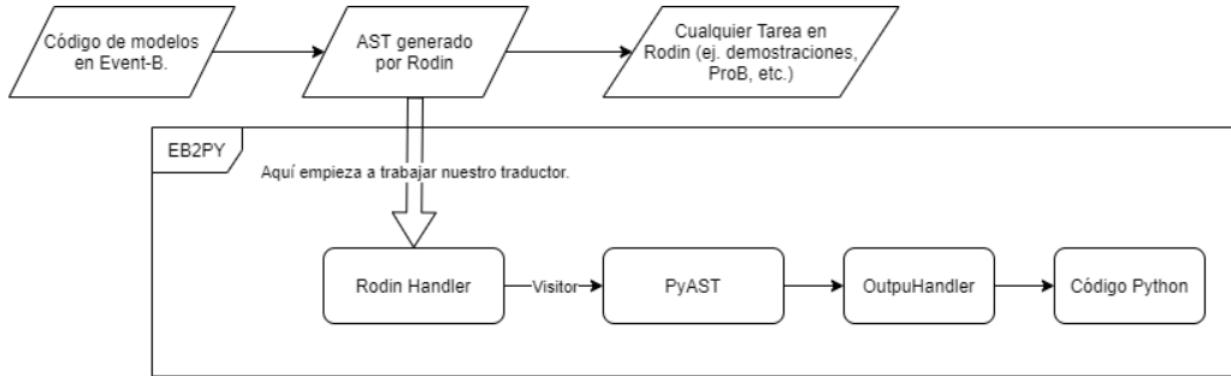


Figura 2.3: *Proceso de traducción de nuestro generador de código (EB2PY).*

en el mismo directorio del espacio de trabajo de la instancia de Rodin donde estén los modelos a traducir), donde se generará una carpeta por cada proyecto que se quiera traducir. En cada carpeta habrá un archivo por cada contexto y cada máquina que se haya traducido.

La gran ventaja del proceso de traducción que acabamos de mencionar es que se modulariza la extracción y organización de datos obtenidos del AST de Rodin, separándola de la etapa de generación de código en el OutputHandler. Esto fue supremamente útil a la hora de desarrollar el código por las siguientes razones: (i) Se obtienen las ventajas innatas que conlleva el hacer uso de la buena práctica de modularización: el código es mucho más mantenible y se facilita su desarrollo. (ii) El proceso se puede hacer más eficiente computacionalmente, por ejemplo, volviendo a la extracción de información del PyAST de Rodin: si el traductor está generando código de las constantes de un contexto con la información que obtuvo del AST de Rodin, luego de extraer las constantes del AST de Rodin y organizarlas en el PyAST (específicamente, en una subclase llamada PyAST\_Context), leerá las constantes del PyAST\_Context que esté traduciendo en el momento. Posteriormente, si vuelve a necesitarlas (por ejemplo, en un contexto concreto que extienda el contexto del que estamos hablando), no necesitamos releer el AST de Rodin, sino que podemos visitar directamente nuestra información procesada y almacenada en el PyAST, ahorrando tiempo de ejecución.

Pero veamos más detalles de la arquitectura del PyAST. Básicamente, en el PyAST se hace un proceso de Organización/Procesamiento/Almacenamiento de la información necesaria para generar nuestra traducción, que se distribuye en las diferentes clases que componen un PyAST, las cuales se pueden observar en el diagrama de clases de nuestro PyAST en la Figura 2.4. En dicha imagen se puede ver todas las relaciones entre los diferentes componentes de nuestro PyAST. Por ejemplo, las variables (PyAST\_Variable) que hacen parte de una máquina (PyAST\_Machine) en Event-B, se representan en nuestro diagrama de clases con una relación de composición, pues efectivamente, eso representaría que las variables hacen parte de una máquina.

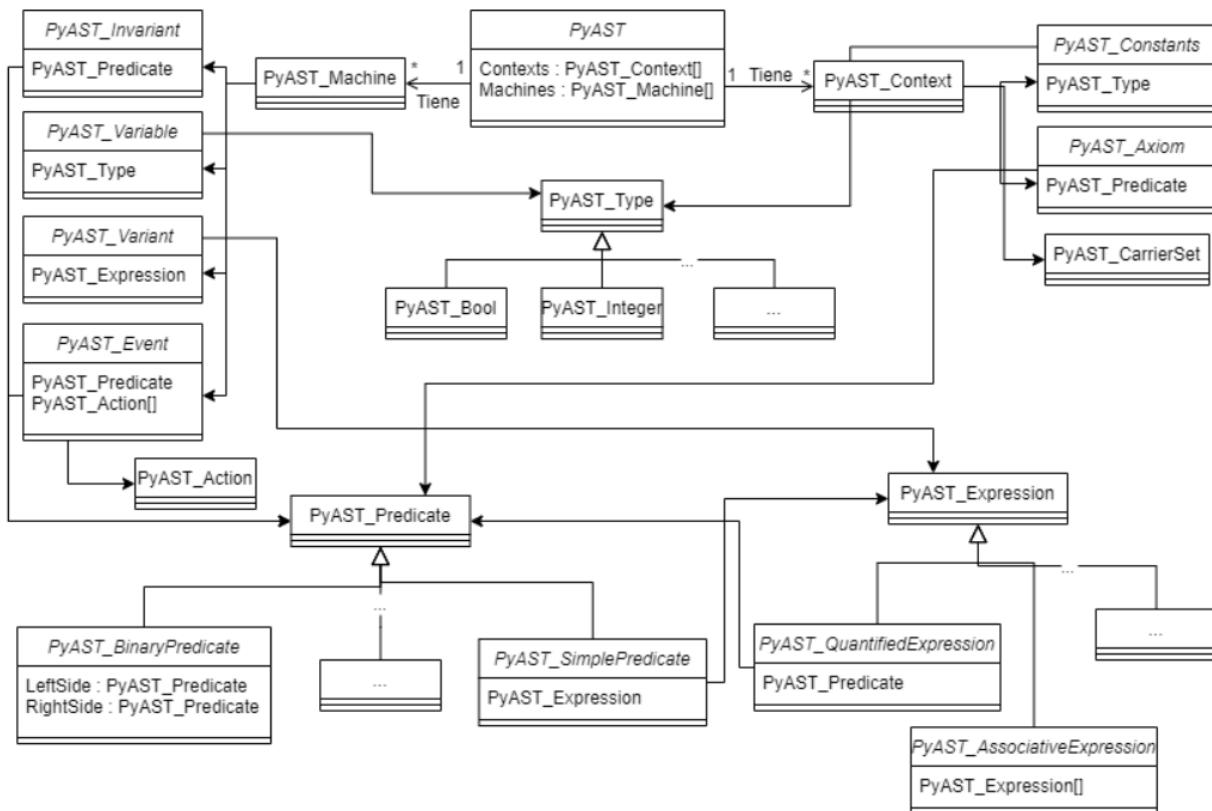


Figura 2.4: Diagrama de clases simplificado referente a la clase PyAST del traductor EB2PY.

```

1 package eventb2python.PyAST_Utils;
2
3 import eventb2python.PyAST_Types.PyAST_Type;
4
5 public class PyAST_Constant {
6
7     //Private Attributes
8
9     //Public Attributes
10    public String ConstantName;
11    public PyAST_Type ConstantType;
12
13    //OutputUtils
14    public String ConstantTypeTranslation;
15    public boolean ConstantTypeTranslated;
16
17    //Constructor
18    public PyAST_Constant(String constantName){
19        ConstantName = constantName;
20        ConstantType = new PyAST_Type();
21
22        ConstantTypeTranslation = "";
23        ConstantTypeTranslated = false;
24    }
25
26    //Methods
27
28 }

```

Figura 2.5: Fragmento de código de nuestro traductor, donde se aprecia la implementación de la clase que almacena la información necesaria de una constante de un contexto.

Apreciemos un ejemplo más del diseño de nuestro traductor, específicamente, veamos la implementación de cómo se almacena una constante en nuestro PyAST en la Figura 2.5. Para nuestras constantes solo nos interesa saber: (i) Su nombre (ConstantName). (ii) Su tipo (ConstantType), que de hecho podemos observar que es un PyAST\_Type, demostrando como se van relacionando las diferentes clases. (iii) Una cadena (ConstantTypeTranslation) que es literalmente el código Python que ya se incluye directamente en el código generado cuando se necesite. (iv) Y un booleano que nos indica si ya se hizo la traducción (para así no tener que repetir este proceso y así acceder directamente a la cadena de traducción); Cuando se extrae toda esta información del AST de Rodin y la guardamos en un PyAST\_Constant para cada constante, almacenamos este objeto en un PyAST\_Context (ya que las constantes hacen parte un contexto), y de esta manera vamos organizando/procesando/almacenando la información.

### 2.3.2. Diseño y Arquitectura General del Código Generado

Lo primero que necesitamos para poder generar el código Python que queremos es un espacio donde definamos cualquier dependencia o clase necesaria que permita ejecutar nuestro código, un ejemplo sencillo de esto es definir un lugar donde podamos declarar nuestra clase PySet para la representación de los conjuntos. Al archivo donde haremos esto nos referiremos a él como el Preludio.

Cualquier componente de un modelo (contexto o máquina) tendrá que importar este archivo.

En este archivo se definirán todas las clases, variables, operaciones, y demás elementos que sean comunes a cualquier contexto o máquina que queramos traducir, por ejemplo, la clase PySet. Específicamente, en el preludio encontraremos:

- Todo lo referente a los PySet (conjuntos) y PyRel (relaciones).
- Todo lo referente a conjuntos o relaciones especiales como: el conjunto de los naturales (PyNAT), el conjunto de los enteros (PyINT), el conjunto de los naturales sin el 0 (PyNAT1), la relación identidad (PyID), entre otros (que veremos en detalle uno por uno eventualmente).
- Todo lo referente a las familias de funciones y familias de relaciones (PyFamilies), tales como, las funciones totales, las funciones parciales, las biyecciones, las relaciones sobreyectivas, etc.
- Operaciones de Event-B como: Cuantificadores, Implicación Lógica, Equivalencia Lógica, entre otras. Cuando un contexto o máquina hagan uso de estas operaciones, se llamará al Preludio. Este apartado también incluye operaciones como la generación aleatoria de valores en el momento en que se anima una máquina, aspecto que veremos con más detalle posteriormente.
- Una clase llamada PyPrelude, donde se almacenan parámetros que el usuario puede manipular según su preferencia (por ejemplo, el parámetro que permite activar/desactivar por completo todo lo referente al diseño por contrato). También es en esta clase donde se definen específicamente las operaciones que se mencionaron en el punto anterior. En el Preludio se crea un objeto único de la clase PyPrelude llamado “P”, y por lo tanto, si queremos hacer uso de una operación del PyPrelude, se haría el llamado a dicho método llamando el objeto: *P.Foo(...)*.

No sobra mencionar que, si hacemos un chequeo de tipado estático de *Mypy* en el archivo del Preludio, *Mypy* dirá que todo el tipado del archivo es correcto.

Ahora, como vimos en el marco teórico, un modelo está compuesto de contextos y máquinas. Por cada contexto y máquina se va a generar un único archivo. Nuevamente, una gran ventaja de representar cada componente en su propia clase es el de modularizar y seguir buenas prácticas. Si un contexto concreto extiende un contexto abstracto, el archivo del contexto concreto importará el archivo del contexto abstracto. Además, se generará un comentario que indique que el contexto concreto está extendiendo la clase del contexto abstracto. Un ejemplo de esto se puede ver en la siguiente Figura 2.6. Similar ocurre con las máquinas. Se importan y generan comentarios según las dependencias de dicha máquina. Por ejemplo, ver Figura 2.7.

Otro aspecto general de nuestro generador de código es la traducción de nombres. A la gran mayoría de componentes que se traducen se les agregará una cadena que permita identificar con facilidad qué representa el componente traducido en Python, por ejemplo, si en Event-B tenemos

```
#Translation of Context: SimpleTwoWayCtxExt
#This context extends the following context: SimpleTwoWayCtx_class

class SimpleTwoWayCtxExt_class():


```

Figura 2.6: Fragmento de un código generado por nuestro traductor, en el cual podemos apreciar los comentarios que se generan cuando un contexto extiende otro.

```
#Translation of Machine: SquareRootMchRefRef
#This machine refines the following machine: SquareRootMchRef_class
#This machine sees the following context: SquareRootCtx_class

class SquareRootMchRefRef_class(SquareRootMchRef_class):
```

Figura 2.7: Fragmento de un código generado por nuestro traductor, en el cual podemos apreciar los comentarios que se generan al tener una máquina que ve un contexto y refina otra máquina.

una invariante que se llama “inv0”, en el código generado se define un método correspondiente que se llame “inv0\_checkInvariant”. Esto no solamente ayuda a hacer nuestro código generado más entendible, sino que además permite que el usuario no tenga qué preocuparse mucho por el nombramiento de componentes en el modelo. Por ejemplo, si no tuviéramos esta política en nuestro traductor, y alguien en un modelo a traducir llamara a un contexto PySet, la clase que represente este contexto en el código generado sobrescribiría la definición de PySet que se importa en el Preludio, lo cual sería supremamente problemático.

Otro aspecto para tener en cuenta en nuestro diseño es sobre cómo planeamos lidiar con los indeterminismos y el infinito, a los cuales les dedicaremos un capítulo entero para estudiar sus detalles. Pero desde el diseño debíamos empezar a planear una cosa muy importante: En Event-B hay múltiples situaciones en las cuales puede haber múltiples valores que permiten cumplir con la correctitud del sistema, en este momento nos interesan los siguientes casos: la inicialización de valores de constantes y el elegir un valor para un parámetro de un evento. Nos interesan estos casos desde el diseño pues una forma de solucionarlos es el de permitir al usuario ingresar los valores como parámetros de los métodos que permitan realizar dichas acciones, así se soluciona la indeterminación y es una solución que usan otros traductores como el de Java. Esto es lo más conveniente también pues: (i) en un modelo completamente refinado, no habrá ninguna indeterminación, y el usuario ya sabrá que valor es el que hará cumplir la correctitud del programa. (ii) le da flexibilidad al usuario de probar el código generado con los valores que él quiera; Más sin embargo, tanto ProB, como el traductor a Java (en varios casos), permiten animar máquinas o contextos generando automáticamente los valores en situaciones de indeterminismos (ProB usa SAT para lograr

lo anterior, y el traductor a Java lo hace ensayando valores aleatorios, tal y como lo haremos en nuestro traductor). Si queremos resaltar la relación entre Event-B (en Rodin) y el código generado, deberíamos permitir una funcionalidad similar.

Teniendo en cuenta lo anterior, en las 2 situaciones que estamos analizando, se va a dar la opción al usuario de especificar el valor indeterminado, pero si no lo hace, el Preludio empezará a generar valores que puedan potencialmente hacer cumplir la correctitud del programa y hacerlo avanzar. Llegados a este punto, podemos introducir algunos de los parámetros que se pueden alterar en el Preludio: `LOWMAXGENATTEMPTS` para definir la cantidad de veces que el Preludio tratará de generar valores para los parámetros en eventos y `HIGHMAXGENATTEMPTS` para la cantidad de veces en que el Preludio tratará de generar valores para inicializar constantes en un contexto. Se definen estos valores para que el código evite quedar en un ciclo infinito en caso no sea capaz de generar aleatoriamente el valor deseado.

Los elementos como los axiomas, invariantes, o guardas, pueden dar indicios sobre cuales valores son los correctos dentro del conjunto de posibilidades que permitirían resolver un indeterminismo, sin embargo, es bastante complicado generar una regla que permita deducir cuáles son los valores correctos basados únicamente en lo que digan las restricciones de los *contratos* (por ejemplo, ProB no es capaz de realizar esta tarea en algunos casos). La tarea anterior es complicada no solo conceptualmente sino también en lo técnico, una alternativa es usar los *SMT Solvers* (herramientas que se usan para solucionar estos problemas de decibilidad), pero en Python no encontramos una librería que pudiera cubrir todos los casos que necesitábamos.

Veamos un ejemplo de lo difícil que puede ser el resolver indeterminismos mediante el análisis de *contratos*: Supongamos que tenemos una constante “x” y un único axioma que se refiere a dicha constante que dice que “ $x=5$ ”. Sabemos claramente que el único valor que cumple ese predicado es asignarle el valor de 5 a “x” cuando vayamos a inicializar el contexto. Según eso, podríamos generar la regla: “cuando veamos un axioma de igualdad, le asignaremos el valor de la derecha a lo que haya a la izquierda siempre y cuando sea una constante”. Sin embargo, no es tan sencillo, ya que una igualdad puede aparecer en muchos tipos de predicados, por ejemplo, en una conjunción, donde la parte de la derecha sea la igualdad. Debido a la forma en que funciona el patrón de diseño Visitor, es difícil identificar si la igualdad está sola o no, y en general sería un gran reto técnico hacer estudios sobre los *contratos*. La única opción, como se mencionó, sería por ejemplo limitar los modelos que se pueden traducir a aquellos que usan igualdades en predicados de la forma: *constante = valor*. Pero eso sacrificaría mucho la flexibilidad de los modelos que soportaría traducir nuestro generador de código. Para resumir, nosotros tampoco buscaremos restringir la cantidad de modelos que se podrán traducir (mejor seguimos el ejemplo de ProB o el traductor a Java), siempre priorizaremos el no reducir la flexibilidad de los modelos que se pueden traducir sobre la resolución de los indeterminismos o tareas similares. Entre menos cambios por parte del usuario se requieran para que el código generado funcione, mejor. Es con este pensamiento como diseñaremos nuestro traductor.

Otro indeterminismo es el de saber cuál evento se ejecutará, como tal, en nuestro traductor el usuario puede elegir el que le plazca (como veremos eventualmente), mientras dicho evento esté habilitado por sus guardas (así también funciona en el traductor a Java). Pero nuevamente, queremos evidenciar una relación entre el modelo formal en Rodin y el código que generaremos, para ello debemos tener en cuenta que con ProB y el traductor a Java podemos animar una máquina como si fuera un programa, y para resolver el indeterminismo de la elección de eventos, lo hace eligiendo un evento al azar que esté habilitado, y lo ejecuta. Por eso, y como lo veremos eventualmente, nuestro traductor ofrecerá un método que permitirá hacer una tarea similar, en la cual se elegirá un evento al azar, y en caso de estar habilitado, se ejecutará e informará al usuario de los cambios.

En general, lo que queremos demostrar es que para todos los indeterminismos y componentes que traten con el infinito, buscaremos encontrar soluciones que permitan generar un código que pueda representar la mayor cantidad de componentes que uno puede tener en Event-B. Desde que se empezó a construir este traductor, siempre se tuvieron en cuenta estos aspectos, y próximamente veremos cómo implementamos soluciones para cada una de las problemáticas o limitaciones que se han expuesto a lo largo de este documento. En síntesis, buscaremos maximizar la cantidad de operaciones que se pueden traducir, minimizar las limitaciones de los modelos que se pueden traducir, y buscaremos evitar que el usuario tenga que hacer cambios al código generado en lo posible.

# CAPÍTULO 3

# Traducción

---

En este capítulo veremos la gran mayoría de descripciones y detalles relativos a la manera en que traduciremos cada componente de Event-B que nuestro generador de código soportará.

## 3.1. Contextos

Como lo mencionamos, por cada contexto que se traduzca, se generará un archivo. El nombre del archivo será el mismo nombre que tenía el contexto en Rodin, pero se le adiciona “`_ctx.py`” (ej. Si el contexto se llama `Foo` en Rodin, el archivo que lo traduzca se llamará `Foo_ctx.py`).

### 3.1.1. Dependencias

Lo primero de cualquier archivo relativo a un contexto que se genere son las dependencias (que estarán bajo el comentario “`#DEPENDENCIES`”). No importa qué modelo se esté traduciendo, siempre se importarán las mismas dependencias: (i) Librería `sys` para que `Mypy` funcione correctamente. (ii) El Preludio. (iii) Dependencias de la librería `typing` para los *type hints* necesarios. (iv) Dependencias de la librería `Enum` para el correcto funcionamiento de las *enumeraciones* (“Carrier Sets”); Lo único que puede variar en esta sección es, en caso de que el contexto en cuestión extienda otro contexto, que se importará el archivo donde se defina dicho contexto abstracto para poder hacer uso de él. Podemos observar lo que se acaba de describir en la Figura 3.1.

Lo siguiente en todo archivo es la definición de las clases que representan las *enumeraciones* de un contexto, las cuales estarán bajo el comentario “`#CarrierSet Types Declaration`”. Aquí viene una pregunta, ¿por qué definir las *enumeraciones* por fuera de la clase del contexto al que pertenecen? La respuesta es que lo que realmente se pretende al crear una clase externamente para cada *enumeración* es crear un tipo de dato global para `Mypy` (mientras que el conjunto de dicha *enumeración* sí lo definiremos como atributo del contexto). Si creamos la clase de la *enumeración* dentro de la clase del contexto, no se creará un tipo de dato global para dicha *enumeración* que pueda ser visto por una máquina o contexto que hagan uso de este archivo (porque lo extienden o porque la máquina lo “ve”), lo cual causaría problemas con el tipado (pues el tipo se definiría únicamente dentro del contexto abstracto). La cantidad de elementos que tendrá cada *enumeración* se tratará en el capítulo de *No Determinismo y el Infinito*. El Preludio se encargará de generar un conjunto que contenga todos los elementos del tipo de dato generado por la *enumeración* que acabos de definir (dentro de muy poco veremos por qué). Veamos un ejemplo en el que se resume lo que se habló en la Figura 3.2.

```

1 #DEPENDENCIES
2
3 #Allow Path Access to the Prelude's Directory
4
5 import sys
6 if not(".." in sys.path):
7     sys.path.append("..")
8
9 #Utilities Dependencies
10 from Py_Preludes import *
11
12 #Typing Dependencies
13 from typing import List
14
15 #Enum Dependencies
16 from enum import Enum,auto,unique
17
18 #Context Extension Dependencies
19 from SimpleTwoWayCtx_ctx import *
20

```

Figura 3.1: Dependencias al inicio de un archivo referente a un contexto. Aquí resalta la importación de un archivo de un contexto abstracto en la línea 19 del código generado.

```

18 #CarrierSet Types Declarations
19
20
21 @unique
22 class COLOURS_CS(Enum):
23
24     # CUSTOM USER CODE BEGIN: Increase or decrease the amount of finite elements as you wish!
25
26     COLOURS0 = auto()
27     COLOURS1 = auto()
28     COLOURS2 = auto()
29
30     # CUSTOM USER CODE END
31
32     #Include this new Type in the Prelude.
33     P.AddCarrierSet("COLOURS_CS",COLOURS_CS)
34

```

Figura 3.2: Definición de la clase que representa el tipo de la enumeración COLOURS en un código generado por nuestro traductor.

### 3.1.2. Constructor

Bueno, finalmente llegamos a la clase que representará al contexto que se está traduciendo.

Lo primero de lo que vamos a hablar es del constructor (método mágico `__init__` de Python). Aquí se realizan varias tareas: (i) Se crean dos atributos privados que estarán en todo contexto (no importa cuál sea) llamados “`Initialized_Context`” y “`Attributes_SetFlag`”. Para entender su propósito, debemos entender una cualidad de los contextos en Event-B: Una vez un contexto asigna valores a sus constantes o enumeraciones, este valor no debe cambiar bajo ninguna circunstancia en el futuro. Permitir un comportamiento similar es la tarea de los dos atributos que acabamos de mencionar (ya que desafortunadamente Python no tiene constantes nativas en su lenguaje), “`Initialized_Context`” es un booleano que permitirá al usuario saber si nuestro contexto ya fue inicializado o no, y “`Attributes_SetFlag`” es un booleano que se encarga de generar una excepción cuando haya un intento de sobrescribir una constante o enumeración de un contexto que ya haya asignado valores a sus atributos. Luego de esto viene la inicialización de cada enumeración (que se encuentra entre los comentarios “`#CarrierSets`” y “`#EndCarrierSets`”), esta tarea la hace el Preludio, y lo que hace es asignarle siempre un conjunto con todos los valores que existan en la clase que representa el tipo de dato correspondiente a la *enumeración* de la que se esté hablando. La razón de ello es que los “`Carrier Sets`” de Event-B implican la creación de un tipo de dato, pero como tal, para hacer uso de dichas *enumeraciones*, Event-B las representa como conjuntos afiliados a un contexto en Event-B.

Basados en lo anterior, para nuestro traductor se buscó generar un comportamiento similar, en el que generamos un tipo de dato (global, externo a la clase del contexto), pero posteriormente creamos un conjunto con todos sus elementos como atributo del contexto, de modo que podamos realmente hacer uso de ese tipo de dato. Trabajar así nos permite traducir aspectos de Event-B como el siguiente: En Event-B no es posible calcular la intersección de dos *enumeraciones* pues son de tipo de distinto. Afortunadamente, en el código traducido de Python ocurre lo mismo, pues *Mypy* identificaría un problema de tipos si se tratara de ejecutar dicha operación; Finalmente, en el `__init__` de la clase se declaran los atributos y tipos de las constantes (que se encuentra entre los comentarios “`#Constants`” y “`#EndConstants`”), y también, en caso de que el contexto del que se hable esté extendiendo un contexto abstracto, entonces se creará un atributo que guardará un objeto que represente el contexto abstracto para poder hacer uso de él (que se encuentra bajo el comentario “`#Context Extended Dependency Object`”). Para un ejemplo en el que se pueden apreciar algunas de las cosas que mencionamos con anterioridad, ver la Figura 3.3.

Antes de proceder con los otros componentes de un contexto traducido, hay un aspecto importante a mencionar de las constantes. Ya que nuestro traductor soportará la traducción de extensiones y refinamientos, cada vez que aparezca una constante, no será trivial la forma en que se llama a dicho atributo del contexto. La razón de esto es que cuando se lee el AST de Rodin, el AST no nos indica a qué contexto pertenece dicha constante. Por lo tanto nuestro traductor debe calcular las

```

77
78 ▼ class ParkingLotCtx_class():
79
80 ▼   def __init__(self) -> None:
81
82     #Context Utils
83     self.__Initialized_Context = False
84     self.__Attributes_SetFlag : bool = True
85
86     #CarrierSets
87     self.Car : PySet[Car_CS] = P.PyCarrierSetGet("Car_CS")
88     self.Door : PySet[Door_CS] = P.PyCarrierSetGet("Door_CS")
89     self.DoorState : PySet[DoorState_CS] = P.PyCarrierSetGet("DoorState_CS")
90     self.ParkingSpace : PySet[ParkingSpace_CS] = P.PyCarrierSetGet("ParkingSpace_CS")
91     #EndCarrierSets
92     self.__Attributes_SetFlag = False
93
94     #Constants
95     self.Closed : DoorState_CS
96     self.Closing : DoorState_CS
97     self.Entrance : Door_CS
98     self.Exit : Door_CS
99     self.Open : DoorState_CS
100    self.Opening : DoorState_CS
101    #EndConstants
102

```

Figura 3.3: Ejemplo del constructor de la clase de un contexto generado por nuestro traductor. Aquí resalta la declaración de los conjuntos de las enumeraciones, y la declaración de las constantes.

dependencias de dónde fue definida dicha constante (afortunadamente, la arquitectura que usamos con el PyAST facilitó esta tarea), y así, por ejemplo, si definimos una constante “x” en el contexto abstracto “ca”, y queremos hacer uso de dicha constante en un contexto concreto (el cual extiende a “ca”) que tiene una constante “y”, para calcular la expresión “x+y”, el traductor generará correctamente el código correspondiente: “self.ca\_get().x + self.y” (donde ca\_get es el método get para acceder al atributo privado que contiene el objeto del contexto abstracto “ca”).

### 3.1.3. Métodos

Luego del `__init__` vienen todos los métodos “set” y “get” de todas las constantes, enumeraciones, y demás componentes de un contexto. Para las constantes y las enumeraciones se usará el decorador `@property`, que es una manera sofisticada de Python que funciona similarmente a los atributos privados de la programación orientada a objetos.

Luego de los “set” y “get” vienen los métodos que representan los axiomas (a los que nos referiremos como los *AxiomChecks*). Cada axioma de un contexto en Event-B tendrá un método correspondiente con el mismo nombre, pero concatenándole la cadena “\_axiomCheck”. Estos métodos no tienen parámetros y retornan un booleano indicando si el axioma se está cumpliendo o no. Cada *AxiomCheck* será un método de una línea donde se evalúa la traducción del predicado que se haya definido en Event-B. Si el axioma es un teorema en Event-B, se genera un comentario indicándolo. Veamos un ejemplo de todo lo que se habló en la Figura 3.4.

```
#Axiom Check Methods

def ax0_axiomCheck(self) -> bool:
    return P.NAT1().PyContains(self.n)

def ax1_axiomCheck(self) -> bool:
    return P.NAT().PyPowerSet().PyContains(self.s)

def ax2_axiomCheck(self) -> bool:
    return PyFamilies(PyFamilyTypes.TotalFunctions, PySet({int_range for int_range in range(1, self.n+1)}), self.s).PyContains(self.f)

#End Axiom Check Methods
```

Figura 3.4: *Ejemplo de la traducción de los axiomas de un contexto.*

Luego de los axiomas vendrá siempre un método llamado *CheckAllAxioms*, que como su nombre lo indica en inglés, retorna un booleano diciendo si todos los axiomas se cumplieron (no requiere parámetros).

Posteriormente viene un método muy importante llamado *checkedInit*. Este método se encarga de inicializar los atributos de un contexto (y bloquearlo para que nadie cambie sus valores a futuro). Este valor recibe como parámetros valores para cada constante (y para un contexto abstracto en caso el contexto del que se está hablando extienda un contexto abstracto). Como se mencionó en el diseño, si no se especifican dichos parámetros, el código intentará generar valores aleatoriamente que hagan cumplir los axiomas. Al final de la ejecución del método se hace un llamado a *CheckAllAxioms*, si no se puede inicializar el contexto, se genera una excepción. No sobra recordar que con el Preludio se puede desactivar el diseño por contrato, o cambiar la cantidad de intentos para tratar de inicializar aleatoriamente el contexto.

Un contexto generado por nuestro traductor finaliza con los métodos mágicos *str* y *repr* de Python. El propósito de estos métodos es (como lo sugieren los objetivos de este proyecto) el de ofrecer herramientas para que el usuario pueda hacer seguimiento a la ejecución del código. Para aquellos que no estén familiarizados con estos métodos mágicos, estos métodos permiten definir la representación del objeto que llame dicho método. En nuestro caso, lo que nos interesa de un contexto es el valor de las constantes cuando éstas ya hayan sido inicializadas (incluyendo las del contexto abstracto que extiende si es el caso). Para un ejemplo de lo anterior, ver Figura 3.5.

Antes de pasar a la sección de las máquinas, expondremos una tabla que resume el soporte de nuestro generador de código respecto a la traducción de los contextos, ver Figura 3.6.

## 3.2. Máquinas

Como lo mencionamos, por cada máquina que se traduzca, se generará un archivo. El nombre del archivo será el mismo nombre que tenía la máquina en Rodin, pero se le adiciona “\_mch.py” (ej. Si la máquina se llama *Foo* en Rodin, el archivo que lo traduzca se llamará *Foo\_mch.py*).

```
>>> print(ce)
###
HospitalCtxExt Constants
MaxMDCap ==> 5
###
HospitalCtx Constants
Available ==> HospitalState_CS.Available
Capacity ==> 5
Unavailable ==> HospitalState_CS.Unavailable
>>> |
```

Figura 3.5: Ejemplo de la impresión del estado de un contexto en consola.

| RESUMEN DE LA TRADUCCIÓN DE CONTEXTOS  |  |   |
|--|--|---|
| Nuestro traductor tiene soporte para generar código relativo a los siguientes aspectos:  |  |   |
| Elementos/Componentes de Event-B   | Características de contextos en Event-B  | Métodos para el Usuario   |
| Axiomas (y teoremas), Constantes y sus tipos de datos, Carrier Sets (enumeraciones) y el tipo de dato generado por los mismos. | Contrato para el cumplimiento de Axiomas, Asignación de Valores a constantes (y enumeraciones), Contrato para el aseguramiento de el no cambio de valores de constantes y enumeraciones luego de su asignación, Extensión de Contextos | Método mágico de Python <code>str</code> , Método mágico de Python <code>repr</code> , <code>CheckAllAxioms</code> , <code>CheckedInit</code> , <code>PyRandValGen</code> |

Figura 3.6: Resumen de la traducción de contextos.

### 3.2.1. Dependencias

Similar a un contexto, lo primero de cualquier archivo relativo a una máquina que se genere son las dependencias (que estarán bajo el comentario “DEPENDENCIES”). Las máquinas importan las mismas dependencias que los contextos, la única diferencia es que como una máquina puede “ver” un contexto y refinar una máquina abstracta, entonces puede potencialmente requerir la importación de dos archivos adicionales (el del contexto que “vea” y el de la máquina abstracta que vaya a refinar).

Directamente luego de las dependencias viene la clase que representa la máquina. Lo primero es que si la máquina en cuestión que se esté traduciendo está refinando una máquina, entonces su clase será hija de la clase de la máquina abstracta.

### 3.2.2. Constructor y Métodos Básicos

Procedamos con el método `__init__`. A diferencia de los contextos, este método si recibirá un parámetro en caso de que la máquina en cuestión “vea” un contexto (el cual puede estar extendiendo otros contextos). El parámetro deberá ser un objeto de la clase de dicho contexto (si el usuario no especifica el parámetro, se generará un objeto de dicha clase automáticamente). Si el objeto del contexto (ya sea especificado por el usuario o generado automáticamente) no tiene inicializadas sus constantes, el código forzará a dicho objeto a tenerlas inicializadas. La razón de esto es que una máquina “ve” un contexto con la intención de usar sus constantes o conjuntos, y por lo tanto para su funcionamiento es necesario que dicho contexto tenga ya sus atributos definidos. Dicho objeto se asigna a un atributo privado de la máquina para su uso.

Seguido a eso, y dentro del mismo método `__init__`, se declaran las variables (y su tipo) como atributos de la clase. Luego viene una sección bajo el comentario “#INITIALISATION of variables” en la cual está la traducción del evento de inicialización que se haya definido en Event-B. Por cada asignación de dicho evento se genera una línea de código correspondiente. Finalmente, se llama un método que revisa el cumplimiento de todas las invariantes (similar al `checkAllAxioms` del contexto), el cual si se incumple genera una excepción indicando que las invariantes se han violado. La razón de lo anterior es lo que se vio en el marco teórico: las invariantes deben cumplirse luego de inicializar las variables de la máquina. Nuevamente, no sobra mencionar que el chequeo de las invariantes se puede obviar si se desactiva el diseño por contrato con el Preludio. Para un ejemplo de todo lo anterior, ver Figura 3.7.

Posterior al `__init__` vienen todos los métodos “set” y “get” de los atributos que representan las variables. Es importante mencionar que también habrá un método llamado `PyMachineVariant` en esta sección en caso de que la máquina en Event-B tenga una variante (recordemos del marco teórico: una variante es una expresión, ya sea un entero o un conjunto, que se usa para demostrar convergencia). El método `PyMachineVariant` no recibe parámetros y retornará el valor de la expresión de la variante. Es importante introducir aquí la primera limitante de nuestro traductor: la

```

26
27 ▼ class HospitalMchRef_class(HospitalMch_class):
28
29 ▼   def __init__( self , HospitalCtxExt_userIn : HospitalCtxExt_class = HospitalCtxExt_class() ) -> None:
30
31     #Assign Parameter to Context Extended Dependency Object
32     self._HospitalCtxExt : HospitalCtxExt_class = HospitalCtxExt_userIn
33     if not(self._HospitalCtxExt.Initialized_ContextGetMethod()):
34       self._HospitalCtxExt.checkedInit()
35
36     #Variables
37     self.HeadNurse : HospitalState_CS
38     self.InSurgeryCnt : int
39     self.MDCnt : int
40     self.NurseCnt : int
41     self.SurgeryCnt : int
42     self.WaitingRoomCnt : int
43     #EndVariables
44
45     #INITIALISATION of variables
46     self.WaitingRoomCnt = 0
47     self.InSurgeryCnt = 0
48     self.HeadNurse = self.HospitalCtxExt_get().HospitalCtx_get().Unavailable
49     self.NurseCnt = 0
50     self.MDCnt = 0
51     self.SurgeryCnt = 0
52
53     #Check ALL Invariants if enabled.
54     if P.DESIGN_BY_CONTRACT_ENABLED() and not(self.checkAllInvariants()):
55       raise Exception("Invariants violated after INITIALISATION.")
56

```

Figura 3.7: Constructor de una máquina en un código generado por nuestro traductor. Aquí resalta que la máquina “ve” un contexto, y por lo tanto lo recibe como parámetro.

variante en nuestro código generado no puede usarse para evaluar expresiones que retornan conjuntos, sin embargo, esto no debería ser mayor problema, pues en Event-B, cuando se retornan conjuntos en una variante, se hace únicamente con el fin de revisar su cardinalidad (un entero). Así que en el modelo, si se quiere trabajar con conjuntos en una variante, lo único que hay que hacer es especificar en el modelo de Event-B que lo que nos interesa es la cardinalidad de dicho conjunto. Posteriormente explicaremos cómo se genera un *contrato* para las variantes en nuestro traductor que haga uso del método *PyMachineVariant*.

Luego viene la sección de los métodos para las invariantes y el método *checkAllInvariants*, que funcionan exactamente igual a los axiomas y al método *checkAllAxioms* que ya explicamos (son métodos que no reciben parámetros y retornan *True* en caso de que la(s) invariante(s) se esté(n) cumpliendo).

### 3.2.3. Eventos

Procedamos a analizar la traducción de los eventos. Para cada evento en una máquina, se generan dos métodos: uno para sus guardas, y otro para sus acciones.

El método para las guardas retorna un booleano (que siempre será verdad si se desactiva el diseño por contrato), donde el valor que retorna depende de si se cumplen las guardas o no (tal y como ocurre en Event-B). Si la guarda es un teorema, se genera un comentario arriba de la guarda

```

338 |     #Finish - Event
339 |
340 |
341 def Finish_eventGuards(self) -> bool:
342     guard_ans : bool = True
343
344     #Set Parameters as an Attribute.
345
346     #Check Event PreConditions.
347     if P.DESIGN_BY_CONTRACT_ENABLED():
348         try:
349             if True and (len(self.visited) == len(self.DijkstraCtx_get().CITIES)) and (self.finished == False):
350                 guard_ans = True
351             else:
352                 guard_ans = False
353             except:
354                 guard_ans = False
355             else:
356                 guard_ans = True
357
358     return guard_ans

```

Figura 3.8: *Ejemplo de un método que evalúa las guardas de un evento generado por nuestro traductor.*

indicando que la guarda es un teorema. Ahora, la guarda puede recibir parámetros (correspondientes a los parámetros del evento que se definan bajo la instrucción *any* en Event-B). Es importante tener en cuenta que el método de las guardas no autogenera parámetros si el usuario no los indica, en otras palabras, es obligatorio que los parámetros sean explícitamente indicados por el usuario para poder evaluar las guardas. Para un ejemplo de un código generado por las guardas de un evento, ver Figura 3.8.

El método para las acciones es un poco más complicado. Lo primero es que este método no retorna ningún valor. Lo segundo para tener en cuenta es que este método también recibe parámetros (correspondientes a los parámetros del evento en Event-B). A diferencia del método de las guardas, si no se indican los parámetros de las acciones, el Preludio esta vez sí tratará por defecto de generar valores para hacer cumplir las guardas y que el evento pueda ejecutarse. Es pertinente mencionar aquí una gran ventaja de haber trabajado con el AST de Rodin: En Event-B existe algo que se llaman los *observadores*, los cuales permiten redefinir los parámetros de un evento abstracto (cuando se están refinando máquinas). Nuestro traductor no tuvo que preocuparse por esto pues el AST de Rodin se encarga de “parsear” este fenómeno e informarnos correctamente de cuáles son los parámetros del nuevo evento (e ignorar los que están siendo reemplazados), por lo tanto, nuestro traductor logra dar soporte a los *observadores* sin tener que definir reglas de traducción para ello.

Lo primero que hace el método de las acciones de un evento es revisar las guardas y generar una excepción indicando que no se pudieron cumplir las guardas del evento si ése es el caso. La razón por la que se genera una excepción es que en un modelo correcto concreto no debe ocurrir que se intente llamar a un evento que no esté habilitado. Si lo que se quiere es revisar el estado de sus guardas, se puede llamar al método de las guardas (para eso se separaron). Cuando las guardas se cumplen, el método continua (todo lo mencionado sería otro *contrato* que refleja el comportamiento de Event-B para asegurar correctitud). Si el método tiene parámetros, pero los parámetros ingresados por el

```

227 #Convergent Event!
228 def inc_eventActions(self) -> None:
229
230     attempt_Count : int = 0
231     while not(self.inc_eventGuards()):
232         if attempt_Count == P.LOWMAXENATTEMPTS():
233             raise GuardsViolated("Guards of the Event could not be fulfilled.")
234         attempt_Count += 1
235
236     #Set Parameters as an Attribute.
237
238     #Event Actions
239
240     #Convergent Event: Value of the variant before the actions.
241     tmp_variant_value : int = self.PyMachineVariant()
242
243     self.p, self.r = (self.r + 1), PySet({int_range for int_range in range((self.r + 1), self.q+1)}).PyChoice()
244
245     #Convergent Event: Check that the Variant decreased.
246     if tmp_variant_value <= self.PyMachineVariant():
247         raise Exception("The Variant did not decrease!")
248
249     #Check Invariants after the actions are executed.
250     if P.DESIGN_BY_CONTRACT_ENABLED() and not(self.checkAllInvariants()):
251         raise Exception("PostConditions of the Event could not be fulfilled.")
252
253
254 #End Event

```

Figura 3.9: *Ejemplo 1 de la traducción de las acciones de un evento. Aquí resalta que el evento es convergente y se verifica la variante en la línea de código 246.*

usuario (opcionalmente), o los valores autogeneratedados no pueden cumplir las guardas, se genera una excepción también.

Al igual que en Event-B, si las guardas se cumplen, se debe proceder con ejecutar las asignaciones de las acciones del evento. Pero es importante introducir aquí una característica de las acciones de un evento, ¡las acciones se deben ejecutar en paralelo como en Event-B! Afortunadamente, Python tiene una manera muy sencilla de representar y ejecutar este efecto (que es el que usó en nuestro traductor), el cual es escribir las asignaciones en una misma línea. Por ejemplo, si yo quiero intercambiar los valores de las variables “x” y “y”, en Python se puede hacer de manera muy sencilla con la instrucción:  $x,y=y,x$ . Luego de ejecutar las acciones se revisan las invariantes (lo que representa otro *contrato*, pues refleja el comportamiento de los eventos en Event-B: donde luego de que las acciones de un evento se ejecutan, se debe asegurar que las invariantes se siguen cumpliendo).

Sólo queda pendiente por mencionar la traducción de un evento convergente y anticipado. Lo primero, es que si el evento es convergente o anticipado, se genera un comentario indicándolo. Pero en el caso de que el evento sea convergente se genera un *contrato* más: Se compara el valor de la variante para antes y después de ejecutar las acciones, si el valor de la variante no disminuye, se genera una excepción. Para un ejemplo de todo lo anterior, ver Figura 3.9 y Figura 3.10.

```

387
388 ▼ |def StartClosing_eventActions(self, door3_userIn : Door_CS = P.NoParam()) -> None:
389
390     if door3_userIn is None:
391         door3_userIn = P.PyRandValGen("Door_CS")
392
393     attempt_Count : int = 0
394     while not(self.StartClosing_eventGuards(door3_userIn)):
395         door3_userIn = P.PyRandValGen("Door_CS")
396         if attempt_Count == P.LOWMAXGENATTEMPTS():
397             raise GuardsViolated("Guards of the Event could not be fulfilled.")
398         attempt_Count += 1
399
400     #Set Parameters as an Attribute.
401     self.door3 = door3_userIn
402
403     #Event Actions
404
405     self.doors = self.doors.PyOverriding(PyRel({{self.door3, self.ParkingLotCtx_get().Closing}}))
406
407     #Check Invariants after the actions are executed.
408     if P.DESIGN_BY_CONTRACT_ENABLED() and not(self.checkAllInvariants()):
409         raise Exception("PostConditions of the Event could not be fulfilled.")
410
411     del(self.door3)
412
413     #End Event

```

Figura 3.10: Ejemplo 2 de la traducción de las acciones de un evento. Aquí resalta que el evento recibe parámetros.

### 3.2.4. Métodos para el Usuario

Luego de los eventos vienen 4 métodos muy importantes. Dos de ellos son *str* y *repr*, que cumplen la misma función que cumplen dichos métodos mágicos en los contextos. Sólo para ilustrar, podemos verlos en acción en la Figura 3.11, donde podremos ver un ejemplo del estado de una máquina. El estado de una máquina se representa por el estado de los contextos que “ve” y por el valor de sus variables en el momento en que se llama al método mágico.

El tercer método se llama *PyGuardsState*, que no recibe parámetros ni retorna nada. Su función es mostrar cuáles eventos están habilitados para ejecutarse (mientras el evento no tenga dos o más parámetros, en cuyo caso se dirá que no se sabe si está habilitado o no). Además, si el evento tiene un parámetro, se presenta uno de los valores para el parámetro que permite la ejecución de dicho evento. La intención de este método es el de ofrecer al usuario una manera más de hacer seguimiento al estado de la máquina. Para un ejemplo de lo anterior, ver Figura 3.12.

Finalmente está el método *PyAutoExecute* (probablemente el método más útil que se le ofrece al usuario), sobre el cual ahondaremos en el capítulo de *No Determinismo y el Infinito*. Sin embargo, para adelantar de una vez su funcionalidad, lo que hace es buscar un evento al azar que esté habilitado, ejecutarlo, e imprimir el nuevo estado alcanzado por la máquina. Lo último que mencionaremos es que en Event-B se pueden refinar o extender eventos, aumentar variables, cambiar parámetros de eventos (entre otras cosas), las cuales, afortunadamente, son procesadas por el AST de Rodin, y por lo tanto no tenemos que lidiar con eso, es suficiente con aplicar las reglas de traducción que

```
>>> print(m)
###
CafeteriaCtxExt Constants
CashierAvailable ==> 1
CashierUnavailable ==> 0
NoPickup ==> 0
PickupReady ==> 1
###
CafeteriaCtx Constants
MaxChairCnt ==> 99
###
Variables
AvailableDishCnt ==> 0
Cashier ==> 1
ClientsEatingCnt ==> 0
ClientsInQueueCnt ==> 0
Pickup ==> 0
>>> |
```

Figura 3.11: Impresión del estado de una máquina en consola, en el que se mapean las constantes y variables del contexto a los valores que tienen en el momento que se llama al método.

```
State of the Guards of every event!
CashierCheckin ==> False
CashierCheckout ==> True
CallForPickup ==> True
CollectPickup ==> False
AddDishes ==> True, e.g. with param = 1
ExitCafeteria ==> False
ArriveInQueue ==> True, e.g. with param = 1
ServeClients ==> Undetermined
LeaveQueue ==> False
>>> |
```

Figura 3.12: Ejemplo de lo que imprime el método *PyGuardsState*, representando cuáles eventos están habilitados.

| RESUMEN DE LA TRADUCCIÓN DE MÁQUINAS  |  |  |
|---|--|--|
| Nuestro traductor tiene soporte para generar código relativo a los siguientes aspectos:   |  |  |
| Elementos/Componentes de Event-B  | Características de contextos en Event-B  | Métodos para el Usuario  |
| Variables y sus tipos de datos, Invariantes (y teoremas), Variantes, Evento de inicialización, Eventos (guardas, acciones, parámetros -y sus tipo de datos- del evento) + eventos convergentes y anticipados. | Contrato para el cumplimiento de invariantes (al inicializar la máquina y luego de la ejecución de cada evento), Contrato para el cumplimiento de guardas, Contrato para el cumplimiento de la variante en métodos convergentes, Refinamiento de Máquinas, Máquinas "viendo" contextos, Hay soporte implícito para observadores que permiten la desaparición de parámetros de evento en un refinamiento. | Método mágico de Python <code>sir</code> , Método mágico de Python <code>repr</code> , <code>CheckAllInvariants</code> , <code>PyRandValGen</code> , <code>PyAutoExecute</code> , <code>PyGuardsState</code> |

Figura 3.13: Resumen de la traducción de máquinas.

hemos mencionado y las que están pendientes por mencionar. Siendo así, concluimos con todos los componentes de una máquina que traduciremos.

Hasta el momento hemos hablado del uso de los asignaciones, predicados y expresiones, sin especificar cómo se traducen los mismos. Lo que ocurre es que la traducción de estos elementos se logra gracias a las reglas de traducción que se explican a continuación en las siguientes secciones de este capítulo (puesto que todo lo que está pendiente son los elementos que componen las asignaciones, predicados, y expresiones en Event-B). No sobra mencionar que las expresiones en Event-B aparecen en las siguientes situaciones: Tipos de datos de las variables, constantes, variantes y parámetros de eventos. También aparecen en asignaciones y dentro de predicados; Los predicados en Event-B aparecen en las siguientes situaciones: Axiomas, Invariantes, Guardas, Cuantificadores, Asignaciones, Acciones, y dentro de otras expresiones o asignaciones; Las asignaciones ocurren en las acciones de un evento.

Antes de pasar a la siguiente sección, queremos presentar una tabla que resume el soporte de nuestro generador de código respecto a la traducción de las máquinas, ver Figura 3.13.

### 3.3. Reglas de Traducción

En este capítulo veremos las reglas de traducción para los elementos que componen los contextos y las máquinas. No sobra mencionar que, aunque las reglas de traducción se presentan por separado, la idea es que un predicado se construya mediante la combinación de una o más de estas reglas, así, podemos tener cosas como: “un cuantificador universal en el que se evalúe la conjunción de dos predicados donde se calculan operaciones entre expresiones que representan conjuntos de enteros”. El AST de Rodin es el que nos permitió lograr lo anterior, y el traductor se encarga de generar los paréntesis y demás elementos necesarios para que se construya correctamente la combinación de las reglas de traducción que veremos a continuación.

| Elemento Especial  |                       | Event-B          |   | Traducción a Python   |  |         |
|--|-----------------------|------------------|---|-----------------------|--|---------|
| Elemento   | Tipo                  | Simbolo          | Descripción   | Representación        | Descripción  | Soporte |
| True (en un predicado)   | Literal de Predicados | T                |   | True                  | Se usan los valores booleanos nativos de Python.   | Sí      |
| False (en un predicado)  | Literal de Predicados | ⊥                |   | False                 | Se usan los valores booleanos nativos de Python.   | Sí      |
| Pareja ordenada (tupla)  | Expresión Binaria     | e ↪ f            |   | (e, f)                | Se usan las tuplas nativas de Python.  | Sí      |
| True (en una expresión)  | Predicado Literal     | TRUE             |   | True                  | Se usan los valores booleanos nativos de Python.   | Sí      |
| False (en una expresión)   | Expresión Atómica     | FALSE            |   | False                 | Se usan los valores booleanos nativos de Python.   | Sí      |
| Conjunto de los números enteros.   | Expresión Atómica     | Z                |   | PyINT() ó P.INT()     | Detalles en el capítulo <i>No Determinismo y el Infinito</i> .   | Sí      |
| Conjunto de los números naturales.   | Expresión Atómica     | N                |   | PyNAT() ó P.NAT()     | Detalles en el capítulo <i>No Determinismo y el Infinito</i> .   | Sí      |
| Conjunto de los números naturales sin el 0.  | Expresión Atómica     | N1               |   | PyNAT1() ó P.NAT1()   | Detalles en el capítulo <i>No Determinismo y el Infinito</i> .   | Sí      |
| Relación identidad.  | Expresión Atómica     | id               |   | PyID() ó P.ID()       | Detalles en el capítulo <i>No Determinismo y el Infinito</i> .   | Sí      |
| Números enteros.   | Literal de Enteros    | Ej.: 37          |   | Ej.: 37               | Se usan los números enteros nativos de Python.   | Sí      |
| Identificadores  | Identificador Libre   | Ej.: constante_x | Se usa para representar constantes, variables, Carrier Sets, etc. | Ej.: self.constante_x | El traductor calcula quién está llamando el identificador (como se dijo, no necesariamente el identificador es atributo de la clase que lo llama).   | Sí      |
| Extra, hay soporte también para los siguientes casos especiales: INT x INT, NAT x NAT, INT x NAT, E(INT), E(NAT), E(NAT1), E1(INT), E1(NAT), E1(NAT1), E1(NAT1), E(INT x INT), E(NAT x NAT), E(INT x NAT), E1(INT), E1(NAT), E1(NAT1), E1(INT x INT), E1(NAT x NAT), E1(INT x NAT) |                       |                  |   |                       | PyINTXINT(), PyINTXNAT(), PyNATXNAT(), y los demás se pueden obtener con ayuda de las operaciones de <i>Producto Cartesiano</i> y <i>Producto Potencia (No Vacío)</i> . Detalles en el capítulo <i>No Determinismo y el Infinito</i> . | Sí      |

Figura 3.14: Reglas de traducción para elementos especiales de Event-B.

Para visualizar con mayor facilidad las reglas de traducción que se presentan en este capítulo, hemos construido unas tablas donde se puede apreciar con mayor claridad cada regla de traducción. Las tablas manejan la siguiente convención: “e” y “f” representan expresiones, “p” y “q” representan predicados, “x” y “y” representan variables libres, “S” y “T” representan conjuntos. “R” y “U” representan relaciones. “g” representa una función. “var” representa un identificador libre de: una constante, una variable, una *enumeración*, o una asignación en una función (Ej.:  $g(x)=7$  ).

### 3.3.1. Elementos Especiales de Event-B

Los elementos especiales en Event-B comprenden algunas expresiones/predicados base que se usan en operaciones del mismo lenguaje. Para visualizar las reglas de traducción de elementos especiales en Event-B, ver la tabla en la Figura 3.14. Se siguen las convenciones que se declararon al comienzo de esta sección.

| Operaciones sobre Predicados |                      | Event-B                 |             | Traducción a Python                                   |   |         |
|------------------------------|----------------------|-------------------------|-------------|---|---|---------|
| Operación                    | Tipo                 | Símbolo                 | Descripción | Representación  | Descripción   | Soporte |
| Conjunción                   | Predicado Asociativo | $p \wedge q$            |             | (e and f)   | Se usa la operación nativa <i>and</i> de Python.  | Si      |
| Disyunción                   | Predicado Asociativo | $p \vee q$              |             | (e or f)  | Se usa la operación nativa <i>or</i> de Python.   | Si      |
| Implicación                  | Predicado Binario    | $p \Rightarrow q$       |             | P.LogicImplication(p, q)                              | Operación del Preludio.   | Si      |
| Equivalencia                 | Predicado Binario    | $p \Leftrightarrow q$   |             | P.LogicEquivalence(p, q)                              | Operación del Preludio.   | Si      |
| Negación                     | Predicado Binario    | $\neg p$                |             | not(p)  | Se usa la operación nativa <i>not</i> de Python. (Funciona también con cuantificadores)   | Si      |
| Cuantificador Universal      | Expresión Unaria     | $\forall x,y \cdot (p)$ |             | P.QuantifiedForAll ((lambda x,y) , [type(x),type(y)]) | El traductor genera un lambda que evalúa el predicado para todos los valores posibles que pueden tomar las variables libres según su tipo de dato. Es una operación del Preludio. | Si      |
| Cuantificador Existencial    | Expresión Unaria     | $\exists x,y \cdot (p)$ |             | P.QuantifiedExists ((lambda x,y) , [type(x),type(y)]) | Similar al cuantificador universal. Pero se hace verdad cuando una sola combinación de valores de las variables libres hace verdad el predicado (no cuando todas o ninguna).      | Si      |
| Igualdad                     | Predicado Relacional | $e = f$                 |             | e == f  | Se usa la operación nativa <i>==</i> de Python.   | Si      |
| Desigualdad                  | Predicado Relacional | $e \neq f$              |             | e != f  | Se usa la operación nativa <i>!=</i> de Python.   | Si      |

Figura 3.15: *Reglas de traducción para operaciones sobre predicados en Event-B.*

### 3.3.2. Operaciones sobre Predicados

Para visualizar las reglas de traducción de operaciones en predicados de Event-B, ver la tabla en la Figura 3.15. Se siguen las convenciones que se declararon al comienzo de esta sección.

### 3.3.3. Operaciones sobre Conjuntos

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.16. Hay que recordar que las relaciones son conjuntos de tuplas, y por lo tanto estas operaciones se pueden usar sobre relaciones.

### 3.3.4. Predicados sobre Conjuntos

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.17.

### 3.3.5. Operaciones sobre Enteros

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.18.

| Operaciones sobre Conjuntos             |                        | Event-B                  |                             | Traducción a Python     |   |         |
|---|------------------------|--------------------------|-----------------------------|-------------------------|---|---------|
| Operación                               | Tipo                   | Símbolo                  | Descripción                 | Representación          | Descripción   | Soporte |
| <b>Conjunto por extensión</b>           | Extensión de Conjuntos | {e, f, ...}              |                             | PySet({e, f, ...})      | Usamos la clase PySet que recibe un Set nativo de Python.                     | Si      |
| <b>Conjunto vacío</b>                   | Expresión Atómica      | Ø                        |                             | PySet() ó PySet(set())  | Usamos la clase PySet, que sin parámetro especificado crea un conjunto vacío. | Si      |
| <b>Conjunto por extensión</b>           | Extensión de Conjuntos | {e, f, ...}              |                             | PySet({e, f, ...})      | Usamos la clase PySet que recibe un Set nativo de Python.                     | Si      |
| <b>Comprensión de Conjuntos (1)</b>     | Expresión Cuantificada | {x : (p   e)}            |                             |                         | Más detalles en el capítulo <i>No Determinismo y el Infinito</i>              | No      |
| <b>Comprensión de Conjuntos (2)</b>     | Expresión Cuantificada | {e   p}                  |                             |                         | Más detalles en el capítulo <i>No Determinismo y el Infinito</i>              | No      |
| <b>Comprensión de Conjuntos (3)</b>     | Expresión Cuantificada | {x   p}                  | Equivalente a {x : (p   x)} |                         | Más detalles en el capítulo <i>No Determinismo y el Infinito</i>              | Si      |
| <b>Unión</b>                            | Expresión Asociativa   | S ∪ T                    |                             | S.PyUnion(T)            |   | Si      |
| <b>Intersección</b>                     | Expresión Asociativa   | S ∩ T                    |                             | S.PyIntersection(T)     |   | Si      |
| <b>Diferencia</b>                       | Expresión Binaria      | S \ T                    |                             | S.PyDifference(T)       |   | Si      |
| <b>Producto Cartesiano</b>              | Expresión Binaria      | S × T                    |                             | S.PyCartesianProduct(T) |   | Si      |
| <b>Conjunto Potencia</b>                | Expresión Unaria       | ℘(S)                     |                             | S.PyPowerSet()          |   | Si      |
| <b>Conjunto Potencia No Vacío</b>       | Expresión Unaria       | ℘1(S)                    |                             | S.PyPowerSet1()         |   | Si      |
| <b>Cardinalidad</b>                     | Expresión Unaria       | card(S)                  |                             | len(S)                  |   | Si      |
| <b>Unión/Intersección generalizada.</b> |                        | union(...)<br>inter(...) |                             |                         |   | No      |
| <b>Unión/Intersección cuantificada.</b> |                        | UNION(...)<br>INTER(...) |                             |                         |   | No      |

Figura 3.16: Reglas de traducción para operaciones sobre conjuntos en Event-B.

| Predicados sobre Conjuntos    |                      | Event-B                           |  | Traducción a Python   |             |         |
|-------------------------------|----------------------|-----------------------------------|--|---|-------------|---------|
| Predicado                     | Tipo                 | Símbolo                           | Descripción  | Representación  | Descripción | Soporte |
| <b>Pertenencia</b>            | Predicado Relacional | $e \in S$                         |  | <code>S.PyContains(e)</code>  |             | Sí      |
| <b>No Pertenencia</b>         | Predicado Relacional | $e \notin S$                      |  | <code>S.PyNotContains(e)</code>   |             | Sí      |
| <b>Subconjunto</b>            | Predicado Relacional | $S \subseteq T$                   |  | <code>S.PySubSet(T)</code>  |             | Sí      |
| <b>No Subconjunto</b>         | Predicado Relacional | $S \not\subseteq T$               |  | <code>S.PyNotSubSet(T)</code>   |             | Sí      |
| <b>Subconjunto Propio</b>     | Predicado Relacional | $S \subset T$                     |  | <code>S.PyProperSubSet(T)</code>  |             | Sí      |
| <b>No Subconjunto Propio</b>  | Predicado Relacional | $S \not\subset T$                 |  | <code>S.PyNotProperSubSet(T)</code>   |             | Sí      |
| <b>Conjunto Finito</b>        | Predicado Simple     | <code>finite(S)</code>            |  | <code>S.PyFinite()</code>   |             | Sí      |
| <b>Partición de Conjuntos</b> | Predicado Múltiple   | <code>partition(S, T, ...)</code> | También,<br><code>partition(S, {e}, {f}, ...)</code> | <code>S.PyPartition([T, ...])</code> ó<br><code>S.PyPartition([e, f, ...])</code> |             | Sí      |

Figura 3.17: Reglas de traducción para predicados sobre conjuntos en Event-B.

| Operaciones sobre Enteros |                      | Event-B             |  | Traducción a Python   |  |         |
|---------------------------|----------------------|---------------------|--|---|--|---------|
| Operación                 | Tipo                 | Símbolo             | Descripción  | Representación  | Descripción  | Soporte |
| <b>Mínimo</b>             | Expresión Unaria     | <code>min(S)</code> |  | <code>P.Minimum(S)</code>   | Operación del Preludio.  | Sí      |
| <b>Máximo</b>             | Expresión Unaria     | <code>max(S)</code> |  | <code>P.Maximum(S)</code>   | Operación del Preludio.  | Sí      |
| <b>Suma</b>               | Expresión Asociativa | Ej.: $3+4$          |  | Ej. $(3+4)$   | Se usan las operaciones nativas de Python.   | Sí      |
| <b>Multiplicación</b>     | Expresión Asociativa | Ej.: $3*7$          |  | Ej. $(3*7)$   | Se usan las operaciones nativas de Python.   | Sí      |
| <b>Resta</b>              | Expresión Binaria    | Ej.: $3-4$          |  | Ej. $(3-4)$   | Se usan las operaciones nativas de Python.   | Sí      |
| <b>División</b>           | Expresión Binaria    | Ej.: $3/7$          |  | Ej. $(3/7)$   | Se usan las operaciones nativas de Python.   | Sí      |
| <b>Módulo</b>             | Expresión Binaria    | Ej.: $9 \bmod 5$    |  | Ej. $(9 \% 5)$  | Se usan las operaciones nativas de Python.   | Sí      |
| <b>Rango</b>              | Expresión Binaria    | Ej.: $3..7$         | Representa un conjunto en el rango que indican los extremos (inclusive). | Ej. <code>PySet( {int_range for int_range in range(3,7+1)} )</code> | Se genera un PySet con los números del rango. Con rangos del tipo $7..3$ genera conjuntos vacíos, como en Event-B. | Sí      |

Figura 3.18: Reglas de traducción para operaciones sobre enteros en Event-B.

| Predicados sobre Enteros |                      | Event-B  |             | Traducción a Python |  |         |
|--------------------------|----------------------|----------|-------------|---------------------|--|---------|
| Predicado                | Tipo                 | Símbolo  | Descripción | Representación      | Descripción                                | Soporte |
| <b>Menor a</b>           | Predicado Relacional | Ej.: 3<4 |             | Ej.: 3 < 4          | Se usan las operaciones nativas de Python. | Si      |
| <b>Mayor a</b>           | Predicado Relacional | Ej.: 3>7 |             | Ej.: 3 > 7          | Se usan las operaciones nativas de Python. | Si      |
| <b>Menor o Igual a</b>   | Predicado Relacional | Ej.: 3≤4 |             | Ej.: 3 <= 4         | Se usan las operaciones nativas de Python. | Si      |
| <b>Mayor o Igual a</b>   | Predicado Relacional | Ej.: 3≥7 |             | Ej.: 3 >= 7         | Se usan las operaciones nativas de Python. | Si      |

Figura 3.19: Reglas de traducción para predicados sobre enteros en Event-B.

| Relaciones                         |                   | Event-B             |   | Traducción a Python  |  |         |
|------------------------------------|-------------------|---------------------|---|--|--|---------|
| Relación                           | Tipo              | Símbolo             | Descripción   | Representación   | Descripción  | Soporte |
| <b>Relación</b>                    | Expresión Binaria | S ↔ T               | Familia de Relaciones.  | PyFamilies( PyFamilyTypes. Relations, S, T)                | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | Si      |
| <b>Relación Total</b>              | Expresión Binaria | S ↔↔ T<br>(S<<->T)  | Familia de Relaciones Totales. (No hay acceso al símbolo oficial)               | PyFamilies( PyFamilyTypes. TotalRelations, S, T)           | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | Si      |
| <b>Relación Sobreyectiva</b>       | Expresión Binaria | S ↔→ T<br>(S<->>T)  | Familia de Relaciones Sobreyectivas. (No hay acceso al símbolo oficial)         | PyFamilies( PyFamilyTypes. SurjectiveRelations, S, T)      | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | Si      |
| <b>Relación Total Sobreyectiva</b> | Expresión Binaria | S ↔↔ T<br>(S<<->>T) | Familia de Relaciones Totales Sobreyectivas. (No hay acceso al símbolo oficial) | PyFamilies( PyFamilyTypes. TotalSurjectiveRelations, S, T) | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | Si      |

Figura 3.20: Reglas de traducción para relaciones en Event-B.

### 3.3.6. Predicados sobre Enteros

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.19.

### 3.3.7. Relaciones

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.20.

### 3.3.8. Funciones

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.21.

| Funciones                            |                   | Event-B                |   | Traducción a Python   |  |           |
|--------------------------------------|-------------------|------------------------|---|---|--|-----------|
| Función                              | Tipo              | Símbolo                | Descripción                                   | Representación  | Descripción  | Soporte   |
| <b>Función (Parcial)</b>             | Expresión Binaria | $S \Rightarrow T$      | Familia de funciones parciales.               | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>Functions, S, T)</code>          | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Total</b>                 | Expresión Binaria | $S \rightarrow T$      | Familia de funciones totales.                 | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>TotalFunctions, S, T)</code>     | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Parcial Inyectiva</b>     | Expresión Binaria | $S \rightsquigarrow T$ | Familia de funciones parciales inyectivas.    | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>PartialInjections, S, T)</code>  | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Total Inyectiva</b>       | Expresión Binaria | $S \rightarrowtail T$  | Familia de funciones totales inyectivas.      | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>TotalInjections, S, T)</code>    | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Parcial Sobreyectiva.</b> | Expresión Binaria | $S \Rightarrowtail T$  | Familia de funciones parciales sobreyectivas. | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>PartialSurjections, S, T)</code> | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Total Sobreyectiva.</b>   | Expresión Binaria | $S \Rightarrow T$      | Familia de funciones totales sobreyectivas.   | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>TotalSurjections, S, T)</code>   | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Función Total Biyectiva.</b>      | Expresión Binaria | $S \rightleftarrows T$ | Familia de funciones totales biyectivas.      | <code>PyFamilies(</code><br><code>PyFamilyTypes.</code><br><code>Bijections, S, T)</code>         | Se usa la clase PyFamilies, que verifica principalmente pertenencia. | <b>Sí</b> |
| <b>Abstracción Lambda</b>            |                   |                        |   |   |  | <b>No</b> |

Figura 3.21: Reglas de traducción para funciones en Event-B.

| Operaciones sobre Relaciones |                      | Event-B  |                                    | Traducción a Python        |             |         |
|------------------------------|----------------------|----------|------------------------------------|----------------------------|-------------|---------|
| Operación                    | Tipo                 | Símbolo  | Descripción                        | Representación             | Descripción | Soporte |
| Dominio                      | Expresión Unaria     | dom(R)   |                                    | R.PyDomain()               |             | Sí      |
| Rango                        | Expresión Unaria     | ran(R)   |                                    | R.PyRange()                |             | Sí      |
| Composición                  | Expresión Asociativa | R ; U    |                                    | R.PyComposition(U)         |             | Sí      |
| Composición Externa          | Expresión Asociativa | R ° U    |                                    | R.PyBackwardComposition(U) |             | Sí      |
| Restricción de Dominio       | Expresión Binaria    | S ⌠ R    |                                    | R.PyDomainRestriction(S)   |             | Sí      |
| Restricción de Rango         | Expresión Binaria    | R ⌢ S    |                                    | R.PyRangeRestriction(S)    |             | Sí      |
| Substracción de Dominio      | Expresión Binaria    | S ⌠ R    |                                    | R.PyDomainSubtraction(S)   |             | Sí      |
| Substracción de Rango        | Expresión Binaria    | R ⌢ S    |                                    | R.PyRangeSubtraction(S)    |             | Sí      |
| Inversa                      | Expresión Unaria     | R~       |                                    | ~R                         |             | Sí      |
| Imagen Relacional            | Expresión Binaria    | R[S]     |                                    | R[S]                       |             | Sí      |
| Sobrescritura (Overriding)   | Expresión Asociativa | (R <+ U) | (No hay acceso al símbolo oficial) | R.PyOverriding(U)          |             | Sí      |
| Producto Directo             | Expresión Binaria    | R ⊗ U    |                                    | R.PyDirectProduct(U)       |             | Sí      |
| Producto Paralelo            |                      | R    U   |                                    |                            |             | No      |
| Proyección 1                 |                      | prj1     |                                    |                            |             | No      |
| Proyección 2                 |                      | prj2     |                                    |                            |             | No      |

Figura 3.22: Reglas de traducción para operaciones sobre relaciones en Event-B.

### 3.3.9. Operaciones sobre Relaciones

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.22. Hay que recordar que las funciones son relaciones, y por lo tanto estas operaciones también pueden ser ejecutadas sobre funciones.

### 3.3.10. Operaciones sobre Funciones

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.23.

### 3.3.11. Asignaciones

Nuevamente, usaremos una tabla con las mismas convenciones, ver Figura 3.24.

| Operaciones sobre Funciones |                   | Event-B |                        | Traducción a Python |             |         |
|-----------------------------|-------------------|---------|------------------------|---------------------|-------------|---------|
| Operación                   | Tipo              | Símbolo | Descripción            | Representación      | Descripción | Soporte |
| Aplicación                  | Expresión Binaria | g(e)    | Imagen de una función. | g(e)                |             | Sí      |

Figura 3.23: *Reglas de traducción para operaciones sobre funciones en Event-B.*

| Asignaciones (Acciones)   |                          | Event-B   |  | Traducción a Python                          |   |         |
|---|--------------------------|-----------|--|--|---|---------|
| Asignación  | Tipo                     | Símbolo   | Descripción  | Representación                               | Descripción   | Soporte |
| Asignación Simple   | Asignación               | var := f  |  | e = f  | Operación nativa de Python.   | Sí      |
| Elección de Conjunto  | Asignación Indeterminada | var :∈ S  | Elige un valor al azar del conjunto S y se lo asigna a e   | e = S.PyChoice()                             | Más detalles en el capítulo No Determinismo y el Infinito   | Sí      |
| Elección por predicado  | Asignación Indeterminada | var :  P  | Elige un valor al azar entre los posibles valores que hagan cumplir el predicado P.  | e = PyBecomesSuchThat(lambda P, [type(var)]) | Más detalles en el capítulo No Determinismo y el Infinito   | Sí      |
| Sobreescritura de Función   | Asignación               | g(e) := f | El AST de Rodin automáticamente interpreta esto como una Asignación Simple que hace uso de la operación de Sobreescritura. | g = g.PyOverriding(PyRel({e,f}))             | No sobra mencionar que este traductor también puede traducir esta operación cuando e es una Tupla, la cual es la forma de Event-B de simular funciones de múltiples parámetros. | Sí      |
| Nota: Event-B permite múltiples asignaciones en una (a excepción de Sobreescritura de Función), nuestro traductor exige que se evite hacer eso y las asignaciones se hagan 1 a 1. |                          |           |  |  |   |         |

Figura 3.24: *Reglas de traducción para asignaciones en Event-B.*



## CAPÍTULO 4

# No Determinismo y el Infinito

---

Sin duda, el mayor reto con el que nos enfrentamos al traducir un modelo en Event-B es todo lo que esté relacionado con el indeterminismo y el infinito. En un artículo científico sobre ProB, podemos ver que incluso la herramienta oficial para animar modelos encuentra este tema como uno de sus mayores obstáculos (traducido al español): “Los retos clave al resolver restricciones en lenguajes de alto nivel como B son los cuantificadores universales y existenciales, así como los conjuntos por comprensión y las lambdas. Cada uno puede estar anidado y no están limitados a valores finitos” [KL16]. También vimos que, para el trabajo más relacionado a este traductor, el generador de código a Java, el no determinismo y el infinito son retos difíciles de afrontar, y por lo tanto es donde más limitantes se encuentran a la hora traducir.

En nuestro traductor se buscaron soluciones a estos problemas, en algunas la solución es completa, en otras, parcial. Pero sin más preámbulos, ¡empecemos!

## 4.1. Métodos no Determinísticos

### 4.1.1. PyAutoExecute

En esta sección hablaremos de un indeterminismo sencillo de solucionar: buscar la manera de ejecutar automáticamente una máquina (con fines de depuración del código generado), considerando que en cualquier estado de una máquina cualquier evento que esté habilitado debería tener la misma probabilidad de ejecutarse.

Para solucionar esto ofrecemos una solución directa (similar a la del traductor a Java), cada máquina tendrá un evento llamado *PyAutoExecute*, el cual elegirá un evento al azar y lo ejecutará si está habilitado (incluyendo aquellos que tengan parámetros, pues como lo hemos mencionado varias veces, nuestro traductor puede generar valores aleatorios a parámetros que no se especifiquen. Tema sobre el cual detallaremos en la siguiente sección). Si el método consigue encontrar un evento habilitado con éxito, imprimirá en pantalla el nombre del evento que se ejecutó incluyendo el nuevo estado de la máquina. Si no consigue hacerlo, se indicará en pantalla (no se genera una excepción). Para ver un ejemplo de lo anterior, ver Figura 4.1.

La razón por la que se decidió construir este método es la de ofrecer herramientas al usuario que le permitan evidenciar con mayor facilidad la relación entre el modelo en Event-B y el código generado, y de poder hacer seguimiento a su ejecución y funcionamiento. Considerando que Rodin

```

progress1 successfully executed!!!
##  

ArrPartCtx Constants  

m ==> 10  

x ==> 5  

###  

Variables  

d ==> 1  

f ==> PyRel([(8, 3), (6, 8), (5, 6), (10, 2), (9, 5), (1, 4), (2, 10), (3, 7), (4, 1), (7, 9)])  

i ==> 1  

j ==> 10  

k ==> -1  

>>> |

```

Figura 4.1: Ejemplo de lo que imprime el método *PyAutoExecute* al poder ejecutar un evento (en dicho ejemplo se ejecuta el evento “progress1”) aleatoriamente con éxito. Específicamente, se imprime el nombre del evento que se ejecutó y el nuevo estado que alcanza la máquina

ofrece la herramienta ProB para animar una máquina, este método sería el equivalente al realizar una tarea muy similar (sin la parte de análisis de correctitud como la búsqueda de *Deadlocks* que ofrece ProB): buscar eventos habilitados, ejecutarlos, y mostrar el estado de la máquina/contexto. No sobra mencionar que este método se usó para todos los test que se ejecutaron para verificar el buen funcionamiento de nuestro traductor (que veremos en el capítulo de *Resultados*).

De aquí solo falta mencionar que el Preludio ofrece un parámetro que el usuario puede ajustar a su gusto llamado *MaxAutoExecuteAttempts*, este parámetro del Preludio evita que este método se quede en un ciclo infinito si por alguna razón no logra ejecutar ningún evento (por ejemplo, esto puede ocurrir si el único evento habilitado requiere un parámetro cuyos valores que lo habilitan son muy pocos o extravagantes como para encontrarlos aleatoriamente).

#### 4.1.2. PyRandValGen

Otro problema donde el indeterminismo se ve involucrado es en las situaciones donde múltiples valores pueden cumplir con la correctitud de un modelo y hacerlo avanzar en la ejecución, este fenómeno ocurre en las siguientes situaciones: Inicialización de constantes y de enumeraciones, parámetros de un evento, y asignaciones no determinadas. Lo único a lo que realmente vincula el AST de Rodin a estos 3 elementos es el tipo de dato, y es que tiene sentido, ya que por más indicaciones que den los axiomas/invariantes/guardas sobre los valores que pueden tomar estos elementos, siguen siendo restricciones que sólo en casos especiales determinan con facilidad el valor que podrían tomar estos elementos. Sin embargo, hacer un análisis de restricciones es un tema muy complicado, no sólo en lo teórico, sino también en lo técnico (ya se había hablado un poco de esto). Es por esta razón que los valores aleatorios que generaremos dependerán únicamente del tipo de dato del elemento, y serán generados por el método del Preludio llamado *PyRandValGen*.

Si el elemento es de tipo booleano, la respuesta es muy sencilla, se elige al azar si el elemento será verdadero o falso. Si es un entero, el Preludio elegirá un valor al azar en un rango que es editable

por el usuario mediante el parámetro *RAND\_INT\_RANGE* (por defecto, el rango será entre -100 y 100). Si es un conjunto/relación, construirá un conjunto/relación (de la clase PySet o PyRel respectivamente) de un tamaño variable (parametrizado por *RAND\_SET\_SIZE* / *RAND\_REL\_SIZE*) con elementos del tipo de dato de dicho conjunto/relación.

Las *enumeraciones* son un caso especial. El indeterminismo aquí es dependiente de la cantidad de elementos que tenga el tipo de dato generado por dicha enumeración (el cual, puede ser cualquiera que haga cumplir las restricciones de los *contratos*). El tamaño del tipo de dato se genera aleatoriamente durante el proceso de traducción. Que el Preludio realice esta tarea no tiene mucho sentido, pues ofrecer un método que pueda alterar el tamaño de un tipo de dato sería como permitir que de la nada uno diga que va a incluir el decimal 3.7 en el conjunto de los enteros. Es por esta razón que, si el usuario quiere cambiar el tamaño de una *enumeración*, debe de hacerlo manual y directamente editando el código generado (la sección del código donde se haría esto ya se mostró en la Figura 3.2). Esta es la única situación en que se requiere una edición manual del código generado por parte del usuario, algo que es necesario en muchos traductores en este ámbito.

Para disminuir un poco el efecto de lo anterior, y aprovechando que las *enumeraciones* frecuentemente sólo se definen en Event-B con particiones o la finitud de un conjunto, se hizo lo siguiente: si hay un axioma de partición asociado a una *enumeración*, el tamaño del tipo de dato generado no será aleatorio, sino que será el tamaño de la cantidad de conjuntos que generen la partición de dicha *enumeración*. La razón de esto es que, en los modelos, la única situación donde normalmente es necesario que el tamaño de la *enumeración* sea uno en específico, es en la que se acaba de mencionar. Esto reduce en gran medida la necesidad de que el usuario realmente tenga que alterar el código generado, de hecho, gracias a esto no fue necesario alterar el tamaño del tipo de dato en ninguno de los modelos con que se testeó este traductor. Finalmente, los elementos que tengan como tipo de dato el de una *enumeración*, se le asigna un valor distinto al de cualquier otro elemento que ya se le haya asignado un valor aleatoriamente, si se cubren todos los valores, vuelve y se reinicia la generación aleatoria (este efecto ayuda a que la inicialización aleatoria funcione con aún más éxito, puesto que normalmente se asignan valores distintos de una *enumeración* a constantes en un modelo).

#### 4.1.3. PyChoice y PyBecomesSuchThat

En esta sección trataremos una indeterminación muy fácil de solucionar, y se refiere a la asignación indeterminada que vimos en una de las tablas del capítulo anterior llamada *Elección de Conjunto*. Lo que hace esta función es elegir un elemento al azar de un conjunto, y eso fue justamente lo que se hizo, implementar una función aleatoria para los PySet/PyRel llamado *PyChoice*, el cual elige un elemento al azar del conjunto.

*PyBecomesSuchThat* funciona similar, se genera el conjunto de valores de cumplen el predicado (detalles de cómo hace esto en la siguiente sección) y se elige un elemento al azar de dicho conjunto.

Es importante mencionar que esta operación no está soportada para tuplas.

## 4.2. Conjuntos Infinitos

### 4.2.1. Conjuntos Infinitos y por Comprensión

En Event-B existen tres conjuntos infinitos especiales (a los que nos referiremos como conjuntos insignia): Los enteros, los naturales, y los naturales sin el 0. Representar estos conjuntos como un PySet es imposible, pues, por ejemplo, no hay forma de poner todos los enteros 1 a 1 por extensión en un conjunto. En el traductor a Java crean una clase especial para cada uno de los conjuntos insignia que permite principalmente la ejecución de un método esencial de los conjuntos: la pertenencia. Para verificar la pertenencia en un conjunto finito, lo normal es recorrer todo el conjunto y ver si el elemento en cuestión está ahí. Recorrer un conjunto infinito no es computable, sin embargo, verificar si un número pertenece por ejemplo a los naturales es sencillo, se verifica que el tipo de dato sea entero y que el valor sea mayor a igual a 0. La pertenencia en los 3 conjuntos insignia se puede verificar revisando una condición sencilla como la que acabamos de mencionar, y calcularla es O(1). Pero no queremos detenernos ahí, si podemos representar dichos conjuntos con una condición de pertenencia, podemos representar cualquier conjunto por comprensión si vemos dicha condición como un predicado. Es por esto que en nuestro Preludio se definió la clase *PyCondSet*, la cual recibe una función de pertenencia. Esta clase tiene 3 hijos, PyINT, PyNAT, PyNAT1, que reciben como función de pertenencia la condición que permite verificar la pertenencia de un entero en dichos conjuntos. En síntesis, se acopló la idea del de permitir las operaciones básicas para los conjuntos insignia de Event-B, pero se aprovechó la oportunidad para también dar soporte a los conjuntos por comprensión que se definen con un predicado.

No sobra mencionar que los conjuntos comprensión son muy útiles para denominar conjuntos infinitos, y también lo son para representar conjuntos finitos muy grandes donde no sea factible definirlos por extensión (ejemplo, el conjunto de los números naturales pares menores a 10'000.000). Nuestro traductor sólo permite traducir un tipo de conjuntos por comprensión (como se aprecia en las tablas del capítulo anterior), sin embargo, ese tipo de conjuntos es suficiente para representar los otros tipos de conjuntos por comprensión si el usuario hace las transformaciones necesarias (que son muy difíciles de automatizar en un traductor, pues, por ejemplo, requieren el cálculo de la inversa de una función).

Adicionalmente se usó los PyCondSet para también soportar los conjuntos potencia y conjuntos potencia no vacíos de los 3 conjuntos insignia de Event-B.

Otro tema importante a tratar aquí tiene que ver con los cuantificadores y la asignación no determinística *Elección por predicado* (operación *PyBecomesSuchThat* que se mencionó en la sección anterior). Para que los cuantificadores y la elección por predicado se puedan evaluar, se debe verificar el predicado asociado a dichas operaciones para todos los posibles valores que puedan tomar las

variables asociadas a dicho predicado. Lo anterior es problemático en un caso en especial: ¡cuando el tipo de dato de dichas variables es los enteros, un conjunto infinito que no podemos recorrer determinísticamente!

Para solucionar lo anterior, la clase PyCondSet recibe un parámetro que permite sobrescribir el método mágico *iter* (el cual permite iterar sobre la clase). Para los conjuntos insignia, se define un iterador especial que por defecto genera una excepción si es llamado (pues por defecto, no es posible recorrer un conjunto infinito). Pero si se asigna *True* al parámetro del Preludio llamado *USEFINITE\_SPECIAL\_SETS*, se permite recorrer conjuntos infinitos a través de recorrer un conjunto finito que lo represente. Por ejemplo, nuestro traductor permite recorrer los enteros como un conjunto finito de enteros entre el rango definido por el parámetro del Preludio llamado *FINITE\_SPECIAL\_SETS\_LIMIT* (por defecto, el rango va de -10 a 10). En síntesis, para poder usar los cuantificadores y la asignación por predicado cuando las variables asociadas exijan recorrer el conjunto infinito de los enteros, se debe activar el parámetro *USEFINITE\_SPECIAL\_SETS*.

PyCondSet también puede recibir como parámetros la cardinalidad y finitud del conjunto, si no se especifican, tratar de averiguar estos dos atributos de dicho conjunto genera una excepción. De igual manera, si no se especifica el predicado (la condición) de pertenencia del PyCondSet o el iterador, y se llaman dichos métodos, se genera una excepción. Más detalles de PyCondSet en la siguiente sección.

#### 4.2.2. Conjuntos Infinitos Extendidos

Aunque la pertenencia es posiblemente la operación más importante (y usada) con los conjuntos insignia o los conjuntos por comprensión, hay otras operaciones válidas en Event-B que son muy importantes en la teoría de conjuntos: Unión, Intersección, y Diferencia de conjuntos.

Para dar soporte a dichas operaciones en conjuntos por comprensión se crearon dos clases. La primera es *PyCondSet\_Ext*, que permite la ejecución de las 3 operaciones en cuestión entre conjuntos por comprensión y conjuntos finitos (básicamente son PyCondSets con una parte finita). Y también se creó la clase *PyCondSet\_TreeExt* que permite la ejecución de estas 3 operaciones entre conjuntos por comprensión. Esta última clase representa internamente los conjuntos como un árbol, donde por ejemplo: Si quiero hacer la unión de dos conjuntos infinitos, genero un nodo de tipo unión, el cual tiene como hijos los dos *PyCondSet\_Ext* que representan los conjuntos infinitos. Para verificar la pertenencia de un elemento a dicha unión de conjuntos por comprensión, se verifica si el elemento pertenece al hijo de la izquierda *jo!* al hijo de la derecha. Si fuera una intersección, se verifica que el elemento pertenezca al hijo de la izquierda *¡y!* al hijo de la derecha. Si es una diferencia, se verifica que pertenezca al hijo de la izquierda *¡y no!* al hijo de la derecha. En síntesis, estas clases permiten, por ejemplo, generar el conjunto de los naturales pares unido al conjunto de los negativos pares, o también se puede obtener el conjunto de los impares sin el número 17. En síntesis, se permiten

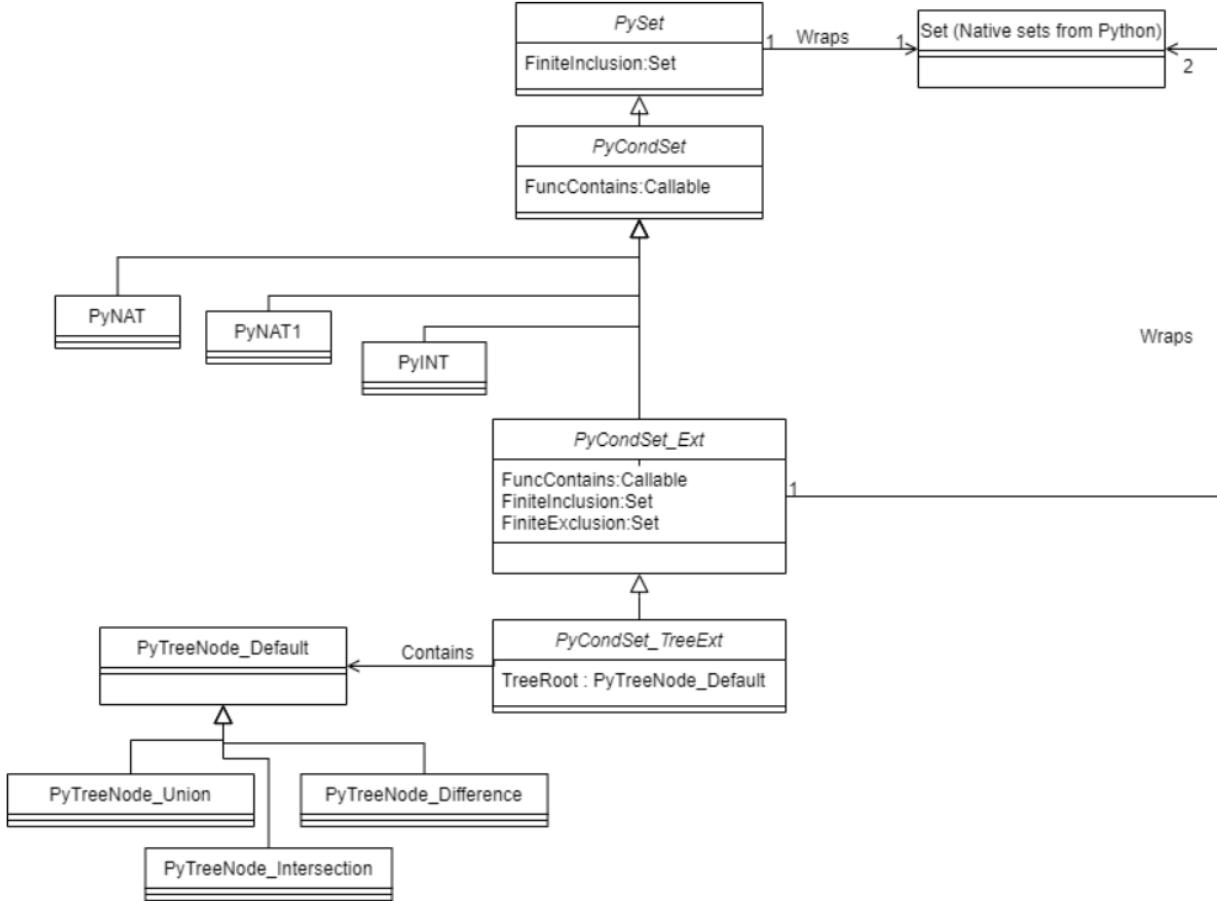


Figura 4.2: Arquitectura de los conjuntos PySet en el Preludio.

operaciones básicas de conjuntos para los conjuntos por comprensión, lo cual es completamente factible en Event-B (y que ProB también es capaz de simular en casos sencillos).

Para ilustrar un poco la arquitectura del Preludio relativa a los conjuntos (en la que se aprecian varias cosas de las que acabamos de hablar), ver la Figura 4.2. Para ver en detalle qué operaciones son posibles con las clases que se introdujeron en este capítulo, ver la Figura 4.3.

## 4.3. Relaciones y Funciones Infinitas

### 4.3.1. Relaciones Infinitas

En esta sección expondremos otra idea original de este traductor, los *PyCondRel*. Como probablemente ya lo están intuyendo, es la misma idea del *PyCondSet*, pero para relaciones. En este caso, en

| Operaciones de los PySet |  |  |   |
|--------------------------|--|--|---|
| Clase                    | Operaciones Soportadas   | Operaciones Soportadas Parcialmente  | Operaciones No Soportadas   |
| PySet                    | Todas las que se relacionan a conjuntos, expuestas en el capítulo anterior.  |  |   |
| PyCondSet                | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>PyUnion, PyIntersection,<br>PyDifference | <code>__len__</code> , <code>__iter__</code> , <code>__eq__</code> ,<br><code>__hash__</code> , PyCartesianProduct,<br>PyIsSubset, PyNotSubset,<br>PyIsProperSubset,<br>PyNotProperSubset, PyFinite  | PyPartition, PyPowerSet,<br>PyPowerSet1, PyChoice   |
| PyINT, PyNAT,<br>PyNAT1  | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>PyUnion, PyIntersection,<br>PyDifference | <code>__len__</code> , <code>__iter__</code> , <code>__eq__</code> ,<br><code>__hash__</code> , PyCartesianProduct,<br>PyIsSubset, PyNotSubset,<br>PyIsProperSubset,<br>PyNotProperSubset, PyFinite,<br>PyPowerSet, PyPowerSet1,<br>PyChoice | PyPartition   |
| PyCondSet_Ext            | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>PyUnion, PyIntersection,<br>PyDifference | <code>__eq__</code>  | <code>__len__</code> , <code>__iter__</code> , <code>__eq__</code> ,<br><code>__hash__</code> , PyCartesianProduct,<br>PyIsSubset, PyNotSubset,<br>PyIsProperSubset,<br>PyNotProperSubset, PyFinite,<br>PyPartition, PyPowerSet,<br>PyPowerSet1, PyChoice |
| PyCondSet_TreeExt        | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>PyUnion, PyIntersection,<br>PyDifference | <code>__eq__</code>  | <code>__len__</code> , <code>__iter__</code> , <code>__eq__</code> ,<br><code>__hash__</code> , PyCartesianProduct,<br>PyIsSubset, PyNotSubset,<br>PyIsProperSubset,<br>PyNotProperSubset, PyFinite,<br>PyPartition, PyPowerSet,<br>PyPowerSet1, PyChoice |

Figura 4.3: Operaciones soportadas por los distintos PySet.

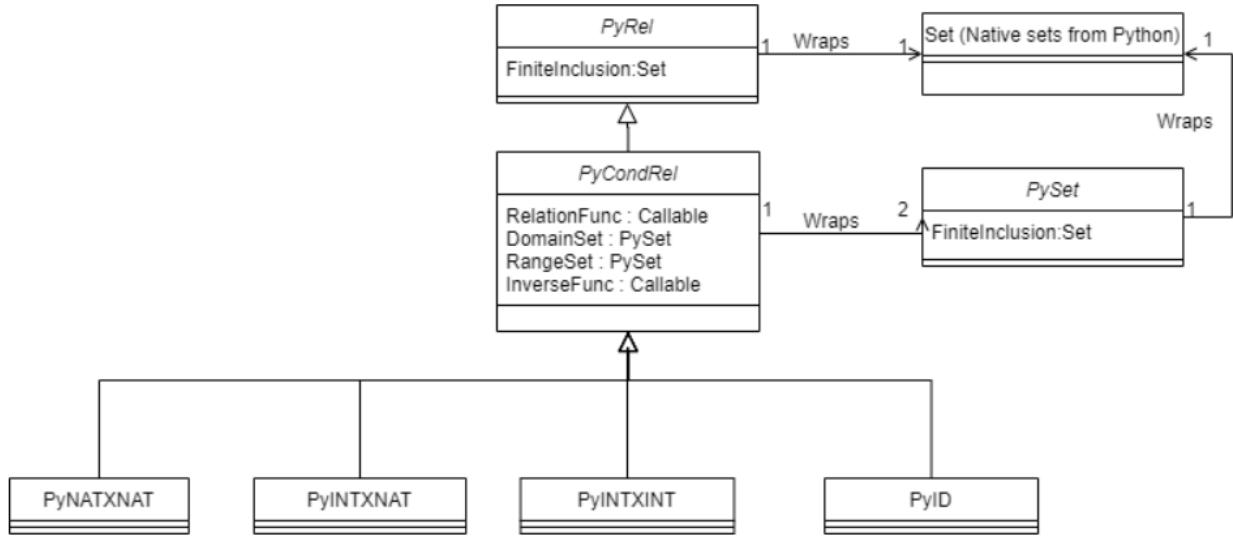


Figura 4.4: Arquitectura de las relaciones *PyRel* en el Preludio.

vez de especificar una condición (predicado) de pertenencia como en los conjuntos por comprensión, lo que se hace es indicar como parámetro una función que permita obtener la imagen relacional de la relación que se busca representar. Por ejemplo, si se quiere representar el conjunto de las tuplas de números enteros (producto cartesiano de enteros a enteros), la función para obtener la imagen relacional de dicha relación recibiría un conjunto de enteros y siempre retornará todos los enteros (o sea, retorna *PyINT*). La razón es que la imagen de cualquier elemento de esa relación serán todos los números enteros, ya que el producto cartesiano empareja todos los elementos del dominio con todos los elementos del rango. Gracias a esta funcionalidad se pueden soportar las siguientes relaciones que tienden a aparecer en modelos en Rodin: NAT x NAT (clase *PyNATXNAT* del Preludio), INT x INT (clase *PyINTXINT* del Preludio), INT x NAT (clase *PyINTXNAT* del Preludio), y los conjuntos potencia (y conjunto potencia no vacío) de las relaciones recién mencionadas. También se usó la clase *PyCondRel* para crear la clase *PyID*, que representa la relación insignia de la identidad en Event-B (y la cual soporta las operaciones de restricción de dominio/rango, las cuales son las operaciones para las que normalmente se usa la identidad en Event-B). Al igual que con la clase *PyCondSet*, si no se especifican sus parámetros, cualquier método que se llame y requiera dicho parámetro generará una excepción. Los otros tres parámetros de la clase *PyCondRel* son el dominio y el rango de la relación que se busca representar (los parámetros son de tipo *PySet*), y la función inversa de la imagen relacional.

Para ilustrar un poco la arquitectura del Preludio relativa a las relaciones, ver la Figura 4.4.

Para ver en detalle qué operaciones son posibles con las clases que se introdujeron en este capítulo, ver la Figura 4.5.

| Operaciones de los PyRel              |  |  |  |
|---------------------------------------|--|--|--|
| Clase                                 | Operaciones Soportadas   | Operaciones Soportadas Parcialmente  | Operaciones No Soportadas  |
| PyRel                                 | Todas las que se relacionan con relaciones y conjuntos, expuestas en el capítulo anterior.   |  |  |
| PyCondRel                             | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>Imagen Relacional, Aplicación  | Todas las operaciones vinculados a relaciones y conjuntos que no se hayan mencionado en las otras columnas de esta misma fila. | PyPartition, PyPowerSet,<br>PyPowerSet1, PyChoice  |
| PyINTXINT,<br>PyNATXNAT,<br>PyINTXNAT | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>Imagen Relacional, Aplicación,<br>PyFinite. También<br>PyIsWellDefined, PyIsInjection<br>que son funciones internas que se usan para el funcionamiento de otras funciones.             | Todas las operaciones vinculados a relaciones y conjuntos que no se hayan mencionado en las otras columnas de esta misma fila. | <code>__len__</code> , PyPartition   |
| PyID                                  | <code>__init__</code> , <code>__str__</code> , <code>__repr__</code> ,<br>PyContains, PyNotContains,<br>PyFinite, Inversa. También<br><b>PyDomainRestriction</b> y<br><b>PyRangeRestriction</b> que es el principal uso que se le da a la identidad en Rodin (según la documentación oficial). | PyIntersection   | Todas las operaciones vinculados a relaciones y conjuntos que no se hayan mencionado en las otras columnas de esta misma fila. |

Figura 4.5: *Operaciones soportadas por los distintos PyRel.*

| n      | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  | 8  | 9  | ... |
|--------|---|---|---|---|---|---|---|----|----|----|-----|
| fib(n) | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | ... |

Figura 4.6: Serie de Fibonacci indexada.

#### 4.3.2. Funciones Infinitas

Representar una función como un conjunto de tuplas no suena muy útil en todas las situaciones, por ejemplo, si un usuario crea una constante en un modelo que representa una función de enteros que mapea los elementos del dominio al mismo elemento pero multiplicado por 2, no suena completamente atractivo que el generador de código se tenga que limitar a un conjunto limitado de tuplas que refleje ese comportamiento. Suena más interesante si el código generado permite realmente especificar una función que reciba un número y lo multiplique por 2. Esta es una funcionalidad que no existe en ningún otro traductor en el ámbito de Event-B.

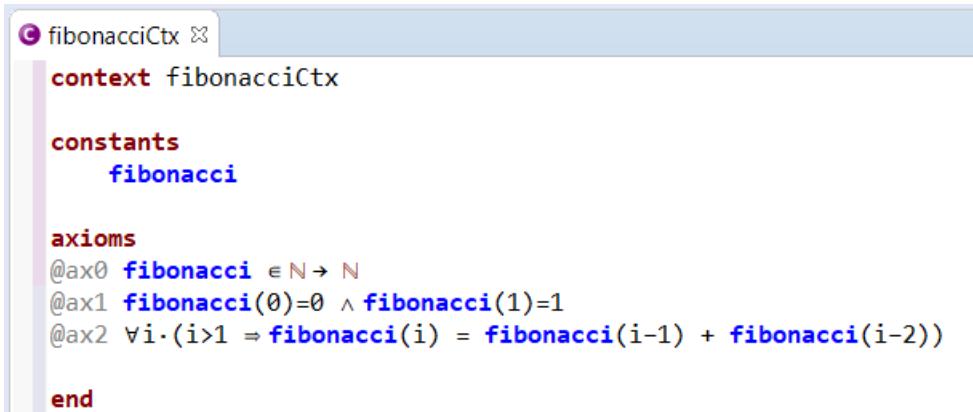
Lograr lo anterior es posible con los *PyCondRel*. Simplemente hay que crear un objeto de dicha clase que reciba como parámetro la función que se busca representar (hay que recordar que la función que recibe como parámetro la clase *PyCondRel* está pensada para ser la función que retorna la imagen relacional de la función, entonces si se quiere representar una función como el algoritmo de ordenamiento, se debe pasar como parámetro la función de ordenamiento pero que reciba un conjunto de arreglos, y devuelva el mismo conjunto de arreglos pero ordenados).

Un ejemplo para facilitar el entendimiento de lo anterior vale más que mil palabras: Supongamos que queremos crear una constante que represente la serie de Fibonacci, en otras palabras, una función que recibe un número natural “n” y retorna el valor que tiene la serie de Fibonacci en dicha posición. Ver en la Figura 4.6 la serie de Fibonacci.

Representar la serie de Fibonacci como una relación usando la clase *PyRel* no es muy factible, ya que la serie de Fibonacci es infinita. Pero afortunadamente tenemos una alternativa que solo se ofrece en nuestro generador de código, la cual es, usar la clase *PyCondRel* para representar funciones infinitas (infinitas en el sentido en que representarlas como un conjunto de tuplas no es posible o tomaría mucha memoria). ¿Cómo se hace exactamente? Ya lo veremos.

Lo primero sería crear un contexto que tenga una constante que busque representar de manera abstracta la función de Fibonacci. Nos tomamos el trabajo de hacerlo, ver la Figura 4.7.

El siguiente paso es usar el traductor para generar código Python que represente ese contexto, lo cual hicimos (el código completo se encuentra en <https://github.com/juanfesba/EB2PY/tree/>



```

fibonacciCtx
context fibonacciCtx

constants
    fibonacci

axioms
@ax0 fibonacci ∈ N → N
@ax1 fibonacci(0)=0 ∧ fibonacci(1)=1
@ax2 ∀i.(i>1 ⇒ fibonacci(i) = fibonacci(i-1) + fibonacci(i-2))

end

```

Figura 4.7: Contexto en el que se especifica de manera abstracta la función de la serie de Fibonacci.

[master/Examples/EB2PYModels/Fibonacci](#)). Ahora viene la parte interesante: Como sabemos, el traductor genera una clase que representa el contexto, pero a la hora de crear un objeto de dicho contexto e inicializarlo, tendría uno que tener mucha suerte para que el método *PyRandValGen* genere casualmente la función de Fibonacci como un conjunto de tuplas. Afortunadamente, nuestro traductor ofrece la opción al usuario de especificar la inicialización del contexto. Es más, ya que podemos elegir el valor que puede tomar nuestra constante, podemos pasar como parámetro un objeto de la clase *PyCondRel* (que no genera problemas de tipado al ser hija de la clase *PyRel*).

¿Pero qué parámetro debo pasarle al *PyCondRel* para que represente la función de Fibonacci? La respuesta es directa, le pasamos como parámetro la función de Fibonacci (adaptada para retornar una imagen relacional). Ya nos tomamos el trabajo de hacerlo (como un método de la clase del contexto para hacerla recursiva y que el código sea más legible), veámoslo en la Figura 4.8.

Con lo visto hasta el momento es suficiente para tener como atributo del contexto a la función de Fibonacci (literalmente), la cual funcionaría y se ejecutaría perfectamente si uno quisiera. Pero podemos ir más allá, ya que, si dejamos las cosas así, tocaría desactivar el diseño por contrato. La razón es que uno de los axiomas está exigiendo verificar que la función de Fibonacci sea efectivamente una función de enteros a enteros, lo cual requiere que sepamos el dominio y el rango de la función de Fibonacci. El inconveniente específicamente es, como lo mencionamos con anterioridad, que la clase *PyCondRel* requiere que uno especifique el dominio y el rango de la relación infinita si se quiere hacer uso de ellos, de lo contrario, se generará una excepción (nuevamente, esto es solo necesario para hacer cumplir los *contratos* de los axiomas, algo que se puede obviar si el objetivo es simplemente tener un código ejecutable y funcional, ya que siempre hemos asumido que se demostró la correctitud del modelo en Rodin, y en teoría el cumplimiento de los *contratos* ya debería de estar asegurado).

```
def fibonacci_func(self, pyset_fibb : PySet) -> PySet:
    ans : Set = set()
    for fibb in pyset_fibb:
        if fibb==0:
            ans.add(0)
        elif fibb==1:
            ans.add(1)
        else:
            ans.add(self.fibonacci(fibb-1) + self.fibonacci(fibb-2))
    return PySet(ans)
```

Figura 4.8: Función que usa como parámetro la clase *PyCondRel* para representar la función de Fibonacci.

Para ir entonces más allá, lo que se puede hacer es justamente lo que los *contratos* nos están exigiendo, especificar el dominio y el rango de la función de Fibonacci. El dominio es muy sencillo, al ser la función de Fibonacci una función total, sabemos que el dominio serán simplemente todos los naturales. En otras palabras, simplemente un objeto de la clase *PyNAT()*. El rango es más difícil de deducir, necesitamos generar un conjunto infinito que tenga únicamente los números de la serie de Fibonacci. Afortunadamente, tenemos la clase *PyCondSet* para representar conjuntos infinitos. Un *PyCondSet* que permita representar la serie de Fibonacci y cumplir los *contratos* nos exige dos cosas: (i) Una función/condición/predicado de pertenencia. (ii) Y un iterador para que el contrato pueda recorrer el conjunto pues tenemos un cuantificador en el axioma (recordar que este iterador buscaría generar un conjunto finito que represente el conjunto infinito de los números de Fibonacci para poder recorrerlo, similar a los iteradores de la clase *PyNAT* o *PyINT*); A continuación, en la Figura 4.9 podemos ver el código que resume la implementación de todo lo que se habló, incluyendo el llamado a la clase del contexto que fue generado con nuestro traductor, e inicializando el objeto del contexto con la función de Fibonacci (literalmente) como parámetro.

Gracias a la clase *PyCondRel*, pudimos traducir un código que representaba un modelo en el cual se quería representar la función de Fibonacci. Otros traductores habrían permitido únicamente generar un código cuya función de Fibonacci sería representada como un conjunto finito de tuplas (algo que también se puede en nuestro traductor, pero no es lo ideal, y por eso ofrecemos una forma más robusta de representar funciones infinitas). Gracias también a que pudimos representar el dominio y el rango con la clase *PyCondSet* y la clase *PyNAT*, no fue necesario desactivar el diseño por contrato. Es más, efectivamente, con el código de la última Figura referenciada, todos los *contratos* se cumplieron (*Mypy* tampoco detectó problemas de tipado). Finalmente, veamos la Figura 4.10 para ver nuestro código en acción.

Para testear esta funcionalidad de nuestro traductor también creamos otro sistema en Rodin que modela el algoritmo Dijkstra (el de encontrar la distancia mínima de un nodo a los demás nodos

```
def fibonacci_range(check_num : int) -> bool:
    if check_num == 0 or check_num == 1:
        return True
    previous = 1
    last = 2
    while(last<check_num):
        previous, last = last,previous+last
    return last==check_num

class PyFIB_Iter(PyBaseIter):

    def __iter__(self) -> Iterator:
        self.boundary : int = 100
        self.previous : int = -1
        self.last : int = 1
        return self

    def __next__(self) -> int:
        if P.USEFINITE_SPECIALSETS():
            self.previous,self.last = self.last,self.previous+self.last
            if self.last > self.boundary:
                raise StopIteration
            return self.last
        raise Exception("You cannot traverse an infinite set.")

P.setUSEFINITE_SPECIALSETS(True)
P.setRAND_INTRANGE((-2,20))
c = fibonacciCtx_class()
c.checkedInit(PyCondRelgetattr(c,"fibonacci_func"),PyNAT(),PyCondSet(fibonacci_range,PyFIB_Iter())))

```

Figura 4.9: Código auxiliar para que el contexto que representa la función de Fibonacci pueda cumplir los contratos.

```
>>>
>>> c.fibonacci(0)
0
>>> c.fibonacci(1)
1
>>> c.fibonacci(2)
1
>>> c.fibonacci(3)
2
>>> c.fibonacci(4)
3
>>> c.fibonacci(5)
5
>>> c.fibonacci(6)
8
>>> c.fibonacci(7)
13
>>> c.fibonacci(20)
6765
>>> c.fibonacci(30)
832040
>>> |
```

Figura 4.10: *Ejemplo de la ejecución de la función de Fibonacci modelada. Aquí se está usando el objeto que creamos con la clase PyCondRel para calcular los números de Fibonacci, lo cual se logra usando (literalmente) la función de Fibonacci internamente.*

en un grafo), en el cual representamos la estructura de datos *Heap* como una lista que se ordena con una función de ordenamiento. Esta última se representó como una función infinita usando la clase *PyCondRel*, lo cual es mucho más ideal que generar un conjunto de tuplas donde cada tupla tiene una lista desordenada a la izquierda, y esa misma lista pero ordenada a la derecha (algo que sería muy poco práctico). En dicho modelo nos saltamos los *contratos* del contexto, ya que el objetivo era demostrar que al ejecutar la máquina efectivamente se generara el efecto del algoritmo de Dijkstra (y así fue). Aunque hay más detalles en el capítulo de *Resultados*, no sobra mencionar que efectivamente se pudo modelar el algoritmo de Dijkstra haciendo uso del algoritmo de ordenamiento, y lo anterior, a través de incluir directamente el algoritmo de ordenamiento como parámetro.



# Detalles de Implementación y Uso del Traductor

---

En este capítulo veremos algunos detalles de la implementación del traductor, y también hablaremos un poco sobre sugerencias de cómo hacer uso del código generado.

## 5.1. Buscando Legibilidad del Código Generado

Cuando se construye un traductor de lenguajes, se debe buscar que el código generado sea lo más legible y menos desordenado posible. Esta tarea no es trivial, puesto que el traductor está pensado para ser genérico y ser capaz de traducir la mayor cantidad de modelos posibles. Por esta razón, en la implementación del traductor se buscó que absolutamente cualquier componente importante que se tradujera tuviera una sección fácilmente identificable en el código generado, para ello, se generan comentarios que le indican al usuario para qué sirve cada porción de código. Esto ya lo hemos visto múltiples veces en capítulos anteriores, por ejemplo, si se van a definir las constantes de un contexto, antes de declararlas hay un comentario que indica que a continuación estarán las constantes. También hay comentarios que dan metainformación al usuario del código generado, por ejemplo, en una máquina hay comentarios indicando si la misma está refinando otra máquina, o si está “viendo” un contexto.

Además de lo anterior: los axiomas, invariantes, constantes, enumeraciones y variables, se van agregando al código generado en orden alfabético. Considerando que el AST de Rodin no mantiene el orden en que esos componentes se definen en Rodin, era importante buscar una manera de organizarlos, así al usuario no le queda tan difícil buscar dichas componentes en el código generado.

Otra manera en que se buscó mejorar la legibilidad del código fue alterando el nombre de varias componentes (ej. De los axiomas, las invariantes, los eventos, los contextos, entre otros). Lo que se hace es que se concatena un identificador a la cadena del nombre (ej. `_axiomCheck` para los axiomas). Esto ayuda al usuario a identificar los componentes del código generado, además de evitar conflictos en el nombramiento de identificadores. Por ejemplo, y como se mencionó, sin esta funcionalidad no se podrían nombrar las *enumeraciones* o los contextos como “PySet”, pues se sobrescribiría la clase PySet del Preludio.

Una gran ventaja de haber hecho el traductor hacia Python es que la legibilidad del código generado se pudo mejorar mucho gracias a que se pudo trabajar con los *métodos mágicos* de Python. Los métodos mágicos de Python permiten que uno pueda implementar la operación que se ejecuta cuando se usa la sintaxis especial de Python, por ejemplo, la palabra reservada *len* se usa para obtener el tamaño de un objeto -cuando tiene sentido que dicho objeto tenga un tamaño-, así, si uno usa el método mágico *len* en una lista o un conjunto nativos de Python, Python retornará la cantidad de elementos de dicho objeto. Veamos ahora específicamente cómo usamos los métodos mágicos en nuestro traductor para mejorar la legibilidad del código generado:

- Digamos que queremos saber la cardinalidad de un conjunto (llámémoslo el conjunto (*PySet*) “S”). En vez de vernos forzados a hacer algo tipo: *S.PyCardinality()*, gracias los métodos mágicos de Python podemos simplemente hacer: *len(S)*. Esto se logra porque la cardinalidad se implementó sobrescribiendo el método mágico *len*.
- Como ya se vio, para ver el estado de las máquinas se sobrescribieron los métodos mágicos *\_\_str\_\_* y *\_\_repr\_\_*. Los cuales, por ejemplo, si tenemos un contexto “ctx”, podemos ver su estado en consola llamando *print(ctx)*, o incluso simplemente escribiendo en consola el nombre de la variable *ctx*. Los métodos *\_\_str\_\_* y *\_\_repr\_\_* se usaron en nuestro traductor para todas las clases que creamos, como los *PySet*, los *PyRel*, los *PyFamilies*, entre otros. Y por lo tanto, se puede imprimir fácilmente su representación.
- Nuestro traductor también hizo uso del método mágico *\_\_eq\_\_*, que permite revisar igualdades con la operación nativa de Python *==*.
- También sobrescribimos el método mágico *\_\_hash\_\_*. Esto sirve por ejemplo para que Python permita crear conjuntos de conjuntos, ya que un conjunto sólo puede tener objetos a los que se les pueda calcular su *hash*. Por ejemplo, en el caso de la clase *PySet*, el hash se obtiene con la estructura de datos nativa de Python llamada *frozenset* (un conjunto inmutable).
- También sobrescribimos el método mágico *\_\_iter\_\_*, esto se hizo principalmente para facilitar la codificación del traductor, pero también permite cosas como: si uno quisiera recorrer el conjunto “S” del contexto “ctx”, el método mágico *\_\_iter\_\_* permite que uno pueda hacerlo nativamente a través de la palabra reservada *for* con la instrucción “*for element in ctx.S:*”, una operación natural de Python, lo que lo hace completamente legible.
- En Rodin uno puede obtener la imagen relacional y la aplicación funcional de una relación con una sintaxis muy sencilla: *r[S]* y *f(x)* respectivamente (siendo “r” una relación, “S” un conjunto, “f” una función, y “x” un elemento del dominio de “f”). Lo fascinante es que en nuestro código generado se usa exactamente la misma sintaxis (algo que difícilmente se podría lograr en otro lenguaje de programación), gracias a que sobrescribimos los métodos mágicos *\_\_getitem\_\_* y *\_\_call\_\_*.
- Finalmente, la inversa de una relación en Rodin se escribe como *R~* siendo “R” una relación. En nuestro código generado lo hacemos de una manera súper similar: *~R* (gracias a la sobrescritura del método mágico *\_\_invert\_\_*).

Otra característica de nuestro traductor es que se buscó que cada axioma, invariante, guarda, acción, variante, entre otros, se pudieran traducir en una sola línea como se hace en Rodin. Esto fue un reto en algunos casos, en los cuales se tuvo que hacer uso de los *lambda* de Python.

## 5.2. Buenas Prácticas

El código que generamos con nuestro traductor busca seguir buenas prácticas de la ingeniería de software. Particularmente, lo que se hizo para que el código generado tuviera buenas prácticas fue encapsular el código de salida en diferentes archivos, clases, y dependencias. Además, se buscó la encapsulación de la programación orientada objetos, donde todos los atributos son privados y tienen sus métodos “get” y “set”. Más aún, para las constantes y variables se usó el decorador `@property`, que es un decorador nativo de Python para definir el acceso y modificación de los atributos de una clase.

De igual manera, se buscó optimizar el código en lo posible. Por ejemplo, el tipo de dato de los identificadores se necesita en distintas etapas de la traducción, y para obtener el tipo de dato de un identificador se requiere un proceso de *parsing*. Pero ello solo es necesario hacerlo una vez porque la traducción se guarda en el objeto que lo requiere. Otro ejemplo es en los cuantificadores, donde necesitábamos una manera de generar todas las combinaciones de valores posibles que tuvieran que evaluarse en el predicado del cuantificador. Para generar las combinaciones posibles se usaron ideas de los algoritmos combinatorios más reconocidos, lo cual ayudó a optimizar este procedimiento.

Finalmente, una de las buenas prácticas que se usaron en nuestro traductor es la de generar excepciones cuando se necesite, deteniendo el código cuando se identifique un problema. Pero, además, la excepción se acompaña de un mensaje que permite identificar que fue lo que causó la excepción (esto ocurre tanto en el código del traductor, como en los códigos generados por el traductor).

## 5.3. Retos de la Implementación

Además de todos los retos técnicos que se causan al tener que buscar una manera de traducir las indeterminaciones y el infinito, hubo otros retos técnicos inherentes a todo este trabajo. El primer reto, es que todas las operaciones de Rodin retornan una nueva copia de los objetos implicados, por ejemplo, si uno quiere unir un conjunto a otro, la operación debe retornar un nuevo conjunto. Asegurarse de que todas las operaciones tuvieran este efecto fue un obstáculo a la hora de implementar el Preludio.

Usar el AST de Rodin como base para nuestro traductor ofrece muchas ventajas como se ha mencionado, pero también plantea muchos retos. El uso correcto de las librerías de Rodin requiere que uno conozca muchos detalles técnicos de la implementación de Rodin. Por ejemplo, la interfaz

de la clase Visitor que usamos para recorrer el AST de Rodin requirió la implementación de 25 métodos abstractos, cada uno con sus propias cualidades y elementos.

Finalmente, el AST de Rodin no tiene un orden a la hora de extraer los contextos y máquinas de un proyecto. Esto es problemático porque en un proyecto de Rodin puede haber múltiples modelos, donde cada uno tiene sus propios contextos y máquinas con relaciones entre ellos. El problema recae en que no se puede traducir un contexto concreto antes de traducir el contexto abstracto que está extendiendo (porque habría identificadores que no se han traducido pero que el contexto concreto necesita). Por esta razón tuvimos que desarrollar un algoritmo que organizara las máquinas y contextos de modo que se puedan traducir en orden sin generar problemas de dependencia.

## 5.4. Instalación del Traductor

Instalar este traductor es supremamente sencillo. Lo único que hay que hacer es instalar Rodin (v3.4 sugerida), y posterior a eso, lo que hay que hacer es descargar el archivo .jar del Plug-in de este proyecto, el cual se puede encontrar en el repositorio de este proyecto en el siguiente enlace: <https://github.com/juanfesba/EB2PY>. Una vez descargado, sólo hay que copiar el archivo .jar en la carpeta plugins de Rodin. ¡Eso es todo!

## 5.5. Uso del traductor

Usar este traductor es muy sencillo. Una vez se tiene el traductor instalado y vinculado a Rodin, lo único que hay que hacer es presionar el botón “EB2PY” en el menú de la parte superior de Rodin, como se ve en la Figura 5.1. Luego de presionar el botón se despliega una lista con una sola entrada llamada “Sample Command”. Al presionar dicha entrada empezará a ejecutarse el traductor, que luego de procesar TODOS los proyectos que estén abiertos, generará una carpeta por cada proyecto en la carpeta de salida llamada Output\_EB2PY (localizada en el mismo directorio del espacio de trabajo de Rodin).

Respecto a la velocidad de traducción, cuando testeábamos el traductor, tomaba aproximadamente 2 segundos la traducción de 4 proyectos (que en conjunto tenían unos 40 componentes entre contextos y máquinas). Luego de la primera ejecución, volver a ejecutar el traductor es prácticamente instantáneo (gracias a la caché). Cuando la traducción se completa (luego de presionar el botón), sale un mensaje que indica el éxito de la traducción, como se ve en la Figura 5.2.

Una vez se traduzcan los proyectos, si se quiere hacer uso de un contexto o máquina, lo que hay que hacer es crear una instancia de la clase del contexto o máquina que nos interese. Si es un contexto, toca llamar al método *checkedInit* para inicializarlo (recordar que se puede especificar los parámetros de este método, o dejar que el Preludio trate de generar valores apropiados), si es una máquina solo hay que crear el objeto porque la inicialización se hace en el constructor de la

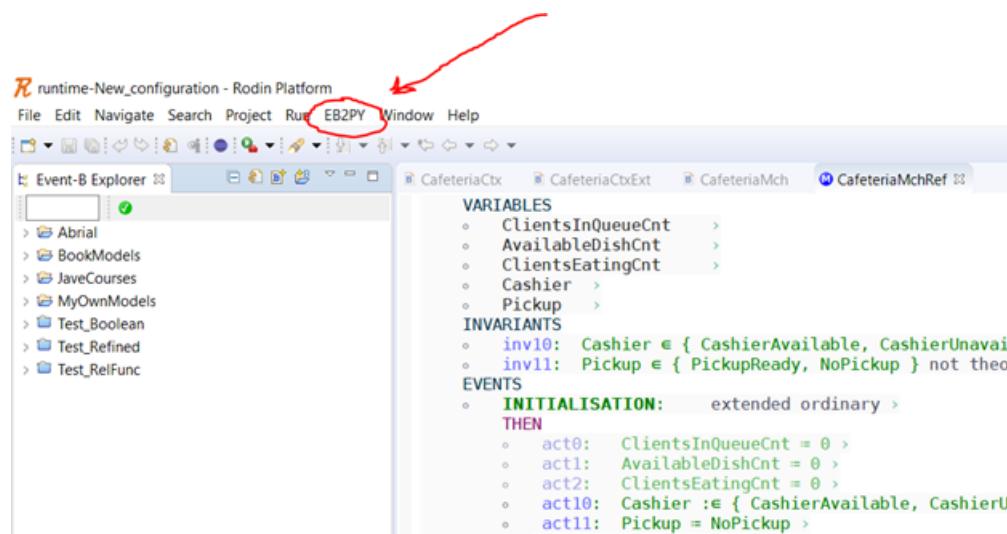


Figura 5.1: Visualización del botón para activar el traductor EB2PY en Rodin una vez instalado.

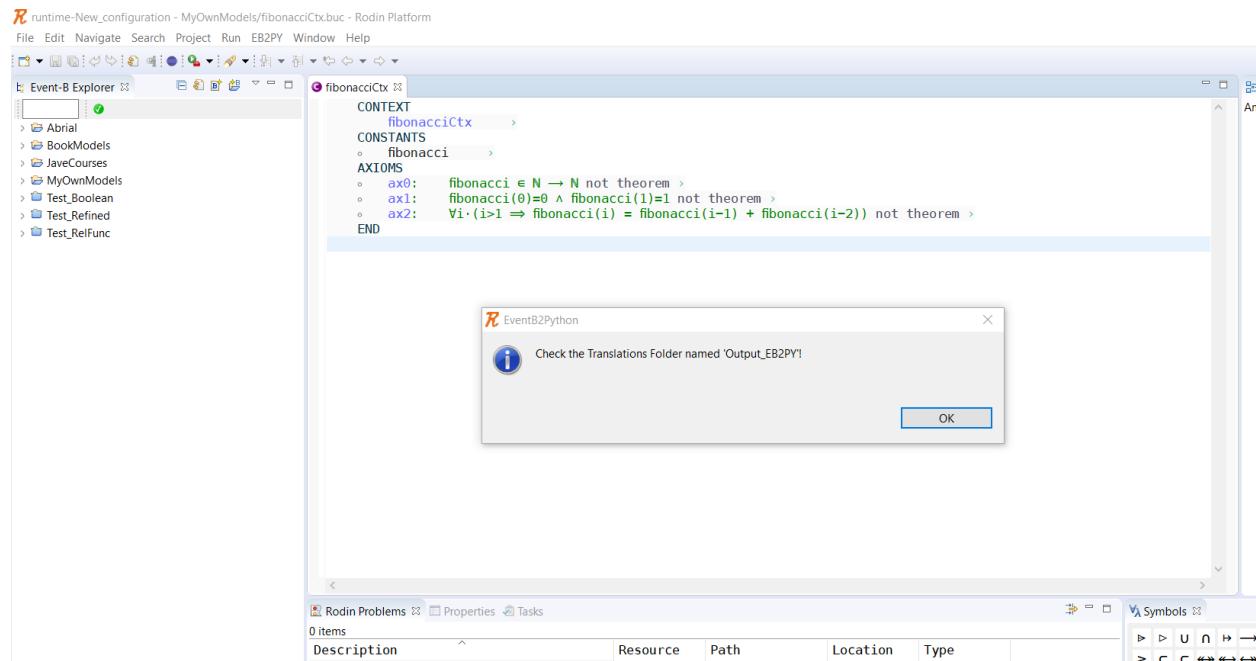


Figura 5.2: Mensaje que se genera luego de que el traductor EB2PY termina de ejecutarse.

```

1  from ArrPartMch_mch import *
2  c = ArrPartCtx_class()
3  c.checkedInit(10,5)
4  m = ArrPartMch_class(c)

```

Figura 5.3: *Ejemplo de instanciación de una máquina que “ve” un contexto para código generado por nuestro traductor.*

clase. Una vez creado el objeto se pueden usar libremente los métodos que se ofrecen. Para las máquinas se recomienda el método *PyAutoExecute*, que permite ver fácilmente el comportamiento de la máquina que se tradujo. Un ejemplo de cómo crear un objeto de una máquina que ve un contexto (al que le parametrizamos sus valores) se ve en la Figura 5.3.

A parte de lo anterior, extender el código generado se puede hacer libremente (por ejemplo, la clase de un contexto funciona como una clase cualquiera, pues se le pueden añadir atributos o métodos de manera natural). Sin embargo, hay que tener cuidado cuando lo que se quiere es específicamente seguir añadiendo componentes de Event-B que se comporten en concordancia con los contratos que ya hay. Por ejemplo, si en un contexto se quieren añadir atributos que sean constantes del modelo, se deben inicializar dichos atributos como todas las demás constantes del modelo, en otras palabras, se debe editar el método *checkedInit* e inicializar dichas constantes ahí, además de proveer métodos “set” y “get” similares a los de las otras constantes (similar ocurriría con las variables de una máquina). Si lo que se quiere es añadir axiomas, se debe crear un método para dicho axioma, pero también se debe añadir el nombre del método en la lista de cadenas (*allAxioms.local*) del método *checkAllAxioms* que guarda los nombres de los métodos que representan axiomas, de ese modo es que realmente se verificará el cumplimiento de los nuevos axiomas al inicializar las constantes del contexto (para las invariantes es similar, donde la lista de cadenas se encuentra en el método *checkAllInvariants*, y se hace un trabajo parecido en los métodos *PyGuardsState* y *PyAutoExecute* si se quiere que los nuevos elementos sean soportados por estos dos métodos para el usuario). En síntesis, extender el código generado no es un limitante, pero se debe tener en cuenta que hay que mantener la arquitectura de los contextos y las máquinas (que se explicó en el capítulo de *Traducción*), cuando lo que se quiere es añadir nuevos componentes que hagan parte del paradigma de los modelos de Event-B (como axiomas, invariantes, etc.).

(Para Windows) Finalmente, para revisar el tipado, se debe instalar *Mypy*, luego en la consola ir al directorio del archivo que se quiere revisar. Y finalmente ejecutar el comando `$mypy <nombredelarchivo>`. El traductor ya habrá generado automáticamente los archivos necesarios para que *Mypy* identifique correctamente las dependencias del Preludio y se haga una revisión correcta del código generado. Si uno quiere extender el código generado, una manera fácil de hacerlo es hacer las ediciones al final de los archivos editados, así *Mypy* seguirá funcionando y revisando las nuevas

ediciones que introduzca el usuario sin necesidad de crear archivos que permitan leer correctamente las dependencias de los archivos.



# CAPÍTULO 6

# Resultados

---

En este capítulo veremos todas las acciones que se tomaron para probar el funcionamiento del traductor.

Lo primero que se hizo para verificar su funcionamiento fue el de traducir todas las operaciones y componentes por separado, que se describen en los capítulos de *Reglas de Traducción y No Determinismo y el Infinito*. La idea de esta verificación era asegurarnos de que el traductor pudiera efectivamente generar código de todo aquello que se especificó debería ser capaz de traducir. Esto incluye también pruebas de modelos con refinamientos y extensiones.

Luego de eso lo que hicimos fue traducir 15 modelos distintos (sin contar los modelos que se usaron para probar todas las operaciones por separado). Los modelos se obtuvieron de diferentes fuentes:

- 5 modelos en un artículo científico de Abrial (el creador de Event-B) [Abr01]. Lo interesante de los sistemas de este artículo es que modelan programas completos, sin indeterminaciones, en otras palabras, programas con las mismas características de los sistemas que se modelan en la vida real luego de un proceso muy largo de refinamiento constante de máquinas y contextos. Claramente, son modelos simples en comparación a los modelos complejos que se usan en aplicaciones reales, pero sirven como perfecto ejemplo para estudiar modelos concretos. Otro aspecto para mencionar es que los modelos del artículo se construyen usando Atelier-B (herramienta para el uso operacional del B-method). Por esta razón se les hacen algunas modificaciones a dichos modelos, de modo que se adapten a la sintaxis de Event-B y a las explicaciones que queremos dar (le recordamos al lector que pueden acceder al código de estos ejemplos en <https://github.com/juanfesba/EB2PY/tree/master/Examples>).
- 4 modelos usados en cursos universitarios: Ya que teníamos acceso a modelos usados en universidades, sería muy interesante ver si el traductor puede efectivamente generar código de dichos modelos.
- 4 modelos de dos libros importantes en el contexto de Rodin: *Rodin User's Handbook* [Jas12] y *System Modelling & Design Using Event-B* [Rob10].
- 2 modelos que construimos para mostrar la traducción de funciones “infinitas”.

```

C SearchArrCtx ✎ M SearchArrMch
context SearchArrCtx
constants
  n
  f
  s
axioms
  @ax0 n in N1
  @ax1 s in P(N)
  @ax2 f in 1..n implies s
end

```

Figura 6.1: Contexto del modelo *SearchArrCtx*.

En las siguientes secciones veremos detalles de la traducción de estos modelos. Por cada sección estudiaremos porciones del código de sólo una de las traducciones de dicha sección. Adicionalmente, incluiremos una tabla por sección donde se podrán observar las características principales de las traducciones (por ejemplo, operaciones importantes involucradas). Ya que no mostraremos el detalle de todas las traducciones, no sobra mencionar que todas las traducciones funcionaron y se pudieron ejecutar sin problemas, cumpliendo las funciones para las que estaban pensadas, mientras cumplían todos los *contratos* necesarios. Si se desea ver todos los códigos de todos los modelos y todas las traducciones, visitar la carpeta de “Examples” del repositorio de este proyecto (en el enlace <https://github.com/juanfesba/EB2PY/tree/master/Examples>).

## 6.1. Traducción de Programas Completos

Los modelos que estudiaremos aquí fueron obtenidos de un artículo de Abrial [Abr01] sobre la construcción de modelos que representen programas secuenciales, se tradujeron 5 de los modelos de dicho artículo.

Empecemos por el modelo más sencillo de esta sección “Searching in an Array”, al que nos referiremos como *SearchArr*. Este modelo consiste en un contexto “*SearchArrCtx*” y una máquina “*SearchArrMch*”. La máquina “*SearchArrMch*” “ve” el contexto “*SearchArrCtx*”.

En el contexto de este modelo hay una constante “*f*” que representa un arreglo (ej. *f*(2) devuelve el valor del arreglo en la posición 2, indexado desde 1). En la máquina, al inicializarse, se elige un elemento del rango de “*f*” al azar. Los eventos buscarán el elemento en dicho arreglo, el programa se detendrá cuando lo encuentre, y la respuesta quedará guardada en la variable “*i*”.

Primero, veamos el contexto en Event-B de este modelo en la Figura 6.1.

```
#Axiom Check Methods

def ax0_axiomCheck(self) -> bool:
    return P.NAT1().PyContains(self.n)

def ax1_axiomCheck(self) -> bool:
    return P.NAT().PyPowerSet().PyContains(self.s)

def ax2_axiomCheck(self) -> bool:
    return PyFamilies(PyFamilyTypes.TotalFunctions, PySet({int_range for int_range in range(1, self.n+1)}), self.s).PyContains(self.f)

#End Axiom Check Methods
```

Figura 6.2: Axiomas del código generado por el contexto del modelo *SearchArr*.

Comparemos por ejemplo los axiomas del modelo en Event-B con los axiomas generados por el traductor, que los podemos ver en la Figura 6.2. Como se puede apreciar, el axioma “ax0” requiere verificar que la constante “n” sea un natural sin el 0, lo que se traduce en el *contrato* llamado “ax0\_axiomCheck”, donde se verifica que P.NAT1() (que como lo vimos representa el conjunto de los naturales sin el 0) contenga la constante “n”. El axioma “ax1” es similar, se quiere verificar que “s” (que define el rango de “f”), sea un conjunto de los naturales (o en otras palabras, que pertenezca al producto potencia de los naturales). En el código generado podemos ver que se genera un *contrato* donde se calcula el producto potencia de los naturales, y se verifica que la constante “s” pertenezca a ese conjunto de conjuntos. Esto se permite ya que, como se mencionó en el capítulo de *Reglas de Traducción*, el conjunto potencia de los conjuntos insignia de Event-B están soportados para las operaciones básicas (como la pertenencia), gracias a la clase *PyCondSet*. Finalmente, tenemos un axioma que nos pide que la constante “f” sea una función total, donde su dominio sean los enteros de 1 hasta “n”, y el rango sea el conjunto “s”. Como se menciona en el capítulo de *Reglas de Traducción*, la clase *PyFamilies* se encarga de representar las familias de funciones (o relaciones), y dado el dominio y el rango, puede verificar la pertenencia de una función al tipo de familia que se le haya indicado (en este caso, funciones totales).

Ahora veamos la mayoría del código de la máquina en Event-B en la Figura 6.3. Ahí podemos apreciar que se definen 4 variables, 5 invariantes, y los eventos.

El código que se genera para las acciones de “aprog” se ve en la Figura 6.4. Como el evento “aprog” no tiene parámetros, el código generado tampoco tendrá parámetros. El código generado verificará las guardas, si las cumple, asignará paralelamente el valor de 0 a la variable “d” y el valor de “k+1” a la variable “i”, tal y como ocurre en el evento correspondiente en el modelo de Event-B.

Finalmente, para probar el código, lo que se hizo fue crear un objeto del contexto pasándole el parámetro necesario para crear el arreglo [70,60,10,30,40,20,50,90,80]. Luego se creó un objeto de la máquina pasándole como parámetro dicho contexto. La máquina eligió un número del rango de “f” al azar, el número que el modelo se supone debería encontrar (particularmente el número 40

```
C SearchArrCtx  M SearchArrMch ✘
machine SearchArrMch sees SearchArrCtx

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:=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
```

Figura 6.3: Porción del código de la máquina en Event-B del modelo SearchArr.

```

144 ▼  def aprog_eventActions(self) -> None:
145
146     attempt_Count : int = 0
147 ▼   while not(self.aprog_eventGuards()):
148         if attempt_Count == P.LOWMAXGENATTEMPTS():
149             raise GuardsViolated("Guards of the Event could not be fulfilled.")
150             attempt_Count += 1
151
152     #Set Parameters as an Attribute.
153
154     #Event Actions
155
156     self.i, self.d = (self.k + 1), 0
157
158     #Check Invariants after the actions are executed.
159     if P.DESIGN_BY_CONTRACT_ENABLED() and not(self.checkAllInvariants()):
160         raise Exception("PostConditions of the Event could not be fulfilled.")
161
162
163 #End Event

```

Figura 6.4: Código generado para representar las acciones del evento “aprog” del modelo SearchArr.

en este test, que se encuentra en la posición 5). La máquina siempre inicializa la variable “i” en 1 (donde eventualmente debería evaluarse en 5, pues ahí está la respuesta). Veamos el estado inicial de la máquina en la Figura 6.5 (la figura está recortada, el conjunto “s” tiene los números del 0 al 100). Posterior a eso, se llama el método PyAutoExecute 20 veces.

Luego de llamar el método PyAutoExecute 20 veces, el modelo empieza a elegir eventos al azar habilitados y empieza a ejecutarlos. Eventualmente, no logra encontrar más eventos habilitados. La razón de lo anterior es que el programa se detuvo porque encontró una respuesta. Afortunadamente, la respuesta es correcta, el último estado registrado por la máquina muestra que la variable “i” es efectivamente el 5, el cual es el índice del número que estábamos buscando. Ver la Figura 6.6.

Enhorabuena, el traductor generó código Python exitosamente, y, además, al ejecutarlo pudimos ver que cumplía la función que se supone debe cumplir. Más aún, si ejecutamos *Mypy* para verificar el tipado del código generado, podremos ver en la Figura 6.7 que *Mypy* afirma que no hay ningún problema de tipos en nuestro programa de Python.

A parte de este modelo se tradujeron 4 más pertenecientes al artículo de Abrial:

- “Searching in a Matrix”: Similar al modelo que acabos de ver, pero este modelo busca un número en una matriz de números naturales.

```

>>> m
###  

SearchArrCtx Constants  

f ==> PyRel({(8, 90), (2, 60), (3, 10), (7, 50), (6, 20), (9, 80), (1, 70), (4, 30), (5, 40)})  

n ==> 9  

s ==> PySet({1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24,  

47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71,  

94, 95, 96, 97, 98, 99, 100})  

###  

Variables  

d ==> 1  

i ==> 1  

k ==> 0  

x ==> 40  

>>> for e in range(20):  

    m.PyAutoExecute()

```

Figura 6.5: Estado inicial de la máquina perteneciente al código Python generado a partir del modelo “Searching in an Array”.

- “Finding Common Number in two Sorted Arrays”: Encuentra un número en común a dos arreglos ordenados (de hecho, este modelo representa el algoritmo más eficiente para esta tarea que se usa en lenguajes de programación de alto nivel).
- “Array Partitioning”: Este modelo tiene un arreglo y lo reorganiza de la siguiente manera, dado un número “x”, todos los elementos de la izquierda deben ser menores a dicho “x”, y todos los de la derecha mayores. Las operaciones se hacen sobre el mismo arreglo.
- “Sorting”: Este es el modelo más complicado que se tradujo de este artículo. Su función es representar el algoritmo burbuja de ordenamiento.

Todos estos modelos se tradujeron exitosamente, de hecho, gracias a que son modelos de programas completos, la función *PyAutoExecute* es suficiente para hacer seguimiento a la ejecución del modelo hasta que el modelo cumple el propósito para el que estaba pensando (por ejemplo, en el modelo que representa el algoritmo de ordenamiento de burbuja, cuando el programa se detiene, efectivamente el arreglo con el que se había empezado queda organizado, esto es algo que encontramos muy interesante).

Otro tema interesante aquí es que estas pruebas nos permitieron encontrar un pequeño inconveniente de nuestro traductor en relación con *Mypy*, detectado únicamente en el modelo “Searching in a Matrix”. A pesar de que el código funciona y se ejecuta sin ningún problema, *Mypy* no es capaz de asemejar un conjunto de tuplas con una relación (esto no ocurre siempre que hayan relaciones, sino en algunos casos cuando se realizan algunas operaciones con las relaciones). Es desafortunado que *Mypy* no logre identificar esta equivalencia en todos los casos, y para arreglarlo habría que refactorizar por completo la clase *PyRel* (y sus clases hijas), además de otros retos técnicos bastante complejos, únicamente con el fin de acoplarse a las exigencias de *Mypy*, que es un logro opcional

```
i ==> 1
k ==> 3
x ==> 40
progress succesfully executed!!!
###
SearchArrCtx Constants
f ==> PyRel({(8, 90), (2, 60), (3, 10), (7, 50), (6, 20), (9, 80), (1, 70), (4, 30), (5, 40)})
n ==> 9
s ==> PySet({1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71
94, 95, 96, 97, 98, 99, 100})
###
Variables
d ==> 1
i ==> 1
k ==> 4
x ==> 40
aprogr succesfully executed!!!
###
SearchArrCtx Constants
f ==> PyRel({(8, 90), (2, 60), (3, 10), (7, 50), (6, 20), (9, 80), (1, 70), (4, 30), (5, 40)})
n ==> 9
s ==> PySet({1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24
47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71
94, 95, 96, 97, 98, 99, 100})
###
Variables
d ==> 0
i ==> 5
k ==> 4
x ==> 40
No event could be AutoExecuted
>>>
```

Figura 6.6: Impresiones en pantalla que demuestran que la máquina que representa el modelo "Searching in an Array" logró llegar al resultado esperado. Particularmente se ve que el número 40 (variable "x") que se estaba buscando en el arreglo (constante "f") se encuentra en la posición 5 (variable "i").

```
D:\Universidad\Tesis\EventB2Python\Output_EB2PY\Py_Preludes>cd ..

D:\Universidad\Tesis\EventB2Python\Output_EB2PY>cd Abrial_Project_0

D:\Universidad\Tesis\EventB2Python\Output_EB2PY\Abrial_Project_0>mypy SearchArrMch_mch.py
Success: no issues found in 1 source file

D:\Universidad\Tesis\EventB2Python\Output_EB2PY\Abrial_Project_0>
```

Figura 6.7: Resultado exitoso de la revisión de tipado que realiza Mypy sobre el modelo ”Searching in an Array” generado en Python.

para este proyecto, y por lo tanto, se deja como trabajo futuro (se puede ver la sección de “Trabajo Futuro” en el capítulo de las *Conclusiones* para más información).

Mostraremos a continuación una tabla donde veremos características principales de los modelos traducidos, los cuales, fueron traducidos todos con éxito. Ver Figura 6.8 para ver la tabla.

## 6.2. Traducción de modelos de Libros de Event-B

Los modelos que estudiaremos aquí fueron obtenidos de los libros *Rodin User’s Handbook* [Jas12] y *System Modelling & Design Using Event-B* [Rob10].

En esta sección veremos características generales de la traducción de los siguientes 4 modelos:

- “Traffic Light System” del libro *Rodin User’s Handbook*. En este modelo de un contexto y dos máquinas (donde una máquina refina otra), lo que se busca es representar una calle donde transitan personas y carros. La idea es que en un momento dado nunca haya personas y carros transitando al tiempo. En este caso en particular se alteró ligeramente la máquina refinada, ya que en el libro hacen algo que no está soportado por este traductor (ni por otros traductores hasta el momento), donde se eliminan variables de la máquina abstracta. Esto se hace en Event-B, entre otras cosas, para simplificar la complejidad del código de la máquina refinada, pero como tal no es necesario. Por eso construimos una máquina exactamente similar sin eliminar las variables de la máquina abstracta.
- “Coffee Club” del libro *System Modelling & Design Using Event-B*. Un modelo de dos máquinas y un contexto (donde una máquina refina otra). Lo que se modela es una tienda de café con miembros, los miembros pueden crear una cuenta en la tienda, depositar dinero, y usarlo para comprar café. Está incluso la acción de representar cuando roban al banco y cambiar el precio del café. Este modelo se usa en el libro para explicar las cualidades básicas de un modelo en Event-B, particularmente, el concepto de los contextos, las máquinas, el estado, los eventos, las pruebas, y los refinamientos. En este modelo *Mypy* no mostró soporte por el

| CARACTERÍSTICAS GENERALES DE LAS TRADUCCIONES DEL ARTÍCULO DE ABRIL  |  |  |               |                  |
|--|--|--|---------------|------------------|
| Searching in an Array  |  |  |               |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados                         | Soporte MyPy  | Éxito Traducción |
| Funciones totales, Operaciones entre conjuntos, Operaciones entre números, Rango de una función, Elección de conjunto, Asignación simple, Conjunción. Entre otros.   | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., <b>Conjunto Potencia de los Naturales.</b> | Elección de Conjunto   | Sí            | Sí               |
| Searching in a Matrix  |  |  |               |                  |
| Funciones totales, Rango, Conjunto de los naturales sin el 0, Operaciones entre conjuntos, Producto Cartesiano, Tuplas, Dominio y Rango, Imagen relacional, Conjuntos por extensión, Sobrescritura de función, Operaciones entre números. Entre otros. | Contexto, Máquina, ..., <b>Conjunto Potencia del Producto Cartesiano de enteros a enteros.</b>                 | Ninguno.   | Casi completo | Sí               |
| Finding Common Number in two Sorted Arrays   |  |  |               |                  |
| Funciones inyectivas. Cuantificador universal. Dominio. Aplicación de una función. Rango. Conjunto Vacío. Rango. Sobreescritura de función. Operaciones y predicados entre números. Igualdad y desigualdad. Entre otros.                               | Contexto, Máquina, ..., <b>Unión de los naturales con un conjunto finito.</b>                                  | Cuantificadores sobre conjuntos infinitos.                       | Sí            | Sí               |
| Array Partitioning   |  |  |               |                  |
| Relaciones, Operaciones entre Conjuntos y Relaciones (ej. Subconjunto), Imagen relacional, Sobrescritura, Entre otros.   | Contexto, Máquina, ..., <b>Unión y Diferencia de los naturales con un conjunto finito.</b>                     | Ninguno.   | Sí            | Sí               |
| Sorting (Bubble algorithm)   |  |  |               |                  |
| Cuantificador universal (¡de dos variables!), Elección de conjunto, Operaciones y predicados entre números, Sobrescritura, Función Inyectiva, Implicación, entre otros.  | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ...   | Cuantificadores sobre conjuntos infinitos. Elección de Conjunto. | Sí            | Sí               |

Figura 6.8: Características principales de los modelos traducidos en la sección "Traducción de Programas Completos".

mismo problema que se identificó en el modelo de la sección anterior en el que se buscaba un número en una matriz (*Mypy* no logra asemejar un conjunto de tuplas con una relación en todos los casos).

- “Simple Two Way” del libro *System Modelling & Design Using Event-B*. Este modelo se introduce en libro para explicar la aplicación de Event-B para modelar sistemas y especificar seguridad (safety) en dichos sistemas. En este caso solo nos interesamos por traducir dos contextos (uno extendiendo al otro).
- Finalmente, y el más importante de esta sección, es un modelo del libro *System Modelling & Design Using Event-B* llamado “SquareRoot”. Como su nombre lo indica en inglés, el modelo se encarga de encontrar la raíz entera de un número. Para ello, se construye un modelo correcto que parte de un contexto y máquina abstractos, pero se le van aplicando refinamientos cada vez más y más complejos. En nuestro caso tradujimos 1 contexto y 3 máquinas de dicho modelo (donde una máquina refina a una máquina que está refinando otra máquina, mientras “ve” un contexto).

No sobra mencionar que todos los códigos generados se tradujeron exitosamente y se pueden ejecutar sin problemas.

Queremos mencionar algo muy particular del modelo “SquareRoot”. En la última máquina refinada, tenemos un evento que sobrescribe dos parámetros del evento abstracto que refina con ayuda de un observador (una de las funcionalidades más sofisticadas para refinamientos en Event-B). El evento en Event-B se puede ver en la Figura 6.9, la traducción de dicho evento se puede ver en la Figura 6.10. Se puede ver que, gracias a que el traductor hace uso del AST de Rodin, se generó ¡correctamente! la firma de la función, puesto que lo correcto es que dicha función tenga un sólo parámetro, tal y como ocurre en el código generado (pues los 2 que tenía el evento abstracto fueron reemplazados por el nuevo parámetro gracias a el observador). Es importante mencionar que aun así, *Mypy* no permite el cambio de firma de una función porque incumple el Principio de Liskov, el cual, como hemos dicho, es una regla por defecto de la herramienta, y por lo tanto no es compatible el tipado. Una solución para esto sería refactorizar la arquitectura de los códigos generados para que las máquinas concretas no hereden de las máquinas abstractas (y así no habría problema con la firma de las funciones), pero esto originaría otros inconvenientes, sobre los cuales no se quiso ahondar en este trabajo (recordando que no estamos considerando dar soporte completo a *Mypy* en este trabajo si eso conlleva a otros problemas). Aparte de lo anterior, queremos agregar que el evento abstracto era un evento convergente el cual requería verificar que la variante del modelo disminuyera (aspecto que se tradujo exitosamente).

Antes de seguir, veamos la ejecución de este modelo. Inicializamos la máquina para buscar la raíz de 49. Activamos los parámetros del Preludio que nos permiten recorrer conjuntos infinitos (la explicación de cómo funciona esto ya se hizo), ajustamos la generación aleatoria de enteros y el

```

event Improve2 refines Improve
any
    m2
where
        @grd1 low+1 ≠ high
        @grd2 m2 ∈ N
        @grd3 low < m2 ∧ m2 < high
        @grd4 m2*m2 > num
with
    @l l=low
    @h h=m2
then
    @act0 high := m2
end
end

```

Figura 6.9: Evento en Event-B del modelo SqrtRoot. Se resalta el uso de un observador.

tamaño de los conjuntos insignia cuando son finitos para que la ejecución se haga más interesante. Un ejemplo de cómo se hizo lo anterior se puede apreciar en la Figura 6.11.

Cuando ejecutamos el código, se inicializa la máquina con los valores que podemos observar en la Figura 6.12. De esos valores nos interesa observar que: Efectivamente la constante a la que queremos buscar la raíz es el número 49, la variable “sqrt” se inicializa con un valor aleatorio (pero eventualmente deberá tener la respuesta), las variables “low” y “high” ayudarán a ir acortando el rango de búsqueda; Luego de inicializar la máquina llamamos al método *PyAutoExecute* 20 veces para que el código se anime por sí mismo y busque la respuesta.

20 llamados al método *PyAutoExecute* fueron más que suficientes para ir reduciendo el rango de búsqueda y eventualmente encontrar la raíz del número 49, como se ve en la Figura 6.13.

Antes de pasar a la tabla que resume las características generales de las traducciones que se tratan en esta sección del capítulo, queremos resaltar algo supremamente importante que nos permite aumentar nuestra confianza en el buen funcionamiento de nuestro traductor: El hecho de que *PyAutoExecute* sea suficiente para animar un modelo como el de “Square Root”, dice mucho de nuestro código generado. La razón es que *PyAutoExecute* funciona buscando eventos al azar que estén habilitados y ejecutándolos. El hecho de que dicho método encuentre constantemente el evento correcto para ejecutar implica: Que los *contratos* de las guardas/invariantes/axiomas se cumplieron

```

142
143     def Improve2_eventActions(self, m2_userIn : int = P.NoParam()) -> None:
144
145         if m2_userIn is None:
146             m2_userIn = P.PyRandValGen("int")
147
148         attempt_Count : int = 0
149         while not(self.Improve2_eventGuards(m2_userIn)):
150             m2_userIn = P.PyRandValGen("int")
151             if attempt_Count == P.LOWMAXGENATTEMPTS():
152                 raise GuardsViolated("Guards of the Event could not be fulfilled.")
153             attempt_Count += 1
154
155         #Set Parameters as an Attribute.
156         self.m2 = m2_userIn
157
158         #Event Actions
159
160         self.high = self.m2
161
162         #Check Invariants after the actions are executed.
163         if P.DESIGN_BY_CONTRACT_ENABLED() and not(self.checkAllInvariants()):
164             raise Exception("PostConditions of the Event could not be fulfilled.")
165
166         del(self.m2)
167
168     #End Event
169

```

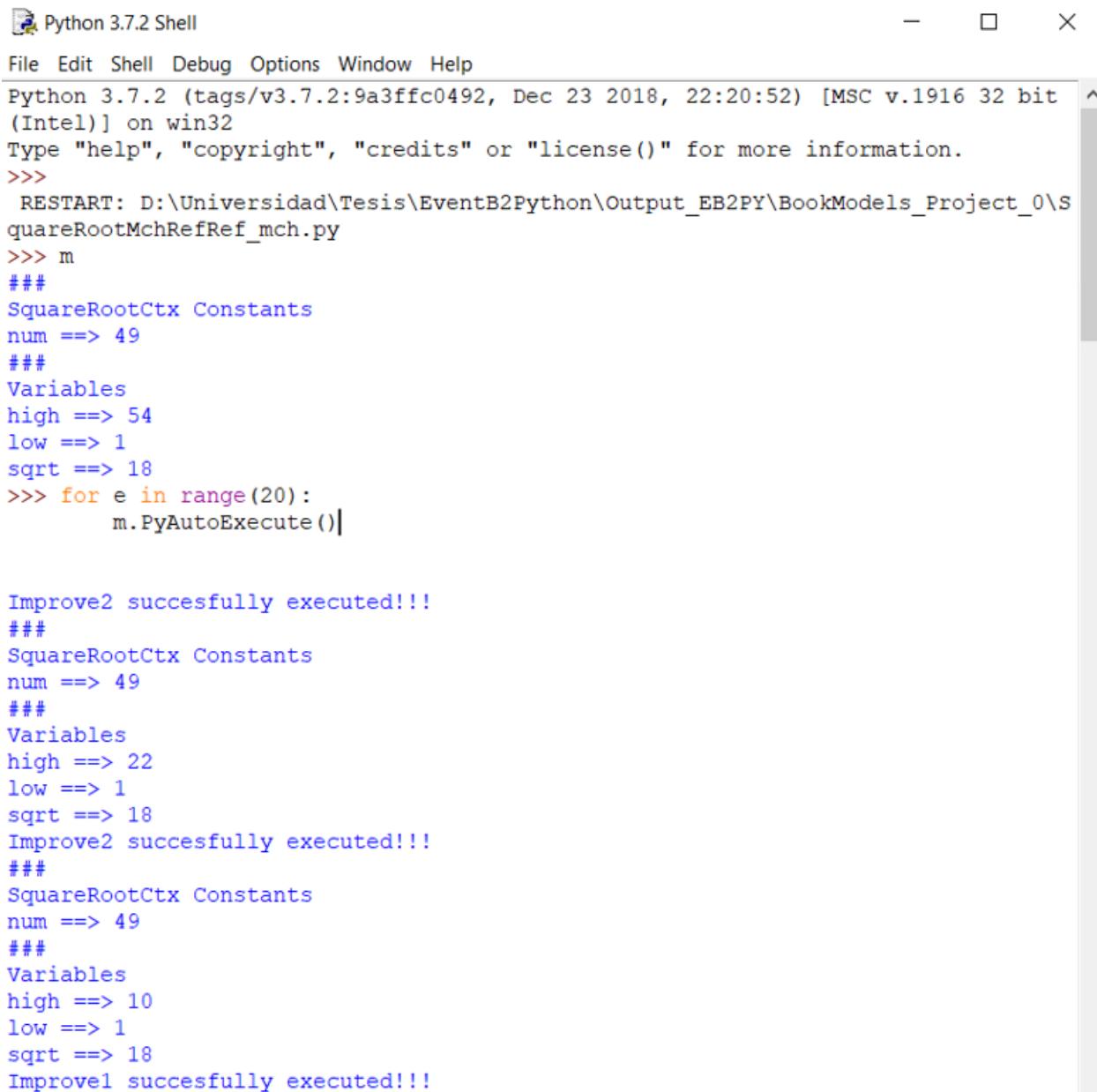
Figura 6.10: Traducción de las acciones del evento mostrado en la Figura anterior.

```

P.setRAND_INT_RANGE((0,30))
P.setUSE_FINITE_SPECIAL_SETS(True)
P.setFINITE_SPECIAL_SETS_LIMIT(90)
c = SquareRootCtx_class()
c.checkedInit(49)
m = SquareRootMchRefRef_class(c)

```

Figura 6.11: Código agregado al generado por la traducción del modelo SqrtRoot para poder ejecutarlo.



The screenshot shows a Python 3.7.2 Shell window. The title bar reads "Python 3.7.2 Shell". The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The main window displays the following text:

```
File Edit Shell Debug Options Window Help
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 22:20:52) [MSC v.1916 32 bit
(Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
RESTART: D:\Universidad\Tesis\EventB2Python\Output_EB2PY\BookModels_Project_0\SquareRootMchRefRef_mch.py
>>> m
###  
SquareRootCtx Constants  
num ==> 49  
###  
Variables  
high ==> 54  
low ==> 1  
sqrt ==> 18  
>>> for e in range(20):  
    m.PyAutoExecute()  
  
Improve2 successfully executed!!!  
###  
SquareRootCtx Constants  
num ==> 49  
###  
Variables  
high ==> 22  
low ==> 1  
sqrt ==> 18  
Improve2 successfully executed!!!  
###  
SquareRootCtx Constants  
num ==> 49  
###  
Variables  
high ==> 10  
low ==> 1  
sqrt ==> 18  
Improve1 successfully executed!!!
```

Figura 6.12: Estado inicial de la máquina, luego de inicializarla a partir del código generado por la traducción del modelo SqrtRoot.

```
Improve1 successfully executed!!!
#####
SquareRootCtx Constants
num ==> 49
#####
Variables
high ==> 8
low ==> 6
sqrt ==> 18
No event could be AutoExecuted
Improve1 successfully executed!!!
#####
SquareRootCtx Constants
num ==> 49
#####
Variables
high ==> 8
low ==> 7
sqrt ==> 18
SquareRootRef successfully executed!!!
#####
SquareRootCtx Constants
num ==> 49
#####
Variables
high ==> 8
low ==> 7
sqrt ==> 7
```



Figura 6.13: Impresiones en consola que demuestran que el código generado del modelo SqrtRoot logró su objetivo de encontrar la raíz.

| CARACTERÍSTICAS GENERALES DE LAS TRADUCCIONES DE LOS LIBROS  |  |   |                |                  |
|--|--|---|----------------|------------------|
| Traffic Light System   |  |   |                |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados      | Soporte MyPy   | Éxito Traducción |
| Enumeraciones, Partición, Booleanos, Predicados y Operaciones sobre Booleanos, Subconjuntos, Operaciones sobre Conjuntos, entre otros.   | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., <b>Eventos con Parámetros, Refinamientos,</b> ..   | Ninguno.                                      | Casi completo. | Si               |
| Coffee Club  |  |   |                |                  |
| Rangos, Elección de conjunto, Operaciones y predicados sobre números, Finitud, Asignación simple, Enumeraciones, Función total, Conjunto vacío y operaciones sobre conjuntos, entre otros. | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., <b>Eventos con Parámetros, Refinamientos,</b> ..   | Elección de Conjunto (Infinito).              | Casi completo. | Si               |
| Simple Two Way   |  |   |                |                  |
| Enumeraciones, Particiones, Función Total, Aplicación de Funciones, Cuantificador universal (sobre conjuntos finitos), Subconjuntos, Implicación, <b>Composición de funciones</b>          | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., <b>Extensión de contextos.</b>   | Ninguno.                                      | Si             | Si               |
| Square Root  |  |   |                |                  |
| Operaciones y Predicados sobre números, Elección de conjunto, <b>Elección por predicado</b> , entre otros.   | Contexto, Máquina, ..., <b>Eventos con Parámetros (parámetros múltiples, y reemplazo de parámetros con observadores), Refinamientos (incluyendo un refinamiento de un refinamiento), Variante, Evento Convergente, Refinamiento de un evento con dos eventos nuevos.</b> | Elección por predicado. Elección de Conjunto. | Casi completo. | Si               |

Figura 6.14: Resumen de las características generales de las traducciones mencionadas en la sección "Traducción de modelos de Libros de Event-B".

constantemente. Que cada vez que se ejecuta un evento se cumplen los *contratos* de las invariantes. Y que para encontrar la respuesta seguramente fue necesario que todas las asignaciones de los eventos llamados se hayan ejecutado de manera correcta; En otras palabras, que un método que funcione al *jazar!* mientras cumpla los *contratos* sea capaz de encontrar la respuesta indica que la traducción fue bastante exitosa, en el sentido que el código generado está representando un modelo cuya correctitud ha sido demostrada, y, por lo tanto, el código que lo represente y ejecute debería de funcionar perfectamente, incluso si hay indeterminismos.

Ahora sí, veamos la tabla que resume las características generales de las traducciones que se tratan en esta sección del capítulo. Ver Figura 6.14.

### 6.3. Traducción de modelos usados en Cursos Universitarios

Los modelos que estudiaremos aquí han sido usados en cursos universitarios sobre métodos formales. Algo interesante de los 3 primeros modelos que mencionaremos es que éstos son modelos abstractos que no representan programas, sino que son sistemas donde la idea es simular el comportamiento de un sistema de la vida real (al contrario de la gran mayoría de modelos que hemos

estudiado y estudiaremos, donde los modelos buscan encontrar una solución a un problema en concreto). Un efecto de lo anterior es que encontramos sistemas donde puede haber más de un evento habilitado y donde los eventos pueden cambiar el estado del sistema indefinidamente.

Específicamente, en esta sección veremos características generales de la traducción de los siguientes 4 modelos:

- “Cafetería”. En este modelo de dos contexto y dos máquinas (donde un contexto extiende otro y donde una máquina refina otra), lo que se busca es representar una cafetería con clientes. La idea es simular la interacción de tener una cola de espera donde llegan clientes mientras se preparan los platos. Luego, los clientes pueden sentarse a comer si hay mesas libres y el cajero está disponible. Finalmente, los clientes pueden irse del restaurante.
- “Hospital”. En este modelo de dos contexto y dos máquinas (donde un contexto extiende otro y donde una máquina refina otra), lo que se busca es representar un hospital al cual van llegando, atendiendo, y despachando pacientes. En este caso los limitantes son la cantidad de salas de cirugía que hay, la cantidad de enfermeros, la presencia del jefe de los enfermeros, entre otros.
- “ParkingLot”. En este modelo de un contexto y una máquina (donde la máquina “ve” el contexto), lo que se busca es representar un parqueadero de carros. En este caso se busca simular la entrada y salida de carros a un parqueadero por medio de *enumeraciones*. En este modelo, *Mypy* nuevamente no es capaz de asemejar conjuntos de tuplas con relaciones.
- Finalmente, tenemos “bbin”. En este modelo tenemos un contexto y dos máquinas que ven dicho contexto (y una máquina refina la otra). Este modelo busca simular el algoritmo de búsqueda binaria, y por lo tanto busca simular un programa. Interesante aquí es que para demostrar su terminación se hacen uso de eventos convergentes (y su respectiva variante).

Todos los códigos generados se tradujeron exitosamente y se pueden ejecutar sin problemas. Esta vez procederemos directamente a la tabla con las características generales de la traducción de los modelos mencionados en esta sección. Ver Figura 6.15.

#### 6.4. Traducción de modelos construidos para este Traductor

Los modelos que estudiaremos aquí fueron construidos con el fin de demostrar aplicaciones de los *PyCondRel* para representar funciones (infinitas), una particularidad que sólo ofrece nuestro traductor. La ventaja potencial de esta funcionalidad es que se pueden hacer modelos complicados de manera más rápida al hacer uso de funciones (de las cuales ya se haya demostrado su correctitud), sin tener que llevar dichas funciones al detalle determinístico, algo que hasta ahora siempre ha sido un requerimiento de los modelos concretos en Event-B. Para aclarar más esta idea, veamos los dos modelos que hacen parte de esta sección:

| CARACTERÍSTICAS GENERALES DE LAS TRADUCCIONES DE MODELOS DE CURSOS UNIVERSITARIOS  |  |  |                |                  |
|--|--|--|----------------|------------------|
| Cafeteria  |  |  |                |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados   | Soporte MyPy   | Éxito Traducción |
| Enteros, Operaciones y Predicados sobre Enteros, Conjuntos y operación de pertenencia de conjuntos, entre otros.   | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., Eventos con Parámetros, Extensiones, Refinamientos.  | Ninguno.                                   | Sí             | Sí               |
| Hospital   |  |  |                |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados   | Soporte MyPy   | Éxito Traducción |
| Enteros, Operaciones y Predicados sobre Enteros, Conjuntos y operación de pertenencia de conjuntos, Enumeraciones, entre otros.  | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., Eventos con Parámetros, Extensiones, Refinamientos.  | Ninguno.                                   | Sí             | Sí               |
| Parking Lot  |  |  |                |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados   | Soporte MyPy   | Éxito Traducción |
| Enumeración, Relación, Función Total, Función parcial inyectiva, Aplicación de Funciones, Conjuntos por extensión, Operaciones sobre conjuntos/relaciones (como unión, diferencia), entre otros. | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., Eventos con Parámetros.  | Ninguno.                                   | Casi completo. | Sí               |
| bbin (búsqueda binaria)  |  |  |                |                  |
| Operaciones/Componentes Involucradas   | Elementos/Situaciones especiales   | Indeterminismos o Infinitos Involucrados   | Soporte MyPy   | Éxito Traducción |
| Función Total, Aplicación de funciones, Enteros, Operaciones y Predicados sobre Enteros, Imagen relacional, Rangos, Elección por predicado, Implicación Lógia, entre otros.                      | Contexto, Máquina, Axiomas, Invariantes, Constantes, Variables ..., Teoremas, Eventos con Parámetros, Variantes, Eventos anticipados y convergentes, Refinamiento, | Cuantificadores sobre conjuntos infinitos. | Sí             | Sí               |

Figura 6.15: Resumen de las características generales de las traducciones mencionadas en la sección “Traducción de modelos usados en Cursos Universitarios”.

- “Fibonacci”: Este modelo ya lo estudiamos en el capítulo de *No Determinismo y el Infinito*, así que solo hablaremos de su uso potencial: Con un contexto tan sencillo de 3 axiomas se pudo modelar de manera abstracta la función de Fibonacci. Este contexto se puede usar por cualquier máquina que haga uso de esta función, la cual tiene un trasfondo teórico muy fuerte en las matemáticas y las ciencias de computación, por lo tanto, sería posible hacer uso de este contexto sin tener que llegar al extremo de modelar la función de Fibonacci en un modelo muy concreto (donde muy seguramente se perdería la posibilidad de representar la función de Fibonacci como una función de infinitas tuplas, sino que tendría que ser finita).
- “Dijkstra”: Un modelo que se usó para reforzar la idea anterior, en este caso modelamos el algoritmo de Dijkstra (el de distancias mínimas) usando como base el algoritmo de ordenamiento en el contexto (en este caso, se usa el algoritmo de ordenamiento para ordenar una lista que representa la estructura de datos heap/montón que se usa para implementar el algoritmo de Dijkstra). Sin entrar a mucho detalle, en este ejemplo desactivamos únicamente el diseño por contrato para el contexto (la razón es que no era el objetivo el construir los axiomas necesarios para especificar la función de ordenamiento). El diseño por contrato se reactiva para la máquina que modela el algoritmo de Dijkstra, y, con ayuda del algoritmo nativo de ordenamiento de Python, se usa el método *PyAutoExecute* para animar dicha máquina. Respecto a la máquina, se especificaron varias invariantes (por ejemplo, el que verifica que nuestra lista ordenada (heap) siempre lo esté), pero no todos los necesarios en un modelo completo, ya que queremos concentrarnos en el heap/montón y su uso para lograr el objetivo que pretende el algoritmo de Dijkstra. No sobra mencionar que los eventos sí se modelaron completamente, y a continuación veremos los resultados que se lograron gracias a eso.

Ya que ya estudiamos bastante el modelo de Fibonacci, veamos el resultado que se obtuvo del modelo de Djkstra. Lo primero es que el programa logra ejecutarse con éxito para obtener la distancia mínima desde el nodo origen a todos los demás nodos. Lo interesante de esto es que se pudo usar el algoritmo de ordenamiento (un algoritmo también con mucho trasfondo en la teoría de las ciencias de la computación, y cuya correctitud ya se ha demostrado) sin tener que especificarlo tan concretamente, y hacer uso de dicho algoritmo para ejecutar el algoritmo de Dijkstra. Esto mostraría el uso potencial de esta funcionalidad particular de nuestro traductor, donde se podrían usar funciones cuya correctitud ya se ha demostrado, omitir su modelamiento concreto (solo modelarlo abstractamente), y hacer uso de dichas funciones para construir sistemas aún más complejos.

Respectivo al modelo de Dijkstra, no sobra agregar un dato interesante: ¡mientras en el sistema del algoritmo de ordenamiento burbuja se requiere una máquina y un contexto relativamente complejos para modelar dicho algoritmo, en el modelo de Djkstra, con las funciones infinitas, pudimos hacer uso de un algoritmo de ordenamiento más sofisticado y que solo requirió de 2 axiomas para especificarse abstractamente!

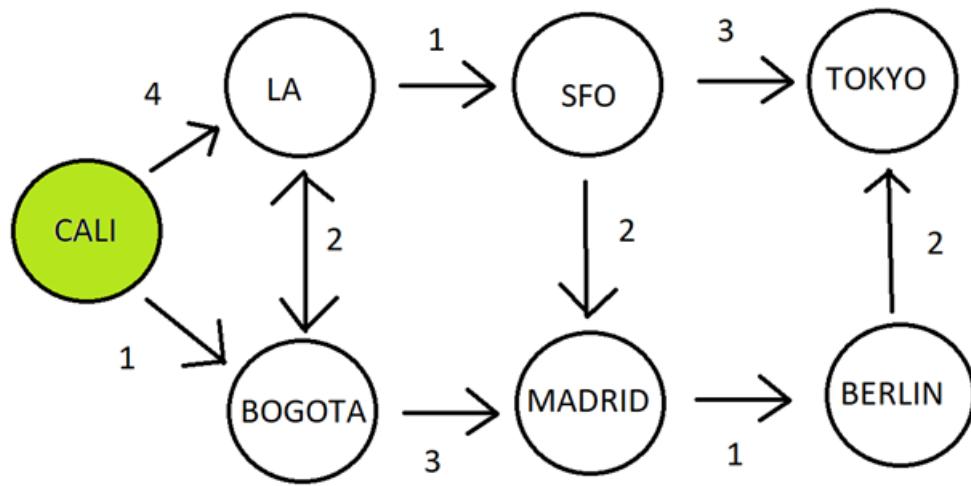


Figura 6.16: Grafo que se piensa resolver con el código generado que representa el modelo del algoritmo de Dijkstra.

Antes de pasar a la tabla que resume las características generales de las traducciones que se tratan en esta sección del capítulo, veamos en particular el grafo sobre el cual se aplicó el modelo que representa el algoritmo de Dijkstra. Ver Figura 6.16. En dicho grafo, la ciudad de CALI es el origen, y se calculará la distancia mínima a todas las demás ciudades.

Lo primero que se hace es inicializar la máquina con el grafo y demás constantes necesarias, en la Figura 6.17 podemos ver el estado inicial de la máquina. Si nos fijamos en la variable “distances” (distancias), al comienzo tiene valores muy altos (representando el infinito) como distancia mínima para cualquier otra ciudad (a excepción del origen que es 0). Esta es la manera tradicional de inicializar dicho arreglo en el algoritmo de Dijkstra.

Luego de eso, con ejecutar el método *PyAutoExecute* 50 veces (lo cual toma menos de 1 segundo en ejecutarse, y eso que se imprime en pantalla por cada vez que se llama al método), se obtiene la respuesta correcta, como se ve en la Figura 6.18. Podemos notar que ahora la variable “distances” tiene las distancias mínimas (correctas y que estábamos buscando) almacenadas. Con esto, verificamos la ejecución correcta del código que representa el algoritmo de Dijkstra.

Antes de terminar este capítulo queremos resumir las limitaciones que encontramos en nuestro traductor:

- Relativo a la herramienta *Mypy*, el código generado tendrá conflictos con el tipado exigido con la herramienta cuando: (i) Hayan relaciones (esto ocurre en pocos casos), (ii) Los parámetros

```

>>> m
###  

Dijkstra Constants  

BERLIN ==> CITIES_CS.BERLIN  

BOGOTA ==> CITIES_CS.BOGOTA  

CALI ==> CITIES_CS.CALI  

LA ==> CITIES_CS.LA  

MADRID ==> CITIES_CS.MADRID  

SANFRANCISCO ==> CITIES_CS.SANFRANCISCO  

TOKYO ==> CITIES_CS.TOKYO  

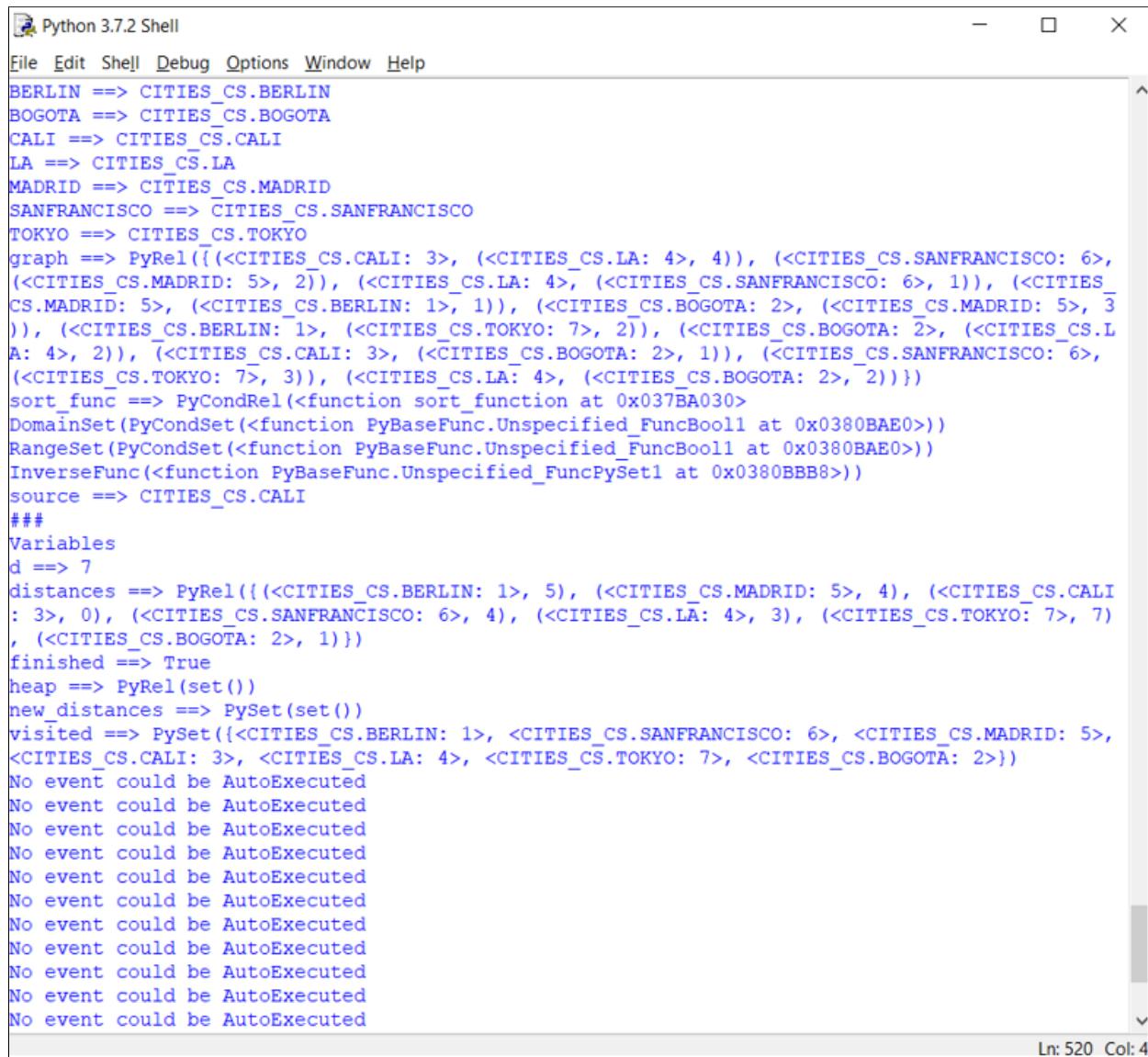
graph ==> PyRel({(<CITIES_CS.CALI: 3>, (<CITIES_CS.LA: 4>, 4)), (<CITIES_CS.SANFRANCISCO: 6>, (<CITIES_CS.MADRID: 5>, 2)), (<CITIES_CS.LA: 4>, (<CITIES_CS.SANFRANCISCO: 6>, 1)), (<CITIES_CS.MADRID: 5>, (<CITIES_CS.BERLIN: 1>, 1)), (<CITIES_CS.BOGOTA: 2>, (<CITIES_CS.MADRID: 5>, 3)), (<CITIES_CS.BERLIN: 1>, (<CITIES_CS.TOKYO: 7>, 2)), (<CITIES_CS.BOGOTA: 2>, (<CITIES_CS.LA: 4>, 2)), (<CITIES_CS.CALI: 3>, (<CITIES_CS.BOGOTA: 2>, 1)), (<CITIES_CS.SANFRANCISCO: 6>, (<CITIES_CS.TOKYO: 7>, 3)), (<CITIES_CS.LA: 4>, (<CITIES_CS.BOGOTA: 2>, 2)))})
sort_func ==> PyCondRel(<function sort_function at 0x037BA030>
DomainSet(PyCondSet(<function PyBaseFunc.Unspecified_FuncBool1 at 0x0380BAE0>))
RangeSet(PyCondSet(<function PyBaseFunc.Unspecified_FuncBool1 at 0x0380BAE0>))
InverseFunc(<function PyBaseFunc.Unspecified_FuncPySet1 at 0x0380BBB8>))
source ==> CITIES_CS.CALI
###  

Variables  

d ==> 0
distances ==> PyRel({(<CITIES_CS.CALI: 3>, 0), (<CITIES_CS.LA: 4>, 9999), (<CITIES_CS.BOGOTA: 2>, 9999), (<CITIES_CS.BERLIN: 1>, 9999), (<CITIES_CS.TOKYO: 7>, 9999), (<CITIES_CS.SANFRANCISCO: 6>, 9999), (<CITIES_CS.MADRID: 5>, 9999)})
finished ==> False
heap ==> PyRel({(0, (<CITIES_CS.CALI: 3>, 0))})
new_distances ==> PySet(set())
visited ==> PySet(set())
>>> |

```

Figura 6.17: Inicialización de la máquina del modelo de Dijkstra. Impresión del estado inicial en consola.



The screenshot shows the Python 3.7.2 Shell window with the following content:

```
Python 3.7.2 Shell
File Edit Shell Debug Options Window Help
BERLIN ==> CITIES_CS.BERLIN
BOGOTA ==> CITIES_CS.BOGOTA
CALI ==> CITIES_CS.CALI
LA ==> CITIES_CS.LA
MADRID ==> CITIES_CS.MADRID
SANFRANCISCO ==> CITIES_CS.SANFRANCISCO
TOKYO ==> CITIES_CS.TOKYO
graph ==> PyRel({(<CITIES_CS.CALI: 3>, (<CITIES_CS.LA: 4>, 4)), (<CITIES_CS.SANFRANCISCO: 6>, (<CITIES_CS.MADRID: 5>, 2)), (<CITIES_CS.LA: 4>, (<CITIES_CS.SANFRANCISCO: 6>, 1)), (<CITIES_CS.MADRID: 5>, (<CITIES_CS.BERLIN: 1>, 1)), (<CITIES_CS.BOGOTA: 2>, (<CITIES_CS.MADRID: 5>, 3)), (<CITIES_CS.BERLIN: 1>, (<CITIES_CS.TOKYO: 7>, 2)), (<CITIES_CS.BOGOTA: 2>, (<CITIES_CS.LA: 4>, 2)), (<CITIES_CS.CALI: 3>, (<CITIES_CS.BOGOTA: 2>, 1)), (<CITIES_CS.SANFRANCISCO: 6>, (<CITIES_CS.TOKYO: 7>, 3)), (<CITIES_CS.LA: 4>, (<CITIES_CS.BOGOTA: 2>, 2)))})
sort_func ==> PyCondRel(<function sort_function at 0x037BA030>
DomainSet(PyCondSet(<function PyBaseFunc.Unspecified_FuncBooll at 0x0380BAE0>))
RangeSet(PyCondSet(<function PyBaseFunc.Unspecified_FuncBooll at 0x0380BAE0>))
InverseFunc(<function PyBaseFunc.Unspecified_FuncPySetl at 0x0380BBB8>))
source ==> CITIES_CS.CALI
## Variables
d ==> 7
distances ==> PyRel({(<CITIES_CS.BERLIN: 1>, 5), (<CITIES_CS.MADRID: 5>, 4), (<CITIES_CS.CALI: 3>, 0), (<CITIES_CS.SANFRANCISCO: 6>, 4), (<CITIES_CS.LA: 4>, 3), (<CITIES_CS.TOKYO: 7>, 7), (<CITIES_CS.BOGOTA: 2>, 1)})
finished ==> True
heap ==> PyRel(set())
new_distances ==> PySet(set())
visited ==> PySet({<CITIES_CS.BERLIN: 1>, <CITIES_CS.SANFRANCISCO: 6>, <CITIES_CS.MADRID: 5>, <CITIES_CS.CALI: 3>, <CITIES_CS.LA: 4>, <CITIES_CS.TOKYO: 7>, <CITIES_CS.BOGOTA: 2>})
No event could be AutoExecuted
Ln: 520 Col: 4
```

Figura 6.18: Resultado final de ejecutar la máquina que representa el modelo de Dijkstra. El resultado es satisfactorio, pues se encuentran todas las distancias mínimas a todos los nodos (almacenadas en la variable “distances”).

de los eventos tengan el mismo nombre, se recomienda no reusar nombres en Event-B, sino *Mypy* confundirá sus tipos. (iii) Cuando hay varios cuantificadores, *Mypy* puede confundir el tipo de las variables asociadas al cuantificador y generar mensajes de precaución (“Warnings”); ¡Sin embargo!, no sobra mencionar que ninguna de las limitaciones que se acaba de mencionar impide el correcto funcionamiento, ejecución, y cumplimiento de *contratos* del código generado. *Mypy* sigue siendo una herramienta en desarrollo, y no es el objetivo del traductor el dar un soporte completo a esta herramienta.

- Otra limitación que no se vio en los modelos con que se testeó, pero que se encontró posteriormente, es que los cuantificadores en este momento no pueden anidarse. Esto es un problema que se origina en una falla técnica del traductor, pero que es perfectamente corregible (trabajo futuro).

# Conclusiones

---

## 7.1. Conclusiones Finales

Como primera conclusión, queremos resaltar que se pudo construir satisfactoriamente un traductor de modelos correctos en Event-B a Python. La herramienta se implementó efectivamente como un Plug-In para Rodin.

Lo segundo que queremos evidenciar es que se logran traducir la gran mayoría de operaciones soportadas por Event-B, aspecto que no todos los traductores en este mismo ámbito han podido lograr. Adicional a esto, se logran traducir varios fenómenos en Event-B que han sido un gran reto para otros traductores en este campo, específicamente, estamos hablando del indeterminismo y del infinito en Event-B. También queremos recalcar que se da soporte para cosas como: conjuntos por comprensión, evaluación de cuantificadores que requieren recorrer un conjunto infinito de manera parcial (recorriendo un conjunto finito que sirva de muestra para representar dicho conjunto infinito que hay que recorrer), extensión de contextos y refinamientos de máquinas, expresiones y predicados construidos a partir de la combinación de diferentes operaciones soportadas por Event-B, etc.

El traductor se construyó para generar código que fuera lo más entendible posible (se genera un código con comentarios que facilitan el entendimiento del código, se generan nombres que facilitan la lectura, se usan herramientas particulares de Python como los métodos mágicos para mejorar la legibilidad del código, se organizan varios componentes alfabéticamente para facilitar la búsqueda de componentes en el código, entre otros), y además se ofrecen herramientas para hacerle seguimiento a su ejecución y facilitar su uso y entendimiento, tales como los métodos *PyGuardsState*, *PyAutoExecute*, y *PyRandValGen*. Todo lo anterior facilita su entendimiento y resalta la relación entre el código generado y el modelo que representa en Event-B, gracias a que se pueden identificar fácilmente las componentes del código generado, y a la hora de ejecutar el código, es fácil hacer un seguimiento al estado de las máquinas y los contextos.

Haber construido el traductor a partir de la extracción de información del AST de Rodin permite con facilidad que nuestro generador de código soporte prácticamente cualquier modelo mientras sea sintácticamente correcto, esto aumenta la libertad y flexibilidad del usuario al hacer uso de nuestra herramienta.

Se buscó que el código generado (y el del traductor) siguiera buenas prácticas de programación, tales como el encapsulamiento de la programación orientada a objetos, la generación de excepciones con mensajes alusivos al error, una arquitectura de clases relacionadas, archivos modularizados por cada contexto y máquina, representación de muchas funcionalidades como métodos de clases (lo que permite que el usuario pueda llamar cada componente del código generado a libertad, por ejemplo, verificar la veracidad de una invariante en un momento dado solo requiere llamar el método asociado al *contrato* de esa invariante).

Se buscó no solamente que el traductor soportara la mayor cantidad de operaciones posibles, sino que también se ofrecieron nuevas herramientas que no se ven en otros traductores en este campo. Por ejemplo, la clase *PyCondSet* y la clase *PyCondRel* aumentan considerablemente la forma en que los usuarios representan conjuntos y relaciones, donde el Preludio deja una arquitectura lista para que el usuario haga un uso libre de dichas clases (que no generan problemas de tipo pues heredan de *PySet* y *PyRel* respectivamente). Y no sobra agregar nuevamente la posibilidad de representar funciones infinitas, cuyo uso vimos en el capítulo de *Resultados*, en el cual se pudieron usar las funciones infinitas para obtener resultados muy satisfactorios, y que además permite una posible nueva manera de modelar en Event-B, donde como se mencionó, lo que se buscaría es hacer uso de funciones especificadas de manera abstracta (pero cuya correctitud en concreto ya esté demostrada por la teoría), y hacer uso de ellos ahorrándonos la necesidad de hacer especificaciones concretas sobre dichas funciones. Esto podría agilizar y facilitar el modelado de sistemas complejos.

Se mejora la confianza de una traducción en Python al ofrecer soporte de chequeo de tipado con la herramienta *Mypy*. Aunque hay que recordar que el soporte es parcial pues hay casos (relativamente pocos) que causan que Mypy no sea compatible con el código generado.

Otra característica de nuestro traductor para comodidad del usuario es el de tener parámetros en el Preludio (editables por el usuario) para ajustarlo con mayor precisión al código generado.

Sobre los resultados, queremos resaltar el hecho de que todos los modelos que se tradujeron fueron traducidos con éxito y pudieron ejecutarse. Pero más aún, el hecho de que para dichos modelos haya sido posible hacerle seguimiento a su ejecución con el método *PyAutoExecute* dice mucho sobre el correcto funcionamiento de este traductor. La razón de lo anterior, como lo habíamos mencionado, es que el método *PyAutoExecute* elige eventos al ¡azar!, ¡únicamente guiado por la correctitud de los *contratos*!. El hecho de que todos los modelos se hayan podido ejecutar (de la manera en que fueron pensados dichos modelos en Event-B y obteniendo los resultados esperados) a través del uso de un método que se guía por la correctitud del modelo, demuestra empíricamente que el código generado también es correcto, pues siguiendo todos los lineamientos que se especificaron, el modelo formal traducido consigue ejecutarse de manera aleatoria sin incumplir ningún *contrato* y cumpliendo el objetivo para el que fue pensado. En otras palabras, si tenemos un modelo correcto, una buena traducción debería por lo menos poder generar un código que no debería tener ningún inconve-

niente en ejecutarse, y más aún si logra cumplir su objetivo de forma aleatoria e indeterminada y cumpliendo todos los *contratos*.

En síntesis, podemos afirmar que se construyó un traductor que cumple la función para la cual estaba previsto, y al mismo tiempo, se buscó generar valor al usuario desde el diseño para poder implementar diferentes herramientas que permitieran fortalecer la calidad, uso, y entendimiento del código generado.

## 7.2. Trabajo Futuro

Este traductor tiene mucho trabajo futuro por delante:

- El primero es corregir las pequeñas fallas técnicas que causan ligeras limitaciones en nuestro traductor.
- Se puede mejorar la demostración de su buen funcionamiento de manera formal y menos empírica, ya que es un poco irónico que se haya desarrollado esta herramienta para el beneficio de los métodos formales sin hacer mucho uso de ellos.
- Algunos traductores en este ámbito permiten una traducción de modelos en Event-B donde se genera código que se puede ejecutar bajo el paradigma de la programación multihilo, una funcionalidad que nuestro traductor no tiene en el momento.
- Se puede mejorar la compatibilidad del código generado con la herramienta *Mypy*. Por ejemplo, para solucionar el limitante con las relaciones, se puede refactorizar la clase *PyRel* de modo que herede de la clase *PySet* y le sea más fácil identificar la relación entre estos dos componentes. Sin embargo, es un trabajo de cuidado, pues por ejemplo, habría que lidiar con el Principio de Liskov, el cual restringiría la firma de los métodos de los conjuntos a cualquier tipo de conjunto en la clase *PyRel*, algo que para las relaciones no es el caso, pues muchas de sus operaciones sólo deberían de darse entre relaciones.
- Se puede mejorar la inicialización aleatoria de constantes/variables con la ayuda de la herramienta ProB, pues hay software que podría permitir el acoplamiento de este traductor con ProB. También se podría buscar el hacer uso de los *SMT Solvers*, que ayudan con esta tarea.
- Se está buscando la manera de incluir este traductor en el conjunto de generadores oficiales de códigos de Event-B.
- Y finalmente, se pueden realizar mejoras o pulir muchas características de este traductor en muchos ámbitos, desde lo técnico, hasta lo conceptual.



# Bibliografía

- [Abr01] Jean-Raymond Abrial. Event driven sequential program construction. 01 2001. Available at <http://www.matisse.qinetiq.com>.
- [CR09] Nestor Cataño and Camilo Rueda. Teaching formal methods for the unconquered territory. In *International Conference on Integrated Formal Methods*, pages 2–19. Springer, Berlin, Heidelberg, 2009. [https://link.springer.com/chapter/10.1007/978-3-642-04912-5\\_2](https://link.springer.com/chapter/10.1007/978-3-642-04912-5_2).
- [CR16] Nestor Cataño and Victor Rivera. Eventb2java: A code generator for event-b. In *NASA Formal Methods Symposium*, pages 166–171. Springer, Cham, Julio 2016. [https://link.springer.com/chapter/10.1007/978-3-319-40648-0\\_13](https://link.springer.com/chapter/10.1007/978-3-319-40648-0_13).
- [CWR<sup>+</sup>11] Néstor Cataño, Tim Wahls, Camilo Rueda, Victor Rivera, and Danni Yu. Translating b machines to jml specifications. *Proceedings of the ACM Symposium on Applied Computing*, 01 2011.
- [EBI<sup>+</sup>12] A. Edmunds, M. Butler, I.Maamria, R. Silva, and C. Lovell. Event-b code generation: Type extension with theories. In *International Conference on Abstract State Machines*, pages 365–368. Springer, Berlin, Heidelberg, 2012. University of Southampton, UK.
- [FHB<sup>+</sup>14] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiro Miyazaki. Code generation for event-b. In *International Conference on Integrated Formal Methods*, pages 323–338. Springer, Cham, 2014. [https://link.springer.com/chapter/10.1007/978-3-319-10181-1\\_20](https://link.springer.com/chapter/10.1007/978-3-319-10181-1_20).
- [Fou20] Python Software Foundation. Typing — support for type hints. <https://docs.python.org/3/library/typing.html>, 2020.
- [Hax10] Anne E. Haxthausen. An introduction to formal methods for the development of safety-critical applications. Technical report, Technical University of Denmark, Agosto 2010.
- [Jas12] Michael Jastram. *Rodin User’s Handbook*. 2012. <https://www3.hhu.de/stups/handbook/rodin/current/pdf/rodin-doc.pdf>.
- [KL16] Sebastian Krings and Michael Leuschel. Constraint logic programming over infinite domains with an application to proof. *Electronic Proceedings in Theoretical Computer Science*, 234:73–87, 12 2016.
- [Lic20] MIT License. python/mypy. <https://github.com/python/mypy>, 2020.
- [MJRV05] C. Métayer, J.-R. Abrial, and L. Voisin. Rigorous open development environment for complex systems. Mayo 2005. <http://rodin.cs.ncl.ac.uk/>.

- [Rob10] Ken Robinson. *System Modelling & Design Using Event-B*. Australia, Octubre 2010. School of Computer Science and Engineering. The University of New South Wales.
- [Rob14] Ken Robinson. A concise summary of the event b mathematical toolkit. <https://wiki.event-b.org/images/EventB-Summary.pdf>, 2014.
- [SG14] Nico Plat Stefania Gnesi. 3rd fme workshop on formal methods in software engineering (formalise 2015). *ACM: ICSE '15 Proceedings of the 37th International Conference on Software Engineering*, 2:977–978, 2014. Florence, Italy.
- [SH15] Stéphane Wojewoda Shane Hastie. Standish group 2015 chaos report - qa with jennifer lynch. <https://www.infoq.com/articles/standish-chaos-2015>, Octubre 2015.
- [SM11] Neeraj Kumar Singh and Dominique Mèry. Eb2all - the event-b to c, c++, java and c# code generator. <http://eb2all.loria.fr/>, Abril 2011.
- [Spi15] Ruslan Spivak. Let's build a simple interpreter. part 7: Abstract syntax trees. <https://ruslanspivak.com/lsbasi-part7/>, 2015.
- [Way19] Hillel Wayne. Why don't people use formal methods? <https://www.hillelwayne.com/post/why-dont-people-use-formal-methods/>, Enero 2019. Revisado: 2020-06-14.
- [WEB10] Wiki-Event-B. B2c plugin. [https://wiki.event-b.org/index.php/B2C\\_plugin](https://wiki.event-b.org/index.php/B2C_plugin), 2010.
- [WEB15] Wiki-Event-B. Code generation activity. [https://wiki.event-b.org/index.php/Code\\_Generation\\_Activity](https://wiki.event-b.org/index.php/Code_Generation_Activity), 2015.
- [Wik20] Wikipedia. Compilador. <https://es.wikipedia.org/wiki/Compilador>, 2020.
- [Wol12] Sune Wolff. Scrum goes formal: Agile methods for safety-critical systems. In *First International Workshop on Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA)*. IEEE, Zurich Switzerland, 06 2012.