EB2Python-Traducción automática de especificaciones Event-B en Rodin a Python

Hernán Felipe Losada Calderón

Director
Camilo Rocha
Co-Director
Wilmer Garzón

Programa de Ingeniería de Sistemas
Escuela Colombiana de Ingeniería Julio Garavito
Bogotá, Colombia
Julio, 2018

Resumen

Event-B es un método formal para el modelado y análisis de sistemas basado en el enfoque de corrección por construcción. Presenta un conjunto de teorías como la elección para la notación de modelado, el refinamiento para representar diferentes niveles de abstracción en los modelos y un sistema de prueba para verificar la consistencia dichos modelos. Este documento presenta una algoritmo para generar programas en el lenguaje de programación *Python* a partir de modelos *Event-B* correctos. El algoritmo presentado aquí es la composición de reglas de traducción; incluye soporte para relaciones, expresiones numéricas y enumeraciones. El código puede ser generado para ejecución secuencial o concurrente (por medio de hilos). Un ejemplo ilustra la traducción de código y su ejecución como programa en el lenguaje de programación *Python*.

Agradecimientos

Quiero agradecer a los directores de proyecto Camilo Rocha y Wilmer Garzón por permitirme trabajar de la mano de ellos. Les agradezco por su apoyo y tiempo. La experiencia y conocimiento que me han transmitido son fuentes de inspiración para mi formación personal y profesional.
A mis padres Alba y Hernán, por su apoyo incondicional. A todas las personas que hicieron parte de mi formación durante mis años de estudio en la Escuela.

Índice general

Capítulo 1

Introducción

El modelado de sistemas se puede usar en diferentes etapas del proceso de desarrollo de software, desde el análisis de requisitos hasta en las pruebas de aceptación [1,10,13]. El modelado formal permite tener una entendimiento más profundo de los sistemas, con mayor coherencia de la especificación y el diseño, que con métodos informales o semiformales. Para gestionar la complejidad de los sistema, su abstracción y su refinamiento es clave estructurar el esfuerzo formal de modelado, debido a la compatibilidad de la separación de preocupaciones y razonamiento en capas [5]. Un enfoque de refinamiento significa que los modelos representan diferentes niveles de abstracción del diseño del sistema; la consistencia entre la abstracción de estos niveles garantiza la verificación formal e incluye pruebas para la verificación de la consistencia.

Event-B es un lenguaje de modelado formal para desarrollo de software. El lenguaje formal disminuye la complejidad del proceso en la descripción de sistemas. *Rodin* permite ejecutar especificaciones *Event-B* [2,11,13]. *Rodin* brinda las bases tecnologicas para la ejecución del algoritmo diseñado en este trabajo.
Este trabajo presenta *EB2Python*, un algoritmo para la generación de código automático de *Event-B* a *Python*3. El código generado corresponde a un sistema del cual se han demostrado matemáticamente propiedades de interés, es decir, de un sistema correcto de *Event-B*. *EB2Python* tiene como característica generar código para diferentes niveles de abstracción de la descripción de sistemas. Además, puede generar código secuencial y multihilo de la implementación del modelo descrito.

Este documento incluye una explicación breve de *Event-B*, las reglas de traducción que permiten la generación de código a *Python* y un caso de estudio para la explicación de las reglas. Un ejemplo de la ejecución de la traducción por medio de sus pruebas también incluida.
El documento está organizado de la siguiente manera:

1. El Capítulo 2 incluye los conceptos requeridos para la lectura y comprensión del documento.
2. El Capítulo 3 presenta las reglas de traducción.
3. El Capítulo 4 describe el caso de estudio desarrollado.
4. El Capítulo 5 presenta las pruebas del caso de estudio en ejecución.

Capítulo 2

Preliminares

2.1. *Event-B*

Event-B es un lenguaje con un enfoque matemático que permite modelar y diseñar sistemas. Usando la idea de refinamiento, permite construir modelos extensibles y facilita un método de razonamiento sistemático mediante pruebas [1,5,10,13].
La notación de máquina abstracta, una evolución de su misma noción en el *B-Method* [11], se considera una notación más simple, con una curva de aprendizaje más alta. *Event-B* tiene como objetivo mejorar el desarrollo de sistemas. Esto hace referencia a diseñar y gestionar proyectos con diferentes niveles de complejidad. Adopta un enfoque interdisciplinario con el fin de definirlos e interactuar con otros sistemas, y permite disminuir la brecha entre requisitos formales y las especificaciones.
Event-B tiene como característica principal el uso de la teoría de conjuntos para el modelado [3,8].

2.2. Modelado

Los modelos formales fácilitan la construcción y el análisis riguroso de sistemas, la reutilización de especificaciones en el desarrollo y la implementación, y la transparencia de recursos de las demostraciones a las especificaciones.

El modelado se basa en dos componentes, el contexto y la máquina. El contexto describe todo lo que es constante en el modelo. Lo que es constante en el sistema no cambia, permanece igual a lo largo de todas las observaciones. Lo que es constante depende del punto de vista sobre el cual se observa. El contexto divide sus propiedades en axiomas y constantes [10,13]. La máquina representa la parte dinámica del modelo y expresa lo que cambia a medida que el sistema interactúa con diferentes elementos en su entorno. Lo realiza por medio de variables cuyos valores cambian a través de la ejecución de eventos. Las constantes y variables deben satisfacee invariantes [10,13].

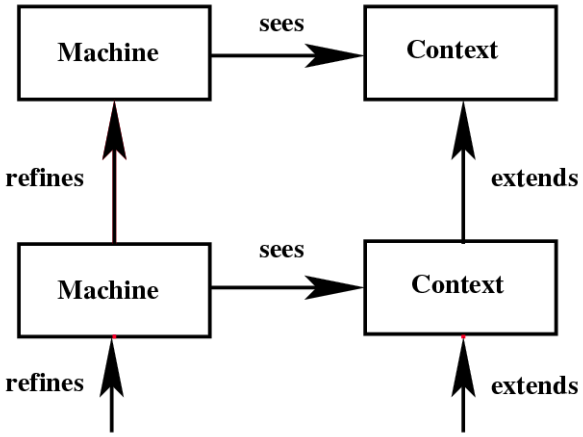

FIGURA 2.1. Relación entre máquina y contexto

Existen relaciones entre máquinas y contextos, como se ilustra en la Figura 2.1 [10]; una máquina puede ser refinada por otra, un contexto puede ser ampliado y una máquina puede ver uno o muchos contextos. Para mayor detalle sobre los elementos del modelado de *Event-B* se puede consultar su documentación en su página event-b.org.

2.2.1. Axiomas. *Event-B* define una sección de axiomas la cual contiene una lista de predicados. Los axiomas definen reglas que siempre serán ciertas para los elementos dados del contexto [10,13].

2.2.2. Teoremas. Los axiomas se pueden marcar como teoremas. Si este es el caso, se declara el predicado para ser probado como teorema. Los teoremas se pueden usar en las demostraciones [10,13].

2.2.3. Invariantes. Los invariantes son predicados que deberían ser verdaderos en cada estado alcanzable. Es una relación que describe todas las observaciones válidas de un sistema [10,13].

2.2.4. Eventos. Un evento se compone de una guarda que es una condición bajo la cual se debe ejecutar y una asignación a un valor nuevo de las variables [10,13].
Para la traducción se definieron los siguientes símbolos de asignación $:=$ y $:\in$ [8], el primero en una asignación convencional y el segundo es una asignación de un valor escogido deterministicamente de valores.

TABLA 1. Tabla de conectivos lógicos

Símbolo	Traducción
$\wedge$	and
$\vee$	or
$\Leftrightarrow$	=
$\neg$	not
$\forall$	all()
$\exists$	any()

2.2.5. Refinamiento. El refinamiento es utilizado para introducir detalles a la complejidad del modelo de un sistema; extiende las funcionalidades de comportamientos declarados de sistemas descritos con anterioridad. Introduce gradualmente los detalles y la complejidad de un modelo. Si una máquina B refina una máquina A, B solo puede comportarse de una manera que corresponda al comportamiento de A, con posibles detalles adicionales sobre dico compotamiento [3].
Un evento de la máquina puede ser refinado por uno o varios eventos de la máquina concreta. El refinamiento puede tomar caraterísticas como el fortalecimiento de una guarda, reutilizar eventos, invariantes y variables. Además, incluir nuevos elementos o extender los anteriores e incluir testigos.

2.3. Notación matemática

Event-B utiliza una notación matemática para describir los sistemas a modelar. Por medio de la sintaxis de la teoría de conjuntos se precisan propiedades en los modelos.

2.3.1. Predicados. Un predicado es una expresión cuyo valor es verdadero o falso. Los predicados se pueden combinar con los operadores lógicos. Se presentan en la Tabla 1 algunos de los operadores lógicos que soporta *Event-B* y la traducción [4,8,10].

2.3.2. Tipos de datos. Para el uso de los operadores y declaración de variables en Event-B se definen tipos de datos. Los datos hacen referencia a algún tipo, como por ejemplo, los números enteros ($\mathbb{Z}$), Booleanos ($BOOL$), conjuntos o relaciones [4,8,10].

2.3.3. Operaciones en conjuntos. Los conjuntos en *Event-B* se definen con la siguiente expresión $\{E\}$, donde E son los elementos del conjunto. Los conjuntos también tienen una serie de operadores entre ellos, que se presentan en la Tabla 2 [4,8,10].

TABLA 2. Tabla de operadores de conjuntos

Símbolo	Traducción
$\emptyset$	BSet()
$\cup$	.union()
$\cap$	.intersection()
$\setminus$	.difference()
$\rightarrow$	.pair()
X	.directProd()
Card(S)	.card(S)
{ E,F }	$Enumerate(E, F)$
SXT	S.cross(T)
$\mathbb{P}$(S)	BSet(S)
Inter(U)	.inter(u)

TABLA 3. Tabla de predicados de conjuntos

Símbolo	Traducción
$\in$	.has()
$\subseteq$	.isSubset()
$\subset$	.isProperSubset()
finite(S)	.finite(S)

Los conjuntos también tienen una serie de predicados y asociados [4,8,10]. En la tabla 3 se presentan aquellos para los cuales se soporta la traducción de *Event-B* a *Python*.

2.3.4. Relaciones. Una relación es un conjunto de pares ordenados, se expresa de dos maneras, $E \rightarrow F$ denota un par cuyo primer elemento es E y el segundo es F y SxT denotado el conjunto de pares donde el primer elemento es miembro de S y el segundo es miembro de T.
Se definen con los símbolos y funciones de la Tabla 4 [4,8,10].

2.3.5. Aritmética. Para los operando de tipos numéricos, cuando se especifican don variables x y y con $x \in \mathbb{Z}$ y $y \in \mathbb{Z}$ se utilizan los siguientes operadores + ,- ,/ y * [4,8,10].

TABLA 4. Tabla operadores de relación

Símbolo	Traducción
dom(r)	.dom(r)
ran(r)	.ran(r)
finite(S)	.domainSubstraction()
⩤	.rangeSubtraction()
▷	.restrictRangeTo()
◁	.restrictDomainTo()

2.4. Rodin

La traducción de *Event-B* utiliza la herramienta *Rodin*. *Rodin* es un software libre, con una colección de herramientas que proporcionan elementos que componen al lenguaje *Event-B*. Define características de sistemas, subsistemas y las interacciones entre ellos [2,11]. *Rodin* cuenta con un editor para escribir modelos y señala posibles errores de escritura [13]. Está diseñada como herramienta para el desarrollo de sistemas, el diseño y gestión de proyectos.

Rodin está basado en Eclipse, una comunidad de código abierto. La comunidad de código abierto está compuesta por marcos extensibles, herramientas y tiempos de ejecución para crear, implementar y administrar el ciclo de vida del software [11,12].

2.5. Caso de estudio

El siguiente caso describe un servicio para el préstamo de bicicletas. Las bicicletas se prestan a un determinado grupo de usuarios que hacen parte de un grupo. Un usuario puede registrarse y desregistrarse del servicio. Volverse activo o inactivo. Realiza estas acciones con el fin pedir prestadas bicicletas y devolverlas de acuerdo con la cantidad de bicicletas que se encuentran disponibles. El sistema describe una serie de condiciones para las cuales una persona puede tomar una bicicleta como no salirse del sistema si tiene una bicicleta en su poder, revisiones de como registrarse y activarse. Por último, define las condiciones para tomar prestada una bicicleta y cómo devolverla.

El caso de estudio, declara una serie invariantes. Las invariantes describen a los usuarios como un subconjunto de personas, a unas personas activas como un subconjunto de usuario, que una bicicleta parquear pertenece a un conjunto de bicicleta que ha tenido una función

```
CONTEXT
    IBSCtx
SETS
  ◦ Bicycle
  ◦ People
  ◦ Lock
AXIOMS
  ◦ ax0:    finite(Bicycle) not theorem
  ◦ ax1:    finite(People) not theorem
  ◦ ax2:    finite(Lock) not theorem
  ◦ ax3:    card(Bicycle) ≤ card(Lock) not theorem
END
```

FIGURA 2.2. Contexto

parcial con las bloqueadas, a una bicicleta que esta en uso pertenezca a unas bicicletas que tiene una función parcial del conjunto de activas y que debe haber una partición de bicicletas con el dominio de parqueadas y en uso.
Se declaran los axiomas donde los conjuntos de bicicletas, personas y bloqueados son finitos y el conjunto de bicicletas debe ser siempre menor a las bloqueadas.
El evento $INITIALISATION$ crea conjuntos vacíos para los usuarios, los activos y las bicicletas en uso, las bicicletas parqueadas tienen una asignación determinista de conjuntos con las bicicletas, con una inyección total de las bloqueadas. El evento $RegistrarUsuarios$ se ejecuta cuando un elemento nuevo no pertenece a sustracción entre las personas y usuarios, ejecuta la unión al conjunto de usuarios. Un usuario se *desregistra* cuando pertenece a una sustracción entre el rango del conjunto de las bicicletas en uso y usuarios, se ejecuta creando una sustracción del usuario al conjunto de usuarios y personas. Un usuario se *activa* cuando se sustrae del conjunto de usuarios y activos, luego se une al conjunto de usuarios activos. Un usuario se *desactiva* cuando pertenece a los usuarios activos y tiene un rango restringido de bicicletas en uso como vacío, se ejecuta sustrayendo del conjunto de activos al usuario que la ejecuta. La *prestacindebicicletas* se activa cuando pertenece al conjunto de usuarios sustraídos del rango de bicicletas en uso y que la bicicleta que tomara pertenezca al dominio de las parqueadas, se ejecuta que la bicicleta pasa a estar en uso con la unión del conjunto del usuario.

La traducción generada se mostrará en el Capitulo 4.

```
MACHINE
    IBSMach
SEES
  ◦ IBSCtx
VARIABLES
  ◦ User
  ◦ Active
  ◦ ParkedAt
  ◦ InUseBy
INVARIANTS
  ◦ inv0:  User ⊆ People not theorem
  ◦ inv1:  Active ⊆ User not theorem
  ◦ inv2:  ParkedAt ∈ Bicycle ⇸ Lock not theorem
  ◦ inv3:  InUseBy ∈ Bicycle ⇸ Active not theorem
  ◦ inv4:  partition(Bicycle,dom(ParkedAt),dom(InUseBy)) not theorem
EVENTS
  ◦ INITIALISATION:  not extended ordinary
    THEN
      ◦ act0:  User ≔ ∅
      ◦ act1:  Active ≔ ∅
      ◦ act2:  ParkedAt :∈ Bicycle ⇸ Lock
      ◦ act3:  InUseBy ≔ ∅
    END

  ◦ RegisterUser:  not extended ordinary
    ANY
      ◦ p
    WHERE
      ◦ grd0:  p ∈ People \ User not theorem
    THEN
      ◦ act0:  User ≔ User ∪ { p }
```

```
  ◦ UnregisterUser:  not extended ordinary
    ANY
      ◦ p
    WHERE
      ◦ grd0:  p ∈ User \ ran(InUseBy) not theorem
    THEN
      ◦ act0:  User ≔ User \ { p }
      ◦ act1:  Active ≔ Active \ { p }
    END

  ◦ ActivateUser:  not extended ordinary
    ANY
      ◦ u
    WHERE
      ◦ grd0:  u ∈ User \ Active not theorem
    THEN
      ◦ act0:  Active ≔ Active ∪ { u }
    END

  ◦ DeactivateUser:  not extended ordinary
    ANY
      ◦ u
    WHERE
      ◦ grd0:  u ∈ Active not theorem
      ◦ grd1:  InUseBy ▷ { u } = ∅ not theorem
    THEN
      ◦ act0:  Active ≔ Active \ { u }
    END

  ◦ BorrowBicycle: not extended ordinary
    ANY
      ◦ u
      ◦ b
    WHERE
      ◦ grd0:  u ∈ Active \ ran(InUseBy) not theorem
      ◦ grd1:  b ∈ dom(ParkedAt) not theorem
    THEN
      ◦ act0:  InUseBy ≔ InUseBy ∪ { b ↦ u }
      ◦ act1:  ParkedAt ≔ ParkedAt \ { b ↦ ParkedAt(b) } ⩤ @act1 ParkedAt ≔ { b } ⩤ ParkedAt
    END

  ◦ ReturnBicycle: not extended ordinary
    ANY
      ◦ u
      ◦ l
    WHERE
      ◦ grd0:  u ∈ ran(InUseBy) not theorem
      ◦ grd1:  l ∈ Lock \ ran(ParkedAt) not theorem
    THEN
      ◦ act0:  InUseBy ≔ InUseBy ⩤ { u }
      ◦ act1:  ParkedAt ≔ ParkedAt ∪ { (InUseBy~)(u) ↦ l }
    END

END
```

FIGURA 2.3. Máquina

Capítulo 3

Reglas de traducción a Python: *EB2Python*

Este capítulo presenta una especificación formal para una traducción del lenguaje *Event-B* a *Python*3. La traducción consta de distintos módulos funcionales. Cada módulo funcional define sus propias reglas. Uno de los módulos define la sintaxis matemática que comparten todos elementos. Los demás módulos son específicos y definen la composición de cada elemento general de los componentes.
En las siguientes secciones se presentan los módulos de extracción de información, traducción de variables, acciones de los eventos, la construcción de máquinas, definición de sintaxis y diseño de pruebas para la traducción.
El algoritmo diseñado se encuentra disponible en un respositorio en Bitbucket. El repositorio contiene las carpetas *Util* y *eventb_prelude*, las cuales son necesarias para la ejecución de las pruebas. La carpeta generateCode con las clases *EB2Python* y *Event* para la generación de código. La carpeta *Rodin_Examples* con algunos ejemplos base diseñados en el aprendizaje de la creación del algoritmo de generación de código. Y finalmente la carpeta *contracts* donde se encuentra un avance de la implementación de los contratos a agregar para futuros trabajos. El repositorio incluye también un *README* que explica como ejecutar el algoritmo.

3.1. Extracción

La interpretación de *Event-B* en *Rodin* está basada en archivos XML. Cada componente está conformado por cinco diferentes archivos. Los archivos se conforman por una raíz que tiene uno o más hijos; cada nodo comienza con el nombre org.eventb.core y se le añade otra palabra de representación [11].

Los archivos XML están diseñados de acuerdo con la sintaxis del lenguaje *Event-B* y se dividen por los elementos de cada componente de la sintaxis. Los componentes de la sintaxis van desde el elemento más general al más específico.
La extracción de los datos de los archivos XML se almacenan en la

TABLA 1. Archivos XML

Máquina	Contexo
bcm	bcc
bpo	bpo
bpr	bpr
bps	bps
bum	buc

clase *Event*. Los datos de la clase *Event* se ejecutan desde la clase *EB2Python*. La clase *EB2Python* tiene una referencia a la clase *Event*. La clase *Event* almacena los elementos de cada evento por separado. Los eventos tienen los siguientes elementos: parámetros con su respectivo tipo, guardas, acciones y el nombre de cada evento. La clase *EB2Python* utiliza un diccionario como estructura de datos identificar cada evento almacenado. *EB2Python* guarda en un arreglo los conjuntos y constantes; en otro arreglo las axiomas e invariantes.
En clase *EB2Python* se define el método *ReadActions*. En esta clase se recorren los archivos que contienen los datos utilizados para la traducción. El método recibe el nombre del archivo de la máquina y así conoce el contexto a evaluar y los archivos a revisar.

El primer archivo a revisar es el de extensión *bum*. El archivo revisa el nodo *seesContext*, el nodo tiene como hijo a *target*, cada nodo hijo identifica los contextos con los cuales la máquina trabaja, la estructura del archivo se observa en el Listing 1. La clase *EB2Python* conoce de esta manera los archivos de los contextos de los cuales se extraerán los datos, además, se tiene el nodo *invariant* y como hijo el nodo *predicate*, el nodo hijo tiene una invariante declarada en el sistema.

LISTING 3.1. Archivo *bum*

```
1 <org.eventb.core.seesContext name="↩
    _QmlXMCnsEeeUR6LcJDd9Nw" org.eventb.core.target=↩
    "contextSee"/>
2 <org.eventb.core.invariant name="↩
    _Lby34CntEeeUR6LcJDd9Nw" org.eventb.core.label="↩
    inv0" org.eventb.core.predicate="X"/>
```

```
3 <org.eventb.core.invariant name="↩
    _Lby34SntEeeUR6LcJDd9Nw" org.eventb.core.label="↩
    inv1" org.eventb.core.predicate="Y"/>
```

El archivo de extensión *bum* contiene los nodos *event*, que tiene un nodo hijo *label*, el cual identifica el nombre de un evento, además de otros nodos hijos como *parameter* con hijo *identifier*, *guard* con *predicate* y *action* con *assignment*. Puede existir el nodo *convergence*; el cual aparece si hay variantes declaradas en el sistema. Si existen variantes declaradas, se revisará el nodo *variant* con su respectivo hijo *expression*; se observan los elementos mencionados en el Listing 2. Los elementos mencionados anteriormente conforman los eventos o subsistemas que interactuaran con la máquina principal. Por último, el archivo *bum* revisa el nodo $refinesMachine$ con su hijo target, si ese nodo existe, la traducción recurre al archivo con extensión bum de la cual se realizó el refinamiento. La clase *EB2Python* realiza el refinamiento con el archivo *bum* de la máquina refinada hasta llegar a la máquina sin refinamiento.

LISTING 3.2. Archivo *bum*

```
1 <org.eventb.core.event name="↩
    _OPV2ECt4EeeUnaXEo56ubg" org.eventb.core.↩
    convergence="0" org.eventb.core.extended="false"↩
     org.eventb.core.label="Name">
2     <org.eventb.core.parameter name="↩
        _OPV2ESt4EeeUnaXEo56ubg" org.eventb.core.↩
        identifier="p"/>
3     <org.eventb.core.guard name="↩
        _OPV2Eit4EeeUnaXEo56ubg" org.eventb.core.↩
        label="grd0" org.eventb.core.predicate="p+2"↩
        />
4     <org.eventb.core.action name="↩
        _OPV2Eyt4EeeUnaXEo56ubg" org.eventb.core.↩
        assignment="X :=  p+2" org.eventb.core.label↩
        ="act0"/>
5 </org.eventb.core.event>
```

El archivo con extensión *buc* se conforma por el nodo *axiom*, con hijos *predicate*. El archivo contiene los datos de los axiomas declarados en la máquina principal. La estructura del archivo se observa en el

Listing 3.

LISTING 3.3. Archivo *buc*

```
1 <org.eventb.core.axiom name="↩
    _Bhg1gSnsEeeUR6LcJDd9Nw" org.eventb.core.label="↩
    ax0" org.eventb.core.predicate="finite(P)"/>
```

El archivo con extensión *bpr* del contexto revisa los nodos *prIdent* y *prProof*, con sus hijos *type* y *prSets* respectivamente. *EB2Python* da una primera revisión a los conjuntos y constantes en este archivo. El archivo *bpo* del contexto revisa los conjuntos y constantes tienen un valor definido. La revisión la realiza con los nodos *poIdentifier* y *action*, con sus hijos *name* y *assignment* respectivamente. La estructura del archivo se observa en el Listing 4.

LISTING 3.4. Archivo *bpr*

```
1 <org.eventb.core.prIdent name="Contabilidad" org.↩
    eventb.core.type="Z"/>
2 <org.eventb.core.prProof name="Event" org.eventb.↩
    core.confidence="1000" org.eventb.core.prFresh="↩
    " org.eventb.core.prGoal="p0" org.eventb.core.↩
    prHyps="" org.eventb.core.prSets="P">
```

Por último, se revisa el archivo *bcm*, el nodo *scEvent* y su hijo *label*. El archivo *bcm* identifica el tipo de parámetro de cada evento, con el nodo *scParameter*. La estructura del archivo se observa en el Listing 5.

LISTING 3.5. Archivo *bcm*

```
1 <org.eventb.core.scEvent name="Contabilidaf" org.↩
    eventb.core.accurate="true" org.eventb.core.↩
    convergence="0" org.eventb.core.extended="false"↩
     org.eventb.core.label="Event" org.eventb.core.↩
    source="/Agencia/AgenciaMach.bum|org.eventb.core↩
    .machineFile#AgenciaMach|org.eventb.core.event#↩
    _OPV2ECt4EeeUnaXEo56ubg">
```

```
2 <org.eventb.core.scParameter name="c" org.eventb.↩
    core.source="/Agencia/AgenciaMach.bum|org.eventb↩
    .core.machineFile#AgenciaMach|org.eventb.core.↩
    event#_OPWdICt4EeeUnaXEo56ubg|org.eventb.core.↩
    parameter#_OPWdISt4EeeUnaXEo56ubg" org.eventb.↩
    core.type="Personas"/>
```

3.2. Máquina principal

La traducción está realizada con la clase *EB2Python*, la cual reescribe algunas reglas de la traducción de $EB2JavadeCodegenerationfor$ *Event-B* [6].
La máquina principal se crea desde el método *mach*, $MultiThreadFile$ o $SequentialFile$ respecto a los que se quiere obtener en la traducción, la diferencia entre estos métodos es la librería que se agrega a la clase principal. Para la primera clase se añade *threading*. El código generado a *Python* inicializa un objeto y crea un arreglo de hilos o variables. El arreglo o variables son las referencias de los eventos a la máquina principal.
A partir de lo mencionado anteriormente, reescribimos las reglas de reescritura de *Java* a *Python*:

$$\frac{\begin{array}{c} EB2Python(sets\ s) = S \\ EB2Python(constants\ c) = C \\ EB2Python(variable\ v) = V \\ EB2Python(variant\ E(s)) = Va \\ EB2Python(events\ e) = E \\ EB2Python(event\ initialization\ then \\ A(s,c,v)\ end) = B1 \end{array}}{\begin{array}{c} EB2Python(\\ context\ ctx \\ sets\ s \\ constants\ c \\ end \\ Machine\ M\ sees\ ctx \\ variables\ v \\ variant\ E(s) \\ event\ initialization\ then\ A(s,c,v)\ end \\ events\ e \\ end) = E \end{array}}$$

Generando el código que se observa en el Listing 3.6.

LISTING 3.6. Generación código máquina principal

```
1  from threading import Lock
2
3  class M:
4
5      def __init__(self):
6          self.S = S
7          self.C = C
8          self.V = V
9          self.lock = Lock()
10         B1
11         #Creation of Python Threads
```

Los métodos de creación de conjuntos, constantes y variables, se definirá en su respectiva sección. La clase de la máquina principal definirá exclusivamente los axiomas e invariantes y se crearán a partir de los contratos, dónde serán traducidos por el compilador. La máquina agrega métodos para iterar los valores de los conjuntos, constantes y variables, también para consultarlos.

3.3. Eventos

Los eventos son creados desde el método *createStandartActions* de la clase *EB2Python*. Este método agrega una referencia de la máquina principal. También crea la guarda y la nueva asignación. Si la traducción es a código multihilo agrega el método *run* extensible de un hilo. Lo mencionado anteriormente se traduce bajo el mismo esquema, cada evento como una clase individual.
Se utilizan tres métodos para creación de los componentes de los siguientes eventos, *addGuard*, *addRun* y $startVariantVariable$. Cada método utiliza los datos que se almacenaron en la clase *Event*. El refinamiento se realiza con los datos agregados en la extracción de todos los archivos refinados, todos los eventos se traducen por igual, sin importar su nivel de refinación.
La ejecución multihilo reutiliza los métodos de *lock* para tener en cuenta la concurrencia, de tal manera que a una misma variable no accedan dos referencial al mismo tiempo.
Se define la regla para la creación de su reescritura.

$$\frac{\begin{array}{c} AddGuard(G(s,c,v,x)) \;=\; G \\ AddRun(A(s,c,v,x)) \;=\; B \\ startVariantVariable(St) \;=\; St1 \end{array}}{EB2Python(evt, St, any\ x\ where\ G(s,c,v,x)\ then\ A(s,c,v,x)\ end)}$$

El método *AddGuard* envia al compilador como parametros la sintaxis matemática declarada en la guarda. El compilador retorna la traducción de la sintaxis matemática recibida en código Python. *AddGuard* concatena todas las guardas declaradas y traducidas; en caso de existir una variante se ejecuta el método $startVariantVariable$ para agregarlo a la guarda; el elemento B en *AddRun* crea la nueva acción, además, agrega una condición de ejecución con referencia a la guarda. *AddRun* crea variables temporales para revisar los valores actuales y posteriormente cambiarlos en los reales. El compilador traduce la sintaxis matemática de *Event-B* asignada a la variable de igual manera que en la guarda.

A partir de lo dicho anteriormente se genera código *Python*, se observa en el Listing 3.7, el código mostrado pertener a la versión multihilo.

LISTING 3.7. Generación código Eventos

```
1  from threading import Thread
2  import threading
3
4  class evt(Thread):
5
6      def __init__(self,m):
7          threading.Thread.__init__(self)
8          this.machine = m
9
10     def guard_evt(x):
11         return G and St1
12
13     def run_evt(x):
14         if(guard_evt(x)==True):
15             B
16
17     def run():
18         while(True):
19             x = GuardValue()
20             self.machine.lock.acquire()
```

```
21          self.run_evt(x)
22          self.machine.lock.release()
```

3.4. Conjuntos, constantes y variables

Los conjuntos, constantes y variables son definidos por una serie de reglas. La clase *EB2Python* define las reglas. Las reglas declaran qué tipo de constante y conjunto son. Adicionalmente, revisa si sus valores fueron asignados en el modelado del sistema o se asignan aleatoriamente.

Los conjuntos se definen por llevar el símbolo $\mathbb{P}$ y se especifican por su contenido. Las constantes se definen por $\mathbb{Z}$ y $BOOL$. Los tipos de las variables se especifican únicamente para las pruebas si son un parámetro. *Python* es un lenguaje que interpreta el tipo de una variable sin ser declarado explícitamente. Las pruebas deben conocer que método se ejecutará para el tipo de variables que se enviarán en los parámetros para su correcta ejecución, también se definen en el método *run* de cada evento.

$$\frac{s = \mathbb{Z}(s = x) \mid BOOL(s = x)}{EB2Python(constant\ s) = x} \quad (\textsc{Constants})$$

$$\frac{s = \mathbb{P}(s)}{EB2Python(set\ s) = Enumerated(self.minInteger, self.maxInteger)} \quad (\textsc{Set})$$

$$\frac{s = \mathbb{P}(a \times b)}{EB2Python(set\ s) = BRelation()} \quad (\textsc{Set})$$

$$\frac{s = \mathbb{P}(a)}{EB2Python(sets) = BSet()} \quad (\textsc{Set})$$

La creación de conjuntos tipo *Enumerate* se define porque el valor al cual se le asigna el conjunto este contenido dentro del mismo, esto se especifica en los datos almacenados de los archivos XML. Los conjuntos *BRelation* se identifican por tener dos conjuntos, constantes o variables operadas por un producto cruz dentro de la asignación $\mathbb{P}$ de los conjuntos. Por último, se declaran los conjuntos *BSet* a los que no

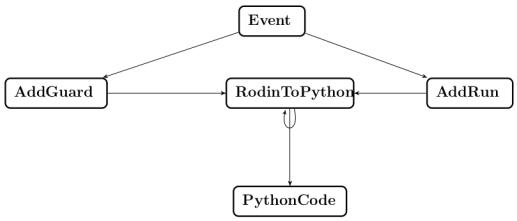

FIGURA 3.1. Diseño del compilador

cumplen las reglas mencionadas anteriormente.

3.5. Compilador

La reescritura de la sintaxis de los símbolos matemáticos de *Event-B* se define en el método *RodinToPython*. Se construye una tabla de símbolos [9], definiendo una estructura de datos que usa el proceso de traducción y un intérprete, dónde cada símbolo está asociado a una acción para la traducción, el tipo de dato y el ámbito de cada conjunto, constante o variable [8]. La tabla diseñada está construida únicamente para las expresiones matemáticas del lenguaje, no para toda su sintaxis.

La tabla se declara según la fuerza de sus operadores y sus operandos, también, recurre nuevamente al método si así lo requiere. El método *RodinToPython* identifica cuando el operador no se puede descomponer más y cuando se debe volver a ejecutar.
Como se menciono en la sección anterior, la guarda y asignación envían como parámetro sus expresiones declaradas de la sintaxis matemática del modelado. El método *RodinToPython* recibe los datos y los traduce, este proceso se refleja en la Figura 3.1.

A partir del diagrama anterior se obtiene una expresión traducida de la siguiente manera:

$$\frac{RodinToPython(r \in D \backslash (R \cup X))}{machine.D().difference(machine.R().union(machine.X())).has(p)}$$

Se observa como cada símbolo define como se agrupa, el orden de operación con los métodos definidos que cada conjunto, constante, variable o función tiene para su ejecución.

3.6. Pruebas

La traducción esta soportada bajo el diseño planteado en *Event-B*. El diseño se soporta con submódulos definidos por los eventos. Se desarrollo un mecanismo de descomposición para cada evento, simulando el comportamiento de la máquina a través del uso de eventos externos.

La ejecución de las pruebas se realizan con variables generadas aleatoriamente que simulan acciones, se envian como parámetros a cada evento.

Se debe conocer qué tipo de parámetro recibe cada evento. *EB2Python* generar variables aleatorias a partir de métodos que crean conjuntos o constantes, por lo cual debe conocer qué tipo de objeto son.

Las pruebas pueden generar un tiempo muerto de respuesta, esto se debe a que puede no ejecutarse la máquina principal al incumplir las invariantes y axiomas declaradas o no cumplirse la condición de las guardas. El valor aleatorio en caso de un sistema complejo que no cumpla sus restricciones debe esperar a generarse nuevamente. El valor generado puede quedar generándose múltiples veces. Se plantea para un futuro trabajo la implementación de SMT [7] para su correcto funcionamiento, actualmente se recomienda que la persona que diseña el sistema agregue sus propios valores a los elementos no asignados, para su correcta ejecución.
Para las pruebas secuenciales se crea un código como se refleja en la Figura 3.8.

LISTING 3.8. Código generado de las pruebas

```
1 class test_Machine():
2     def main():
3         M = Machine()
4         #Number of events
5         n = x
6         r = random.randint(0,n)
7         for i in range(0,r):
8             if(machine.guard_event1(x)==True):
9                 machine.run_event1(x)
```

Para las pruebas multihilo solo se ejecutará la clase máquina principal. La máquina principal ejecutara cada evento con sus respectivos

métodos *run* heredados de la clase *Threading*, estos llevarán a cabo el proceso de las pruebas.

Capítulo 4

Caso de estudio

Este capítulo muestra cómo se traduce el caso de estudio explicado en el Capítulo 2, a partir de las reglas definidas en el Capítulo 3.

La traducción se hace a partir de la extracción de datos de los archivos XML generados por *Rodin*. A lo largo del capítulo se mostrará el contenido de los archivos en la sección específica en la cual se utilizan para la traducción.

4.1. Máquina

La reescritura del caso de estudio comienza por máquina principal. La máquina principal almacena los conjuntos, constantes y variables declarados en la descripción del sistema.

Las Figuras 2.2 y 2.3, declaras tres conjuntos en el contexto y 4 variables en la máquina. Los conjuntos del contexto no son asignados a un valor específico dentro de la máquina, a diferencia de las variables a la cual se les asigna un valor en el evento $INITIALISATION$.

Todos los conjuntos definidos en el contexto tienen los datos de la forma $People$=$\mathbb{P}$($People$) en el XML, por lo cual se utiliza la regla *Enumerate* de la Sección 3.4. Las variables $User$ y *Active* tienen un valor inicial de $\emptyset$, por lo cual se declaran como un conjunto; estas variables tienen una estructura en los archivos de la forma $User$=$\mathbb{P}$($People$), por lo cual se traducen a partir la regla $BSet$. La variable $InUseBy$ toma el valor $InUseBy$=$\mathbb{P}$($Bicycle$x$People$), lo cual genera la aplicación de la regla $BRelation$. Por último, la variable $ParkedAt$ es asignada con un símbolo de conjuntos y se utiliza la regla $InUseBy$=$\mathbb{P}$($Bicycle$x$Lock$); combinando dos elementos diferentes. El algoritmo al conocer todas los conjuntos, constantes y variables que esta contiene, crean los métodos get y set de cada una.

El diccionario que tiene las referencias a la clase *Event* extrae las llaves, que corresponden a los nombres de los eventos del sistema. A partir de los nombres se crean las referencias a la máquina principal para cada evento.

Los invariantes y axiomas se traducen a partir del proceso de las acciones definidas de la Sección 3.2. Los invariantes y axiomas reescriben su expresión descrita a través del compilador de *EB2Python*, y este retorna el código *Python* definido. La expresión para reescribir se almacena de igual manera como se modela en el sistema de *Event-B*. La máquina es traducida a código *Python* de como se observa en el Listing 4.1.

LISTING 4.1. Máquina caso de estudio

```
1  class IBSMach:
2
3      def __init__(self):
4          self.max_integer = Utilities.getMaxInteger↩
               ()
5          self.min_integer = Utilities.getMinInteger↩
               ()
6                  #Numero de eventos declarados
7          self.n_events = 6
8          self.events = []
9          self.lock = Lock()
10         self.events.append(RegisterUser(self))
11         self.events.append(UnregisterUser(self))
12         self.events.append(ActivateUser(self))
13         self.events.append(DeactivateUser(self))
14         self.events.append(BorrowBicycle(self))
15         self.events.append(ReturnBicycle(self))
16         tmp = Test_IBSMach()
17         self.Bicycle = Enumerated(self.min_integer,↩
               self.max_integer)
18         self.InUseBy = BRelation()
19                 self.ParkedAt = Utilities.someVal(↩
                       BRelation().cross(self.Bicycle,↩
                       self.Lock).pow())
20         self.Lock = Enumerated(self.min_integer,↩
               self.max_integer)
21         self.ParkedAt = BRelation()
22         self.People = Enumerated(self.min_integer,↩
               self.max_integer)
```

```
23          self.User = BSet()
24          self.Active = BSet()
25          for i in range(self.n_events):
26              self.events[i].start()
27
28      def get_Bicycle(self):
29          return self.Bicycle
30
31      def set_Bicycle(self,Bicycle):
32          self.Bicycle = Bicycle
33
34      def get_InUseBy(self):
35          return self.InUseBy
36
37      def set_InUseBy(self,InUseBy):
38          self.InUseBy = InUseBy
39
40      def get_Lock(self):
41          return self.Lock
42
43      def set_Lock(self,Lock):
44          self.Lock = Lock
45
46      def get_ParkedAt(self):
47          return self.ParkedAt
48
49      def set_ParkedAt(self,ParkedAt):
50          self.ParkedAt = ParkedAt
51
52      def get_People(self):
53          return self.People
54
55      def set_People(self,People):
56          self.People = People
57
58      def get_User(self):
59          return self.User
60
61      def set_User(self,User):
62          self.User = User
63
64      def get_Active(self):
65          return self.Active
66
67      def set_Active(self,Active):
```

```
68          self.Active = Active
```

4.2. Eventos

Todos los eventos se traducen bajo un mismo esquema. La primera parte de la traducción se define por la declaración del objeto, si es secuencial o multihilo; la declaración se usa para agregar o no la extensión de la librería *threading*. En ambos casos se agrega la referencia a la máquina principal.

Como se menciona en la Sección 3.3 se realiza primero el método *AddGuard*, el cual extrae los parámetros y las guardas declaradas en *Event-B*. Las guardas son extraídas con la misma sintaxis de su declaración del sistema y los parámetros se traducen como fueron extraídos. Se reescriben igual dentro de la declaración del método en *Python*. Por último, cada guarda se envía al compilador (Referencia a la sección); el método recibe su traducción y la concatena con las demás guardas.

El método *AddRun*, al igual que *AddGuard*, realiza el procedimiento de traducción con los parámetros, e incluye una instrucción condicional con referencia al método de la guarda creada. El condicional valida si la guarda se cumple. *AddRun* primero construye un set de las variables a reasignarse, se guardan en una variable temporal. Posteriormente se utilizan para ser operadas y reasignadas con su nuevo valor. La sintaxis de asignación se pasa al compilador, el cual retorna la reescritura de la sintaxis en código *Python*. La sintaxis enviada se extrae de igual manera que se define en la declaración del sistema.

La traducción queda como se observa en el Listing 4.2.

LISTING 4.2. Evento caso de estudio

```
1 class ActivateUser(Thread):
2
3     def __init__(self,m):
4         threading.Thread.__init__(self)
5         self.machine = m
6
7     def guard_ActivateUser(self,u):
8         return self.machine.get_User().difference(↩
              self.machine.get_Active()).has(u)
```

```
9
10
11      def run_ActivateUser(self,u):
12          if self.guard_ActivateUser(u):
13              Active_tmp = self.machine.get_Active()
14              self.machine.set_Active(self.machine.↩
                    get_Active().union(BSet([u])))
15              print('ActivateUser executed u: '+str(u↩
                    )+ ' ')
16
17          def run(self):
18          while(True):
19              u =Utilities.someVal(BSet(Enumerated(1,↩
                    Utilities.getMaxInteger())))
20              self.machine.lock.acquire()
21              self.run_ActivateUser(u)
22              self.machine.lock.release()
```

A continuación, se muestran algunas traducciones creadas para la reasignación de valores en el método addRun, se realiza para no agregar la estructura del mismo código anterior:

$$\frac{Active \backslash \{u\}}{self.machine.get_Active().difference(BSet([u]))}$$

$$\frac{InUseBy \cup \{b \mapsto u\}}{self.machine.get_InUseBy().union(BRelation([Pair(b,u)]))}$$

$$\frac{ParkedAt \backslash \{b \mapsto \text{ParkedAt}(b)\}}{self.machine.get_ParkedAt().difference(BRelation([Pair(b, self.machine.get_ParkedAt().apply(b))]))}$$

$$\frac{ParkedAt \rhd \{u\}}{self.machine.get\_InUseBy().rangeSubtraction(BSet([u]))}$$

4.3. Pruebas

Para la creación de las pruebas se utilizan la información extraída del tipo de variable de los parámetros, los parámetros que recibe cada evento y de los eventos.

Se definen de manera aleatoria los valores de los parámetros. El método que genera el valor aleatorio se define por medio de qué tipo de parámetros se recibe en la guarda.

Con esto se define la ejecución de los valores aleatorios como parámetros para el método *guard* y *run* de cada evento. Se observa la traducción secuencial en el Listing 4.3. Para la traducción multihilo se realiza por medio del método *run* del hilo, incluyendo funciones para sincronización. En la traducción secuencial, se genera un número aleatorio para lanzar las pruebas cantidad aleatoria de veces.

LISTING 4.3. Código generado de las pruebas

```
1  def main():
2      test = Test_IBSMach()
3      machine = IBSMach()
4      n = 6
5      p = Utilities.someVal(BSet(Enumerated(1,↩
           Utilities.getMaxInteger())))
6      u = Utilities.someVal(BSet(Enumerated(1,↩
           Utilities.getMaxInteger())))
7      b = Utilities.someVal(BSet(Enumerated(1,↩
           Utilities.getMaxInteger())))
8      l = Utilities.someVal(BSet(Enumerated(1,↩
           Utilities.getMaxInteger())))
9      total = random.randint(0,n)
10     for i in range(0,total):
11         if machine.evt_RegisterUser.↩
               guard_RegisterUser(p):
12             machine.evt_RegisterUser.↩
                   run_RegisterUser(p)
13         if machine.evt_UnregisterUser.↩
               guard_UnregisterUser(p):
14             machine.evt_UnregisterUser.↩
                   run_UnregisterUser(p)
15         if machine.evt_ActivateUser.↩
               guard_ActivateUser(u):
16             machine.evt_ActivateUser.↩
                   run_ActivateUser(u)
17         if machine.evt_DeactivateUser.↩
               guard_DeactivateUser(u):
18             machine.evt_DeactivateUser.↩
                   run_DeactivateUser(u)
```

```
19          if machine.evt_BorrowBicycle.↩
               guard_BorrowBicycle(u, b):
20             machine.evt_BorrowBicycle.↩
                  run_BorrowBicycle(u, b)
21          if machine.evt_ReturnBicycle.↩
               guard_ReturnBicycle(u, l):
22              machine.evt_ReturnBicycle.↩
                   run_ReturnBicycle(u, l)
```

Capítulo 5

Ejecución

Este Capítulo muestra la ejecución de los modelos generados, de manera secuencial y multihilo, además, como son ejecutados los modelos por medio de las pruebas. El código generado se encuentra en una carpeta con el nombre del modelo seguido de la forma de ejecución, se debe comprobar que existan las carpetas *Util* y *eventb_prelude* como apoyo a la correcta ejecución de los modelos [6].

5.1. Código versión secuencial

Luego de genera el código traducido se crea la carpeta nombrada de la forma $nombreModelo\_sequential$. Para la ejecución del modelo es necesario dirigirse a esa carpeta y ejecutar desde consola el archivo de las pruebas generadas. El archivo se nombra de la forma $Test\_nombreModelo.py$. Para ejecutar el archivo se realizará desde consola, por ejemplo en linux se realiza con el comando $python3$ $Test\_nombreModelo.py$.
Para el caso de estudio se observan diferentes resultados. Los resultados varían respecto a los valores aleatorios generados. Los resultados obtenidos pueden cambiar de valor, pero no el comportamiento. El comportamiento se mantiene respecto al sistema modelado. Los resultados se observan en Figura 5.1

La Figura 5.1 muestra la generación de números aleatorios para p, u, b y l. Se ejecutan los eventos por medio del proceso a llamar "adecuado", que el sistema plantea. Comienza con una cantidad de 75 usuarios registrados y activados, tomando ciclas y retomándolas, finalizando con poner un usuario inactivo y desactivando su registro. El número de usuarios fue dado aleatoriamente y se trasnmitio a los eventos, el número se ilustrar en el ejemplo.

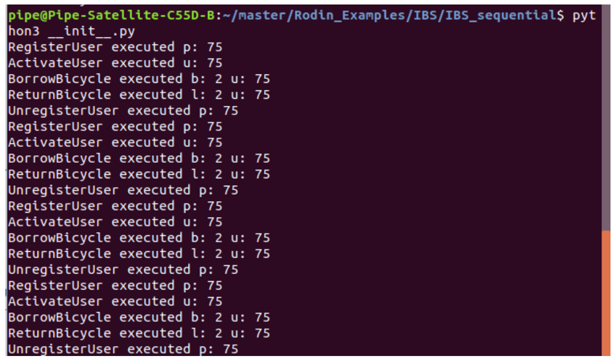

FIGURA 5.1. ejecución secuencial

5.2. Multihilo

Al igual que se mencionó en la Sección anterior se creó una carpeta nombrada de la forma *nombreModelo_multi_threaded*. Dentro de la carpeta se encuentra el archivo *Test_nombreModelo.py*. Para la ejecución se realizará desde consola de la misma manera que en el modelo secuencial, observamos resultados de la siguiente forma:

La figura 5.2 muestra como son generados aleatoriamente más números y se activan todos los eventos al tiempo. La ejecución del modelo muestra que siempre prevalecen las condiciones de las bicicletas disponibles respecto a la cantidad de usuarios activos que desean tomarlas, por lo cual el evento *BorrowBicycle* no se ejecuta hasta que *ReturnBicycle* se lleve a cabo, y que nunca habrá más usuarios desregistrándose que la cantidad actual.

5.3. Fallo de pruebas

Como se ha mencionado las anteriormente las pruebas pueden llegar a fallar o quedarse en espera cuando uno de sus valores aleatorios no cumple las condiciones dadas. Para esto se genera un ejemplo donde el valor aleatorio generado se genera con el fragmento de código $Utilities.someSet(BSet(Enumerated(1, Utilities.getMaxInteger())))$. El fragmento de código presentado genera conjuntos aleatorios de los cuales pueden no coincidir sus elementos con las condiciones dadas y en la operaciones realizadas de las guardas.
Las pruebas generadas para estos casos se quedan en espera hasta que se generen conjuntos con valores similares. Estos valores pueden tomar

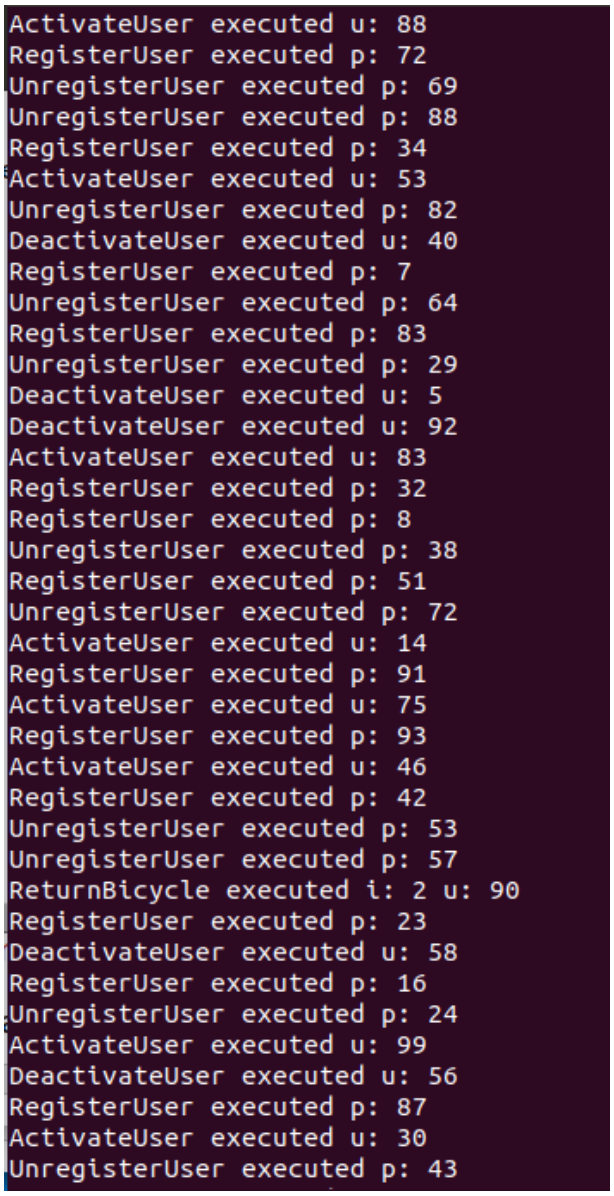

FIGURA 5.2. Ejecución multihilo

un tiempo de espera muy alto. De igual manera puede reincidir con valores númericos de alta complejidad y multiples condiciones.

Capítulo 6

Conclusión

El modelado de sistemas reduce la complejidad para la creación de Software. El generar código a partir de especificaciones formales permite comprobar las propiedades de interés descritas. Además, permite tomar más recursos en la descripción del modelado sin ir directamente a la implementación del código.
El algoritmo creado brinda herramientas para la traducción del modelo diseñado. El código que se genera brinda una ejecución y revisión de lo desarrollado desde una perspectiva diferente.
El documento brinda pautas para futuros trabajos de autogeneración de código y un producto para ser utilizado en sistemas que no requieren valores complejos aleatorios para ser ejecutados.
Se plantea como trabajo futuro la inclusión de contratos en el código traducido, el cual permita abarcar todos los componentes descritos en *Event-B*, además, adicionar SMT para generar valores aleatorios respecto a las condiciones declaradas en el modelado del sistemas.

Bibliografía

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press.

[2] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. *Rodin: an open toolset for modelling and reasoning in Event-B*. STTT, 12(6):447–466, 2010.

[3] Jean-Raymond Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press. 2005.

[4] Michael Butler. *Towards a cookbook for modelling and refinement of control problems*. 2009.

[5] Michael J Butler. *Mastering system analysis and design through abstraction and refinement.*, 2013.

[6] Victor Rivera, Néstor Catano, Tim Wahls, and Camilo Rueda. *Code generation for event-b*. International Journal on Software Tools for Technology Transfer, 19(1): 31–52, 2017.

[7] David D'eharbe, Pascal Fontaine, Yoann Guyot, and Laurent Voisin. *Smt solvers for rodin*. In International Conference on Abstract State Machines, Alloy, B, VDM, and Z, pages 194–207. Springer, 2012.

[8] Christophe Métayer and Laurent Voisin. *The event-b mathematical language.Systerel*, March, 2009.

[9] Jacinto Ruiz Catalán. *Compiladores: teoría e implementación*. RC Libros, 2010.

[10] C. Métayer (ClearSy) J.-R. Abrial, L. Voisin (ETH Zürich). *EventB Language*. Information Society, 2005.

[11] Jastram, M., & Butler, P. M. (2014). *Rodin User's Handbook: Covers Rodin v. 2.8*.

[12] Requirements Specification Tutorial. http://wiki.event-b.org/index.php/Requirements_Tutorial

[13] Camilo Rueda. *Introducción al Modelado de Sistemas*, 2016.