Pontificia Universidad JAVERIANA Cali

Facultad de Ingeniería
Secretaría de la Facultad

Acta de Correcciones al Proyecto de Grado
Ingeniería de Sistemas y Computación

Fecha: **4 Agosto 2014**

Autores: Carlos Eduardo Gómez Hernández

Nombre del Proyecto de Grado: Especificación de Patrones de Diseño de Event-B como procesos NTCC

Director: Carlos Alberto Olarte Vega

Como indica el artículo 2.27 de las Directrices de Trabajo de Grado, he verificado que los estudiantes indicados arriba han implementado todas las correcciones que los Jurados del Proyecto de Grado definieron que se efectuaran, como consta en el Acta de Calificación correspondiente.

Firma de Director(a) del Proyecto de Grado

Nota de Aceptación

Aprobado por el Comité de Trabajo de Grado en cumplimiento de los requisitos exigidos por la Pontificia Universidad Javeriana para optar el título de Ingeniero de Sistemas y Computación.

Dr. JAIME AGUILAR ZAMBRANO
Decano de la Facultad de Ingeniería

ING. ANTAL ALEXANDER BUSS
Director Carrera Ingeniería de Sistemas.

ING. CARLOS ALBERTO OLARTE VEGA
Director Trabajo

ING. GERARDO MAURICIO SARRIA
Jurado 1

ING. CAMILO RUEDA CALDERON
Jurado 2

Especificación de Patrones de Diseño de Event-B como procesos NTCC

Pontificia Universidad Javeriana Cali

Carlos Eduardo Gómez Hernández

Junio 24, 2014

Resumen

Los métodos formales son importantes para modelar sistemas críticos, ya que con ellos se tiene una certeza del correcto funcionamiento de dichos sistemas. Por ejemplo, en el contexto de cirugías asistidas por computador, uno esperaría que cada acción del robot corresponda a la acción que el usuario (en este caso el doctor) esté realizando, después de todo, está en juego la vida de una persona.

Dentro del diseño de software, Event-B usa modelos llamados patrones de diseño, y tal como se realiza en programación clásica, el reutilizar código es importante para hacer mejores programas y evitar pensar en problemas que previamente fueron resueltos. Event-B usa los patrones de diseño para reutilizar modelos que previamente han sido verificados, haciendo el proceso de modelamiento más eficiente.

NTCC es un lenguaje concurrente que permite expresar de manera declarativa sincronización de procesos en diferentes unidades de tiempo. La idea general del proyecto es crear modelos de patrones de diseño en NTCC y proponer una forma de traducir dichos modelos a una estructura en Event-B.

La manera para poderlo realizar es creando modelos generales de algunos patrones de diseño de dichos sistemas, modelarlos en ambos tipos de especificación de sistemas, observar semejanzas y finalmente indicar una traducción generalizada sobre la especificación desde NTCC hasta Event-B.

El proyecto de grado entonces explorará estos modelos, analizará sus semejanzas y propondrá un modelo de traducción entre modelos. Para realizarlo propondrá un lenguaje de alto nivel para que el diseñador le sea más sencillo especificar dichos patrones. Este lenguaje, como veremos, se puede traducir de manera simultánea a los dos tipos de especificaciones mencionados anteriormente manteniendo la estructura general del modelo que se quiere especificar. Así, la tarea del modelador resulta más sencilla utilizando el método propuesto en este trabajo de grado.

Abstract

Formal methods are important since they specify critical systems. Take for instance a system for assisted surgeries where we expect that each movement controlled by the system corresponds exactly to the doctor's actions. After all, there is a life in danger.

Related to software design, Event-B uses design patterns, similar to other programming languages, in order to reuse already defined models that solve well-known problems. Such patters thus ease the modeling process.

NTCC is a process calculus that allows to model in a declarative manner synchronization of process along time units. The general idea of this project is to create models of design patterns in NTCC and propose one way to translate those models into an Event-B structure. For that, we proceed as follows. We propose a NTCC model of the design pattern. Then, by identifying similarities among different models, we propose a general translation from NTCC specifications into Event-B. We also propose a high level language to make more comfortable the specification of patterns. This language, as we shall see, handles different kinds of pattern specifications.

Dedicación

Le dedico este trabajo de grado a mis padres, por el apoyo económico y emocional a lo largo de todos los semestres, a mi hermana, a mis compañeros de carrera que dieron su ayuda a cualquier problema que tuve en la carrera.

Agradecimientos

Agradezco a mis seres queridos, a mi familia por todo el apoyo que me brindaron a lo largo del desarrollo de mi título profesional y me enseñaron muchas cosas para así no desfallecer o perderme en el transcurso de dicho proceso.

Agradezco a mi director de tesis, Carlos Alberto Olarte Vega por toda la dedicación para la realización de este proyecto, no solo por el tiempo dedicado sino por su asertiva opinión y sugerencia para el desarrollo de la misma.

Agradezco a todos los profesores que hicieron parte de mi formación académica. Ya sea que me hayan gustado o no su metodología de enseñanza, siempre harán parte de la cómo fui formado en esta universidad.

Y agradezco a mis compañeros de carrera, por todo lo que vivimos, me ayudaron y me brindaron su apoyo a lo largo de este tiempo y siempre con alegría y diversión en todo lo que se hacía.

Índice general

Lista de Tablas

Lista de Figuras

Capítulo 1

Introducción

En el mundo de los métodos formales, existe un sin número de lenguajes que permiten modelar los sistemas de la vida real basado en lenguajes matemáticos precisos. Entre estos lenguajes, se destacan dos que son el objeto de interés de este proyecto: el lenguaje de procesos concurrentes NTCC [NPV02] y el framework de modelado de sistemas Event-B [Abr10].

El lenguaje NTCC (Non-Deterministic Concurrent Constraint Programming) es un lenguaje que permite modelar procesos concurrentes de manera no determinista, así como hacer pruebas sobre el modelo creado, todo esto basado en la teoría de cálculos de procesos.

El framework Event-B, es un modelo que permite definir y modelar de manera formal un sistema completo. Adicionalmente, Event-B ofrece herramientas para probar el correcto funcionamiento del sistema, algo que es muy difícil de asegurar en el campo de la programación.

Al diseñar un sistema, hay que tener en cuenta el tiempo en que cada proceso debe ser ejecutado. El diseño de patrones de diseño requiere planear de cierta manera la secuencia en que cada proceso debe ser ejecutado, y esta planeación a la larga se hace de manera "manual". Esta forma de realizarlo genera un desgaste en tiempo de implementación ya que estos patrones pueden ser reutilizados para ciertos modelos en general.

Por esta razón, el presente trabajo va encaminado a buscar una forma que permita reutilizar patrones de diseño, empezando desde un modelo en NTCC que pueda ser luego esquematizado (modelado como una estructura general) en Event-B. Para realizar tal tarea, el conocimiento de éstos lenguajes formales es indispensable, así como el conocimiento de los patrones de diseño ya existentes ya que con ellos, se desea buscar ciertos patrones capaces de

ser utilizados para muchos problemas.

Para esto vamos a utilizar un lenguaje de alto nivel que permite dar al diseñador una facilidad a la hora de modelar estos patrones gracias a su estructura que permite definir de manera "natural" dichos modelos. Además de lo anterior, el lenguaje de alto nivel permite generalizar los modelos tanto en NTCC como en Event-B.

La manera de lograr este objetivo va desde la fase de exploración hasta la fase de análisis y pruebas de los modelos. En el capítulo 2 se mencionará las definiciones generales que se requieren para el desarrollo del proyecto. En el capítulo 3 se analiza la plataforma Rodin y NTCC. El capítulo 4 abarca la Fase Exploratoria en la que se analizan algunos modelos de patrones de diseño para finalmente en el capítulo 5 crear el lenguaje de alto nivel y en el capítulo 6 definir las traducciones desde este lenguaje a NTCC y Event-B.

Cabe anotar que la idea de patrón de diseño en Event-B no es nueva [CMR07]. Sin embargo, normalmente dichos patrones ofrecen una visión limitada de las sincronizaciones de los procesos y se reducen a interacciones entre dos agentes. El fin último de este trabajo es proveer una visión mucho más general de sincronización de procesos como patrones de diseño.

Capítulo 2

Marco Referencial

2.1 Áreas Temáticas

- D.3.2: Software - Programming Languajes - Language Classifications - Constraint and logic languages.
- F.4.1: Mathematical Logic and Formal Languages - Mathematical Logic - Logic and constraint programming.
- F.4.1: Mathematical Logic and Formal Languages - Mathematical Logic - Model Theory

2.2 Marco Teórico

2.2.1 Patrones de Diseño

Los patrones de diseño (de ahora en adelante PD), son un concepto de ingeniería cuyo objetivo es facilitar el desarrollo de sistemas. Tal como lo dice [HFA09], los PD no necesariamente son un producto terminado, pero son un *template* de cómo resolver problemas que se pueden usar en diferentes situaciones. Así que se puede decir que los PD son una estructura general de ciertas propiedades de un sistema. Un patrón de diseño clásico es el MVC o Modelo Vista Control, donde el Modelo es la representación de la información que posee el sistema como los accesos a ella, la vista es la parte gráfica del modelo (Interfaz de Usuario) y el controlador responde a las acciones que le manda la vista.

2.2.2 CCP

Concurrent constraint programming, es un formalismo de concurrencia basado en el modelo de compartimiento de variables. En [NPV02], se menciona que sus orígenes están dados por las ideas de "computar con restricciones", "programación lógica concurrente" y "programación lógica por restricciones". Lo anterior es la base en la cual se ha creado CCP que permite realizar pruebas matemáticas sobre el modelo para verificar su correctitud. CCP ha venido siendo desarrollado durante más de 20 años. En [Sar93] explican su estructura y funcionamiento y en [NPV02] mencionan el estado de arte completo de este formalismo. CCP está basado en el cálculo de procesos. Éste lenguaje utiliza un pequeño conjunto de operadores primitivos que le permite realizar un sin número de operaciones y modelos concurrentes (lo que lo hace atractivo a los modeladores). En este lenguaje se tiene un Store que es el lugar donde todos los *contraints* se encuentran y donde los procesos pueden realizar sus operaciones con los operadores del modelo. Entre estos operadores se encuentran el "Operador Tell" y el "Operador Ask" que guarda y pregunta respectivamente un constraint en el Store. También existe la "Composición Paralela", que simplemente combina procesos concurrentes (realiza más de 1 proceso) y un operador de localidad que introduce variables locales y restringe las interfaces de los procesos que interactúan con los demás, es decir, solo actúa con ciertos procesos. Con todo lo anterior CCP puede entonces modelar procesos reactivos de manera formal y hacer verificaciones sobre el modelo para evitar estados o comportamientos no deseados sobre este.

2.2.3 TCC

TCC, o *Temporal Concurrent Constraint Programming* es un formalismo que combina la idea de CCP con los lenguajes sincrónicos [NPV02].

En [NPV02] mencionan que TCC está dividido en *intervalos discretos*, esto es, unidades de tiempo discretas, que por cada tiempo un proceso *determinístico CCP* es ejecutado (recibe un *estímulo*) y aquel proceso que no pueda ser ejecutado, se espera hasta la otra unidad de tiempo para ejecutarlo si está bajo un operador *Next*.

TCC extiende del modelo de CCP con los operadores temporales *Next P* y *unless c Next P*. El primer operador indica que el proceso P se ejecutará en la próxima unidad de tiempo y el segundo operador indica que el proceso P se ejecutará en la próxima unidad de tiempo al menos que c pueda deducirse

del Store al final del intervalo de tiempo presente.

2.2.4 NTCC

NTCC proviene de las palabras *Non-Deterministic Time Concurrent Constraint Programming* [NPV02]. NTCC extiende los operadores de TCC que a su vez extiende a CCP [NPV02].

NTCC es una extensión del modelo de TCC. Esto implica que NTCC obtiene todos los operadores de TCC (que incluyen los de CCP) y además extiende a TCC agregando *guarded-choices* que son guardas para que se seleccione de manera no determinista los procesos [ORV13].

Los nuevos operadores que extiende NTCC de TCC son la *sumatoria "+"* y la *estrella "*"*. El primer operador selecciona cualquier proceso que esté dentro de la suma, y el segundo operador ejecuta un proceso P en una unidad de tiempo futura pero aleatoriamente. Los operadores mencionados anteriormente funcionan como siguen:

Skip Es un proceso que no realiza ninguna acción.

Tell(c) Agrega una restricción (*constraint*) al *Store*.

When c Do p Indica que cuando se pueda deducir del *Store* la restricción c entonces ejecutará en un mismo tiempo p.

$\boldsymbol{P} \parallel \boldsymbol{Q}$ Representa la composición paralela de P y Q, es decir, se ejecuta P y Q en un mismo tiempo.

!P Representa la replicación del proceso P en cada unidad de tiempo.

Next P Ejecuta el proceso P en la siguiente unidad de tiempo.

local ($\bar{x}$ **c**) **P** Se comporta como P pero toda la información en las variables $\bar{x}$ producidas por P solo lo puede ver P y la información creada por otros procesos no la puede ver P.

$\sum_{i \in I}$ **when** c_i **do** P_i Escoge no deterministicamente cualquier proceso P_i que su condición c pueda ser deducido del Store. Si no hay ninguna condición que se pueda ejecutar queda el proceso bloqueado hasta que ingresen nuevos *constraints* al Store. Por ejemplo $whenadob + whenadoc$ entonces si se tiene a puede ejecutarse tanto b como c.

$\star$ **P** El proceso P se ejecuta en un tiempo ilimitado pero finito, es decir, el tener $\star P$ indica que P puede ejecutarse en el tiempo 1, o en el tiempo 2 o en el tiempo 3 y así sucesivamente.

El comportamiento del modelo es como sigue: el *store* empieza vacío, y mediante los procesos de *tell* agrega *contraints* a él. Además de lo anterior, se puede obtener información del store mediante otros operadores para poder agregar más *constraints* al store. Cuando ya no se pueda agregar más restricciones al *store* (cuando no existe una operación que se pueda realizar con el *store* actual) entonces se culmina el tiempo en ese proceso y se cambia el próximo tiempo. En este momento los *constraints* que quedaron en el *store* son los que se dicen que se "observaron" en este tiempo. Al próximo tiempo el *store* vuelve a quedar vacío y se reinicia el proceso.

Para mejor entendimiento supongamos que se tiene el siguiente proceso y queremos ver su comportamiento a lo largo del tiempo:

```
P = tell(a) || tell(b)
    || when (a ∧ b) do (Next( tell(c)) )
```

El proceso empieza con el *store* vacio. Los procesos *tell(a)* y *tell(b)* adicionan al store las restricciones "a" y "b" respectivamente. Luego, el proceso *when* $a \wedge b$ *do next (tell c)* puede deducir del store la guarda $a \wedge b$, por lo que el proceso *Next (Tell c)* puede ser ejecutado.

Como el proceso *Next (tell c)* hace un delay (espera 1 tiempo), entonces en la primera unidad de tiempo se observa la ejecución de las restricciones "a" y "b". En la segunda unidad de tiempo el proceso *Tell c* es ejecutado y este tiempo finaliza pudiendo observar en el *store* la restricción *c*.

El proceso explicado anteriormente se puede observar a continuación. Se supone que los procesos se observan mediante $\langle P, Store \rangle$, siendo P los procesos a ejecutar y el Store aquellas restricciones que se encuentran en el proceso:

Tiempo 1 :

$$\langle\ tell(a) \parallel tell(b) \parallel where\ (a \wedge b)\ do\ (Next(tell(c))), \emptyset \rangle$$

$$\langle\ where\ (a \wedge b)\ do\ (Next(tell(c)))\ ,\ (a \wedge b)\ \rangle$$

$$\langle\ Next\ (tell(c))\ ,\ a \wedge b\ \rangle$$

$$\langle\ Next\ (tell(c))\ ,\ a \wedge b\ \rangle$$

Tiempo 2 :

$$\langle\ tell(c)\ , \emptyset\ \rangle$$

$$\langle\ \emptyset\ ,\ c\ \rangle$$

En conclusión se dice que este proceso obtiene “a” y “b” en el tiempo 1 y “c” en el tiempo 2.

2.2.5 EVENT-B

Event-B es un método formal de modelado de sistemas [Abr10]. Sus principales características están dadas por su uso de la teoría de conjuntos para hacer el modelo y las pruebas matemáticas. Usa refinamientos como método de abstracción del modelo y las pruebas matemáticas garantizan la consistencia entre los distintos niveles de atracción.

Event-B fue creado por Jean-Raymond Abrial [Abr10], un científico de la computación francés conocido por ser el inventor de los métodos formales Z y B. Luego de concebir B, creó Event-B, para realizar operaciones de B con una interfaz amigable.

Los modelos creados por Event-B son descritos en términos de dos constructores básicos: contextos y máquinas [Abr10]. Los contextos contienen la parte estática del modelo, mientras que las máquinas contienen la parte dinámica. Los contextos pueden tener los conjuntos, constantes, axiomas, donde los conjuntos pueden ser similares a tipos. Las máquinas en cambio, contienen variables, invariantes y eventos.

El proceso es como sigue: se crea un contexto que contiene lo estático del mundo a modelar, luego se crea la máquina, que es la que contiene las variables que queremos 'observar'. Al inicio del modelo, se crea una máquina llamada máquina abstracta, un nivel de refinamiento muy general, para luego introducir nuevos detalles con nuevas máquinas (utilizando más refinamientos). El proceso de prueba lo definen en [Abr10], mostrando que en cada refinamiento demuestra que los invariantes de la máquina se mantienen por evento y que las variables mantienen su estado consistente.

Un ejemplo de uso de este lenguaje para modelar sistemas se menciona en el capítulo 3, al hablar de Event-B.

2.3 Trabajos Relacionados

En [HFA09], Hoang et al. describen la forma de modelar patrones de diseño en Event-B. Al inicio da una introducción al modelamiento en Event-B, describe formalmente los problemas que va a tratar y describe la forma de como los desarrolla en Event-B.

Este paper es importante para el desarrollo del proyecto ya que, además de dar un resumen al modelamiento en Event-B, describe y explica un sistema que incorpora patrones de diseño, además muestra la manera de usar este patrón de diseño en otro sistema con ayuda de una herramienta propia de Event-B.

Este paper está muy ligado al presente trabajo porque tiene el mismo objetivo: modelar patrones de diseño en Event-B, solo que el paper no menciona en ninguna parte el modelo de PD en NTCC, que es la extensión que haría el presente proyecto.

Sin embargo, a diferencia de [HFA09], nuestra idea es utilizar un lenguaje de alto nivel para poder modelar estos patrones de diseño. Este lenguaje, que se explicará en el capítulo 5, proporcionará una forma más natural de representar el modelo de patrones de diseño y, bajo el mismo objetivo de lo que se expresa en este paper, el lenguaje permitiría el manejo de reutilización de código.

En [OR14] los autores utilizan *Multiparty Session Types* [DY12] para especificar la comunicación global de un protocolo de varios agentes. El tipo global es el modelo abstracto del sistema. Dentro de este modelo se encuentra el Contexto que especifica las constantes necesarias para representar los estados del tipo global. Los eventos que se crean en este modelo corresponden a los sistemas de transición, en otras palabras, todo el modelo abstracto representa la "coreografía" que los agentes deben de seguir. Los tipos locales vienen a extender el modelo previamente definido, es decir, este tipo viene a ser el "refinamiento" del modelo. El fin último es especificar patrones de comunicación utilizando *Session Types* para facilitar la tarea de modelado.

Capítulo 3

Conocimiento de la Plataforma

3.1 Plataforma Rodin

Como se ha mencionado anteriormente, Event-B es un método formal para el modelado de sistemas. La plataforma Rodin es un ambiente donde se puede especificar estos programas `http://www.event-b.org/`. Desarrollado por Jean-Raymond Abrial, Rodin permite el uso de una interfaz de usuario para que se pueda modelar mas fácil los sistemas.

Rodin es extendible libremente y sus interfaces, modelos, y métodos de prueba son extensiones propias que han hecho a esta plataforma más "amigable". A continuación presentamos un vistazo general de esta plataforma.

3.1.1 Interfaz Rodin

Rodin posee dos vistas dentro de su aplicativo. Estas dos vistas fueron divididas así por la forma ya explicada de la creación de los modelos; la parte estática del modelo denominado *Contexto* y la parte que posee las variables, invariantes y eventos llamado *Máquina*. No olvidar que se crea el *Contexto* primero y luego se especifica cómo se va a comportar en la *Máquina*.

Workspace

Basado en Eclipse, Rodin requiere que el usuario especifique una ruta donde se va a guardar los proyectos denominado *Workspace*.

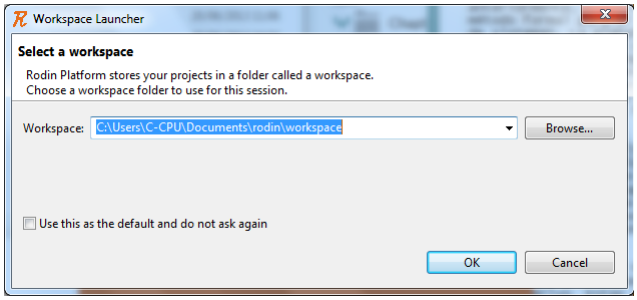

Figura 3.1: Workspace de Rodin

Interfaz Principal

La interfaz principal de Rodin, tal como se muestra en la figura 3.2, divide la interfaz entre los proyectos abiertos, y los archivos tal y como se muestra en Eclipse. Los símbolos se muestran en la parte inferior derecha y para realizar los proyectos se tienen varios editores, cada uno por gusto de cada persona.

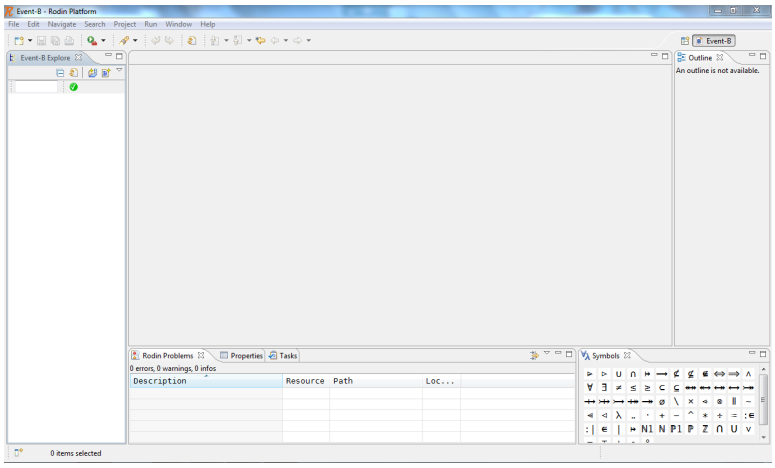

Figura 3.2: Interfaz de Rodin Principal

Interfaz Contexto

La interfaz que se presenta a continuación es la interfaz del contexto de Event-B. Es una interfaz donde el usuario decide qué elementos agregar y

quitar gracias a su buena UI, que es perfecta para la gente que está iniciando en este tipo de framework.

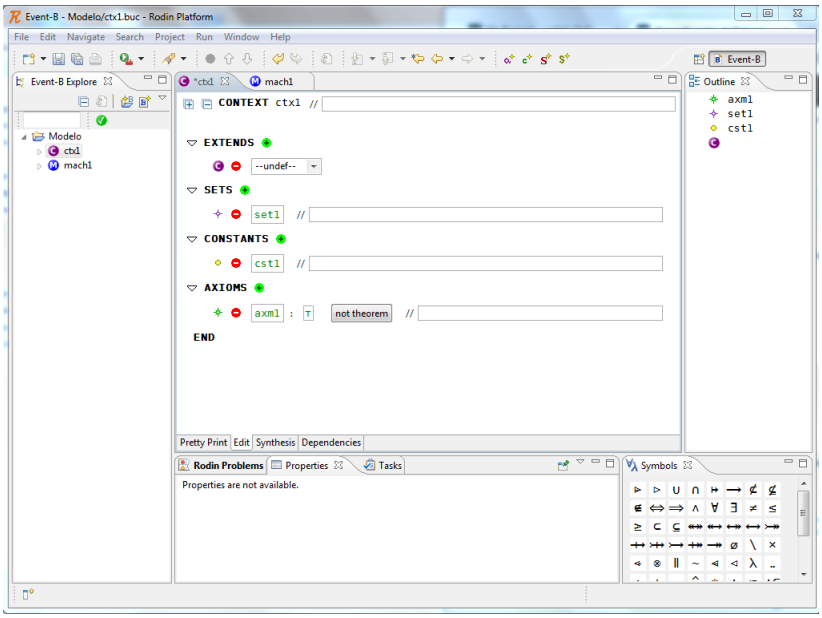

Figura 3.3: Interfaz del Contexto

A grandes rasgos, cada una de las variables que se presentan en esta interfaz se muestran a continuación:

Extends Se refiere a indicar sobre que otro contexto debe extenderse, esto es, sobre qué contexto se complementa, algo así como sucede en programación orientada a objetos cuando se refiere a herencia de clases; en este caso este contexto sería quien busca heredar las propiedades de otro contexto.

Sets Conjuntos que la persona quien hace el modelo puede construir para su uso específico en la máquina.

Constants Como su nombre lo indica, son los valores constantes que tiene el modelo. Estos valores no cambian en ningún momento y son globales para todas las máquinas que vean este contexto (ver Interfaz Maquina).

Axioms Reglas globales sobre las constantes y conjuntos (sets) del modelo.

Interfaz Máquina

La máquina es la parte del modelo que representa los eventos y todos los valores cambiantes de este. La máquina es quien se encarga de tener las variables que "observan" los cambios por evento en el modelo.

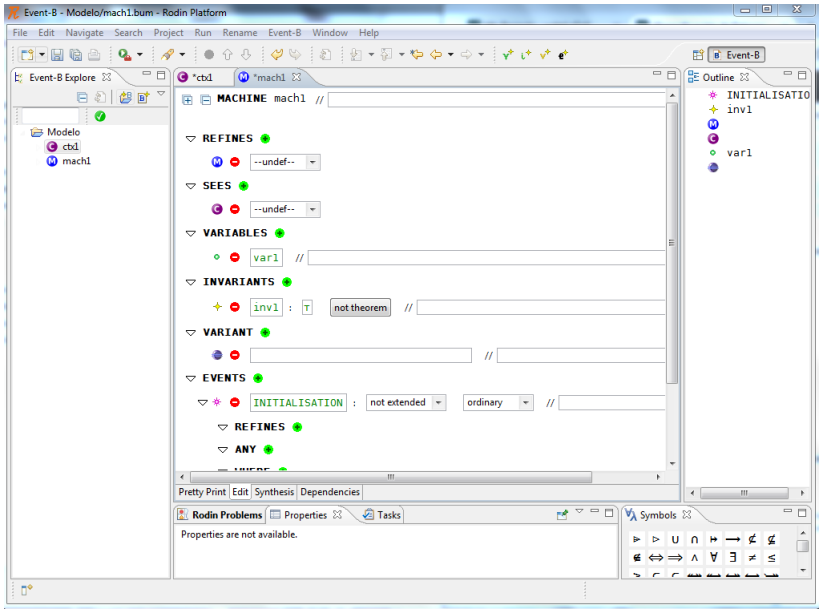

Figura 3.4: Interfaz de la Máquina

Con respecto a lo anterior, los elementos que componen la máquina son los siguientes:

Refines Al igual que Extends en el Contexto, Refines extiende o complementa las propiedades de otras máquinas.

Sees Indica el Contexto que debe usar para la ejecución de los eventos.

Variables Como indica su nombre son los elementos cambiantes del modelo. Estos valores cambian a través de los eventos que se especifiquen en la máquina.

Invariants Son las reglas generales del modelo. Estas reglas indican las propiedades de las variables y reglas generales que quien modela el sistema desea que se cumplan para que el modelo esté correctamente modelado.

Variant Se refiere a los valores que cambian por evento, pero a diferencia de las variables, el variante debe cambiar en cada evento y generalmente se especifica de tal manera que vaya disminuyendo. Los variantes se usan más para especificar programas que modelos de sistemas.

Events Se le puede denominar el *core* de la máquina. En ella se especifica las acciones que se ejecutan dependiendo si se cumplen ciertas condiciones.

3.2 Lenguaje de Event-B

Ya se ha mencionado brevemente en el capítulo 2 acerca de cómo se maneja Event-B. Ahora, para ilustrar mejor el modelamiento en este lenguaje, se va suponer un modelo sencillo: manejo de puertas de seguridad. Se pretende modelar un sistema de puertas de seguridad para el ingreso en un edificio. Existen dos puertas, A y B. La puerta A conecta al interior del edificio mientras que la puerta B conecta al exterior de este. Una puerta se puede abrir si y solo si la otra puerta está cerrada. Se quiere entonces modelar el estado de las puertas (si están abiertas o no).

Para ello entonces se conoce que existen 2 ambientes en este lenguaje; el contexto, que es todo lo relacionado con valores constantes, y la máquina, que ofrece toda la información sobre los valores variantes. En la primera parte se busca conocer qué elementos deben ser constantes y qué elementos deben ser variables, en otras palabras, qué elementos van en el contexto y qué elementos van en la máquina.

El lector puede darse cuenta de que no existen valores constantes en este modelo, dejando el contexto vacio tal como se muestra en la tabla 3.2.

Context
$ctx1$
Constants
Axioms
end

Tabla 3.1: Contexto modelo de Puertas de Seguridad

Ahora, la segunda parte en la definición del modelo es crear la máquina. En la máquina se definen los invariantes, las variables y los eventos del modelo. Para este ejemplo, el invariante, es decir, la regla que siempre el

modelo debe cumplir sería “Una puerta solo se puede abrir si la otra está cerrada”.Además de este invariante, se crea las variables “A” y “B” que son el estado de las puertas, siendo 1 abierto y 0 cerrado.

$Machine$
$mach1$
$Sees$
$ctx1$
$Variables$
A
B
$Invariants$
$A \in \{0, 1\}$
$B \in \{0, 1\}$
$A = 1 \Leftrightarrow B = 0$
$Events$
...
end

Tabla 3.2: Máquina 1ra parte modelo de Puertas de Seguridad

El lector puede notar que el invariante define lo que se quiso mencionar sobre la restricción de la apertura de las puertas. Finalmente se define los eventos, y este ejemplo nos entrega 4 eventos: “Abrir puerta A”, “Cerrar Puerta A”, “Abrir puerta B”, “Cerrar puerta B” nombrados en el modelo como “AbrirA”, “CerrarA”, “AbrirB” y “CerrarB” respectivamente. En la tabla 3.2 se definen los eventos con sus condiciones necesarias para cumplir los invariantes. Nótese que solo se puede abrir la puerta si inicialmente no está abierta y la otra puerta está cerrada también. Con base a lo anterior el modelo queda como se muestra en 3.2.

Los eventos que representan el cambio del estado de la puerta B se representa en la tabla 3.3 y la tabla 3.4.

Notese que en este modelo existe un patrón de diseño; El evento “Abrir puerta A” depende del evento “cerrar Puerta B” y viceversa. Este modelo luego ya se puede extender para agregar al usuario que entra. Pero con el modelo anteriormente creado ya se tiene la estructura de sincronización de los procesos.

Initialization Begin A := 0 B := 0 end	Event AbrirA when A = 0 B = 0 then A := 1 end
Event CerrarA when A = 1 then A := 0 end	

Tabla 3.3: Eventos del modelo de las Puertas de Seguridad

Event AbrirB when B = 0 A = 0 then B := 1 end	Event CerrarB when B = 1 then B := 0 end

Tabla 3.4: Eventos del modelo de Puertas de Seguridad (2da parte)

Capítulo 4

Fase Exploratoria

Como el nombre del capítulo lo menciona, en esta parte vamos a explorar algunos patrones de diseño, que más adelante, en los capítulos siguientes se explotarán para crear los modelos.

En la sección 4.1 vamos a explorar los modelos de PD que suelen aparecer recurrentemente en los modelos de Event-B. Luego en la sección 4.2 se construirá modelos generales sobre estos patrones para el diseño en el lenguaje de Alto Nivel.

4.1 Modelos de Patrones de Diseño

Para realizar la especificación de los procesos de cálculo NTCC primero hay que conocer cuales son los Patrones de Diseño que se quieren modelar. Por tal razón se tienen tres modelos que representan patrones generales de diseño.

4.1.1 Question - Response Model

El modelo es como sigue: Existen dos sistemas que ejecutan dos procesos distintos, *Q* y *R*. El proceso Q realiza la operación *Question* al sistema y luego de ejecutarlo, el proceso R ejecuta la acción *Response*. El proceso *Question-Response* se realiza de manera indefinida.

Un diseño de modelo en NTCC sería de la siguiente manera:

```
P = tell(a=0) || tell(b=0)
  || !( when a = b do (tell(Q) || next tell(a=a'+1 ∧ b=b')))
```

|| where a ≠ b do (tell(R)|| next tell(b=b'+1 ∧ a=b')))

En el diseño anterior, las variables "a" y "b" representan el condicional que requiere el modelo para poder activar uno u otro modelo. Éstas variables representan el número de ocurrencias de los procesos P y Q respectivamente y cada vez que las ocurrencias sean iguales, se ejecuta P y cuando sean diferentes se ejecuta Q. Como el modelo inicia con $a = b = 0$, entonces inicia ejecutando Q.

Además de "a" y "b", las variables a' y b' indican el valor que tienen los procesos a y b respectivamente en el tiempo actual y se usa para de alguna manera "guardar" los valores de estas variables en el otro tiempo.

Así mismo se tiene su diseño de modelo como eventos de Event-B, tal como lo muestra 4.1.1. Se puede apreciar que este modelo es tomado bajo los mismos parámetros que el modelo de NTCC.

Initialization a=0 b=0 end	Event_Question where a=b then a=a+1 end	Event_Response where a≠b then b=b+1 end

Tabla 4.1: Question/Response Model

Notese que los eventos *Question* y *Response* de la tabla 4.1 solo se activan si se cumple la condición anteriormente descrita: "Si las ocurrencias de procesos son iguales haga Question, de lo contrario haga Response", y las variables que se encargan de realizar esta cuenta son "a" y "b" para Question y Response respectivamente.

4.1.2 Synchronous Request/Confirm/Reject Model

Se tienen dos procesos *Sender* y *Receiver* que realizan las operaciones *request* y *Confirm/Reject* respectivamente.

El proceso Sender envia la petición Request y una vez ejecutada el proceso Receiver envia o una señal de Confirm o una de Reject de manera aleatoria.

En este modelo es importante señalar que existe un evento que sucede aleatoriamente, por tanto las guardas y la ejecución de los eventos deben garantizar que no se ejecute el otro evento, ya sea el Confirm o el Sender.

```
P = tell(a=0) || tell(b=0)
|| !( when a=b do (tell(R) || next(tell(a=a'+1 ∧ b=b')))
     || when a ≠ b do (tell (c) + tell (Rj)
                       || next (tell(b=b'+1 ∧ a=a'))) )
```

No olvidar que en el modelo NTCC, las variables a' o b' representan el valor de la variable en la unidad de tiempo anterior.

Initialization	Event Request	Event Confirm	Event Reject
a=0	where	where	where
b=0	a=b	a≠b	a≠b
end	then	then	then
	R	C	Rj
	a=a+1	b=b+1	b=b+1
	end	end	end

Tabla 4.2: Synchronous Request/Confir model

Las acciones R, C y Rj indican la ejecución de ese proceso y se usa para fines prácticos, ya que pueden indicar también acciones diferentes a las que se especifican en los patrones de diseño. Pero para mejor comprensión, en los modelos generales que se presentarán más adelante no se agregarán estas variables ya que no representan la estructura para el modelo de los patrones de diseño.

4.1.3 Asynchronous Multiple Messages Model

Se obtienen los mismos procesos *sender* y *receiver* que el modelo anterior, sin embargo el Sender envía cuantos mensajes quiera sin restricción de turno mientras que el Receiver puede hacer lo mismo siempre y cuando haya recibido un mensaje.

Este modelo es la versión asincrónica del modelo anterior. En ella la dependencia importante es el envío asincrónico de los mensajes por parte de

ambos procesos, por lo que en el diseño del modelo es importante las guardas para satisfacer la condición de que los mensajes del Sender deben ser mayores a los mensajes que el proceso Receiver haga.

```
P = Tell(a=0) || tell(b=0)
  || !( where a≥b do (tell(M) || Next(tell(a=a'+1 ∧ b=b')))
      + where a > b do (tell (R) + tell (C)
                              || Next (tell(b=b'+1 ∧ a=a'))) )
```

Para evitar confusiones, en este modelo las acciones son: *Message*, indicando el envío del mensaje y se expresa con la letra "M", *Confirm* que acepta el mensaje y se observa con la letra "C" y finalmente *Reject* representado con "Rj". De la misma forma que se explicó en la sección anterior, en la tabla 4.3 se aprecia el modelo en Event-B.

Initialization	Event Sender	Event Confirm	Event Reject
a:=1 b:=0 end	where a≥b then M a:=a+1 end	where a>b then C b:=b+1 end	where a>b then Rj b:=b+1 end

Tabla 4.3: Asynchronous Multiple Messages Model

4.1.4 Otros Modelos

Otros modelos se pueden ver en [Abr10]. En el capítulo 3 de [Abr10] el autor menciona procesos Reactivos Débiles y Fuertes. Éstos tipos de modelos son parecidos a la versión sincrónica y asincrónica de los modelos anteriormente mencionados en los que el autor indica las guardas necesarias en Event-B para mantener el modelo propuesto, es decir, definir el modelo sin que se pierda la estructura general del modelo. Estos modelos se especificarán más adelante en la "evolución de los modelos descritos".

4.2 Patrones de Diseño Generales

En base a los modelos anteriores se va a especificar las instancias que permiten conocer el comportamiento en que los distintos procesos se pueden ejecutar. Para ello, recurrimos a los operadores que típicamente se utilizan en expresiones regulares [Sip05].

Sintaxis para los Modelos

Para especificar los procesos que se mencionarán más adelante se tiene la siguiente sintaxis, que tiene una relación muy estrecha con la especificación que se hace al escribir lenguajes regulares [Sip05] combinándolo con la sintaxis que tiene NTCC para la ejecución de sus procesos.

(α) Se refiere al grupo de los procesos que se encuentran en el modelo. Un ejemplo sería (A,B) indicando que existen dos procesos; A y B dentro del modelo. Para el conocimiento del orden de los procesos se usa la coma "," tal como se mostró en el ejemplo anterior.

$$(A, B) = A \rightarrow^* B \tag{4.1}$$

Esto quiere decir que primero se ejecuta el proceso A y luego se ejecuta el proceso B.

***** La estrella indica la ejecución del proceso de manera cíclica e infinita.

$$(\alpha)^* = \alpha \rightarrow^* \alpha \rightarrow^* ... \tag{4.2}$$

Por ejemplo, si se tiene el proceso A, y se aplica el operador estrella A^*, entonces en cada unidad de tiempo se podrá observar A: $A, A, A...$

α^n Representa la ejecución finita de eventos. También se usa al indicar condicionales sobre uno o más eventos, por ejemplo, al mencionar que un evento A debe ejecutarse más veces que otro evento B.

$$(\alpha)^n = \alpha \rightarrow^* \alpha \rightarrow^* ... \rightarrow^* \alpha; n\ veces \tag{4.3}$$

Hay que notar que la diferencia entre este modelo y el de la estrella radica en que las ejecuciones de los procesos son finitos. Por ejemplo A^2 indicaría la ejecución de A 2 veces: A, A.

$+$ Para los modelos indica el *o-exclusivo* si al referirse a la ejecución de procesos.

$$(\alpha + \beta) = \alpha \vee \beta \tag{4.4}$$

$\|$ Indica el *y-lógico* de ejecución de procesos.

$$(\alpha \| \beta) = \alpha \wedge \beta \tag{4.5}$$

Modelos Generales

Usando la sintaxis anterior, los modelos generales que se van a revisar en este trabajo de grado se describen a continuación:

1. $\boldsymbol{(A, B)^*}$: Indica la ejecución de los procesos A y B de manera cíclica.
2. **(A,B+C)**: Se ejecuta A, luego se manera aleatoria se ejecuta el proceso B o C.
3. $\boldsymbol{(A \| B, C)}$: El proceso C solo puede ejecutarse cuando el proceso A y B se hayan ejecutado.
4. $\boldsymbol{(A^n, B^m); n > m}$: Se refiere a la ejecución no determinista de los procesos A y B siempre y cuando se cumpla la condición mencionada (las ocurrencias deben ser mayores en A que de B).

Los modelos generales resultan de utilizar un subconjunto de las expresiones que se pueden generar de la sintaxis mencionada en la sección anterior.

Capítulo 5

Lenguaje de Alto Nivel

Este trabajo de grado tiene como objetivo especificar patrones de diseño en NTCC para luego poderlo modelar en el lenguaje de procesos en Event-B. No obstante, la idea general detrás de este objetivo es hacer que los modeladores puedan realizar dichos modelos de manera más sencilla. Esto se puede lograr usando un lenguaje de alto nivel para luego traducirlo a NTCC y más adelante a Event-B.

El lenguaje de alto nivel será la ayuda para los diseñadores. En ella podrán especificar de manera más amigable los Patrones de Diseño mencionados anteriormente y luego podrán observar su especificación mediante NTCC.

5.1 Gramática

Basados en la sintaxis de NTCC, a continuación definiremos la gramática que posee el lenguaje de alto nivel:

$$L := Do\,\alpha \mid \alpha\ And\ \beta \mid \alpha\ Or\ \beta \mid If\ \alpha\ Then\ \beta \mid Num\ \alpha \mid Always\ \alpha\ End \quad (5.1)$$

Se usará cada uno de los operadores y se mencionará cómo se pueden usar basado en la gramática anterior.

$< process >$ Se refiere a cualquier proceso α que tenga el sistema. Se considera el valor atómico del lenguaje de alto nivel.

[**Statement**] Los corchetes indican que se puede usar el tipo *Statement* que se encuentre dentro.

{ [**Stat1** [Stat2] }] Cuando está dentro de estas llaves, indica que se pueden usar cualquiera de las sentencias que estén dentro de ellas.

5.1.1 Or Statement/And Statement

Al igual que en los lenguajes de programación conocidos, las palabras *AND* y *OR* hacen referencia a los condicionales lógicos que nombran. No obstante, éstas palabras también definirán sentencias de ejecución de procesos, ya sea utilizando un *y-obligatorio* o usando un *o-exclusivo* al mencionar que procesos usar.

$$< process > \ OR \ < process > \tag{5.2a}$$
$$< process > \ AND \ < process > \tag{5.2b}$$
$$< process > \ OR \ [Or - Statement] \tag{5.2c}$$
$$< process > \ AND \ [And - Statement] \tag{5.2d}$$
$$\tag{5.2e}$$

5.1.2 Do Statement

La sentencia *Do* indica la ejecución de un proceso. Esta ejecución es única y no se repite al menos que se encuentre dentro de otra sentencia. La forma general es:

$$Do(\alpha) \tag{5.3}$$

Donde α es un proceso.
La sentencia *Do* puede tener un proceso ó una sentencia *And/Or*.

$$Do \ < Process > \tag{5.4a}$$
$$Do \ [Or - Statement] \tag{5.4b}$$
$$Do \ [And - Statement] \tag{5.4c}$$

5.1.3 If Statement

Para NTCC es muy común observar la forma *Next* que indica la ejecución de un proceso en otra unidad de tiempo. Para la sintaxis de este modelo, la acción que realiza *Next* se encuentra dentro del operador *if*.

La sentencia *If* indica la ejecución de un proceso cuando se cumple la condición dada, es de la forma:

$$If\ \alpha\, then\, \beta \tag{5.5}$$

Donde α indica cualquier sentencia condicional y β cualquier sentencia que involucre ejecución de procesos.

En este caso, el proceso que se ejecuta después de *then* se realiza en otra unidad de tiempo, siendo este una simulación al operador *Next* de NTCC.

$$If\ a\ then\ B\ =\ when\ a\ do\ (Next(b))$$

$$[IFStatement]\ =\ [If-stat]\ THEN\ [Do/AndStat]\ [AlwaysStat] \tag{5.6}$$

Donde *If-Stat* son todos los condicionales que puede tener el operador 5.7 y *Do/And Stat* y *Always Stat* son los procesos de tipo *Do/And* y *Always* que puede tener el lenguaje.

$$(If-Stat) = \{< proceso > [Or/AndStat] \tag{5.7}$$
$$[CondicionalStat]\} \tag{5.8}$$

Donde *Condicional Stat* son los operadores condicionales básicos y usando la sentencia *Num* (explicado más adelante):

$$(Operador)\ =\ >, <, =, \leq, \geq, \neq \tag{5.9}$$

Num Statement

Al igual que las sentencias *And* y *Or*, la sentencia *Num* es un condicional, pero a diferencia de ellas no funciona como ejecuciones de procesos, es decir, puede ir entre las palabras *if* y *then* pero no despúes de *then*. Su estructura es como se ve en la ecuación 5.10.

$$Num \ < process > \tag{5.10}$$

Con lo anterior se puede definir entonces la sentencia condicional:

$$[NumStatement]\ (Operador)\ < VarNatural > \tag{5.11a}$$
$$[NumStatement]\ (Operador)\ [NumStatement] \tag{5.11b}$$

5.1.4 Always Statement

Finalmente, la sentencia *Always* representa la ejecución de las sentencias anteriormente mencionadas de manera cíclica, dado que en NTCC estas sentencias solo son ejecutadas en un solo tiempo si no se usa el símbolo *'!'*. Esta sentencia es de la forma:

$$Always\ \alpha\ end\ ;\ \alpha\ cualquier\ sentencia \tag{5.12}$$

Los comandos *Always...End* realiza los procesos de manera repetida. Al igual que pasó con la definición de *If Statement*, dentro del Always pueden incluir mas de un tipo de sentencia diferente, por lo que hay que tener mucho cuidado al definirlo.

Para definirlo de forma mas sencilla, se especificará de la siguiente manera: cada acción que se realice se especifica con un salto de línea "$\backslash n$", es decir, cada nueva instrucción dentro del Always es un salto de línea, lo que genera crear un "and" en el proceso NTCC.

Para que el usuario pueda utilizar acciones que contengan condicional OR, deberá hacerlo en la línea continua o si no, no le será posible especificarlo.

Su forma general es:

$$[AlwaysStatement]\ =\ [OperacionesStat]\ END \tag{5.13}$$

Donde *Operaciones Stat* son las operaciones que puede tener el lenguaje separados por un salto de línea, es decir:

$$[OperacionesStat]\ =\ [Or/AndStat][DoStat][IfStat] \tag{5.14}$$

5.2 Ejemplos

Para entender mejor la estructura de la sintáxis del lenguaje de alto nivel, se presenta a continuación una serie de ejemplos de escritura en este lenguaje.

Para la sentencia *Do* se puede indicar los procesos que se quiera en él usando los condicionales *Or* y *And*.

$$Do\ A \tag{5.15a}$$
$$Do\ A\ and\ B\ and\ C\ and\ ... \tag{5.15b}$$
$$Do\ A\ or\ B\ or\ C\ or\ ... \tag{5.15c}$$

En todos los ejemplos las letras en mayúscula, (ejemplo A o B) representan procesos. Estos procesos, además de estar dentro de la sentencia *Do* pueden estar dentro de la sentencia If. Usando la sentencia *Statement-If* tenemos los siguientes ejemplos:

$$If\ A\ then\ B \tag{5.16a}$$
$$If\ A\ then\ B\ and\ C\ and\ D\ and\ ... \tag{5.16b}$$
$$If\ A\ then\ B\ or\ C\ or\ D\ and\ ... \tag{5.16c}$$
$$If\ A\ and\ B\ ...\ Then\ C \tag{5.16d}$$
$$If\ A\ or\ B\ ...\ Then\ C \tag{5.16e}$$
$$If\ Num\ A\ >\ \ Num\ B\ Then\ C \tag{5.16f}$$

La primera línea de los ejemplos es un condicional simple. Las 2 líneas subsiguientes usan un condicional simple con *And* y *Or* como ejecución continua de procesos. La 3ra y 4ta línea indican condicionales *And* y *Or* y la última indica el condicional con el contador de procesos Num.

Finalmente, la sentencia *Always* que representa ejecuciones cíclicas e infinitas puede contener cualquier otra sentencia anteriormente descrita separados por un *and* que en este caso sería un salto de línea.

$$Always\ Do\ A\ End \tag{5.17a}$$
$$Always\ If\ A\ then\ B\ End \tag{5.17b}$$
$$Always\ If\ Num\ A\ <\ Num\ B\ Then\ C\ End \tag{5.17c}$$

Y como se mencionó anteriormente, un ejemplo de la sentencia *always* usando saltos de línea.

$$\begin{aligned} &Always \\ &\qquad DoA \\ &\qquad NextB \\ &End \end{aligned} \tag{5.18}$$

Capítulo 6

Traducción de Modelos

En este capítulo introduciremos modelos generales de patrones de diseño denominados Dependencias. A lo largo del capítulo explicaremos el significado de cada dependencia y con ello su traducción a los lenguajes de NTCC y Event-B, para finalizar con las diferentes construcciones de este modelo, empezando por la extensión con la noción del tiempo y finalizando con la noción de relaciones fuertes.

6.1 Modelos Basados en Dependencias

Se tienen 5 modelos que indican procesos o eventos que dependen de otros, por eso son llamados de esa manera. Cada uno de ellos son generalidades de dependencias básicas, que luego se pueden mezclar y formar otras, además que con esas dependencias se pueden recrear los 4 patrones de diseño especificados en el capítulo 4.

6.1.1 Dependencia Simple

$\{If\ A\ Then\ B\}$

Esta dependencia indica que el proceso B solo tiene otro proceso que debe "Observar" para poderse ejecutar. En la figura 6.1.1 esta "Observación" se aprecia en las guardas del evento B.

6.1.2 Dependencia Compuesta: And

$\{If\ A\ AND\ B\ Then\ C\}$

NTCC: !(When (A) do Next(Tell(B)))	Event_A where then a_up=1 end	Event_B where a_up=1 then a_up=0 end

Tabla 6.1: Modelo de Dependencia Simple

Esta dependencia sobre C tiene la misma dependencia simple que en la sección anterior, pero en este caso C depende de que 2 procesos se ejecuten, A y B. De la misma forma que se mencionó anteriormente, las guardas que observan la ejecución de estos procesos en 6.1.2 se logran a través del evento C y los eventos A y B solo se ejecutan y levantan un *flag* que indica que el proceso está activo.

NTCC: !(When(A $\wedge$ B) do Next(Tell(C)))	Event_A where then a_up=1 end	Event_B where then b_up=1 end	Event_C where a_up=1 b_up=1 then a_up=0 b_up=0 end

Tabla 6.2: Modelo Dependencia Compuesta: And

6.1.3 Dependencia Compuesta: Or

$\{If\ A\ OR\ B\ Then\ C\}$

En este caso C espera que A o B se ejecuten. Cuando vea que esto sucede se activa. En la tabla 6.1.3 la guarda en el evento C simula la estructura de

esta dependencia gracias al condicional *or*. Nótese que tanto en este modelo como en los anteriores, la acción extra de C es "apagar" los procesos A y B, siendo más específicos, C baja el *flag* de ambos procesos para seguir el ciclo.

NTCC:	Event_A	Event_B	Event_C
!(When(A ∨ B) do Next(Tell(C)))	where then a_up=1 end	where then b_up=1 end	where (a_up=1 ∨ b_up=1) then a_up=0 b_up=0 end

Tabla 6.3: Modelo Dependencia Compuesta: Or

6.1.4 Dependencia Futura Compuesta: And

$\{If\ A\ Then\ B\ AND\ C\}$

Se le denomina Dependencia Futura Compuesta porque en este caso la dependencia es compartida; los procesos B y C dependen de manera simple con A y ambos están conectados para que se ejecuten cuando A suceda. Para esto se tuvo mucho cuidado en bajar el *flag* en A. En la tabla 6.4 se muestra la solución a este problema: las guardas en Event-B que se usan son los condicionales sobre si está arriba el *flag* de A ó el de B ó C. En otras palabras, B observará si está arriba A ó si en vez de eso está C arriba (ya que si C está arriba entonces B también puede estar arriba), y sucede lo mismo con C.

6.1.5 Dependencia Futura Compuesta: Or-Exclusivo

$\{If\ A\ Then\ B\ OR\ C\}$

Aunque B y C dependen de que se ejecute A, este modelo se vuelve una competencia en el cual el ganador es el proceso que puede ejecutarse. En este caso la competencia es sobre B o C, esperando que A se active para

NTCC:	Event_A	Event_B	Event_C
!(When (A) do Next(Tell(B) || Tell(C)))	where then a_up=1 b_up=0 c_up=0 end	where (a_up=1 ∨ c_up=1) then a_up=0 b_up=1 end	where (a_up=1 ∨ b_up=1) then a_up=0 c_up=1 end

Tabla 6.4: Modelo Dependencia Futura Compuesta: And

activarse y desactivar a los demás. La tabla 6.1.5 muestra que cuando B o C se ejecutan bajan el *flag* de los otros procesos.

NTCC:	Event_A	Event_B	Event_C
!(When (A) do Next(Tell(B) + Tell(C)))	where then a_up=1 b_up=0 c_up=0 end	where a_up=1 then a_up=0 end	where a_up=1 then a_up=0 end

Tabla 6.5: Modelo Dependencia Futura Compuesta: O-Exclusivo

6.2 Evolucion del Modelo: Uso del Tiempo

A diferencia de Event-B, NTCC posee en sus modelos la noción de tiempo. Ésta noción dificulta la traducción de los dos modelos debido a la falta de esta noción en Event-B. NTCC permite tener actores sobre un solo tiempo con sus procesos mientras que en Event-B los eventos se realizan siempre, es

decir, en cada momento (la noción de tiempo no existe por lo que los eventos no tienen una restricción en observar sus guardas y ejecutar sus eventos).

Por tal motivo, en las traducciones se deberá construir la noción de tiempo para Event-B. La manera de hacerlo es crear un evento dentro del modelo que simule el paso discreto del tiempo, además de agregar las variables o guardas necesarias para que los modelos de procesos mantengan su correctitud.

Con ello se tiene entonces un evento que genera un cambio en el tiempo discreto y una variable extra por cada evento que se asegura, en su guarda, que no se repita el evento hasta haber terminado el evento.

```
Event_Time
where

then
 TIME := TIME+1
end
```

Tabla 6.6: Event Time

El evento Time de la tabla 6.6, es el que crea la noción de la transición discreta del tiempo. Usando una variable TIME, que se incrementa de uno en uno, simula el tiempo en el cuál los otros eventos se ejecutan. En cada uno de los eventos entonces, se tendrá una variable que indica la última ejecución del proceso y en la guarda habrá un condicional que restringe la ejecución de este proceso si no ha avanzado en el tiempo.

6.2.1 Dependencia Simple Tiempo

$\{ALWAYS\ If\ A\ Then\ B\ END\}$

La extensión de la dependencia es como sigue: el modelo ahora posee un nuevo operador *Always*. Este operador indica que el patrón se repetirá indefinidamente. Este modelo utiliza las mismas guardas que la Dependencia Simple, pero en la tabla 6.7 se observa las guardas extras *Time_A* y *Time_B* representando el tiempo en que se ejecuta A y B respectivamente.

6.2.2 Dependencia Compuesta: And Tiempo

$\{ALWAYS\ If\ A\ AND\ B\ Then\ C\ END\}$

Event_A where TIME > Time_A then a_up=1 Time_A = TIME end	Event_B where a_up=1 TIME > Time_B then a_up=0 Time_B = TIME end

Tabla 6.7: Modelo Dependencia Simple con Tiempo

En esta dependencia solo se extiende, al igual que la anterior, las variables y guardas que restringen la ejecución de un mismo proceso en un mismo tiempo. En la tabla 6.8 se puede ver la extensión a este modelo.

Event_A where TIME > Time_A then a_up=1 Time_A = TIME end	Event_B where TIME > Time_B then b_up=1 Time_B=TIME end	Event_C where a_up=1 b_up=1 TIME > Time_C then a_up=0 b_up=0 Time_C=TIME end

Tabla 6.8: Modelo Dependencia Compuesta And con Tiempo

6.2.3 Dependencia Compuesta: OR Tiempo

$\{ALWAYS\ If\ A\ OR\ B\ Then\ C\ END\}$

En la tabla 6.9 se ve la extensión de esta dependencia. Al igual que la anterior, C espera que cualquiera de los 2 procesos, A o B se ejecuten para ejecutarse, y esto lo hace mirando sus *flag*.

Event_A where TIME > Time_A then a_up=1 Time_A=TIME end	Event_B where TIME > Time_B then b_up=1 Time_B=TIME end	Event_C where a_up=1 V b_up=1 TIME > Time_c then a_up=0 b_up=0 Time_c=TIME end

Tabla 6.9: Modelo Dependencia Compuesta Or con Tiempo

6.2.4 Dependencia Futura Compuesta: And con Tiempo

$\{ALWAYS\ If\ A\ Then\ B\ AND\ C\}$

Con las dependencias futuras compuesta no tienen mayor cambio que lo que se ha hecho anteriormente. Ya se puede apreciar en la tabla 6.10 la extensión sobre las guardas usando TIME.

Event_A where TIME > Time_A then a_up=1 b_up=0 c_up=0 Time_A=TIME end	Event_B where (a_up=1 ∨ c_up=1) TIME > Time_B then a_up=0 b_up=1 Time_B = TIME end	Event_C where (a_up=1 ∨ b_up=1) TIME > Time_C then a_up=0 c_up=1 Time_C = TIME end

Tabla 6.10: Modelo Dependencia Futura Compuesta And con Tiempo

6.2.5 Dependencia Futura Compuesta: Or-Exclusivo con Tiempo

$\{ALWAYS\ If\ A\ Then\ B\ OR\ C\}$

Todos los eventos realizan las mismas acciones: levantan un *flag* para que otros procesos lo vean y además agrega el tiempo en el que se ejecutó en la variable $Time_A$ si es el evento A y así sucesivamente. La tabla 6.11 muestra la dependencia futura compuesta con el o-exclusivo en tiempo.

Event_A where TIME > Time_A then a_up=1 b_up=0 c_up=0 Time_A = TIME end	Event_B where a_up=1 TIME > Time_B then a_up=0 Time_B = TIME end	Event_C where a_up=1 TIME > Time_C then a_up=0 Time_A = TIME end

Tabla 6.11: Modelo Dependencia Futura Compuesta Or-Exclusivo con Tiempo

Si el lector se dio cuenta, el uso de la guarda del tiempo no cambia para nada la estructura de los eventos mencionados anteriormente, son simplemente una guarda y una acción extra. Los eventos que dependen de otros eventos como lo son B y C en la tabla 6.11 no tienen necesidad de tener estas guardas ya que dependen del tiempo de A, pero se ponen con fines de explicación.

Con respecto a la evolución de este modelo, aparecen nuevas estructuras que vienen siendo las instrucciones que solo aparecen en un tiempo. Al igual que los modelos anteriores la única manera para que se repitan es activando la guarda del tiempo y la acción que copia el tiempo en el que la acción se tomó. Por esta razón la forma de romper con esta secuencia es eliminando esta acción del evento que no depende de ningún otro.

```
Event_A
 where
  //Condiciones Generales//
  TIME > Time_A
 then
  //Condiciones Generales//
end
```

Tabla 6.12: Tabla de los eventos singulares en tiempo

6.2.6 Sentencia Num

Usa la palabra reservada *Num* para representar la cuenta de la ejecución de los procesos.

$\{IF\ NUM(A)\ >\ NUM(B)\ THEN\ B\}$

NTCC:	Event_B where TIME > Time_A A_count>B_count then B_count=B_count+1 Time_B = TIME end
When (count_a>count_b) Do (Next (Tell(b) || count_b'=count_b+1))	

Tabla 6.13: Tabla de los eventos singulares en tiempo

Sin embargo, estos modelos siguen teniendo una limitante: todos los procesos no dependientes pueden ser activados en cualquier tiempo sin restricción. A esto se le llama "Reacción Débil" o *Weak Reaction*.

No obstante, existen modelos que requieren que esto no suceda ya que los procesos se necesitan que estén consecutivos sin que se repitan antes de que los procesos dependientes se ejecuten. Por tal razón hay que extender el modelo para que apliquen estos tipos de modelo.

6.3 Evolución del Modelo: Weak y Strong Reactions

En los modelos anteriores se aprecian dos tipos de procesos: los procesos Dependientes y los procesos Independientes. Los procesos Dependientes son aquellos que requieren de otros procesos para poder ser ejecutados. En el lenguaje de alto nivel se pueden apreciar después de *then* y sus condiciones para activarse se refieren entre *if then*.

Los procesos independientes son todos los que pueden ser ejecutados en cualquier momento ya que no requieren de otros procesos en su comportamiento. Estos procesos son los que hacen que los modelos sean débiles o fuertes dependiendo de cómo se especifiquen.

Para que los modelos sean fuertes se requiere que tanto los procesos dependientes como los independientes tengan una restricción en su ejecución y que sucedan uno tras otro sin dejar que estos procesos puedan ser ejecutados en tiempos no deseados. Una aproximación sería volver de alguna manera "Dependientes" los eventos independientes creando los modelos Dependientes en ambos sentidos. Desafortunadamente, esta aproximación solo traería comportamientos no deseados en el modelo y no representaría la definición de reacción fuerte.

Por tal motivo la solución es usar unas guardas que permitan la dependencia de los procesos frente al tiempo y frente al proceso que le sigue. Las guardas que se usan son la bandera de activación de los eventos dependientes, así se mantiene el modelo y se espera a que todo ocurra antes de volver al evento independiente.

6.3.1 Extensión del Lenguaje de Alto Nivel

Para la persona quien diseña los procesos pueda modelar de forma débil o fuerte los sistemas reactivos, se debe tener la sintaxis que los diferencie. Por ese motivo se extiende la sintaxis anterior y se agrega una nueva palabra reservada al lenguaje de alto nivel.

6.3.2 IFF: If and Only If

La extensión al lenguaje es entonces la palabra "Si y solo si" o *Iff* por sus siglas en inglés. Esta extensión indica que la relación entre los eventos que

tiene se hace con relación fuerte, por ejemplo, si tenemos $IFF\ A\ Then\ B$ y el proceso A se ejecutó, entonces el proceso A no podrá volver a ejecutarse y el patrón de diseño acaba cuando B se haya ejecutado.

La diferencia con las dependencias anteriores se observa en la ejecución de estos procesos "libres" (independientes). Ninguna guarda en los eventos en Event-B le restringe la posibilidad a ejecutarse de nuevo a los eventos que no dependen de otro evento. Entonces para mencionar esta "restricción" se crea entonces el operador *iff*.

$$IFF\ A\ THEN\ B := A \rightarrow^* B \rightarrow^* A \rightarrow^* ... \tag{6.1}$$

6.3.3 Dependencia Simple

En los eventos que se presentan a continuación no se muestran las guardas de tiempo.

$\{ALWAYS\ If\ A\ Then\ B\ END\}$

Event_A where ... B_up=1 then a_up=1 B_up=0 ... end	Event_B where ... A_up=1 then A_up=0 B_up=1 ... end

Tabla 6.14: Dependecia Simple Iff

6.3.4 Dependencia Compuesta: And con IFF

{ALWAYS Iff A AND B Then C END}

En la tabla 6.15 se observa que en las guardas de los eventos A y B dependen del *flag* de C o de A o B según corresponda. Al ser ejecutado A o B, se baja la bandera en C y este evento observará como lo hace normalmente

si A y B están arriba para ejecutarse. Es importante hacer énfasis en la guarda $a\_up = 0$ o $b_up = 0$ ya que restringen por completo la ejecución de este proceso de nuevo hasta que C se ejecute.

Event_A where ... (c_up=1 ∨ b_up=1) a_up=0 then a_up=1 ... c_up=0 end	Event_B where (c_up=1 ∨ a_up=1) b_up=0 then c_up=0 b_up=1 ... end	Event_C where a_up=1 b_up=1 then a_up=0 b_up=0 c_up=1 end

Tabla 6.15: Dependecia Compuesta And Iff

6.3.5 Dependencia Compuesta: OR con IFF

{*ALWAYS Iff A OR B Then C END*}

En este caso, y tal como se muestra en la tabla 6.16 las guardas en los eventos independientes A y B dependen de que C esté arriba, y luego bajan el *flag* de C para que el otro evento no pueda ser ejecutado. Sin embargo, C ahora puede ejecutarse con estos condicionales subiendo su *flag* y bajando la de los otros procesos.

6.3.6 Dependencia Futura Compuesta: And con Iff

{*ALWAYS Iff A Then B AND C*}

Al igual que la Dependencia Compuesta And, se requiere que las guardas ahora en los eventos B y C tengan la restricción de uso del evento, es decir, $b\_up = 0$ y $c\_up = 0$ respectivamente. El modelo se puede observar en 6.17

Event_A where c_up=1 then a_up=1 c_up=0 end	Event_B where c_up=1 then b_up=1 c_up=0 end	Event_C where (a_up=1 $\vee$ b_up=1) then a_up=0 b_up=0 c_up=1 end

Tabla 6.16: Dependecia Compuesta Or Iff

6.3.7 Dependencia Futura Compuesta: Or-Exclusivo con Iff

{*ALWAYS Iff A Then B OR C*}

En esta dependencia se observa la competencia que se mencionaba en el modelo sin el tiempo. La tabla 6.18 muestra las guardas que satisfacen esta competencia; una vez que B o C se activen, bajan el *flag* de A haciendo que el otro evento no sea capaz de ejecutarse (y de paso que el mismo evento no pueda volverse a ejecutar hasta que A lo haga).

6.3.8 Modelos no Cíclicos

La palabra reservada *iff* también aplica a eventos no cíclicos, es decir, aquellos en los que *always* no aplica. Estos procesos entonces, solo son ejecutados una vez. Pero de alguna manera la guarda principal debe mantenerse.

Es por eso que la guarda principal se aplica a los procesos que son independientes, pero no olvidar que pueden existir 3 tipos de procesos:

Procesos Independientes Son los procesos que no dependen de ningún otro proceso en el sistema.

Procesos Independientes Relativos Son procesos que en algunas instrucciones del modelo no dependen de nadie (un proceso externo lo requiere) pero en el sistema en general tiene una dependencia con otro proceso.

Event_A where b_up=1 c_up=1 then a_up=1 b_up=0 c_up=0 end	Event_B where (a_up=1 ∨ c_up=1) b_up=0 then a_up=0 b_up=1 end	Event_C where (a_up=1 ∨ b_up=1) c_up=0 then a_up=0 c_up=1 end

Tabla 6.17: Dependecia Futura Compuesta And Iff

Procesos Dependientes Son los que dependen de otros procesos para poder accionarse.

P := when (A) do (Next (b))

$\{Iff\ A\ Then\ B\}$

Event_A where (b_up=1 ∨ c_up=1) then a_up=1 b_up=0 c_up=0 end	Event_B where a_up=1 then a_up=0 b_up=1 end	Event_C where a_up=1 then a_up=0 c_up=1 end

Tabla 6.18: Dependecia Futura Compuesta Or-Exclusivo Iff

Event_A where a_up=0 a_on=1 then a_up=1 a_on=0 end	Event_B where a_up=1 then a_up=0 b_up=1 end

Tabla 6.19: Modelo Dependencia Simple Procesos Independientes

Capítulo 7

Aplicaciones

Se presentarán a continuación dos modelos donde se aplique la traducción desde el Lenguaje Formal de Ato Nivel a NTCC. El primero se llama *Simple Transfer Protocol*, que es propuesto como un ejemplo de Abrial [Abr10]. El segundo modelo es sobre procesos de Sistemas Operativos, conveniente para explicar procesos reactivos fuertes y condicionales.

7.1 Simple File Transfer Protocol

El modelo de Simple File Transfer Protocol fue escrito en el libro de Abrial [Abr10] en el Capítulo 4 al explicar el uso de las funciones en Event-B. Sin embargo, no es propósito del proyecto el explicar los distintos refinamientos y variables para llegar al modelo final de este modelo, pero sí es explicar como sería este modelo usandose desde los lenguajes anteriormente mencionados.

El modelo se presenta como sigue: Se desea efectuar un envío secuencial de un archivo entre dos sistemas o agentes. El agente que tiene el archivo, el *sender*, efectuará la operacion de envío desde un canal de datos y cuando el agente *receiver* haya obtenido estos datos envía un *ack* para que el agente *sender* envie la otra parte del archivo. Este proceso se repite las veces que sea necesario para que el archivo original esté copiado en el otro agente.

7.1.1 Canales Ideales

Al modelar dicho sistema hay que tener en cuenta todas las variables implicadas en el caso. Para la primera parte del modelo se observa que el canal

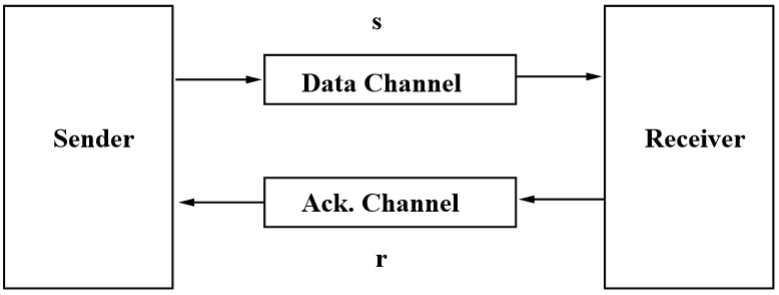

Figura 7.1: Modelo de Simple File Transfer Protocol

de datos es ideal, esto es, que la comunicación entre los dos sistemas (el libro lo llama agentes) siempre es existente.

Lo siguiente por definir es el modelo de comportamiento del sistema. Se observa entonces que existe una acción *send* y por consiguiente una acción *ack*. La acción *send* se ejecuta inicialmente y en el tiempo siguiente se ejecuta la acción *ack*.

Estas acciones poseen una relación estrecha: ninguna de ellas se ejecuta hasta que la otra lo haya realizado y en el tiempo inicial empieza la acción de *send*. Por tal motivo se aprecia la siguiente ejecución de eventos:

$$P = send \rightarrow^* ack \rightarrow^* send \rightarrow^* ack \rightarrow^* ...$$

Con el modelo anterior descrito se aprecia que se está hablando de un modelo de la forma $(A, B)^*$ por lo que se sabe que tiene una relación fuerte y cíclica.

Por consiguiente, el modelo especificado en lenguaje de alto nivel se puede apreciar en la tabla 7.1:

```
Always Iff send Then ack end
```

Con esto se crea el siguiente proceso en NTCC:

```
!( when (send) Do (Next (tell(ack))) ||
   when (ack) Do (Next(tell(send)) )
```

Y con ello los modelos en Event-B (no olvidar que se maneja tiempo):

Initialization where then ack_up:=1 TIME:=0 Time_Send:=0 Time_Ack:=0 end	Event_Time where then TIME:=TIME+1 end
Event_Send where ack_up=1 TIME > Time_send then send_up:=1 ack_up:=0 Time_send := TIME end	Event_Ack where send_up=1 TIME > Time_ack then ack_up:=1 send_up:=0 Time_ack := TIME end

Tabla 7.1: Simple File Transfer Protocol con Canales ideales

7.1.2 Canales Reales

Ahora suponga que con el modelo anterior por alguna u otra causa falla, los canales no siempren envian los datos. Esto se traduce a que el evento que genera el agente *Sender* debe siempre estar cada cierto tiempo re-enviando los datos para evitar pérdida de información. Este manejo se hace mediante otras guardas que se explican en el refinamiento del modelo en el capítulo 4 de Abrial [Abr10]. Sin embargo para nuestro propósito hay que observar un ejemplo de cómo sería la ejecución de los procesos.

$$P = send \rightarrow^* send \rightarrow^* ack \rightarrow^* send \rightarrow^* ...$$

Lo que indica el modelo es que ya sea que el canal de datos falle o el canal de ack falle, la forma global como se comporta el sistema es mediante

el modelo $(send^n, ack^m); n \geq m$. Por tal motivo, este puede ser un ejemplo de sistemas reactivos débiles ya que no siempre se ejecutará un evento sobre otro.

```
Always If send Then ack end
```

```
!(when (send) Do (Next ( tell(send)+tell(ack) ) )
  when (ack) Do (Next ( tell(send) ) ))
```

Note que el modelo especificado en la tabla 7.2 indica que la acción *send* se podrá ejecutar las veces que sea necesario. Dentro del modelo en Event-B deberá especificar el diseñador qué condiciones debe efectuarse para que "Falle" el envío de datos.

Nótese que este modelo también podría haberse escrito según el patrón de diseño no determinista:

```
Always
    If num(send)>num(ack) Then (ack)
  end
```

Por lo que las reglas cambiarían pero mantendrían el modelo principal de acciones entre los modelos.

7.2 Process Flow Graphs

Este ejemplo es sobre un sistema que maneja control de procesos, utilizado generalmente en los sistemas operativos. Por ejemplo, en [Lub02] estos modelos revisan la forma en que los procesos dentro del sistema se deben ejecutar, por lo que las restricciones están bien definidas y es un buen ejemplo para el proyecto.

El sistema consta de cinco procesos. Cada uno de ellos se puede ejecutar si y solo si se han ejecutado otros procesos, tal como se muestra en la figura 7.2:

Initialization where then send_up:=0 TIME:=0 Time_Send:=0 Time_Ack:=0 end	Event_Time where then TIME:=TIME+1 end
Event_Send where TIME > Time_send then send_up:=1 Time_send := TIME end	Event_Ack where send_up=1 TIME > Time_ack then send_up:=0 Time_ack := TIME end

Tabla 7.2: Simple File Transfer Protocol con Canales Reales

En la gráfica 7.2 se puede apreciar que el proceso 3 depende del proceso 1 y el proceso 4 depende del 3. El proceso 5 depende de que se ejecute el proceso 2 y el proceso 4.

Basados en la gráfica y en la explicación anterior se puede notar que los procesos que se detallan tienen dependencia simple y compuesta, además estos procesos son ejecutados una sola vez (un gran proceso que pide el sistema, luego lo volveran a llamar, pero es un proceso único).

Por lo tanto, el modelo anterior se puede especificar de la siguiente manera:

```
Iff p1 Then p2 And p3
Iff p3 Then p4
Iff p2 And p4 then p5
```

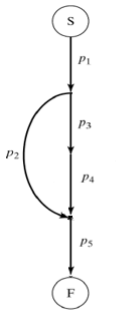

Figura 7.2: Process Flow Graph Modificado

Se puede notar que tal como se mencionó, naturalmente se especifica en el lenguaje de alto nivel. Ahora, observe que *p2* y *p3* dependen únicamente de *p1*, entonces se modela como si no existieran otros procesos. Lo mismo pasa con *p5* que, aunque requiera a *p2* y *p4* y éstos dependan de *p1* y *p3* respectivamente, el proceso solo debe "observar" la ejecución de estos procesos no de sus antecesores.

Con la especificación de este modelo en lenguaje de alto nivel se traduce a NTCC de la siguiente manera:

```
P := when (p1) do ( Next(tell (p2) ∧ tell(p3)) )
        || when (p3) do ( Next(tell(p4)) )
     || when (tell(p2) ∧ tell(p4)) do ( Next(tell(p5)) )
```

Y con este modelo se extiende a la especificación en Event-B representada en la tabla 7.3 y 7.4.

Notese que se requirió de las guardas *on* para todos los procesos que eran *independientes relativos*, esto es, que eran independientes frente a otros procesos. Sin embargo, el único que no era independiente relativo era el proceso *p1* por lo que fue el único en el cuál la guarda $p1\_up = 0$ se aplicó.

Initialization where then TIME := 0 p1_up := 0 p1_on := 1 p2_up := 0 p2_on := 1 p3_up := 0 p3_on := 1 p4_up := 0 p4_on := 1 p5_up := 0 Time_p1:=0 Time_p4:=0 Time_p2:=0 Time_p5:=0 Time_p3:=0 end	Event_Time where then TIME := TIME + 1 end
Event_P1 where TIME > Time_p1 p1_up = 0 p1_on = 1 p2_up = 1 $\wedge$ p3_up = 1 then Time_p1 := TIME p1_up := 1 p1_on := 0 end	Event_P2 where TIME > Time_p2 p1_up = 1 $\vee$ p3_up = 1 p2_on = 1 then Time_p2 := TIME p2_up := 1 p2_on := 0 end

Tabla 7.3: Process Flow Graph Problem (1ra parte)

Event_P3
  where
  TIME > Time_p3
  p3_on = 1
  p1_up = 1 ∨ p2_up = 1
  then
  Time_p3 := TIME
  p3_up := 1
  p3_on := 0
end
	Event_P4
	  where
	  TIME > Time_p4
	  p4_on = 1
	  p3_up = 1
	  then
	  Time_p4 := TIME
	  p4_up := 1
	  p4_on := 0
	end

Event_P5
  where
  TIME > Time_p5
  p5_on = 1
  p2_up = 1 ∧ p4_up = 1
  then
  Time_p5 := TIME
  p5_up := 1
  p5_on := 0
end	

Tabla 7.4: Process Flow Graph Problem (2da parte)

Capítulo 8

Conclusiones

De acuerdo a lo realizado a lo largo del proyecto, se mencionan a continuación algunas conclusiones.

Inicialmente, se analizó el lenguaje de especificación de modelos Event-B determinando sus limitaciones, haciendo pruebas y revisando posibles maneras de realizar los patrones de diseño a través de ejemplos. Más adelante se analizó los procesos de cálculo NTCC, especificando los patrones de diseño, realizando pruebas y validando con otras personas conocedoras del tema para luego crear los patrones de diseño que permiten la especificación de modelos en NTCC y Event-B. Con esto se puede concluir que los mecanismos de sincronización de procesos en NTCC permiten especificar el tipo de patrones que se querían diseñar en Event-B gracias a la noción de tiempo y no determinismo que tiene este lenguaje.

La sincronización de sistemas reactivos requieren, por un lado, el uso del tiempo para conocer el orden de la ejecución de los procesos, y por otro lado, el no determinismo, para poder sincronizar sistemas como el envío de mensajes sin respuesta (tipo streamming, por ejemplo). La manera como en Event-B se realizaría esto sería creando guardas y variables que permitan levantar una *bandera* que indicara qué procesos (en este caso eventos) ejecutar. Sin embargo y como se observó a lo largo del proyecto, realizar directamente este proceso resulta muy exhaustivo por lo que debe existir otra manera de sincronizarlos. Es por esta razón que aparece el lenguaje NTCC, ya que con la noción del tiempo que posee y sus operadores para representar acciones no determinísticas permiten crear modelos de sincronización de sistemas reactivos de manera mucho más eficiente y más cómoda.

Es así como el lenguaje NTCC se convierte en la opción viable para

realizar estos modelos. Sin embargo, cuando los modelos se vuelven muy grandes, es decir, cuando los procesos y la sincronización de ellos son demasiados, la especificación de procesos en este lenguaje resulta ser muy repetitivo y tedioso. Por tal razón se llega a la conclusión de usar un lenguaje de alto nivel que sea capaz de mapear a NTCC.

El lenguaje de alto nivel permite, además de una "limpieza" en el modelado de la sincronización de procesos (se realiza en forma más natural y como código de programación), modularizar los tipos de patrones, es decir, crear lo que se denominó Dependencias en el trabajo presentado. Éstas dependencias sirven para especificar patrones genéricos en la sincronización de procesos, y fueron necesarios justamente para poder indicar qué tipo de patrones de diseño se iban a modelar (en las limitaciones del proyecto).

Con respecto a otras formas de modelar la sincronización de procesos, la razón de modelar desde NTCC radica en poder probar los modelos creados, además que otros lenguajes no tienen la noción del tiempo definida y NTCC fue creado para procesos reactivos y concurrentes, así que era la opción viable para la especificación de los sistemas reactivos.

Con base a lo anterior, un modelador de sistemas podrá especificar la sincronización de los procesos de una manera más sencilla, sin tener que pasar por el uso de guardas y variables que permitan esta sincronización, además de preguntarse si existe algún error en la sincronización de procesos mostrando los invariantes dentro del modelo acerca de la sincronización. Ésta facilidad en el diseño de los modelos se observan en el capítulo 7 del presente trabajo.

Finalmente, se puede concluir que como trabajo futuro podría usar modelos que incluyan los otros operadores de NTCC, de tal forma que se tengan más patrones de diseño sobre los procesos reactivos y extiendan de alguna manera la traducción del lenguaje de alto nivel a NTCC y luego a Event-B.

8.1 Trabajo Futuro

A pesar del análisis de los modelos, es necesario mencionar posibles extensiones al trabajo teniendo en cuenta las limitaciones de este sobre tiempo y espacio. A continuación se presentan futuras extensiones probables en el desarrollo futuro del proyecto:

- Pruebas con otros modelos más fuertes, es decir, extensión del modelo con otros tipos de patrones de diseño.

- Extensión de los modelos creados en el presente trabajo como un Plug-In dentro del sistema de Event-B.
- Prueba de correctitud del modelo, generalización mediante automatas.
- Extensión del modelo, uso de todos los condicionales posibles.
- Revisión en una aplicación práctica, evaluación por parte de pares o personas que sean capaces de modelar dichos sistemas.

Bibliografía

[Abr10] Jean-Raymond Abrial. *Modeling in event-b, system and software engineering (1ra ed.)* Cambridge University Press, 2010.

[CMR07] Dominique Cansell, Dominique Méry, and Joris Rehm. "Time Constraint Patterns for Event B Development". In: *B*. 2007, pp. 140–154.

[DY12] Pierre-Malo Deniélou and Nobuko Yoshida. "Multiparty Session Types Meet Communicating Automata". In: *ESOP*. Ed. by Helmut Seidl. Vol. 7211. Lecture Notes in Computer Science. Springer, 2012, pp. 194–213. ISBN: 978-3-642-28868-5.

[HFA09] Thai Son Hoang, Andreas Fürst, and Jean-Raymond Abrial. "Event-B Patterns and Their Tool Support". In: *SEFM*. 2009, pp. 210–219.

[Lub02] Alan C. Shaw Lubomir F. Bic. *Operating Systems Principles (1ra ed.)* Prentice-Hall/Pearson Education, 2002.

[NPV02] Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia. "Temporal Concurrent Constraint Programming: Denotation, Logic and Applications". In: *Nord. J. Comput.* 9.1 (2002), pp. 145–188.

[OR14] Carlos Olarte and Camilo Rueda. *Communicating Systems in Event-B*. Tech. rep. 2014. URL: `http:%20//cic.puj.edu.co/~caolarte/sessionB/`.

[ORV13] Carlos Olarte, Camilo Rueda, and Frank D. Valencia. "Models and emerging trends of concurrent constraint programming". In: *Constraints* 18.4 (2013), pp. 535–578.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[Sip05] Michael Sipser. *Introduction to Theory of Computation (2nd Ed.)* Thompson Course Technology, 2005.