Herramientas y mecanismos formales para el tratamiento de la tolerancia a fallas

por

Araceli Acosta

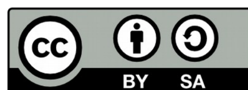
Presentado ante la Facultad de Matemática, Astronomía, Física y Computación como parte de los requerimientos para la obtención del grado de Doctora en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CÓRDOBA

septiembre, 2022

Director: **Nazareno Aguirre**

Tribunal Especial:
Titulares
Dr. Silvio Miguel GONNET (CONICET - UTN, Santa Fe)
Dr. Raúl Alberto FERVARI (FAMAF)
Dr. Valentín CASSANO (UNRC)
Suplentes
Dr. Maximiliano CRISTIA (UNR - CONICET)
Dr. Pedro Octavio SÁNCHEZ TERRAF (FAMAF)



Este trabajo se distribuye bajo una licencia Creative Commons Attribution-ShareAlike
CC BY-SA https://creativecommons.org/licenses/by-sa/4.0/deed.es

Índice general

Prefacio **5**

1. Marco conceptual **9**
1.1. Conceptos básicos 9
1.2. Clasificación de las fallas 14
1.3. Estrategias de tolerancia a fallas 20
1.4. Reflexiones sobre el modelado de sistemas 24
1.4.1. Sistemas reactivos vs. sistemas transformacionales 24
1.4.2. Modelos orientados a estados y orientados a eventos 25
1.4.3. Prescripción vs. descripción 26
1.4.4. Componentes 26
1.4.5. Modelado de la ocurrencia de fallas 27
1.5. Conclusiones y notas metodológicas 28

2. Sistemas Reactivos para el Modelado ... **29**
2.1. Especificación 30
2.2. Modelo del Sistema 31
2.3. LTS Reactivos 34
2.4. Modelo del Sistema con Fallas 35
2.5. Definiciones y Propiedades 40
2.6. Modelo de fallas 45
2.7. Consideraciones finales 46

3. fCTL, una nueva lógica basada en estados **47**
3.1. Antecedentes 47
3.1.1. Estructuras de Kripke 47
3.1.2. CTL (Computation Tree Logic) 48
3.1.3. Deontic Computation Tree Logic 49
3.2. Definición de fCTL 53
3.3. Casos de estudio 58
3.3.1. Celda de memoria 58
3.3.2. Protocolo Token Ring 60
3.4. Tolerancia a fallas con fCTL 62
3.5. Consideraciones finales 65

4. Verificación de una lógica deóntica temporal **67**
4.1. Antecedentes 68
4.1.1. u-calculus 68
4.2. Presentación de la lógica 70
4.2.1. Lógica Deóntica Proposicional (PDL) 70
4.2.2. Lógica Deóntica Temporal DTL 73
4.2.3. Complejidad de PDL 75
4.3. Primera Traducción 77
4.3.1. Traducción de PDL a u-calculus 77
4.3.2. Corrección de Tr 80
4.3.3. Traducción de DTL a u-calculus 83
4.3.4. Corrección de la extensión de Tr 85
4.3.5. Lemas auxiliares 91
4.4. Segunda Traducción 93
4.4.1. Segunda traducción de PDL a u-calculus 94
4.4.2. Corrección de la segunda traducción Tr 96
4.4.3. Segunda traducción de DTL a u-calculus 99
4.4.4. Corrección de la segunda traducción extendida Tr 100
4.5. Casos de estudio 104
4.5.1. Productor-consumidor 104
4.5.2. Muller C-Element 108
4.6. Consideraciones finales 112

5. Conclusiones **115**

A. Código Mucke de casos de estudio **119**

Prefacio

La confiabilidad es uno de los atributos de calidad más importantes asociados al desarrollo de software, y la *corrección de software* (es decir, en qué grado el software cumple con el propósito para el cual fue desarrollado) es, tal vez, el factor más importante asociado a la confiabilidad [GJM03]. Además de los defectos de software tradicionales (bugs) que afectan a la corrección, pueden existir eventos *externos*, fuera del control del programa, que provoquen comportamiento erróneo o no deseado. Algunos ejemplos son los correspondientes a fallas en dispositivos de hardware, cortes de energía eléctrica, fallas de comunicación, etc. Todo análisis de confiabilidad, especialmente para sistemas críticos, es decir, sistemas en los cuales el mal funcionamiento del software o hardware pueden llevar a drásticas consecuencias (sistemas de control de plantas nucleares, vehículos espaciales, etc.), necesita poder razonar acerca de los efectos de estos eventos y cómo el sistema los tolera o reacciona ante los mismos.

La importancia de la garantía de confiabilidad ante eventos externos como los mencionados anteriormente se refleja en el hecho de que la *tolerancia a fallas* [Dub13], el área de estudio en las Ciencias de la Computación que se centra en cómo modelar, operar y razonar con sistemas sujetos a fallas externas y a su tolerancia, se ha vuelto un área de investigación muy activa en las últimas décadas [BJRT09].

La tolerancia a fallas se asocia generalmente con los sistemas de hardware y redes de computadoras, y a mecanismos vinculados al manejo de bajo nivel de dispositivos, y protocolos en sistemas distribuidos. De hecho, en el diseño de hardware y protocolos de redes, se han diseñado diversos mecanismos para proveer cierto comportamiento esperado a pesar de la ocurrencia ocasional de fallas. Estas fallas en general son conocidas *a priori*, y pueden deberse a una variedad de razones, tales como ruido electromagnético afectando un circuito integrado, o partículas energéticas afectando el estado de una componente electrónica. Este tipo de errores, denominados usualmente *errores suaves* (soft errors), constituyen una de las principales causas de la caída de sistemas, y uno de los dominios tradicionales de la tolerancia a fallas [Sah06].

A pesar del contexto tradicional de redes y hardware vinculado a la tolerancia a fallas, en las últimas décadas el problema de la tolerancia a fallas ha visto un enorme desarrollo, en cierto sentido más general, con el surgimiento de nuevas áreas como las de sistemas *self managed*, *self healing* y *self adaptive*

[dLF02] [GS02] [GMK02] (distintos tipos de sistemas que tienen la habilidad de auto administrarse en base eventos externos, y en particular de reconfigurarse o "curarse" a sí mismos como respuesta a tales eventos). Estas áreas encaran la tolerancia a fallas desde una perspectiva más abstracta y, también, desde una etapa más temprana en el proceso de desarrollo de software.

Una característica sorprendente de la tolerancia a fallas, como área de investigación dentro de las Ciencias de la Computación, es el relativamente poco trabajo en el desarrollo de soluciones de propósito general, o ambientes para el análisis de sistemas tolerantes a fallas. La amplia mayoría de las soluciones a problemas de tolerancia a fallas son *ad hoc*, y atacan problemas específicos en contextos específicos. Si bien existen soluciones genéricas, éstas ofrecen esquemas para lidiar con la tolerancia a fallas en contextos específicos, sin ofrecer formas de analizar el impacto en cada contexto. Por ejemplo, entre las técnicas para tolerancia a fallas en software, las soluciones más importantes tienen que ver con formas de aprovechamiento (o imposición) de redundancia o diversidad. La diversidad puede ser diversidad de diseño, como en *N-version programming* [Avi85] y bloques de recuperación, o diversidad de datos, como en los bloques de reintento y la *N-copy programming* [Pul01]. La redundancia puede ser redundancia temporal, como en el caso de la retransmisión de mensajes, o redundancia espacial, como es el caso de código de corrección de errores. Como vemos, en cada uno de estos casos está presente la redundancia y/o diversidad, aunque su materialización en cada contexto es muy diferente.

Por otra parte, si nos enfocamos en los *métodos formales* [O'R17], es decir, las técnicas rigurosas de especificación, construcción y verificación de software, cuyo objetivo es ofrecer mayores garantías del correcto funcionamiento del software, veremos que en su amplia mayoría se concentran en garantizar la ausencia de defectos de software (bugs) [MH01]. Si entendemos a estos métodos como enfoques que contribuyen a la construcción de sistemas libres de defectos, y observamos que de hecho los defectos de software son una de las causas fundamentales de fallas, podemos decir que los métodos formales más tradicionales son también técnicas para la construcción de sistemas tolerantes a fallas. Sin embargo, el enfoque tradicional de los métodos formales no ataca el problema de lidiar con fallas provenientes de eventos externos al software, o al módulo que se está desarrollando, y por lo tanto, en general resultan insuficientes. Cabe mencionar que algunas propuestas más recientes se enfocan específicamente en la aplicación de métodos formales a la tolerancia a fallas. Por ejemplo, métodos formales orientados a modelos para ciertas formas de tolerancia a fallas [JHAL09], o técnicas formales para la verificación de sistemas con mecanismos específicos de tolerancia a fallas (a bajo nivel) [BF15], además de la definición de lógicas particulares para tolerancia a fallas [CM09b].

Objetivos

El objetivo general de esta tesis es, en primer lugar, reflexionar sobre la tolerancia a fallas desde una perspectiva formal, y a la vez analizar y diseñar herramientas formales generales que nos ayuden en el proceso de construcción y razonamiento sobre sistemas tolerantes a fallas. Entre las tareas que llevaremos adelante para alcanzar estos objetivos, se destacan la definición formal de conceptos y propiedades fundamentales relevantes en el contexto de sistemas tolerantes a fallas, el desarrollo de formalismos que faciliten la expresión de propiedades relacionadas con la tolerancia a fallas, y la presentación de algunos resultados técnicos relativos a los formalismos estudiados y desarrollados para la verificación automática, en particular model checking.

Este trabajo se enfoca principalmente en el desarrollo de formalismos y herramientas de análisis que apuntan a etapas "tempranas" en la construcción de sistemas tolerantes a fallas, de diseño y análisis de este tipo de sistemas, sobre modelos abstractos de los mismos. Alrededor de este objetivo, elaboraremos diferentes aspectos, tales como:

- Definición de un marco teórico general de especificación de sistemas, que permita comprender el anclaje principal de los desarrollos de la tesis.
- Definición de conceptos y propiedades fundamentales de sistemas tolerantes a fallas.
- Propuesta de formalismos lógicos que faciliten la expresión formal de propiedades relacionadas con la tolerancia a fallas.
- Estudio de propiedades de los formalismos lógicos en sí mismos, y en relación a traducciones entre diferentes formalismos, motivados por análisis automático.

Alcance de esta Tesis

Como mencionamos anteriormente, para que un sistema sea considerado tolerante a fallas es necesario en general hacer explícito el conjunto específico de fallas que el sistema debe tolerar [Gär99]. Algunas estrategias para diseñar sistemas tolerantes a fallas consisten en soluciones *ad hoc*, a través de componentes que introducen redundancia, y la definición de mecanismos para tomar decisiones en base a las componentes redundantes (*voting*, por ejemplo). A este tipo de enfoques se suman otras estrategias de "diseño diversificado", que consisten en multiplicar el proceso de diseño e implementación de cada subsistema por parte de equipos independientes de desarrollo, de manera que la redundancia obtenida de lugar a implementaciones diversificadas, con menos chances de contener los mismos defectos.

Este tipo de soluciones de diseño de sistemas tolerantes a fallas trabaja sobre la premisa de que el desarrollo puede ser defectuoso, es decir, contener errores. Existen numerosos enfoques basados en métodos formales para lidiar con la construcción de software “libre de defectos”, en el sentido de poder garantizar que el mismo satisface cierta especificación formalmente planteada (enfoques para la verificación formal de programas). En esta tesis, nos concentraremos en las fallas que *no* pueden resolverse a través de estos enfoques más clásicos, es decir, nuestro foco estará en las fallas producidas por eventos externos no controlables. Las fallas correspondientes a errores en el desarrollo de software pueden ser indirectamente contempladas en nuestro escenario considerando los módulos externos con los que interactúa el sistema bajo análisis. El diseño de estos módulos externos no están en nuestro dominio, y las fallas que generan son consideradas fallas externas que nuestro sistema debe tolerar; más allá de que éstas sean de diseño, desarrollo u otro tipo de fallas.

En relación a las etapas de desarrollo, se recorrerán aspectos conceptuales de la tolerancia a fallas, para luego enfocarnos en herramientas para la especificación formal de sistemas tolerantes a fallas, sin avanzar en etapas de diseño o implementación.

En el capítulo 1 haremos un recorrido conceptual informal sobre la tolerancia a fallas y conceptos relacionados. En el capítulo 2 analizaremos un formalismo basado en sistemas reactivos [Gab08] que, además de permitir modelar cambios de comportamientos luego de la ocurrencia de fallas, será nuestra primera aproximación para modelar formalmente algunos conceptos abordados previamente de manera informal, a partir de diferenciar la especificación del sistema (las características básicas que debe garantizar) del modelo de implementación (modelos que explicitan algunas decisiones de implementación). En el capítulo 3 introducimos dCTL y fCTL, lógicas basadas en estados cuyos modelos son variantes de estructuras de Kripke para la descripción de sistemas con fallas. Los modelos de estas lógicas se basan en estados, en contraste con el capítulo previo cuyos modelos ponen el foco en las acciones o eventos. Exploramos en este capítulo varias propiedades caracterísitcas de la tolerancia a fallas que se pueden expresar con estas lógicas. Finalmente en el capítulo 4 abordamos una lógica cuyo modelo incorpora tanto acciones como propiedades de estados, y trabajamos sobre una traducción a μ-calculus que nos permite realizar verificación automática de propiedades sobre esta lógica.

[...] while, before, a program with only one bug used to be almost correct, afterwards a program with an error is just wrong.

EWD1036-0.

1 Marco conceptual

Por tolerancia a fallas entenderemos dos cuestiones. Por un lado, hacemos referencia a la capacidad de los sistemas de mantener un grado de comportamiento correcto o aceptable, ante la ocurrencia de fallas. Por otro lado, nos referimos al área de estudio dentro de las Ciencias de la Computación, y conjunto de técnicas de desarrollo de la misma, que nos ayudan a construir sistemas tolerantes a fallas.

En general en contexto de validación y verificación de software, y en particular en el ámbito de los sistemas tolerantes a fallas, suele hacerse referencia a diversos conceptos, como el de *falla* (fault), *error* y *fallo* (failure). Además, en el contexto de la tolerancia a fallas suelen mencionarse procesos o tareas, como la *detección* y la *recuperación*, o mecanismos como la *redundancia*. Este capítulo revisará estos conceptos, ampliamente utilizados por ingenieros y desarrolladores de sistemas críticos, y las intuiciones alrededor de ellos, para luego avanzar en formalizarlos sin alejarnos de sus definiciones informales más aceptadas.

1.1. Conceptos básicos

La primera idea intuitiva que surge cuando se habla de un sistema con *tolerancia a fallas* es la de un sistema que sigue funcionando a pesar de la ocurrencia de fallas. Es decir, la *tolerancia a fallas* es la habilidad de un sistema de comportarse "correctamente", o con un nivel de corrección aceptable, una vez que las fallas ocurren.

Para hablar de sistemas tolerantes a fallas debemos empezar por hablar de *sistemas*. Comenzaremos con una primera definición de este concepto intro-

ducida por P. Jalote en [Jal94].

> Un *sistema* se define como un mecanismo identificable que mantiene un patrón de comportamiento en su interfaz entre el sistema y su entorno. Esta definición implica que un sistema debe interactuar con su entorno, y que el entorno del sistema es necesario para definirlo. Esencialmente, la especificación del entorno define los límites de las atribuciones del sistema. Entonces, el "patrón de comportamiento" en la interfaz, es el comportamiento externo del sistema, que es observado por el entorno. [Jal94][1]

Lo que queremos destacar de esta definición es la idea de que *un sistema se define en relación a su entorno*. Refinaremos esta noción más adelante cuando hablemos de fallas, y si éstas son parte del entorno, de otros subsistemas, o de componentes de más bajo nivel del mismo sistema.

En la bibliografía encontramos tres términos muy utilizados para definir tolerancia a fallas: *falla* (fault), *error* y *fallo* (failure). En el artículo titulado *Fault-Tolerant Computing: Fundamental Concepts* [Nel90], V. Nelson define estos conceptos de la siguiente manera:

> *Fallo* denota la incapacidad de un elemento para realizar la función diseñada debido a un *error* en el elemento o su entorno, que a su vez es causado por varias *fallas*. Una *falla* es una condición física anómala. Sus causas incluyen errores de diseño, como equivocaciones en la especificación del sistema o su implementación; problemas de fabricación; daño, fatiga, u otro deterioro; y alteraciones externas como duras condiciones ambientales, interferencia electromagnética, radiación ionizante, entradas no previstas, o mal uso del sistema.[...] Un *error* es la manifestación de una *falla* en la que el estado lógico del sistema es diferente de su valor previsto. [...] El término *error-logico*, en inglés *soft error*, se aplica frecuentemente a errores que perduran aún después de que la falla que los origina desaparece. Una vez corregidos, los errores lógicos, no dejan ningún daño en el sistema.[2]

L. Pullum [Pul01] define estos conceptos de la siguiente manera:

> Una *falla* es la causa identificada o supuesta de un error, a veces denominada *bug*. Puede ser vista como una simple consecuencia de un *fallo* de algún otro sistema (incluido el desarrollador) que brinda servicios al sistema dado. Una falla activa es una que produce un error.

[1] Traducción propia.
[2] Traducción propia.

> Un *error* es parte de un estado del sistema que podría generar un *fallo*. Puede no estar reconocido como un error (i.e., encontrarse *latente*) o haber sido detectado. Un error se puede propagar, esto es, producir otros errores. Las *fallas* se reconocen como presentes cuando los errores son detectados.
>
> Un *fallo* ocurre cuando el servicio que brinda el sistema se desvía del servicio especificado, o provee un resultado incorrecto. Esto implica que el servicio esperado es descripto, típicamente por una especificación o un conjunto de requisitos. [3]

Las dos definiciones anteriores ponen importante énfasis en que una falla puede provenir de un bug, de un error de diseño o implementación. De hecho, esta característica es bastante recurrente en la bibliografía de tolerancia a fallas “ajena” a las técnicas de verificación formal de programas.

Por su parte, F. Gärtner, previo a introducir una formalización de estos conceptos en [Gär99], lo describe de la siguiente manera:

> El término *falla* se utiliza normalmente para nombrar un defecto en un nivel de abstracción más bajo, e.g., una celda de memoria que siempre devuelve 0 [Jal94]. Una *falla* puede causar un *error*, que es una categoría del estado del sistema. Un error, en efecto, puede llevar a un *fallo*, es decir, que el sistema se desvíe de su especificación de corrección.[4]

Sobre Nuestro enfoque

El presente trabajo se centra en el objetivo de brindar herramientas para especificar y diseñar sistemas tolerantes a fallas, usando métodos formales. La propuesta es complementaria a los métodos formales tradicionales, los cuales ya brindan herramientas para garantizar la corrección de programas. En nuestro caso, supondremos que la fuente de fallas es externa al programa, y no producto de un error de programación (bug) en el sentido clásico. Más precisamente, supondremos que los eventos asociados a las fallas están por fuera del control del sistema, aunque sus efectos afecten al mismo. Las fallas, entonces, pueden provenir de desperfectos de más bajo nivel (capas subyacentes o hardware), o de otros sistemas o subsistemas con los que el sistema bajo análisis interactúa. Como mencionamos anteriormente, si bien no consideramos aquí defectos provenientes del desarrollo de la propia componente, sí es posible contemplar indirectamente defectos de desarrollo de otras componentes, simplemente interpretando como fallas las interacciones con comportamientos incorrectos de estos sistemas.

[3] Traducción propia.

[4] Traducción propia.

El problema de determinar qué comportamientos son correctos o aceptables ante la manifestación de una falla depende de los requisitos establecidos para cada sistema; es decir, no sólo se distingue entre comportamiento correcto e incorrecto, sino que pueden existir matices que consideramos aceptables. En relación a esto último, por ejemplo, en el protocolo UDP es aceptable que se pierdan paquetes, en el protocolo TCP no lo es. En este punto cabe resaltar que, dado que no nos interesa razonar sobre defectos en el sentido tradicional (bugs), cualquier divergencia respecto de los requisitos del sistema proviene de una falla. Es decir, en general podemos utilizar la definición de comportamiento aceptablemente correcto, como la definición de comportamiento correcto.

Esta reflexión nos marca tres condiciones necesarias para formalizar estos conceptos:

- Diferenciar comportamiento interno del externo o de más bajo nivel. En estos últimos se producen las fallas.
- Contar con especificaciones del propio sistema y aquellos con los que se interactúa, para diferenciar comportamiento correcto del de falla.
- Describir el comportamiento del entorno, con un modelo que incluya los comportamientos que afectan el sistema.

Resumiendo las definiciones recorridas anteriormente, podemos definir los conceptos de falla, error y fallo de la siguiente manera:

Definición 1.1 *Una* ***falla*** *(o* fault *en inglés) es un comportamiento externo, o de más bajo nivel, e incorrecto (identificado como tal) que afecta el sistema.*

Definición 1.2 *Un* ***error*** *es un estado del sistema provocado por una falla, que eventualmente puede llevar a generar un comportamiento incorrecto del propio sistema (fallo).*

Definición 1.3 *Un* ***fallo*** *(o* failure *en inglés) es un comportamiento incorrecto del sistema; es decir, una violación a alguno de los requisitos del sistema.*

Nos queda pendiente, entre otras cosas, describir mejor a qué nos referimos con comportamiento cuando definimos fallas. Gärtner [Gär99] afirma que una falla puede ser modelada como una transición no deseada, aunque posible. Es decir, muchas veces se identifica una falla con un evento o acción. Si bien utilizaremos principalmente esta noción, resta plantear algunas consideraciones para contemplar diferentes situaciones.

Por otro lado, desde una perspectiva formal, cuando se habla de tolerancia a fallas queda implícito que las fallas están identificadas de alguna manera. Caso contrario, no sería posible calificar si es tolerante un sistema en relación a una falla determinada. En consecuencia, para que un sistema sea tolerante a fallas en general es necesario saber qué fallas debe tolerar [Gär99].

En relación a la definición de la tolerancia a fallas, es importante mencionar el trabajo de Gärtner [Gär99], quien que a partir del trabajo de Arora y Gouda [AG93] sobre *self-stabilization* define la *tolerancia a fallas* en base a un conjunto de fallas F y un invariante P, que permite debilitar la noción tradicional de la tolerancia a fallas. En general se entiende por tolerancia a fallas a la capacidad de un sistema de mantener siempre un correcto funcionamiento, a pesar de la ocurrencia de fallas. En términos formales, que no se viole el invariante P. Esto representa la exigencia de una propiedad de *safety*. Para Gärtner, éste es sólo un tipo de tolerancia a fallas que define como *masking*. Sin embargo, en algunos casos es posible aceptar un comportamiento "no deseado" por un corto plazo si se puede garantizar que, sin la ocurrencia de nuevas fallas, eventualmente se cumpla P. Esta propiedad que se pide es de *liveness*. Este tipo de tolerancia a fallas es llamada *nonmasking*, ya que el efecto de la falla se muestra al no cumplirse el invariante. [5]

En este sentido, podemos arriesgar una segunda definición informal teniendo en cuenta los siguientes elementos:

- La tolerancia a fallas se define en relación a un conjunto de fallas; por lo tanto es necesario identificarlas previamente.
- Formalmente podemos caracterizar el correcto funcionamiento de un sistema, a través de un predicado sobre el estado del sistema y/o en relación a las interacciones con otros subsistemas y el entorno.
- El concepto de tolerancia a fallas no necesariamente implica que el sistema funcione *continuamente* de manera correcta ante la ocurrencia de fallas. Es decir, en algunos casos se puede contemplar un funcionamiento transitorio de menor calidad, pero dentro de los márgenes aceptables.
- En estos casos, el concepto de "margen aceptable" de funcionamiento también es una característica del sistema que debemos formalizar. Esto puede incluir tiempo máximo de recuperación y/o la diferenciación de comportamientos permitidos de comportamientos inaceptables.

Definición 1.4 *Dado un conjunto de fallas, se dice que un* ***sistema es tolerante a fallas*** *si, luego de la ocurrencia de cualquiera de ellas, el sistema nunca deja de funcionar dentro de los márgenes aceptables de corrección. A su vez, si las fallas dejan de ocurrir, el sistema vuelve a funcionar correctamente dentro de un tiempo aceptable.*

Algunos autores proponen dimensionar la tolerancia a fallas de acuerdo al comportamiento del sistema en relación a la ocurrencia de las fallas. Un

[5] Para los términos masking y nonmasking, al ser términos técnicos con significado bien definido, preferimos usarlos en inglés, ante las posibles alternativas en español como enmascaramiento u ocultamiento.

ejemplo de este enfoque es dimensionar la tolerancia a fallas en relación a la frecuencia y probabilidad de que las fallas ocurran [Rav11]. La *confiabilidad* (en inglés, *dependability*) de un sistema puede ser medida en términos de su fiabilidad (reliability) y disponibilidad. La función de fiabilidad (reliability) $R(t)$ es la probabilidad condicional de que un sistema puede realizar la función diseñada en el tiempo t, dado que estaba operativo en el tiempo $t = 0$. Entonces $R(t)$ es función del proceso de fallas que afectan el sistema y los mecanismos que previenen las averías (fallos) cuando ocurre una falla. [6] [Jal94] La función disponibilidad (availability) A(t) se define como la probabilidad que un sistema esté operacional en el tiempo t, por lo que está relacionada con las fallas y el tiempo de recuperación. Se puede medir como una probabilidad de que el sistema esté caído un determinado tiempo aleatorio, o como una relación entre el tiempo no operativo sobre un periodo de tiempo. [Jal94]

En esta tesis no exploraremos aspectos cuantitativos de la tolerancia a fallas, como los basados en medidas de probabilidad, que acabamos de citar.

1.2. Clasificación de las fallas

En esta sección haremos un recorrido de algunas clasificaciones de fallas con el objetivo de identificar aquellas que son pertinentes desde la perspectiva del desarrollo de herramientas formales para especificación y diseño de la tolerancia a fallas.

Nelson [Nel90] clasifica las fallas según su *duración*, su *naturaleza* y su *alcance*.

duración: de acuerdo a su duración, una falla puede ser *transitoria*, *intermitente* o *permanente*.

naturaleza: las fallas pueden de naturaleza *lógica*, cuando éstas se pueden representar dentro de la lógica del sistema, o *indeterminadas*, cuando no pueden ser representadas lógicamente (por ejemplo, una falla causada al estar el voltaje de una celda entre el 0 y el 1).

alcance: el alcance de una falla puede ser *local*, cuando afecta sólo a una componente (donde se origina), o *global*, cuando afecta a varias porciones del sistema (por ejemplo, diversas componentes).

Otras clasificación utilizada es de acuerdo a la forma en que la falla se manifiesta. En este caso, las fallas pueden ser *benignas* o *maliciosas* [KK07]. Las maliciosas (o bizantinas) refieren a aquellas que comunican distintos resultados a diferentes módulos o subsistemas.

En [PST13] se retoma la clasificación de fallas de [ACD$^+$06], considerando los siguientes siete atributos para la clasificación de fallas:

[6] traducción propia.

fase de creación u ocurrencia refiere a si las fallas ocurren durante la ejecución del sistema o fueron introducidas durante el desarrollo o el mantenimiento del sistema.

márgenes del sistema refiere a si las fallas son internas al propio sistema, o fueron generadas fuera del sistema y se propagan dentro del mismo mediante interacción o interferencia (externas).

dimensión si se originan o afectan el *hardware* o si afectan al *software*, ya sea al programa o los datos.

causa fenomenológica se identifican como *humanos*, si son causadas por humanos, o *naturales*, cuando en la causa no interviene un ser humano.

objetivo cuando la falla tiene por objetivo causar daño en el sistema se identifica como *maliciosa*. En caso contrario es *no-maliciosa*.

capacidad puede ser accidental, intencionada, o debido a la incompetencia.

persistencia en el sentido de si sus efectos persisten en el tiempo, *permanente*, o si desaparecen en un período acotado de tiempo, *transitoria*.

Se puede considerar también el “estado” de la falla. Si bien una falla está asociada a un evento y no a un estado, se puede considerar que una falla transitoria en un canal de comunicaciones no tiene ningún efecto en la medida en que no se utilice ese canal. Se puede distinguir entonces las fallas entre activas, inactivas y latentes. La última categoría corresponde a las fallas que pueden tener una determinada duración y, mientras el efecto perdure, se las considera latentes en tanto no se manifiesten debido al comportamiento del resto del sistema.

Otro tipo de fallas que se puede identificar son las fallas de precisión. Éstas están relacionadas con la naturaleza física del mundo, y los dispositivos de digitalización o modelado de los mismos que hacen imposible “capturar la realidad” tal como es. Consideremos por ejemplo un sensor de temperatura: por más preciso que fuere, al representar digitalmente con valores discretos información naturalmente contínua estamos “perdiendo” información. Otros ejemplos similares surgen de observar otras magnitudes a través de sensores, como por ejemplo el uso de sensores para el cálculo de la aceleración de un vehículo: en general, estos sensores tienen en cuenta la aceleración de la gravedad; si el dispositivo debe proveer información relativa (la aceleración del vehículo), es necesario restar la aceleración de gravedad a la lectura; pero, ¿cuál es la aceleración de la gravedad en un punto determinado de la tierra? Por más complejo que sea nuestro modelo de la tierra, no es posible contar con absoluta precisión con esta información, lo que provoca que en lecturas sucesivas, la información de aceleración obtenida no sea precisa sino aproximada.

Este tipo de fallas es de más difícil abordaje, ya que son parte intrínseca de los dispositivos físicos o de digitalización.

Siguiendo con la descripción de clasificaciones de fallas, en [Wil00, SWH94] puede encontrarse una clasificación, similar que mostramos en la tabla 1.1.

Tabla 1.1: Clasificación de fallas [Wil00, SWH94]. Traducción propia.

Criterio	Tipo de Falla
Actividad	Latente vs. Activa
Duración	Transitoria vs. Permanente (intermitente)
Percepción	Simétrica vs. Asimétrica
Causa	Aleatoria vs. Genérica
Intención	Benigna vs. Maliciosa
Cantidad	Aislada vs. Múltiple
Tiempo (fallas múltiples)	Coincidentes vs. Diferenciadas
Causa (fallas múltiples)	Independientes vs. Causa común en múltiples componentes

En estas clasificaciones, además de las mencionadas hasta el momento, se incorporan las siguientes:

percepción distingue las fallas simétricas de las asimétricas. Las fallas simétricas son aquellas percibidas de la misma manera por los diferentes componentes del sistema; una falla asimétrica se percibe de distinta forma por distintos componentes del sistema.

causa diferencia fallas aleatorias, asociadas con causas del entorno (humedad, calor, vibración) o causadas por la degradación de los componentes, de las causas generales, que se asocian con fallas de construcción y diseño. Esta clasificación se relaciona fuertemente con la *fase de creación* y *causa fenomenológica* de la clasificación anterior.

intención al igual que la clasificación por *objetivo*, distingue fallas entre benignas y maliciosas, aunque se entiende por benigna a una falla que es detectable por todos los subsistemas “buenos”, y maliciosa en caso de no ser detectable.

cantidad se asocia a los efectos de las fallas en diferentes subsistemas. Una falla simple se asocia a una falla en un solo subsistema; si se afectan múltiples subsistemas, se identifica como un grupo de fallas múltiples.

tiempo (fallas múltiples) está relacionada a la granularidad del tiempo. Las fallas *múltiples coincidentes* son aquellas que aparecen en un mismo in-

tervalo de tiempo. Las fallas *múltiples no coincidentes* (tiempo) son aquellas que ocurren en diferentes intervalos de tiempo.

causa (fallas múltiples) corresponde sólo al caso de fallas múltiples. Se dice que las fallas son *independientes* si se originan por diferentes causas o naturaleza. Se dice que las fallas son *de modo común* cuando son fallas que tienen una causa común y están presentes en múltiples componentes.

Jalote [Jal94] propone clasificar las fallas en un sistema distribuido basándose en cómo se comportan las componentes del sistema cuando éste falla.

Crash Fault La falla que causa que la componente se detenga o pierda su estado interno. Con este tipo de fallas, una componente, cuando falla, nunca experimenta una transición incorrecta de estados.

Omission fault Una falla que causa que una componente no responda a algunas entradas.

Timing fault Una falla que causa que una componente responda o muy temprano o muy tarde. Esto también es llamado una falla de performance.

Bizantina Una falla que ocurre de forma arbitraria, y causa que la componente se comporte en forma arbitraria.

Jalote luego incorpora el tipo de fallas **incorrect computation faults**, que se incluye como un subconjunto de las bizantinas y que refieren a fallas que generan que el sistema produzca una salida incorrecta.

Gärtner [Gär99] propone una caracterización similar para modelar distintos tipos de fallas en sistemas distribuidos.

> Ejemplos bien conocidos son el modelo *crash-failure* (en el cual el poceso simplemente se detiene en un punto específico del tiempo), *fail-stop* (en el cual el proceso colapsa, pero esto puede ser detectado fácilmente por sus vecinos), o bizantino (en el cual los procesos se pueden comportar de manera arbitraria, incluso con intenciones maliciosas). La corrección del sistema siempre se prueba con respecto a un modelo específico de fallas.

Una observación pertinente en este punto atañe a las **fallas bizantinas**. Tales fallas, de acuerdo a las definiciones que se encuentran usualmente en la literatura, abarcan prácticamente cualquier tipo de falla, sin restricciones en su periocidicidad, su causa o su efecto. En tal sentido, resulta inviable, desde una perspectiva formal, abarcarlas como una *clase* de fallas, tanto desde el punto de vista de su formalización, como del de mecanismos de tolerancia asociadas a las mismas. Si bien en esta tesis abordaremos fallas que con

frecuencia son clasificadas como bizantinas (por ejemplo, fallas de comunicación o de errores en lectura y escritura de datos), en cada caso puntual las fallas tendrán una especificación concreta de su comportamiento. Por esta razón, evitaremos hablar del tratamiento de fallas bizantinas como una clase de fallas.

Nuestro enfoque

Nuevamente, si consideramos que la nuestra es una perspectiva formal y que apunta a generar herramientas para las primeras etapas de desarrollo, podemos seleccionar estas dimensiones (clasificaciones de fallas) según el interés y la pertinencia a este propósito. El recorrido sobre diferentes caracterizaciones de fallas nos es de utilidad para modelar apropiadamente las fallas y contar con un modelo formal más completo.

Recordemos que en [PST13] se hace referencia a *márgenes del sistema* en relación a si una falla es interna o externa, distinguiendo esta última entre falla de interacción o interferencia. En relación a fallas internas, deberíamos contemplar las de hardware o servicios de más bajo nivel, pero no abordaremos aquí las de implementación. En relación a las externas, asumiremos que las de interferencia en realidad afectan los dispositivos de hardware por lo que se encuentran entre las primeras. El resto son fallas externas de interacción. Por lo tanto aquí se nos abren dos universos en el modelado. Aquellos modelos en los que la falla ocurre dentro del sistema (generada por fallas de más bajo nivel), y las que ocurren fuera del sistema, que tienen que ver con las interacciones con otros sistemas y el entorno.

Es posible que una falla provoque efectos en múltiples subsistemas. Esto se puede modelar como múltiples fallas que se relacionan entre sí. En estos casos, se podría contemplar la categoría percepción, que diferencia las fallas simétricas (cuando todos los subsistemas la perciben de la misma manera) de las asimétricas. Análogamente, se podría pensar que la ocurrencia de múltiples fallas con una causa común también corresponde a este caso. En el caso de diferentes eventos de falla asociados con una falla común de más alto nivel, se podría capturar su relación a través de restricciones que formen parte del modelo de fallas. Caso contrario, las fallas se consideran independientes.

Otra característica que distingue los efectos de las fallas, es si son fallas de protocolo o de semántica.

Independientemente de cómo decidamos modelar una falla (anteriormente se mencionó la posibilidad de modelarlas como eventos), en un nivel más abstracto sería deseable poder representar la *duración* o *persistencia* de las fallas. Por ejemplo, la pérdida de un mensaje en un canal de comunicación, es una falla instantánea, pero la caída por un largo período de tiempo puede ser considerada permanente o intermitente, si el canal está en malas condiciones o cada tanto funciona bien. En los casos, por ejemplo, de un desperfecto

en una memoria (asociada a una falla de hardware) o la pérdida completa de un nodo en un sistema distribuido que se podría atribuir a errores graves de conexión, entendemos que la falla puede modelarse con un evento en la medida que podamos contemplar como parte de los efectos de ésta el cambio de comportamiento del entorno o los subsistemas fallidos.

Otra característica que podríamos contemplar es el *estado* de la falla. Si está activa, inactiva o latente. Es necesario revisar si esta característica está asociada a una falla o a un estado del sistema, en el caso de que las fallas fueran modeladas como eventos.

Finalmente, en relación a la detectabilidad mencionada en [Wil00], en la categoría de intención, consideramos que en lugar de ser una atribución de una falla es una capacidad del sistema y un herramienta más para la tolerancia. En la próxima sección desarrollamos más este aspecto.

Algunos modelos tradicionales, en sistemas distribuidos, consideran las fallas como de crash, omisión, falla temporal y falla bizantina. Estos modelos podrán servir de ejemplos, pero no hacen a las características generales de las fallas, así que lo dejaremos en un segundo plano.

Otros aspectos dejan de tener sentido, ya sea por su naturaleza o el marco de nuestro enfoque formal:

naturaleza de las fallas y fallas de precisión sobre la distinción entre fallas lógicas o no lógicas, si bien en nuestros modelos es necesario incluirlas a todas, lo cierto es que las fallas no lógicas se modelarían en un marco lógico (dado nuestro enfoque), por lo que no es necesario distinguirlas en el modelo del tipo de fallas. Las fallas de precisión en esta problemática tienen un interés particular, pero consideramos que su abordaje está asociado a las características de cada problema en particular, por lo que no las consideraremos en esta instancia.

fase de creación u ocurrencia Excluiremos las fallas internas al desarrollo. Dado que la nuestra es una perspectiva formal, esperamos poder aprovechar mejor otras herramientas para evitar este tipo de fallas, sin descartar las consideraciones hechas en la sección 1.1.

dimensión No es necesario diferenciar si las fallas son de hardware o de software, dado que su formalización permite un abordaje teórico de las mismas que nos permite abstraernos de estas características.

causa fenomenológica En el modelo formal no es de nuestro interés diferenciar si las fallas son causadas por humanos o por causas naturales. Consideramos que no es una categoría apropiada para diferenciar las particularidades que pueden tener las interacciones con humanos, en particular en casos accidentales o intencionales, ya que esa interacción puede ser mediada por otros sistemas.

objetivo En relación al objetivo, no es necesario diferenciar si son maliciosas o accidentales. Al contemplar un modelo de falla, simplemente se describe su comportamiento y posibilidad de ocurrencia, independientemente de su objetivo. Esto no quita que se tenga presente, al hacer el relevamiento de las posibles fallas, la posibilidad de ocurrencia de fallas maliciosas.

capacidad al igual que el objetivo y la causa fenomenológica, proponemos no diferenciar las fallas en esta categoría.

1.3. Estrategias de tolerancia a fallas

La especificación del sistema permite definir formalmente el comportamiento deseado del sistema. Una vez identificadas las posibles fallas, para lograr que el sistema sea tolerante a fallas, se incorporan al diseño e implementación elementos que hacen posible esa tolerancia. Los conceptos de redundancia, recuperación, detección de fallas aparecen en esta instancia. En esta sección describiremos algunos de ellos para destacar los que nos interesan desde el abordaje formal.

En la tolerancia a fallas, también se suelen utilizar otras técnicas como self healing y self management [dLF02, GS02, GMK02, GSRRU07, DvdHT02] que permiten abordar la tolerancia a fallas desde el diseño mismo de los sistemas. Aunque no indagaremos demasiado en esta tesis sobre estas técnicas, las mismas presentan una perspectiva interesante para pensar la tolerancia a fallas a nivel de diseño, como la que proponemos en este trabajo.

Gärtner [Gär99] menciona dos elementos a tener en cuenta en la tolerancia a fallas. Por un lado afirma que no importa cuan bien diseñado o cuan tolerante a fallas sea un sistema, siempre está la posibilidad de un *fallo* si las fallas son muy frecuentes o muy severas. Por otro lado, para que un sistema sea tolerante a fallas debe haber algún tipo de *redundancia*. El significado usual que le damos a redundancia es de un elemento presente (en este caso en el software), pero que no es indispensable porque algún otro elemento cumple con su función [Gär99]. Por ejemplo, en teoría de códigos, los bits de una transmisión que no llevan información son redundantes. Gärtner distingue dos tipos de redundancia: en espacio y en tiempo.

Definición 1.5 *Redundancia espacial: Un programa que no es redundante en espacio debería recorrer, eventualmente, todos los estados considerando todas las posibles ejecuciones (sin fallas).*

Definición 1.6 *Redundancia temporal: Un programa que no es redundante en tiempo debería ejecutar, eventualmente, toda acción posible, considerando todas las ejecuciones (sin fallas).*

Otra observación importante es que resulta inviable diseñar un sistema tolerante a fallas cuando el mismo está sujeto a la ocurrencia de fallas "bizantinas", entendiendo éstas como fallas totalmente arbitrarias dentro del sistema lógico. Debemos poder caracterizar las fallas, por lo menos, desde la perspectiva de la relación con los otros subsistemas o componentes. Por ejemplo, si un nodo de la red no está disponible, ya sea porque ocurrió un problema de alimentación energética o porque se perdió la conexión de red con el mismo, desde la perspectiva de los otros nodos la falla es que el nodo ya no está activo. En este punto juega un rol central la modularización en la especificación y el diseño del sistema, y así poder describir las interacciones de manera más precisa.

Por su parte, Jalote [Jal94] identifica las siguientes fases para abordar la tolerancia a fallas: detección de errores, confinamiento de daños, recuperación de errores y tratamiento de fallas y continuidad del servicio del sistema. La *detección de errores* es la fase en la cual la presencia de una falla se deduce detectando un error en el estado de algunos subsistemas. Cualquier daño causado por un fallo tiene que ser identificado y delimitado en la fase de confinamiento de daño. Frecuentemente, el diseño del sistema tiene que incorporar mecanismos que ayuden a limitar la difusión de errores en el sistema, y así confinar el daño a límites predefinidos. Dado que hay un error en el estado del sistema, en la fase de recuperación de errores es necesario eliminar el error para que no se propague en acciones futuras. Con la *recuperación de errores*, el sistema alcanzará un estado libre de errores. En la fase final de *tratamiendo de fallas y continuidad del servicio del sistema*, la falla o el componente fallido deben ser identificados y el sistema tolerante a fallas debe funcionar de manera tal que el componente fallido no se utilice, sea usado de otra manera o con otra configuración, para que la falla no genere fallos nuevamente [7].

De manera similar, Nelson [Nel90] menciona los siguientes elementos para diseñar sistemas tolerantes a fallas:

Ocultamiento o Enmascaramiento (Masking): corrección dinámica de los errores generados.

Detección (Detection): Detección de un error (síntoma de una falla).

Contención (Containment): Evitar que un error se propague fuera de los límites definidos.

Diagnóstico (Diagnosis): Identificación del módulo fallido responsable del error detectado.

Recuperación (Recovery): Corrección del sistema a un estado aceptable para seguir operando.

[7]Traducción libre del texto referenciado

Intuitivamente estas clasificaciones parecen razonables. De todas maneras, en la bibliografía informal usualmente se abordan como procedimientos *ad hoc* que se agregan en la etapa de implementación del sistema y muchas veces no forman parte integral del diseño del mismo. En nuestro abordaje formal no consideraremos esta distinción como elementos esenciales, sino como fuente de ideas para incorporar nuevas estrategias que nos permitan garantizar ciertas propiedades en los sistemas, y elementos en lenguajes de especificación que nos permitan expresar mejor estas cuestiones.

Para explicar esta distinción, tomemos como ejemplo el caso de la memoria en sistemas espaciales. Un fenómeno conocido es el efecto que pueden producir los rayos cósmicos en el valor registrado en la memoria que puede alterar el valor almacenado, pero sin dañar la misma. Identificaremos como falla el evento que causa el cambio en la memoria, y como error el estado resultante del mismo. El fallo (o failure) ocurrirá cuando utilicemos la información luego de la lectura de la celda. Ahora bien, una de las soluciones más frecuentes para este problema es la triple redundancia que, con un mecanismo de votación, permite identificar (si no hubiera más de una falla) la información correcta. Para evitar que el estado de error permanezca mucho tiempo, se implementa (a veces integrado en el mismo sistema de hardware) un sistema de refresco que actualiza el valor de las celdas con el valor de mayoría. En este diseño no están identificadas cada una de las fases mencionadas, como por ejemplo la detección y el confinamiento de daños, sino que es el diseño integrado como un todo lo que garantiza ciertas propiedades asimilables con la tolerancia a fallas.

La pregunta que nos hacemos entonces es ¿qué representan estos conceptos desde una perspectiva formal y cómo caracterizarlos? Discutiremos aquí algunos elementos.

Los mecanismos de *detección* necesitan nuevos estados o acciones de programas. Como estos mecanismos no son utilizados en ejecuciones libres de fallas forman parte de la redundancia del sistema [Gär99]. Para Arora y Kulkarni [AK98] un *detector* es un componente del sistema que "detecta" si un predicado del sistema se satisface en un estado dado del sistema. Detectores conocidos son comparadores, códigos de detección de errores, snoopers, programas watchdog, alarmas, procedimientos de snapshot, tests de aceptación y condiciones de excepción. Estos elementos se deben incorporar como parte constitutiva del sistema, aunque sean considerados como redundancia.

Un error puede ser detectado como consecuencia de la ocurrencia de un evento no deseado del sistema (o no esperado en situaciones sin la ocurrencia de fallas), como por ejemplo un evento de timeout producido porque no llegó a tiempo la confirmación de un mensaje enviado; o a partir de un procedimiento que ejecuta el sistema (o alguno de sus subsistemas) para identificar errores, es decir, estados no deseados del sistema provocados por una falla. Llamaremos a la primera detección pasiva y a la segunda detección activa.

Definición 1.7 *Dado un sistema, llamamos* ***detección pasiva*** *de una falla cuando la ocurrencia de la misma desencadena la ocurrencia de un evento determinado que nos permite detectarla.*

Definición 1.8 *Dado un sistema, llamamos* ***detección activa*** *de una falla cuando el sistema cuenta con procesos específicos que ejecuta el sistema o algún subsistema del mismo que permite identificar errores, es decir, estados no deseados del sistema provocados por una falla.*

El **masking**, que podríamos traducir como "enmascaramiento", se podría entender cuando es imposible distinguir el comportamiento normal de aquel que se dio debido a la ocurrencia de una falla. Por ejemplo, cuando se utiliza un código de corrección de errores, como por ejemplo un código de Hamming, el mensaje que llega luego de su decodificación es indistinguible de aquel que se transmitió con una falla. Por otro lado, Arora y Kulkarni [AK98] definen *masking* en relación a una falla dada cuando la ocurrencia de la falla no genera ningún comportamiento indeseado del sistema.

Esta última acepción nos ayuda a distinguir dos definiciones posibles de tolerancia a fallas: aquella tolerancia que enmascara las fallas, y aquella que puede llegar a tener una baja de performance y/o funcionalidad dentro de los límites aceptables, y eventualmente se recupera.

Definición 1.9 *Decimos que el sistema* ***oculta o enmascara****, masking, una falla si el comportamiento del sistema a pesar de la ocurrencia de la misma, mantiene el mismo comportamiento que en caso de no ocurrir la falla.*

Definición 1.10 *Decimos que el sistema* ***no oculta o no enmascara****, non-masking, una falla si el comportamiento del sistema cambia, luego de la ocurrencia de la misma.*

Estas definiciones no pretenden ser precisas sino introductorias. Es necesario analizar con mayor profundidad a qué nos referimos con comportamiento del sistema. Por otro lado, nos pueden ayudar a definir o caracterizar mejor la tolerancia a fallas.

Estas características podrían pensarse como propiedades de *safety* para el modelo de sistemas tolerantes a falla: el sistema garantiza que se cumple determinada propiedad, incluso ante la ocurrencia de fallas. Se pueden distinguir dos propiedades diferentes en relación al funcionamiento óptimo del sistema y al funcionamiento mínimo requerido. Las fallas están ocultas si el sistema garantiza el funcionamiento óptimo, incluso en presencia de fallas, y decimos que las fallas se manifiestan, si trasitoriamente no se cumple con la propiedad requerida, aunque se sigue entendiendo que el sistema es tolerante a fallas si puede garantizar los requerimientos mínimos de funcionamiento.

La *contención* o confinamiento es cuando el sistema evita la propagación del error fuera de cierta "área" de contención al resto del sistema. Para definir este

concepto más precisamente tendremos que describir mejor el estado de error y el "área de confinamiento", y en particular será necesario contar con una descripción modularizada del sistema. En todo caso nos interesará definir si un determinado error está contenido dentro de ciertos límites, donde esos límites podrán ser, por ejemplo una componente determinada, o alguna caracterización sobre el conjunto de estados. En relación a este concepto nos interesaría poder describir y verificar si una falla está contenida.

Definición 1.11 *Dado un sistema con tolerancia a fallas, decimos que un error está* ***contenido*** *en una componente si no afecta el comportamiento de otras componentes del sistema o el entorno.*

En relación a la *recuperación*, puede referirse a mecanismos de recuperación que devuelven el sistema a un estado correcto anterior o a un punto de recuperación predeterminado. Por ejemplo, reiniciar la componente a un estado inicial correcto, siempre y cuando eso sea una alternativa válida.

La característica de recuperación también puede estar asociada con propiedades de *liveness*, es decir, aunque esté en un estado malo, alguna cosa buena ocurrirá en el futuro y se volverá a garantizar un comportamiento bueno. Magee y Maibaum en [MM06] presentan como ejemplo un modelo con estados OK, BAD y REALLY BAD. Si el sistema se encuentra en un estado REALLY BAD y nada malo ocurre, se puede llegar a un estado BAD y luego a uno OK. Un problema que surge de este tipo de propiedades es que, aunque se cumplan, no es posible determinar cuándo se cumplirán.

Definición 1.12 *Decimos que el sistema se* ***recupera*** *si vuelve a un estado sin errores.*

1.4. Reflexiones sobre el modelado de sistemas

Contar con modelos apropiados para la descripción de los sistemas es central en el abordaje de la tolerancia a fallas. En esta sección planteamos algunas reflexiones que nos servirán de guía para esta tesis.

1.4.1. Sistemas reactivos vs. sistemas transformacionales

Es importante hacer un paréntesis en los aspectos de la tolerancia a fallas, para analizar el tipo de sistema, o más bien la perspectiva de análisis de los sistemas, en los que hacemos foco cuando hablamos de tolerancia a fallas.

Entendemos que la tolerancia a fallas no es una dimensión de análisis y/o diseño interesante en el dominio de los sistemas transformacionales. Para estos sistemas no interesan centralmente los estados intermedios, sino los estados iniciales y finales, y su relación. Los estados intermedios pueden ser útiles para la verificación y análisis de los mismos, pero no en términos de

su comportamiento y relación con el entorno. Para analizar la corrección de este tipo de sistemas, existen múltiples herramientas formales. En este sentido, considerar la ocurrencia de fallas asumiendo posibles errores internos de programación, es decir, errores al momento del diseño y la implementación del sistema, refleja una mala política de diseño.

Los sistemas reactivos, en cambio, son sistemas que interactúan contínuamente con su entorno, es decir, reciben estímulos del entorno y actúan sobre él. En estos sistemas, la tolerancia a fallas -o más precisamente esa dimensión de análisis- podría jugar un rol importante a la hora de construir sistemas confiables y con ciertas garantías de funcionamiento, incluso con la presencia de fallas en el hardware u otros sistemas del entorno.

Podríamos suponer que todos estos sistemas se integran con componentes (partes) transformacionales e interacciones con el entorno. De todas maneras, un modelo apropiado que se ajusta a nuestro interés de análisis permitiría abstraernos de los procedimientos internos transformacionales a través de la representación de un cambio de estado interno y sólo enfocarnos en las interacciones con el hardware y otros sistemas y subsistemas.

Es importante observar que la tolerancia a fallas, como metodología de construcción de un sistema que interactúa con su entorno susceptible a fallas, permitiría considerar casos de errores de diseño o implementación en módulos externos considerados como entorno. De todas maneras, es importante tener una correcta caracterización de las fallas, para lograr buenos resultados.

1.4.2. Modelos orientados a estados y orientados a eventos

A la hora de implementar una herramienta que pueda ser utilizada en términos prácticos para la verificación de propiedades debemos contar, además de la lógica para la descripción de éstas, con un lenguaje para describir los sistemas. Dentro de las herramientas conocidas podemos encontrar a lenguajes como `spin` (o más precisamente Promela, el lenguaje asociado al model checker) y `SMV`, que enfocan la descripción a la especificación de los estados y sus transiciones dadas en términos generales de acuerdo a los cambios de los valores de las variables. Por otro lado existe una gran variedad de lenguajes que enfocan la descripción en las transiciones, como por ejemplo FSP. En este sentido, podemos diferenciar dos tipos de modelos, aquellos "orientados a eventos" y aquellos "orientados a estados". En el primer caso, lo que cobra relevancia en la descripción del sistema, son los eventos o transiciones posibles, como es el caso de los LTS. En el segundo caso, cobra centralidad la descripción de los estados y sus propiedades, como las estructuras de Kripke. En el capítulo 2 nos enfocamos en modelos orientados a eventos, en el capítulo 3 nos enfocamos en modelos modelos orientados a estados. Finalmente, en el capítulo 4 integraremos ambas.

1.4.3. Prescripción vs. descripción

El concepto de modelado de sistema está asociado a una descripción detallada (aunque más abstracta) del artefacto real y su entorno. Con el modelo nos abstraemos de las cuestiones más concretas de la implementación, y nos podemos enfocar en el comportamiento general y sus propiedades. Al momento de diseñar un sistema lo que queremos hacer es *prescribir* su comportamiento, en el sentido de indicar cómo debe comportarse. En contraposición, el entorno u otros sistemas con los que se interactúa, son elementos sobre los que usualmente no tenemos control. De todas maneras, para contar con una descripción del comportamiento global del sistema es necesario *describir* estos últimos de la manera más precisa posible. Usamos descripción en este caso en referencia a detallar el comportamiento de algo sobre lo que no tenemos control o posibilidad de cambiar, sino que está predeterminado por la naturaleza o agentes externos. En algunos casos los propios sistemas reactivos que queremos diseñar requieren modelos internos de la naturaleza para su implementación. Por ejemplo una computadora de navegación, que cuenta con sensores de giro y aceleración para el cálculo de la posición, requiere un modelo preciso de la aceleración de gravedad para su funcionamiento.

El modelo de Parnas de 4 variables [Par06] también propone realizar una distinción entre variables controlables y variables observables. Esta distinción puede ser pensada desde la perspectiva que discutimos anteriormente, dado que las variables observables se utilizan en el modelo de Parnas para modelar sensores que interactúan con el ambiente; en contraposición con las variables controlables que son aquellas que pueden afectar el sistema.

1.4.4. Componentes

Magee y Maibaum [MM06] identifican los siguientes elementos para especificar tolerancia a fallas:

- describir el comportamiento de las componentes,
- describir los mecanismos de interacción entre componentes (conectores y coordinación),
- describir cómo los sistemas se configuran en términos de componentes y conectores,
- definir la estructura jerárquica de componentes,
- caracterizar el comportamiento de estados que se consideran anómalos,
- describir actividades de recuperación.

Estos elementos, más allá de estar enunciados desde una perspectiva particular en el diseño de sistemas, nos introduce el concepto de componente, que

se presenta como esencial para modularizar la descripción de los sistemas. Desde nuestra perspectiva nos resulta importante, a su vez, distinguir: componentes, relaciones, modelos de fallas, modelo de entorno y comportamiento deseado (especificación).

La distinción de jerarquías entre componentes y de subsistemas basados principalmente en el principio de ocultamiento de la información, nos permitirá abordar la tolerancia a fallas para sistemas más complejos. Pensemos, por ejemplo, en el modelo de capas de las redes de computadoras. La capa de enlace de datos le “garantiza” a la capa de red que las comunicaciones nodo a nodo se realizan sin errores. La capa de red se encarga entonces de otras tareas.

1.4.5. Modelado de la ocurrencia de fallas

Gärtner [Gär99] afirma que una falla puede ser modelada como una transición no deseada, aunque posible. Arora y Gouda sugieren la incorporación de nuevas variables que amplíen el espacio de estados para simular cualquier tipo de falla. Por ejemplo las ocurrencia de un *crash* puede ser modelado con una variable booleana up (que indica que el sistema está funcionando) que condicione cada comando con guarda.

Al introducir fallas a nuestro modelo, estamos trasformando el modelo, con eventos y estados que pertenecen al contexto y no efectivamente al sistema que estamos modelando o diseñando. Esto nos aleja de la especificación del “artefacto” real que estamos diseñando. Una buena herramienta teórica debería poder distinguir la especificación del sistema de la ocurrencia de fallas y estados de error.

Por otro lado, puede ser deseable describir el comportamiento y la relación entre fallas. Por ejemplo, podemos tener dos fallas posibles en un sistema de comunicaciones, una que represente un crash (el nodo deja de funcionar por completo), debido por ejemplo a algún corte energético, y una falla que, debido al congestionamiento de la línea, represente un funcionamiento con menor performance. Vemos que hay cierta dependencia entre ambas fallas: si está caída la comunicación por un crash, no puede ser el caso de que ocurra la otra falla.

Tener un modelo diferenciado que describa el comportamiento de las fallas puede ser una herramienta interesante a explorar, ya que permitiría

- Modelar comportamientos o relaciones entre fallas.
- Poder tratar cada falla al interior de cada componente como eventos independientes.
- Explorar nuevas variantes de la tolerancia a falla, en relación a comportamientos más definidos de las fallas.

1.5. Conclusiones y notas metodológicas

En este capítulo hicimos un recorrido informal de conceptos fundamentales de la tolerancia a fallas. Además de capturar la idea de la tolerancia a falla y sus desafíos, recorrimos diferentes clasificaciones y estrategias de la toleracia, y evaluamos su pertinencia desde la perspectiva formal que abordaremos en el resto de la tesis.

A partir del recorrido y las reflexiones de este capítulo remarcamos algunas notas que tendremos en cuenta cuando abordemos algunos ejemplos o temáticas enfocadas a estos aspectos.

En relación a los **sistemas**:

- Diferenciar modelo del Entorno y modelo del Sistema.
- Tanto entorno como sistema podrán ser modelados con múltiples componentes.
- Las componentes podrán tener una relación jerárquica que implica que unas componentes se componen (valga la redundancia) de otras componentes.
- Las componentes podrán tener una relación simétrica, que implicará interacción entre subsistemas. Esa interacción también se dará entre componentes del sistema y del entorno.
- Es deseable poder diferenciar, en cada componente, los elementos[8] observables y/o controlables de aquellos que no lo son.

En relación a las **fallas**:

- Para modelar las fallas es necesario identificarlas.
- Las fallas pueden ser generadas a) por el entorno, b) por componentes de más bajo nivel (teniendo en cuenta la relación jerárquica entre componentes), c) por otras componentes con las que se interactúa d) por la relación con otras componentes.
- las fallas pueden ser modeladas como eventos, pudiendo éstos provocar cambios de estado que cambien el comportamiento de las componentes. Por ejemplo la caída de un canal de comunicación, provoca que ningún mensaje llegue a destino.
- Es posible modelar interrelaciones de fallas con componentes adicionales del entorno que las describan.

[8]Con elemento nos referimos a variables, estados, acciones o determinados eventos, según el formalismo que estemos utilizando para modelar el sistema y el entorno.

... so reduce the use of the brain
and calculate!

EWD 1036-0.

2
Sistemas Reactivos para el Modelado de la Tolerancia a Fallas

Presentaremos aquí un primer enfoque para capturar sistemas tolerantes a fallas. Éste nos servirá como un primer abordaje formal a los conceptos presentados en el capítulo anterior. En este enfoque se distingue explícitamente la descripción del sistema y la descripción del comportamiento asociado a una falla. El comportamiento asociado a la falla, considerado comportamiento de entorno, se superpone sobre el comportamiento del sistema. Esto prohíbe que en la descripción del sistema pueda hacerse uso de eventos que "comuniquen" la ocurrencia de una falla al software, lo cual resulta poco realista. Por otro lado, esta formalización provee una noción más precisa de tolerancia a fallas, la cual es dependiente de cada falla, y requiere una especificación de los requerimientos del sistema.

Para la formalización, proponemos una variante de los *Modelos Reactivos de Kripke* diseñados por D. Gabbay [Gab08]. En nuestro caso, utilizamos sistemas de transiciones de estados etiquetadas (LTS, por sus siglas en inglés) como semántica y como vehículo para la especificación de sistemas tolerantes a fallas. Con este formalismo expresamos la superposición del comportamiento asociado a la falla sobre el comportamiento del sistema de una manera natural, que nos permite caracterizar una amplia variedad de tipos de fallas y sus consecuencias. Más precisamente, utilizamos sistemas de transiciones de estados etiquetadas para caracterizar la combinación de la funcionalidad normal del sistema, y los mecanismos que el desarrollador utiliza para lidiar con las fallas. La descripción del comportamiento de entorno asociado a la falla lo expresaremos con las características dinámicas de los LTS reactivos (una variante de los definidos por Gabbay).

Los principales aportes de este capítulo son:

- Introducir una definición formal inicial de conceptos de tolerancia a fallas.
- Desarrollar una variante de los modelos reactivos de Kripke para describir sistemas.
- Introducir un enfoque metodológico para la descripción y modelado de sistemas tolerantes a fallas, consistente con los criterios delineados en el capítulo anterior.
- Presentar algunos ejemplos simples de modelos con fallas y tolerancia a fallas, que ayuden a introducir conceptos.

2.1. Especificación

En los procesos de construcción de software, se denomina **especificación** a una descripción del comportamiento deseado del sistema. La especificación de un sistema de software captura los requisitos asociados al mismo, en general desde una perspectiva más abstracta que la implementación del sistema. En nuestro caso, la especificación de un sistema será *formal*, en el sentido de que tendrá una semántica precisa. La especificación del comportamiento esperado de un sistema es esencial para hablar de tolerancia a fallas: un sistema será tolerante a fallas si ante la presencia u ocurrencia de fallas, el sistema continúa cumpliendo con la especificación del comportamiento esperado del sistema (ver definición 1.4).

Si bien la especificación de un sistema de software se describe usualmente de forma declarativa (por ejemplo, a través de una narrativa en lenguaje natural, o una fórmula lógica en algún formalismo), en el contexto de enfoques formales a la construcción de sistemas concurrentes o sistemas distribuidos, es usual utilizar *LTS* para capturar tanto la especificación como el sistema.

Definición 2.1 *Un* sistema de transiciones de estados etiquetadas *o* LTS *(por sus siglas en inglés) es una tupla de la forma $\langle S, \mathcal{E}, \delta, S_0 \rangle$, donde S es un conjunto de estados, $\mathcal{E}$ es un conjunto de eventos (o etiquetas), $\delta \subseteq S \times \mathcal{E} \times S$ es un conjunto de transiciones, y $S_0 \subseteq S$ es un conjunto de estados iniciales.*

Dado un LTS, el hecho de que $(s, e, s') \in \delta$ se denota usualmente con $s \xrightarrow{e} s'$. Los *LTS* suelen describirse mediante grafos de forma similar a la utilizada con otros formalismos como los autómatas finitos.

Definición 2.2 *Dado un LTS $\langle S, \mathcal{E}, \delta, S_0 \rangle$ y un conjunto de estados $A \subseteq S$, definimos $Reach_\delta(A) = \bigcup_{i \geq 0} A_i$, donde:*

- $A_0 = A$

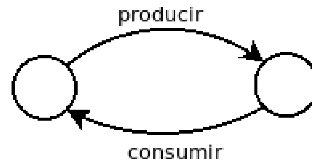


Figura 2.1: Especificación mediante LTS de *Productor-Consumidor*.

- $A_{i+1} = \{s' \in S \mid \exists s \in A_i \cdot \exists e \in \mathcal{E} \cdot (s, e, s') \in \delta)\}$

Definición 2.3 *Dado un LTS* $L = \langle S, \mathcal{E}, \delta, S_0 \rangle$*, decimos que* $\pi = e_0, e_1, e_2, \ldots$*, con* $e_i \in \mathcal{E}$*, es una* traza *de* L*, si existen* $s_0, s_1, s_2, \ldots$*, con* $s_i \in S$*, tal que* $s_0 \in S_0$ *y* $s_i \xrightarrow{e_i} s_{i+1} \in \delta$*, para todo* i*. Además, para* $k \geq 0$*, decimos que* $\bar{\pi} = e_0, e_1, e_2, \ldots, e_{k-1}$ *es un* prefijo *de* π*, que denotaremos* $\bar{\pi} \prec \pi$*, y* $\vec{\pi} = e_k, e_{k+1}, \ldots$ *es un* sufijo *de* π*, que denotaremos* $\vec{\pi} \succeq \pi$*.*

Denotaremos con $Tr(L)$ al conjunto de todas las trazas de un LTS L.

Definición 2.4 *La* ***especificación*** *de un sistema es un LTS de la forma* $E = \langle S^e, \mathcal{E}^e, \delta^e, S_0^e \rangle$*.*

Notemos que los requisitos del sistema pueden especificarse con una variedad de lenguajes, y es relativamente poco usual tener una descripción de naturaleza operacional como la que proponemos. Sin embargo, utilizamos este formalismo para la representación de requisitos por razones de simplicidad en la presentación de las ideas. Especificaciones en formalismos de más alto nivel pueden llevarse a esta forma de manera directa. Analizaremos algunos lenguajes en los próximos capítulos de esta tesis.

Ejemplo 2.1 *Para hacer más clara la función del* LTS *como especificación de requisitos de sistema, consideremos como ejemplo el modelo* Productor-Consumidor*. Este modelo consiste de dos procesos, un productor y un consumidor. El productor genera mensajes y los transmite por un canal; estos mensajes son luego recibidos por el consumidor. Un requisito usual asociado con este patrón es que todo mensaje "producido" por el productor es eventualmente "consumido" por el consumidor. Podemos especificar una versión más restrictiva de esta propiedad como se muestra gráficamente en la Figura 2.1. Esta versión es más restrictiva porque exige que todo mensaje producido es consumido antes de que se produzca el siguiente mensaje (es decir, que producción y consumo se alternan).*

2.2. Modelo del Sistema

Consideramos al sistema como aquella componente, o conjunto de componentes, que son objeto del desarrollo y análisis. En este sentido, lo distingui-

mos del entorno que incluye no sólo elementos de la naturaleza, sino también el resto de componentes externos con los cuales interactúa, en particular, dispositivos de más bajo nivel que pueden ser objeto de fallas. Recordemos que en nuestro enfoque no consideramos las fallas del propio sistema provocadas por errores en la implementación.

El comportamiento del sistema también será descripto usando un *LTS*. Supondremos entonces que el comportamiento del sistema estará dado por una tupla $Sys = \langle S, \mathcal{E}, \delta, S_0 \rangle$, donde, al igual que para las especificaciones, S es un conjunto de estados, $\mathcal{E}$ es un conjunto de eventos, $\delta \subseteq S \times \mathcal{E} \times S$ es un conjunto de transiciones con etiquetas, y $S_0 \subseteq S$ es el conjunto de estados iniciales.

Definición 2.5 *Un **sistema** (sin fallas internas) es un LTS de la forma $Sys = \langle S, \mathcal{E}, \delta, S_0 \rangle$ en el que se cumple $S = Reach_\delta(S_0)$.*

La condición $S = Reach_\delta(S_0)$ exige que todos los estados del sistema sean alcanzables mediante la aplicación de secuencias finitas de transiciones desde el conjunto de estados iniciales. Esta condición si bien puede parecer exigente, no nos condicionará el modelo, ya que pensaremos el sistema sin fallas como una parte diferenciada del sistema con fallas. Por lo tanto, estados que pueden no ser alcanzables en nuestro diseño original de sistema, serán representados en el sistema con fallas.

Es importante destacar que si bien el formalismo utilizado para especificar tanto requisitos de sistema como el modelo (más concreto) del comportamiento del sistema es el mismo, éstos representan conceptos diferentes. Más precisamente, E resultará ser un modelo abstracto de Sys (bajo alguna definición apropiada de abstracción).

Ejemplo 2.2 *Continuando con el ejemplo anterior, consideremos dos posibles implementaciones de* Productor-Consumidor *que se muestran gráficamente en las Figuras 2.2 y 2.3. En ambos modelos, la comunicación entre productor y consumidor se realiza mediante un canal confiable (es decir, sin pérdida de mensajes), y se han añadido estados y transiciones respecto de la especificación incorporando eventos para representar la comunicación. Si bien la primera puede ser una implementación más simple del sistema, no cumple con la especificación restrictiva que nos propusimos, en la que deben alternarse los eventos de producción y consumición. La implementación de la Figura 2.3, al incorporar un mensaje de confirmación (*`ack`*) fuerza la alternancia de los eventos de producción y consumición.*

Varias nociones de *refinamiento* han sido definidas sobre este tipo de modelos formales. No nos preocuparemos aquí por cuál de ellas es la más adecuada para nuestro propósito; simplemente supondremos contar con una relación de refinamiento $\leq_r$, que nos permita establecer que Sys satisface E cuando $Sys \leq_r E$.

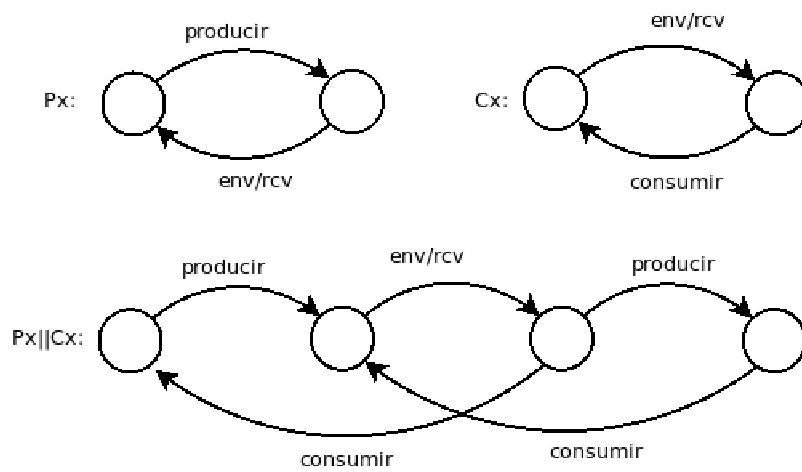


Figura 2.2: Primer intento de implementación de *Productor-Consumidor*.

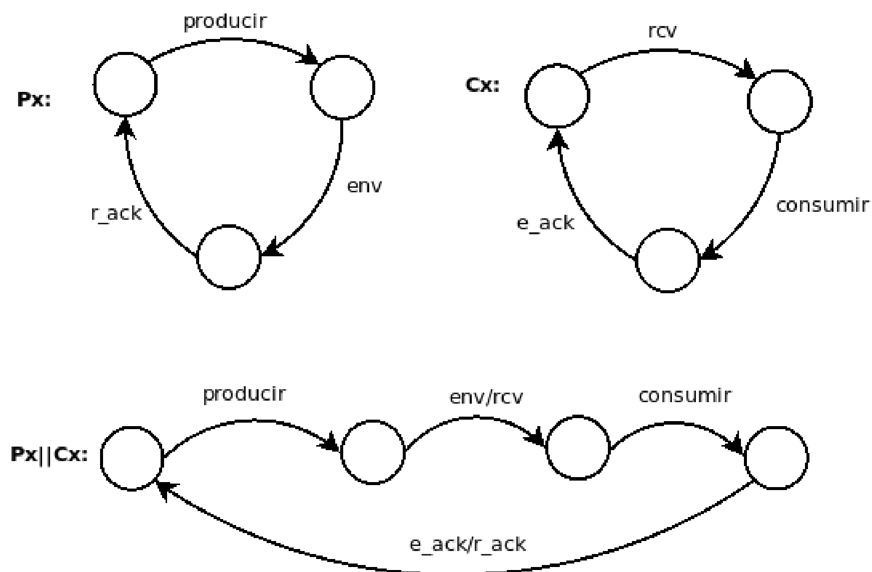


Figura 2.3: Implementación de *Productor-Consumidor*.

2.3. LTS Reactivos

A diferencia de las descripciones del sistema y los requisitos asociados al mismo, el comportamiento asociado al entorno y en particular a las *fallas* será descripto mediante una variante de *modelos reactivos de Kripke* [Gab08] [Gab12]. Los modelos reactivos de Kripke son un formalismo introducido por D. Gabbay, similar a las estructuras de Kripke convencionales. Un modelo reactivo de Kripke, a diferencia de una estructura de Kripke convencional, puede *reconfigurarse* cuando una transición es "disparada", habilitando transiciones que antes estaban deshabilitadas o deshabilitando transiciones habilitadas, cambiando así el comportamiento del sistema, dinámicamente.

Definición 2.6 *Un sistema de transiciones etiquetadas reactivo o **RLTS** se define como una tupla $R = \langle S, \mathcal{E}, \delta, \delta^+, \delta^-, S_0, \delta_0 \rangle$, donde S en un conjunto de estados, $\mathcal{E}$ es un conjunto de eventos, $\delta \subseteq S \times \mathcal{E} \times S$ es un conjunto de transiciones etiquetadas, $\delta^+ \subseteq \delta \times \delta$ y $\delta^- \subseteq \delta \times \delta$ son conjuntos de* meta-transiciones, *$S_0 \subseteq S$ es un conjunto de estados iniciales, y $\delta_0 \subseteq \delta$ es el conjunto de transiciones habilitadas inicialmente.*

Intuitivamente, las transiciones δ^+ y δ^- permiten habilitar y deshabilitar, respectivamente, las transiciones en δ cuando otras transiciones se ejecutan.

La idea principal es que a un sistema Sys representado con un LTS, se le puede incorporar el comportamiento asociado al entorno y las fallas extendiendo Sys con componentes reactivos que darán lugar a un LTS reactivo. Esto quedará más claro en la siguiente sección.

Podemos extender los conceptos de traza, prefijo y sufijo para LTS reactivos de la siguiente manera.

Definición 2.7 *Sea $R = \langle S, \mathcal{E}, \delta, \delta^+, \delta^-, S_0, \delta_0 \rangle$ un LTS reactivo. Decimos que $\pi = e_0, e_1, e_2, ...$ con $e_i \in \mathcal{E}$ es una **traza** de R, si existen $s_0, s_1, s_2, ...$ con $s_i \in S$ tal que $s_0 \in S_0$ y $(s_i, e_i, s_{i+1}) \in \delta$ para todo i, y además se cumple alguna de las siguientes condiciones*

- *$t_i \in \delta_0$ y no existe $j < i$ tal que $(t_j, t_i) \in \delta^-$*
- *existe $j < i$ tal que $(t_j, t_i) \in \delta^+$ y para todo k tal que $j < k < i$ $(t_k, t_i) \notin \delta^-$*

donde $t_i = s_i \xrightarrow{e_i} s_{i+1}$

Denotaremos con $Tr(R)$ al conjunto de todas las trazas de R.

*Para $k \geq 0$, decimos que $\bar{\pi} = e_0, e_1, e_2, ..., e_{k-1}$ es un **prefijo** de π, que denotaremos $\bar{\pi} \prec \pi$, y $\vec{\pi} = e_k, e_{k+1}, ...$ es un **sufijo** de π, que denotaremos $\vec{\pi} \succeq \pi$.*

Las últimas condiciones establecen que la transición debe estar habilitada al momento de ejecutarse.

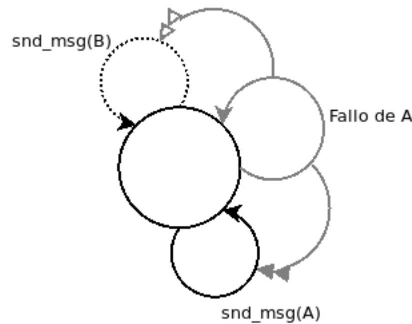


Figura 2.4: Ejemplo gráfico de LTS Reactivo.

Modelos Reactivos de Kripke

Basamos la definición de los LTS Reactivos en los *Modelos Reactivos de Kripke (MRK)* introducidos por D. Gabbay en [Gab08] y luego desarrollados en varios trabajos posteriores en el que se destaca principalmente el libro *Reactive Kripke Semantics* [Gab13], en el cual desarrolla la semántica de una lógica asociada y varias variantes y aplicaciones. Estas estructuras son similares a las estructuras de Kripke usuales con características particulares, entre las que queremos destacar:

- Son una generalización a las estructuras de Kripke.
- No agregan expresividad a las estructuras de Kripke usuales.
- Permiten modelar un sistema con menor cantidad de estados.
- Tiene una lógica asociada con varios resultados y aplicaciones.
- Podría permitir realizar análisis basado en model checking más eficientemente

Gráficamente los LTS reactivos pueden ser representados como grafos. Identificaremos con flechas con líneas de puntos a las transiciones que no están habilitadas inicialmente, y en gris las transiciones asociadas a las fallas. Las flechas con cabezas dobles son meta-transiciones, siendo las de cabezas rellenas (negras) las pertenecientes a δ^- y las de cabezas vacías (blancas) las de δ^+ (las que, al "dispararse", inhabilitan y habilitan otras transiciones, respectivamente). Se puede ver un ejemplo en la Figura 2.4.

2.4. Modelo del Sistema con Fallas

Para representar la integración del sistema con las fallas utilizaremos LTS reactivos.

Definición 2.8 *Dado un sistema sin fallas* $Sys = \langle S, \mathcal{E}, \delta, S_0 \rangle$ *decimos que* $Sys^f = \langle S, S^f, \mathcal{E}, \mathcal{E}^f, \delta, \delta^f, \delta^+, \delta^-, S_0 \rangle$ *es un* ***sistema con fallas*** *sii*

- $\langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0, \delta \rangle$ *es un LTS reactivo.* [1]
- *para todo* $(\delta_1, \delta_2) \in \delta^+ \cup \delta^-$ *existe* $s \in S \uplus S^f$, $s' \in S \uplus S^f$ *y* $e \in \mathcal{E}^f$ *tal que* $\delta_1 = s \xrightarrow{e} s'$.

Utilizaremos la notación $\langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *para referirnos a* Sys^f.

Observar que, S^f y δ^f son los estados y transiciones que, como consecuencia de considerar la ocurrencia de las fallas, son agregados al comportamiento del sistema o el entorno. También se agregan como parte del entorno y la falla las meta-transiciones δ^+ y δ^-. El conjunto de transiciones iniciales habilitadas del LTS Reactivo se corresponden con las transiciones del sistema original δ. La última condición de la definición establece que las meta-transiciones deben ser disparadas únicamente por eventos de falla.

Notemos que la incorporación del comportamiento de la falla *preserva* las transiciones y estados del modelo del sistema Sys, complementando a éste con nuevos estados y transiciones (algunas de las cuales pueden ser meta-transiciones). Es importante resaltar que Sys debe contener tanto el comportamiento propio del sistema, como los mecanismos de detección y tratamiento de fallas que el desarrollador haya pensado; en otras palabras, el comportamiento superpuesto al sistema sólo tiene que ver con el efecto de las fallas y el comportamiento del entorno, no su tratamiento. De todas maneras, como para el modelos del sistema pedíamos que todos los estados sean alcanzables, será en el modelo con fallas en el que aparecerán estados y transiciones previstas por el diseñador para la tolerancia a fallas, pero que no eran alcanzables sin la presencia a fallas.

Ejemplo 2.3 *Continuemos con el modelo* Productor-Consumidor. *Una falla que podríamos incorporar al sistema es la potencial pérdida de mensajes por parte del canal de comunicación. El efecto de la falla sobre el sistema* Productor-Consumidor *puede ser descripto por el LTS reactivo que se muestra en la Figura 2.5. Recordar que graficamos en gris los eventos de falla.*

Técnicamente la falla de pérdida de mensajes en el canal se traduce como dos eventos diferentes: `env/err`*, el evento de pérdida de mensaje del transmisor y* `e_ack/err`*, la pérdida del mensaje de confirmación que realiza el receptor. En este caso, el modelo del sistema con las fallas superpuestas es un LTS convencional (no se agregan meta-transiciones).*

Como vimos en el ejemplo 2.2 el modelo sin fallas satisface la especificación, pero claramente no lo hace si consideramos la ocurrencia de estas fallas. En resumen, el sistema no es tolerante a la falla de pérdida de mensajes, ya que

[1] $\uplus$ representa la unión disjunta

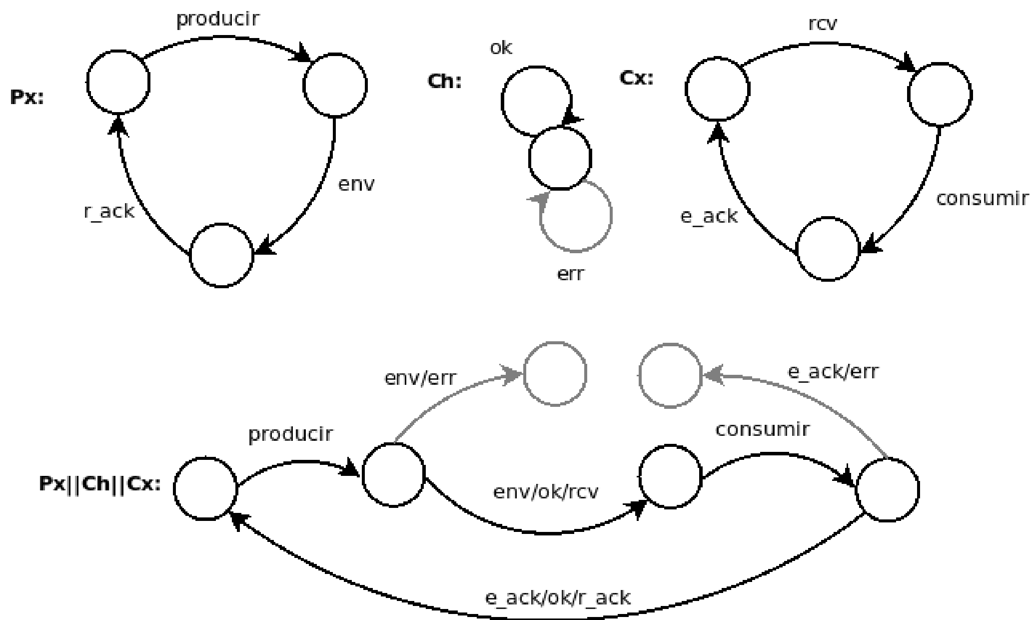


Figura 2.5: Pérdida de mensajes en el modelo *Productor-Consumidor*.

su inclusión en el comportamiento del sistema lleva a una violación de los requisitos del mismo, es decir, a un *fallo* (failure).

Como mencionamos anteriormente, es la implementación del sistema la que califica de tolerante a fallas (específicamente a una falla determinada) o no. La implementación del ejemplo anterior no es tolerante a fallas, pero podemos considerar una implementación con reenvío de mensajes. Por razones de simplicidad en la presentación del ejemplo, consideraremos sólo la falla `env/err`.

Ejemplo 2.4 *En la Figura 2.6 se muestra el problema de productor consumidor que es tolerante a la falla de pérdida de mensajes* `env/err` *en relación a la especificación (Figura 2.1).*

En este caso, la estrategia de tolerancia fue incorporar al sistema un mecanismo de reenvío de mensajes que, al combinarla con el mecanismo de confirmación, condiciona el funcionamiento del sistema para que se ajuste a la especificación. Es interesante preguntarse cuándo se activa la acción de reenvío de mensajes. Este cambio en la implementación, sería una estratégia para la tolerancia a fallas, como las que discutimos en la sección 1.3. La activación de este reenvío, por ejemplo, podría codificarse con el uso de temporizadores. No incorporamos al modelo el evento de `timeout`, pero también se podría incorporar.

Consideremos otro ejemplo de falla superpuesta sobre el sistema de productor-consumidor introduciendo como falla la caída total del canal de comunicación.

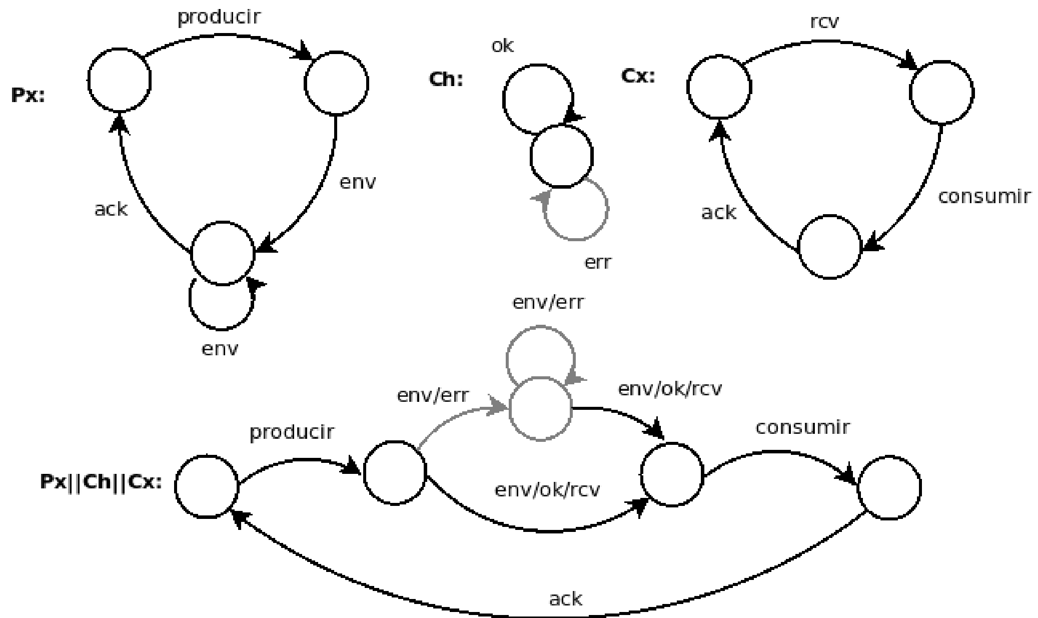


Figura 2.6: Pérdida de mensaje, representado como falla superpuesta al modelo con confirmación (ack) de *Productor-Consumidor*.

Ejemplo 2.5 *En la Figura 2.7 se presenta nuevamente el caso del productor-consumidor en el cual la falla es la caída total del canal de comunicación. El modelo se presenta como un LTS reactivo que extiende al sistema original. En este modelo, el evento asociado a la ocurrencia de la falla es `caida`. La caída del canal de comunicación produce cambios en el comportamiento del sistema y el entorno. En este caso, cualquier mensaje enviado por el productor (`Px`) no llega al consumidor (`Cx`). Ese cambio de comportamiento permanece hasta que el canal se recupera (`recup`). A diferencia del ejemplo anterior en que la falla sólo podía ocurrir al momento del envío, es decir, está asociada a un evento o acción determinada, en este caso la caída del canal de comunicación puede ocurrir en cualquier momento y afectar acciones que se realicen en el futuro. Observar que para simplificar la gráfica, los eventos de falla y recuperación no se graficaron en el LTS integrado. Para ser precisos, los eventos de falla y recuperación deberían graficarse en cada estado del sistema.*

En este ejemplo vemos claramente que la falla cambia el comportamiento del sistema. Esa situación se codifica con las meta-transiciones que se activan con la ocurrencia de la falla. También podemos observar, que la falla nos lleva a un estado de *error*, en el sentido que la falla no se manifestará, hasta tanto el productor no intente enviar un mensaje a través del canal.

La pregunta interesante sería: si el sistema descripto es tolerante a fallas o no. Según la especificación, y si incorporamos algún principio de *fairness*, el

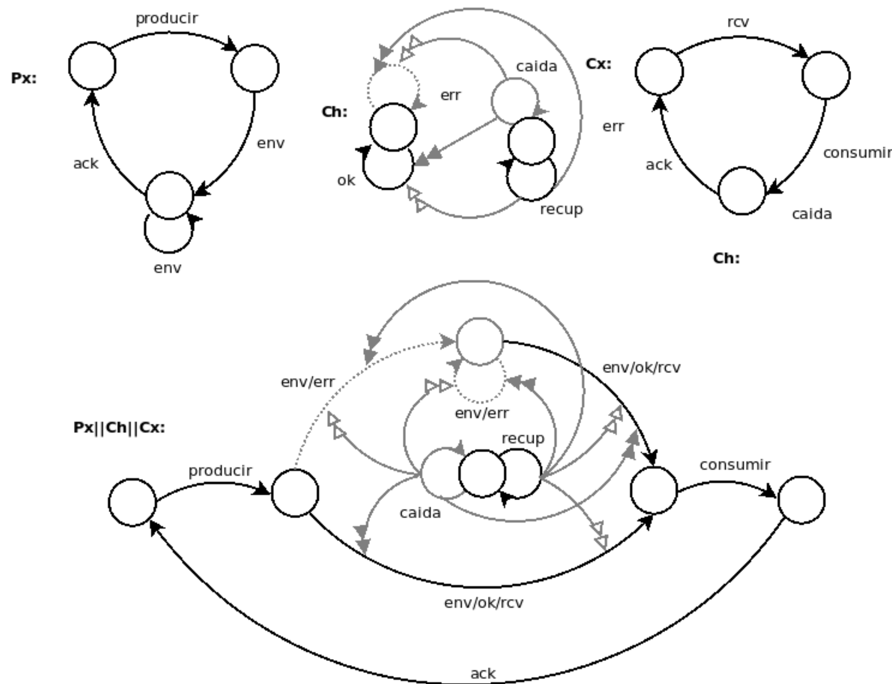


Figura 2.7: Caída del canal de comunicación en el modelo *Productor-Consumidor*.

sistema sería tolerante a fallas. Sin embargo, si pensamos la caída del canal de comunicación como la rotura física de una antena de transmisión, o una placa de red, por ejemplo, modelar que en algún momento ocurrirá el evento de recuperación (`recup`), no sería apropiado si no se toman medidas concretas. En este sentido, aunque la reparación del canal sea responsabilidad humana, deben preverse mecanismos detección y alerta, y protocolos a seguir, que garanticen la recuperación.

Por otro lado, tal vez evaluemos que la especificación es muy laxa al no incluir, por ejemplo, límites temporales para describir mejor el “comportamiento esperado” del sistema. Es responsabilidad del analista/diseñador incorporar los elementos necesarios para describirlo, dado que no sería aceptable, por ejemplo, que el evento de recuperación tome un año en ocurrir.

En esta propuesta, identificamos y modelamos las fallas de manera diferenciada. Al igual que la recuperación, los eventos de falla, pueden depender de otras variables, o tener cierta dependencia entre sí. Esta perspectiva nos permite razonar sobre el comportamiento de éstas de manera independiente del sistema.

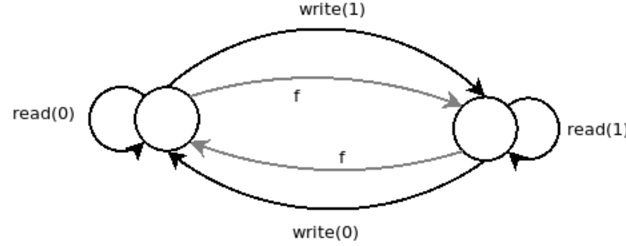


Figura 2.8: Modelo de una celda de memoria con falla `f`

2.5. Definiciones y Propiedades

La formalización de sistemas tolerantes a fallas presentada nos permite realizar algunas tareas importantes. Una de ellas es describir de manera diferenciada las fallas de acuerdo a la forma en que éstas alteran al sistema original. A su vez, nos permite abordar una caracterización de las fallas de manera más formal. Avanzaremos en esta tarea en esta sección.

Una primera observación que podemos realizar sobre las fallas es que éstas pueden tener diferentes tipos de consecuencias sobre el sistema. En nuestro modelo, es evidente que una falla puede provocar alguno de los siguientes sucesos: *(i)* un cambio de estado a un estado normal, *(ii)* un cambio de estado a un estado anormal, *(iii)* habilita y/o inhabilitar transiciones, o *(iv)* una combinación de los anteriores.

Los casos *(ii)* y *(iii)* los podemos observar en los ejemplos 2.4 y 2.5, respectivamente. Para entender el caso *(i)* veamos el siguiente ejemplo:

Ejemplo 2.6 *En la Figura 2.8 se puede observar el modelo de una celda de memoria. Luego de la escritura de un* 0 *(*`write(0)`*), sólo se puede leer* 0 *(*`read(0)`*)*[2]*. Una falla que ocurre en sistemas espaciales es que el valor de la memoria cambie por algún fenómeno de interferencia (aunque sin dañar la misma). Esta falla está identificada en la figura como* `f`*. Es evidente que si la especificación nos indica que debo leer un* 0 *cuando la última operación de escritura fue un* 0*, este sistema no es tolerante a fallas.*

Para poder definir de manera más precisa la tolerancia a fallas, necesitamos algunas definiciones básicas.

Definición 2.9 *Dado un sistema con fallas* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$*, una* ***falla*** f *es un elemento de* $\mathcal{E}^f$ *identificado como tal.*

Podríamos decir que la *ocurrencia de una falla* en este modelo queda caracterizada por la ejecución de una transición $s \xrightarrow{a} s' \in \delta^f$ siendo a una falla.

[2] Análogamente para 1

Definición 2.10 *Dado un sistema con fallas* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$, *una* ***falla no trivial*** *es cualquier evento de* $\mathcal{E}^f$ *que cumple con alguna de las siguientes condiciones*

- *existe* $(s \xrightarrow{f} s', \delta')' \in \delta^-$ *con* $\delta' \in \delta$
- *existe* $(s \xrightarrow{f} s', \delta')' \in \delta^+$ *con* $\delta' \notin \delta$
- *existe* $s \xrightarrow{f} s' \in \delta^f$ *con* $s \neq s'$

Básicamente una falla no trivial es aquella que nos desvía del funcionamiento normal del sistema, ya sea porque nos cambia de estado o porque nos habilita o deshabilita alguna transición de δ que no sería posible sin las ocurrencia de la misma.

Definición 2.11 *Dado un sistema con fallas* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$, *llamaremos* ***estados normales*** *a los elementos de* S, *y* ***estados de error*** *a los elementos de* S^f.

Definición 2.12 *Dado un sistema con fallas* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$, $\vec{\pi}$ *el sufijo de alguna traza* π *de* Sys^f, *decimos que* $\vec{\pi}$ *es un* ***sufijo sin fallas*** *si no contiene eventos de falla y lo denotamos* $\vec{\pi} \succeq_N \pi$. *Si además* $\vec{\pi}$ *es una traza, la llamaremos* ***traza sin fallas*** *o* ***traza normal***.

Denotaremos con $Tr^N(Sys^f)$ *al conjunto de todas las trazas normales de* Sys^f.

Definición 2.13 *Dado un sistema con fallas* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$, $\vec{\pi}$ *el sufijo de alguna traza* π *de* Sys^f, *decimos que* $\vec{\pi}$ *es un* ***sufijo con fallas*** *si contiene fallas entre sus eventos y lo denotamos* $\vec{\pi} \succeq_f \pi$. *Si además* $\vec{\pi}$ *es una traza, la llamaremos* ***traza con fallas*** *o* ***traza anormal***.

Observar que la noción de normal, la consideramos sólo en el caso de una traza completa, dado que un determinado sufijo sin fallas, podría ocurrir luego de la ocurrencia de una falla previa, por lo que los estados y eventos recorridos por el sufijo no serían posibles en una instancia de normalidad, es decir sin la ocurrencia de fallas.

Teniendo en cuenta que el comportamiento de los sistemas está caracterizado por las trazas, podemos definir una noción de **refinamiento** entre la especificación, el sistema y el sistema con fallas. Decimos que el sistema (o el sistema con fallas) refina, implementa o satisface la especificación, si el conjunto de trazas del sistema, restringidas al conjunto de eventos de la especificación, está incluida en el conjunto de trazas de la especificación.

Definición 2.14 *Sea* $\pi = e_0, e_1, ...$ *una secuencia de eventos tal que* $e_i \in \mathcal{E}$ *y* A *un conjunto de eventos. Denotaremos como* $\pi|_A = e_{i_0}, e_{i_i}, e_{i_2}...$ *a la secuencia*

obtenida de eliminar los eventos de la secuencia original que no pertenecen a A. *Diremos que* $\pi|_A$ *se obtiene de* ***restringir*** π *a* A.

Si R *es un LTS reactivo con conjunto de eventos* $\mathcal{E}$, *definimos* $Tr|_A(R) = \{\pi|_A | \pi \in Tr(R)\}$

Definición 2.15 *Dado un LTS* $L = \langle S, \mathcal{E}, \delta, S_0 \rangle$ *y un LTS reactivo* R, *diremos que* R ***satisface*** L *sii* $Tr|_{\mathcal{E}}(R) \subseteq Tr(L)$ *y lo denotaremos* $R \sqsubseteq L$.

Como primera aproximación, diremos que un sistema con fallas Sys^f es *tolerante a fallas*, dada una especificación E del mismo, si $Sys^f \sqsubseteq E$. A esta relación la llamaremos tolerancia a fallas con ocultamiento (enmascaramiento) de fallas (masking fault tolerance).

Definición 2.16 *Dada una especificación* E *y un sistema con fallas* Sys^f, *decimos que* Sys^f *es* ***tolerante a fallas con ocultamiento o enmascaramiento de fallas*** *si y sólo si* $Sys^f \sqsubseteq E$

Esta definición de tolerancia a fallas es la más fuerte, ya que exige que un sistema cumpla con la especificación, a pesar de la ocurrencia de fallas. Una forma de debilitar la definición, es abordando la definición de tolerancia a fallas no enmascarada. Para ello requerimos la descripción del comportamiento de máxima esperado, y el comportamiento de mínima, en el sentido de las propiedades de comportamiento óptimo del sistema, y aquel que no puede dejar de exigirse en todo momento, respectivamente.

Definición 2.17 *Dada dos especificaciones* $E^{op} = \langle S^{op}, \mathcal{E}^{op}, \delta^{op}, S_0^{op} \rangle$ *y* E^{min} *que describan el comportamiento óptimo y el comportamiento mínimo esperado de un sistema, respectivamente; y un sistema con fallas* Sys^f, *decimos que* Sys^f *es* ***tolerante a fallas*** *si y sólo si*

- $Sys^f \sqsubseteq E^{min}$
- $Tr^N|_{\mathcal{E}^{op}}(Sys^f) \subseteq Tr(E^{op})$
- $\forall \pi \in Tr(Sys^f)$ *si existe* $\vec{\pi} \succeq_N \pi$ *entonces existen* $\vec{\pi'} \succeq_N \pi'$ *con* $\pi' \in Tr(E^{op})$ *tal que* $\vec{\pi}|_{\mathcal{E}^{op}} = \vec{\pi'}$

La segunda condición establece que si no hay fallas, se satisface el comportamiento óptimo. La última condición establece que si las fallas dejan de ocurrir, entonces vuelvo a un comportamiento óptimo en algún momento. A esta última característica se le puede llamar *recuperación*.

Duración o persistencia de las fallas En relación a la duración o persistencia de las fallas, podemos identificar fallas instantáneas, transitorias o permanentes. Las instantáneas son aquellas que no producen cambios en el comportamiento del sistema, es decir, no habilitan ni deshabilitan transiciones.

Definición 2.18 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas y* f *una falla. Diremos que* f *es una* ***falla instantánea*** *si para todo* $(s \xrightarrow{e} s', t) \in \delta^+ \cup \delta^-$ *se cumple* $e \neq f$.

En el caso de que la falla no sea instantánea, puede ser transitoria o permanente. Informalmente una falla permanente es aquella que nunca se recupera, o requiere una intervención drástica; por ejemplo, reemplazar el dispositivo de hardware dañado. Una falla transitoria es aquella que tiene asociada un evento de recuperación que retorna al estado normal las transiciones afectadas; por ejemplo, un corte de luz, puede provocar el cese de funcionamiento de un nodo determinado, que podrá recuperarse cuando el servicio se habilite nuevamente.

Definición 2.19 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas y* f *una falla. Diremos que* f *es una* ***falla transitoria*** *si existe* $r \in \mathcal{E}^f$ *tal que si para todo* $(s \xrightarrow{f} s', t) \in \delta^+$ *existe* $w, w' \in S \uplus S^f$ *tal que* $(w \xrightarrow{r} w', t) \in \delta^-$ *y análogamente para todo* $(s \xrightarrow{f} s', t) \in \delta^-$ *existe* $w, w' \in S \uplus S^f$ *tal que* $(w \xrightarrow{r} w', t) \in \delta^+$.

Llamaremos a r ***evento de recuperación****.*

Esta definición de fallas instantáneas es un primer intento de formalización, pero no necesariamente refleja lo que queremos definir, dado que no garantiza que en algún momento ocurra el evento de recuperación r. Se nos hace evidente la necesidad de poder describir mejor le comportamiento de las fallas, por ejemplo caracterizando las trazas posibles, que permitan describir correctamente la transitoriedad de las trazas.

Definición 2.20 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas y* f *una falla. Diremos que* f *es una* ***falla permanente*** *si existe* $(s \xrightarrow{f} s', t) \in \delta^+$ *y no existe* t' *tal que* $(t', t) \in \delta^-$ *o, análogamente, existe* $(s \xrightarrow{f} s', t) \in \delta^-$ *y no existe* t' *tal que* $(t', t) \in \delta^+$.

Es decir, si una falla habilita o deshabilita transiciones, ésta es permanente, si no hay ningún evento que las vuelva a habilitar o deshabilitar, respectivamente.

Autonomía de fallas En relación al momento de ocurrencia de las fallas, podemos clasificar a las fallas como ligadas, en el caso de que dependan del estado del sistema o alguna acción en particular, o como autónomas, en el caso de que puedan ocurrir en cualquier momento (estado del sistema). El primer caso se puede ejemplificar con la pérdida de mensaje en un canal de comunicación. Esta falla puede ocurrir sólo cuando enviamos un mensaje. Por otro lado, una falla autónoma podría ser la caída del canal de comunicación, ya

que puede ocurrir en cualquier momento, independientemente de si estamos utilizando el canal o no.

Definición 2.21 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas y* f *una falla. Diremos que* f *es una* ***falla autónoma*** *si para todo* $s \in S$ *existe* $s' \in S \uplus S^f$ *tal que* $s \xrightarrow{f} s' \in \delta^f$*.*

Definición 2.22 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas y* f *una falla. Diremos que* f *es una* ***falla ligada*** *si no es autónoma.*

La *falla ligada* puede darse como consecuencia del comportamiento indeseado de una acción determinada que pretende ejecutar alguna componente, como el ejemplo discutido previamente de la pérdida de mensaje en un canal de comunicación.

Estado de las fallas Para caracterizar el estado de la falla, deberíamos contemplar trazas. En el caso de las fallas transitorias o permanentes, podríamos clasificarlas en activas o inactivas, en los intervalos de las trazas en que están habilitados o deshabilitados los efectos de la falla. La noción de latente sería mientras nos mantengamos en un estado normal del sistema.

Como esta definición está más asociada a propiedades de los estados, no es muy natural definir estos conceptos con este formalismo. De todas maneras tratamos de abordarlo con la siguiente definición:

Definición 2.23 *Sea* $Sys^f = \langle S \uplus S^f, \mathcal{E} \uplus \mathcal{E}^f, \delta \uplus \delta^f, \delta^+, \delta^-, S_0 \rangle$ *un sistema con fallas,* f *una falla transitoria,* r *su evento de recuperación y* $\pi = e_0, e_1, e_2, ...$ *una traza de* Sys^f*. Tomemos* $\pi_{ij} = e_i, e_{i+1}, ..., e_{j-1}, e_j$ *una subtraza de* π*. Diremos que*

a) f *está* ***inactiva*** *en* π_{ij} *si para todo* $k \leq j$ *tal que* $e_k = f$ *existe* m *tal que* $k < m < i$ *y* $e_m = r$*.*

b) f *está* ***activa*** *en* π_{ij} *si existe* $k < i$ *tal que* $e_k = f$ *y para todo* m *tal que* $k < m \leq j$ *se cumple* $e_m \neq r$

El concepto de latente cuando una falla está activa es más difícil de capturar, porque se relaciona con el comportamiento del sistema, en particular con los estados.

Esta clasificación también se aplica a las fallas permanentes.

Otras clasificaciones de fallas La caracterización en relación a *márgenes del sistema* identificando fallas internas y externas no la podemos hacer con nuestra caracterización, al no identificar diferentes componentes. Tampoco podremos diferenciar las fallas de semántica o de protocolo por la misma razón.

Sin embargo, ampliando este formalismo podríamos modelar cierta dependencia de fallas, que permitiría representar fallas múltiples, o incluso refinar el

concepto de tolerancia a fallas. Para ello introduciremos el concepto de modelo de fallas.

2.6. Modelo de fallas

En muchos casos, pedir fairness en la ocurrencia de fallas nos llevaría a una situación poco real. Por otro lado nos gustaría poder describir algunas relaciones entre la ocurrencia de las fallas, como por ejemplo, dependencia o causalidad. Otra característica que nos interesaría es contemplar diferentes niveles de tolerancia a fallas, por ejemplo teniendo en cuenta las fallas permanentes (rotura de un dispositivo de hardware). Decir que un sistema es tolerante a una o dos fallas de ese tipo nos aportaría más información que decir simplemente que no es tolerante a fallas. Así como en el formalismo trabajado utilizamos el concepto de especificación para modelar el comportamiento esperado del sistema, utilizaremos LTSs para describir el modelo de fallas.

Definición 2.24 *Un* ***modelo de fallas*** MF *lo representaremos con un LTS* $\langle S^{mf}, \mathcal{E}^{mf}, \delta^{mf}, S_0^{mf} \rangle$.

El modelo de fallas también debería incluir los eventos de recuperación y otros eventos del sistema que permitan describir su comportamiento.

A partir de este concepto, podemos definir la tolerancia a fallas en relación al modelo de fallas específico.

Definición 2.25 *Dada dos especificaciones* $E^{op} = \langle S^{op}, \mathcal{E}^{op}, \delta^{op}, S_0^{op} \rangle$ *y* E^{min} *que describan el comportamiento óptimo y el comportamiento mínimo esperado de un sistema, respectivamente; un modelo de fallas* $MF = \langle S^{mf}, \mathcal{E}^{mf}, \delta^{mf}, S_0^{mf} \rangle$ *y un sistema con fallas* Sys^f*, decimos que* Sys^f *es* MF*-**tolerante** si y sólo si para todo* $\pi \in Tr(Sys^f)$ *tal que* $\pi|_{\mathcal{E}^{mf}} \in Tr(MF)$ *se cumple*

- $\pi|_{\mathcal{E}^{min}} \in Tr(E^{min})$
- *si* $\pi \in Tr^N(Sys^f)$ *entonces* $\pi \in Tr(E^{op})$
- *si existe* $\vec{\pi} \succeq_N \pi$ *entonces que existen* $\vec{\pi'} \succeq_N \pi'$ *con* $\pi' \in Tr(E^{op})$ *tal que* $\vec{\pi}|_{\mathcal{E}^{op}} = \vec{\pi'}$

La nueva definición difiere de la anterior en el sentido que sólo observamos las trazas del sistema que cumplen con el modelo de fallas.

Observar que si tomamos $E^{op} = E^{min}$, podemos definir MF**-tolerante con ocultamiento o enmascaramiento**.

Ejemplo 2.7 *Tomemos* $MF1$ *y* $MF2$*, modelos de falla, como se muestran en la Figura 2.9, en la que modelamos fallas permanentes* f*. Con estos modelos podemos expresar que un sistema con fallas tolera una falla si es* $MF1$*-tolerante, pero no tolera dos fallas si no es* $MF2$*-tolerante.*

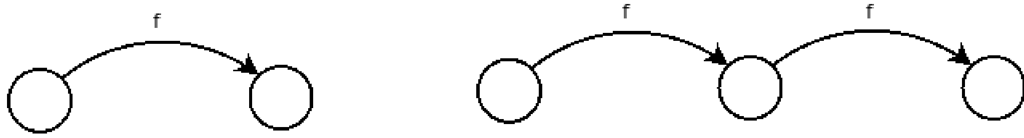


Figura 2.9: Modelo de fallas $MF1$ (izquierda) y $MF2$ (derecha).

2.7. Consideraciones finales

En este capítulo abordamos una primera definición de los principales conceptos de la tolerancia a fallas y algunas clasificaciones. El abordaje formal permitió analizar con mayor detalle estos conceptos, e introducir otros que pueden colaborar a un mejor entendimiento del tema. Sin embargo encontramos varias limitaciones que resumimos en los siguientes puntos:

- La particularidad de los LTS reactivos, no aportan demasiados elementos diferenciales en relación a los LTS estándar.
- Es necesario trabajar con lenguajes que permitan describir las especificaciones y propiedades de manera más simple, y, por otro lado, el comportamiento de las fallas y su interrelación con los sistemas.
- Debemos contar con elementos que permitan describir cada componente y su interrelación. Para dar un ejemplo, cuando queríamos describir las diferentes componentes en la presentación de nuestros ejemplos, las presentamos por separado para agegar claridad.
- La distinción del comportamiento de fallas nos puede permitir afinar los conceptos de tolerancia a fallas; por ejemplo, incorporando cantidad de fallas, relación entre las fallas, tiempos de recuperación, etc.

We must not forget that it is not our [computing scientists] business to make programs, it is our business to design classes of computations that will display a desired behaviour.
EWD340 The Humble Programmer

3

fCTL, una nueva lógica basada en estados

La exploración de diferentes lenguajes de especificación nos permite evaluar beneficios y limitaciones para la tarea de especificación y diseño de sistemas tolerantes a fallas. En el capítulo 2, formalizamos los conceptos de tolerancia a fallas con modelos basados en transiciones, utilizando LTSs y LTS reactivos. En este capítulo introducimos dCTL y fCTL, lógicas basadas en estados utilizando variantes de estructuras de Kripke para la descripción de sistemas con fallas. fCTL surge a partir de revisar nuestro trabajo sobre dCTL [CKAA11] y proponiendo una nueva representación para el modelo de fallas.

Este capítulo refleja parte de nuestro trabajo realizado con lógicas deónticas para especificar sistemas tolerantes a fallas. Como explicaremos en el transcurso del capítulo, a partir del análisis de los casos de estudio desarrollados para la lógica dCTL, elaboramos una nueva lógica, fCTL, para capturar otras propiedades de la tolerancia a fallas.

3.1. Antecedentes

En esta sección introducimos las definiciones de Estructuras de Kripke y Estructuras de Kripke coloreadas, y presentamos las lógicas CTL y dCTL que nos sirven como antecedente y motivación para presentar fCTL.

3.1.1. Estructuras de Kripke

Las estructuras de Kripke son utilizadas para interpretar fórmulas modales o temporales, y para caracterizar comportamiento de sistemas reactivos [CGP99] (Ver 1.4.1).

Definición 3.1 *Sea AP un conjunto de proposiciones atómicas. Una* ***Estructura de Kripke*** *sobre AP es una 4-upla $\langle S, S_0, R, I\rangle$, donde S es un conjunto de elementos llamados* estados, *$S_0 \subseteq S$ es un conjunto de* estados iniciales, *$R \subseteq S \times S$ es una relación de* transición *entre estados, y $I : S \to 2^{AP}$ es una función de interpretación, que indica el conjunto de proposiciones que se cumplen en cada estado.*

Dada una estructura de Kripke $M = \langle S, S_0, R, I\rangle$, normalmente la interpretación de operadores lógicos modales se realiza haciendo referencia a I y la estructura de R. Para operadores temporales, usualmente es necesario emplear la noción de *traza* para definir su semántica.

Definición 3.2 *Dada una estructura de Kripke $M = \langle S, S_0, R, I\rangle$, una* ***traza*** *es una secuencia de estados maximal, con estados adyacentes en relación a R. Una* ***ejecución*** *de M es una traza que comienza en un estado inicial, y una* ***ejecución parcial*** *es una una secuencia no maximal de estados adyacentes, que comienza en un estado inicial. Dada una traza $\sigma = s_0, s_1, s_2, s_3, \ldots$, denotaremos con $\sigma[i]$ al estado i-ésimo de σ, con $\sigma[i..]$ al segmento final de σ a partir de la posición i, y con $\sigma[i, i+1]$ a la transición $(\sigma[i], \sigma[i+1])$. Finalmente, denotaremos con $Tr(M)$ al conjunto de las trazas de M.*

Sin pérdida de generalidad, se puede asumir que cada estado tiene un sucesor, como se acostumbra para varias lógicas temporales [BK08].

3.1.2. CTL (Computation Tree Logic)

CTL, por sus siglas en inglés Computation Tree Logic, es una lógica temporal de ramificaciones por tiempo, con importantes aplicaciones en model checking [CES86]. Esta lógica permite describir propiedades sobre una estructura de Kripke complementando conectores proposicionales con cuantificadores sobre las ejecuciones y operadores temporales combinados de una cierta manera. Es una lógica de árboles de ejecución (computation trees) porque permite expresar propiedades que refieren a los árboles construidos a partir de una estructura de Kripke, comenzando por un estado inicial como raíz y desdoblando la estructura, a través de la relación entre estados, en una forma de un árbol de estados, normalmente infinito.

Definición 3.3 *Sea AP un conjunto de proposiciones atómicas $\{p_0, p_1, \ldots\}$; el conjunto Φ de fórmulas bien formadas de CTL se define recursivamente como:*

$$\Phi ::= \top \mid p_i \mid \neg\Phi \mid \Phi \to \Phi \mid \mathsf{EX}\Phi \mid \mathsf{AX}\Phi \mid \mathsf{E}(\Phi\ \mathcal{U}\ \Phi) \mid \mathsf{A}(\Phi\ \mathcal{U}\ \Phi)$$

Definición 3.4 *Dada una estructura de Kripke $M = \langle S, S_0, R, I\rangle$, y un estado $s \in S$, la semántica de una fórmula CTL se define de la siguiente manera.*

- $M, s \models \top$

- $M, s \models p_i \Leftrightarrow p_i \in I(s)$, *donde* $p_i \in AP$.
- $M, s \models \neg\varphi \Leftrightarrow$ *no* $M, s \models \varphi$.
- $M, s \models \varphi \rightarrow \varphi' \Leftrightarrow (M, s \models \neg\varphi)$ *o* $(M, s \models \varphi')$.
- $M, s \models \mathsf{EX}\varphi \Leftrightarrow$ *para algunas trazas* σ *tal que* $\sigma[0] = s$, $M, \sigma[1] \models \varphi$.
- $M, s \models \mathsf{AX}\varphi \Leftrightarrow$ *para todas las trazas* σ *tal que* $\sigma[0] = s$, $M, \sigma[1] \models \varphi$.
- $M, s \models \mathsf{E}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ *para algunas trazas* σ *tal que* $\sigma[0] = s$, *existe* $j \geq 0$ *tal que* $M, \sigma[j] \models \varphi'$, *y para todo* k *con* $0 \leq k < j$, $M, \sigma[k] \models \varphi$.
- $M, s \models \mathsf{A}(\varphi\, \mathcal{U}\, \varphi') \Leftrightarrow$ *para todas las trazas* σ *tal que* $\sigma[0] = s$, *existe* $j \geq 0$ *tal que* $M, \sigma[j] \models \varphi'$, *y para toda* k *con* $0 \leq k < j$ *se cumple* $M, \sigma[k] \models \varphi$.

Algunos model checkers, en particular SMV, utilizan CTL como un lenguaje para expresar propiedades temporales de los sistemas. Para la lógica CTL el problema de model checking es lineal respecto al tamaño del sistema y la fórmula verificada. CTL se diferencia de otras lógicas tales como CTL* (una lógica de árboles de ejecución más expresiva), para la cual el problema de model checking es exponencial en el tamaño de la fórmula verificada [CGP99].

3.1.3. Deontic Computation Tree Logic

dCTL, presentada en [CKAA11], además de los componentes temporales de CTL, incorpora elementos deónticos que permiten diferenciar el comportamiento normal (o normativo, en términos deónticos) del comportamiento de falla o anormal. Esta característica de los sistemas deónticos permite describir varios aspectos de la tolerancia a fallas de manera más natural dentro de la lógica.

dCTL, por sus siglas en inglés *Deontic Computation Tree Logic*, es una lógica cuyas fórmulas expresan propiedades sobre sistemas modelados con estructuras de Kripke Coloreadas. En estas estructuras la distinción entre estados *normales* y *anormales* permite diferenciar comportamientos con fallas de aquellos en los que no ocurren fallas.

Definición 3.5 *Una* ***estructura de Kripke coloreada*** *es una 5-upla* $\langle S, S_0, R, I, \mathcal{N}\rangle$, *donde* $\langle S, S_0, R, I\rangle$ *es una estructura de Kripke y* $\mathcal{N} \subseteq S$ *es un conjunto de estados* normales.

Las trazas sobre una estructura de Kripke coloreada $\langle S, S_0, R, I, \mathcal{N}\rangle$ son las trazas sobre la estructura de Kripke $\langle S, S_0, R, I\rangle$. Las transiciones que llevan de estados normales a estados anormales se pueden interpretar como transiciones de falla, lo cual representa nuestra interpretación de *falla*. Entonces, las ejecuciones normales son aquellas que recorren sólo estados normales y denotaremos con $\mathcal{NT}$ al conjunto de todas las ejecuciones normales.

Asumimos que en cada estructura de Kripke coloreada existe por lo menos un estado inicial normal, y para cada estado normal existe un sucesor normal. Esto garantiza que cada sistema tiene por lo menos una ejecución normal, es decir $\mathcal{NT} \neq \emptyset$.

La lógica dCTL se define sobre CTL con la incorporación de nuevos operadores deónticos de obligación $\mathbf{O}(\psi)$, permiso $\mathbf{P}(\psi)$ y recuperación $\mathbf{R}(\psi)$, que se aplican sobre cierto tipo de fórmulas de trazas ψ. La idea de estos operadores es capturar la noción de *obligación*, *permiso* y *recuperación* sobre trazas. Intuitivamente, podemos describir el significado de estos operadores de la siguiente manera:

- $\mathbf{O}(\psi)$: la propiedad ψ se cumple (está obligada) en cada estado futuro alcanzable mediantes trazas normales.
- $\mathbf{P}(\psi)$: existe una traza normal, comenzando en el estado actual, en el cual la propiedad ψ se cumple en cada estado de la misma.
- $\mathbf{R}(\psi)$: la propiedad ψ se cumple en cada estado futuro anormal, es decir, luego de la ocurrencia de una falla.

Obligación y permiso nos habilitan a expresar propiedades que se cumplen en todas las ejecuciones o algunas ejecuciones normales, respectivamente. Por el otro lado, la recuperación nos permite expresar propiedades que deberían cumplirse luego de que la falla ocurre. Este último operador puede servir para imponer restricciones sobre lo que debe suceder cuando las fallas ocurren, para poder garantizar ciertas propiedades.

Estos operadores deónticos expresan implícitamente características temporales, ya que ψ es una fórmula de traza. Como se verá más adelante, estos operadores, en combinación con las formulas de traza $\psi \rightsquigarrow \psi'$ (el operador $\rightsquigarrow$ es una *“implicación”* entre propiedades de trazas), proveen alguna expresividad adicional con respecto a CTL y serán útiles para expresar propiedades de tolerancia a fallas de manera más directa. Presentemos la sintaxis de la lógica.

Definición 3.6 *Sea AP un conjunto* $\{p_0, p_1, \ldots\}$ *de proposiciones atómicas. El conjunto de fórmulas* Φ *y* Ψ *de estados y trazas, respectivamente, se definen recursivamente como sigue:*

$$\Phi \ ::= \ \top \mid p_i \mid \neg\Phi \mid \Phi \to \Phi \mid \mathsf{A}(\Psi \rightsquigarrow \Psi) \mid \mathsf{E}(\Psi \rightsquigarrow \Psi) \mid \mathbf{O}(\Psi \rightsquigarrow \Psi) \mid \mathbf{P}(\Psi \rightsquigarrow \Psi) \mid \mathbf{R}(\Psi \rightsquigarrow \Psi')$$

$$\Psi \ ::= \ \mathsf{X}\Phi \mid \Phi\, \mathcal{U}\, \Phi \mid \Phi\, \mathcal{W}\, \Phi$$

Otros conectivos booleanos (operadores de estado) como $\wedge$, $\vee$, etc. pueden ser definidos de la forma usual. También, los operadores temporales G y F pueden ser expresados como $\mathsf{G}(\phi) \equiv \phi\ \mathcal{W}\ \bot$ y $\mathsf{F}(\phi) \equiv \top\ \mathcal{U}\ \phi$, respectivamente.

Los operadores booleanos estándares y los cuantificadores A y E de CTL tienen la semántica usual. De todas maneras, ambos cuantificadores y los operadores deónticos se aplican a fórmulas que involucran el operador $\rightsquigarrow$. Este operador relaciona dos fórmulas de trazas, y representa un condicional. Por ejemplo $\mathbf{O}(\psi \rightsquigarrow \psi')$ indica que, para cada traza normal σ que comienza en el estado actual si σ satisface ψ entonces también satisface ψ'.

Definamos formalmente la semántica de la lógica. Comenzamos definiendo la relación $\vDash$ de satisfacción de fórmulas de estado de dCTL sobre estructuras de Kripke coloreadas.

Definición 3.7 *Dada una estructura coloreada de Kripke $M = \langle S, S_0, R, I, \mathcal{N} \rangle$ se definen recursivamente las relaciones de satisfactibilidad sobre estados y sobre trazas (utilizaremos el mismo símbolo $\vDash$ para ambas) de la siguiente manera:*
Para todo $s \in S$

- $M, s \vDash \top$.
- $M, s \vDash p_i \Leftrightarrow p_i \in I(s)$, *donde* $p_i \in AP$.
- $M, s \vDash \neg\varphi \Leftrightarrow$ *no* $M, s \vDash \varphi$.
- $M, s \vDash \varphi \rightarrow \varphi' \Leftrightarrow (M, s \vDash \neg\varphi)$ *o* $(M, s \vDash \varphi')$.
- $M, s \vDash \mathsf{A}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \vDash \psi$ *implica* $M, \sigma \vDash \psi'$, *para toda traza* σ *tal que* $\sigma[0] = s$.
- $M, s \vDash \mathsf{E}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \vDash \psi$ *y* $M, \sigma \vDash \psi'$, *para alguna traza* σ *tal que* $\sigma[0] = s$.
- $M, s \vDash \mathbf{O}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *para todo* $\sigma \in \mathcal{NT}$ *tal que* $\sigma[0] = s$ *se cumple que para todo* $i \geq 0$ *se cumple* $M, \sigma[i..] \vDash \psi$ *implica* $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{P}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *existe* $\sigma \in \mathcal{NT}$ *con* $\sigma[0] = s$ *tal que para todo* $i \geq 0$ *se cumple* $M, \sigma[i..] \vDash \psi$ *y* $M, \sigma[i..] \vDash \psi'$.
- $M, s \vDash \mathbf{R}(\psi \rightsquigarrow \psi') \Leftrightarrow$ *para toda traza* σ *tal que* $\sigma[0] = s$ *se tiene que para todo* $i \geq 0$ *si* $s[i] \notin \mathcal{N}$ *entonces* $M, \sigma[i..] \vDash \psi$ *implica* $M, \sigma[i..] \vDash \psi'$.

La relación de satisfacción anterior hace uso de la satisfacción de fórmulas sobre trazas de dCTL, definidas de manera estándar:

- $M, \sigma \vDash \mathsf{X}\varphi \Leftrightarrow M, \sigma[1] \vDash \varphi$.
- $M, \sigma \vDash \varphi\ \mathcal{U}\ \varphi' \Leftrightarrow$ *existe* $j \geq 0$ *tal que* $M, \sigma[j] \vDash \varphi'$ *y para toda* $0 \leq k < j$, *se cumple* $M, \sigma[k] \vDash \varphi$.
- $M, \sigma \vDash \varphi\ \mathcal{W}\ \varphi' \Leftrightarrow$ *o existe* $j \geq 0$ *tal que* $M, \sigma[j] \vDash \varphi'$ *y para toda* $0 \leq k < j$ *se cumple* $M, \sigma[j] \vDash \varphi$, *o para toda* $j \geq 0$ *se tiene* $M, \sigma[j] \vDash \varphi$.

Denotamos con $M \vDash \varphi$ cuando $M, s \vDash \varphi$ se cumple para cualquier estado s de M, y con $\vDash \varphi$ cuando $M \vDash \varphi$ se cumple para cualquier estructura de Kripke coloreada M. Normalmente utilizamos la notación $\mathbf{O}(\psi)$ para expresar $\mathbf{O}(\top \rightsquigarrow \psi)$ (de la misma manera para otros operadores y cuantificadores). También aplicamos operadores de trazas sobre fórmulas de estado; esto se puede hacer expresando cualquier fórmula de estado φ como la fórmula de trazas $\bot\, \mathcal{U}\, \varphi$.

Los operadores deónticos introducidos aquí dan lugar a algunas propiedades interesantes, algunas de las cuales enumeramos a continuación. Utilizamos φ y φ' para fórmulas de estado y ψ para fórmulas de trazas.

1. $\mathbf{O}(\bot) \equiv \mathbf{O}(\psi) \wedge \mathbf{O}(\neg\psi)$, donde $\neg\psi$ denota la negación de ψ, obtenida utilizando los operadores temporales duales de ψ y negando hacia adentro.
2. $\mathbf{O}(\top) \equiv \top$
3. $\mathbf{P}(\bot) \equiv \bot$
4. $\mathbf{R}(\bot) \vdash \mathsf{AG}\varphi \leftrightarrow \mathbf{O}(\varphi)$
5. $\mathbf{R}(\bot) \vdash \mathsf{EG}\varphi \leftrightarrow \mathbf{P}(\varphi)$
6. $\mathbf{R}(\top) \equiv \top$
7. $\mathbf{R}(\bot) \rightarrow \mathbf{P}(\top)$
8. $\mathbf{O}(\varphi) \wedge \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \wedge \varphi')$
9. $\mathbf{O}(\varphi) \vee \mathbf{O}(\varphi') \rightarrow \mathbf{O}(\varphi \vee \varphi')$
10. $\mathbf{P}(\varphi \wedge \varphi') \rightarrow \mathbf{P}(\varphi) \wedge \mathbf{P}(\varphi')$
11. $\mathbf{P}(\varphi) \vee \mathbf{P}(\varphi') \rightarrow \mathbf{P}(\varphi \vee \varphi')$
12. $\mathbf{R}(\varphi) \wedge \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \wedge \varphi')$
13. $\mathbf{R}(\varphi) \vee \mathbf{R}(\varphi') \rightarrow \mathbf{R}(\varphi \vee \varphi')$

Expliquemos brevemente estas propiedades. La propiedad 1 establece que expresar que $false$ está obligado (que es equivalente a decir que eventualmente habrá una falla) es lo mismo que tener obligaciones contradictorias. La propiedad 2 expresa que decir que $true$ está obligado es equivalente a $true$. Para el operador de permiso se cumple una propiedad similar. La propiedad 3 indica que $false$ no puede estar permitido. Las reglas de deducción 4 y 5 establecen que, en la ausencia de fallas, los operadores deónticos pueden ser expresados utilizando CTL estándar. La propiedad 6 del operador $\mathbf{R}$ establece que $true$ siempre vale después de una falla, mientras que la propiedad 7 expresa que si no ocurre ninguna falla en el futuro ($\mathbf{R}(\bot)$) entonces existen ejecuciones normales ($\mathbf{P}(\top)$). Las propiedades de 8 a 13 relacionan los operadores deónticos con los conectores lógicos estándares.

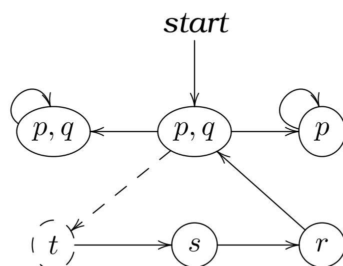


Figura 3.1: Ejemplo simple de una estrucura de Krike coloreada.

Para ilustrar la semántica de los operadores deónticos, consideremos la estructura de Kripke coloreada de la Figura 3.1, donde el conjunto de variables proposicionales involucradas es $\{p, q, r, s, t\}$ y cada estado es etiquetado por un conjunto de variables proposicionales que se cumplen en ese estado. También, las transiciones que llegan a los estados anormales, aquellos graficados con líneas discontinuas, las graficamos con lineas discontinuas y son las que consideramos fallas en nuestro modelo. En la figura, el único estado de falla es el que tiene la etiqueta t.

Es obvio que en cada estado de una ejecución normal (considerando el estado *start* como el estado inicial), la proposición p vale. Esto puede ser expresado en dCTL como $\mathbf{O}(p)$. También existe alguna ejecución normal para la cual $p \wedge q$ se cumple en cada estado de la traza; que puede ser expresada en dCTL como $\mathbf{P}(p \wedge q)$. Por otro lado, el operador de recuperación nos permite expresar propiedades con respecto a estados de falla, y por lo tanto sobre ejecuciones con fallas. Por ejemplo, podemos expresar que inmediatamente después de la ocurrencia de cualquier falla alcanzable, t se cumple y el estado en el cual r se cumple es alcanzable. En dCTL estas propiedades pueden se expresadas como $\mathbf{R}(t)$ y $\mathbf{R}(\mathsf{F}r)$, respectivamente.

Otros operadores deónticos, por ejemplo el operador de *prohibido*, pueden ser expresados usando los operadores definidos. En particular, prohibido se puede definir como $\mathbf{F}(\psi) = \neg\mathbf{P}(\psi)$. Intuitivamente, una propiedad (de traza) está prohibida cuando no puede ser verdadera en un comportamiento normal. En otras palabras, si dicha propiedad es siempre $true$ en una traza, la traza contiene fallas.

3.2. Definición de fCTL

En dCTL el modelo nos permite diferenciar comportamientos normales o de falla a través de colorear estados. De esta manera, modelamos las fallas como transiciones a estados “anormales” o que podríamos llamar de error, en térmi-

nos de tolerancia a fallas. Si embargo, podríamos querer modelar un sistema en que una falla nos lleva a un estado *normal*, por ejemplo el cambio de un valor en una celda de memoria que no cuenta con redundancia. Consideramo entonces que, en lugar de colorear los estados, sería conveniente colorear las transiciones. Esto nos permitiría una descripción más precisa de los sistemas con fallas, y a su vez nos daría nuevas herramientas para expresar propiedades y características más interesantes de los sistemas.

Con este objetivo definimos estructuras de Kripke con fallas o transiciones coloreadas y, a partir de estos modelos, proponemos una nueva lógica que permite describir sistemas con fallas.

Definición 3.8 *Una* ***estructura de Kripke con fallas*** *es una 5-upla* $\langle S, S_0, R, F, I\rangle$*, donde* $\langle S, S_0, R, I\rangle$ *es una estructura de Kripke y* $F \subseteq R$ *es un conjunto de transiciones diferenciadas que llamaremos* fallas*.*

Llamamos *transiciones normales* a aquellas que no son fallas, es decir al conjunto $R - F$. Las ejecuciones normales son aquellas que recorren exclusivamente transiciones normales.

Para definir fCTL retomamos las ideas principales de dCTL considerando las estructuras de Kripke con fallas. En dCTL se utilizan estructuras de Kripke coloreadas, es decir, estructuras de Kripke en la que diferenciamos estados, como estados anormales o de falla. En las estructuras de Kripke con falla, son las transiciones las que diferenciamos como transiciones de falla, lo cual nos permite representar mejor un evento de falla.

Como las transiciones sólo se referencian a partir de las trazas, tomamos principalmente la lógica presentada anteriormente con dCTL agregando expresividad a fórmulas sobre trazas. Los operadores que incorporan características deónticas se pueden definir a partir de estos operadores, por lo que no es necesario incluirlos ahora.

Definición 3.9 *Sea AP un conjunto* $\{p_0, p_1, \dots\}$ *de proposiciones atómicas. El conjunto de fórmulas* Φ *y* Ψ *de estados y trazas, respectivamente, se definen recursivamente como sigue:*

$$\begin{aligned}
\Phi &::= \top \mid p_i \mid \neg\Phi \mid \Phi \to \Phi \mid \mathsf{A}(\Psi_l \rightsquigarrow \Psi_b) \mid \mathsf{E}(\Psi_l \rightsquigarrow \Psi_b) \\
\Psi_b &::= \mathsf{X}\Phi \mid \Phi\,\mathcal{U}\,\Phi \mid \Phi\,\mathcal{W}\,\Phi \mid \mathsf{X}_f\Phi \mid \Phi\,\mathcal{U}_f\,\Phi \mid \Phi\,\mathcal{W}_f\,\Phi \\
\Psi_l &::= \top \mid p_i \mid \neg\Psi_l \mid \Psi_l \to \Psi_l \mid \\
&\quad\;\; \mathsf{X}\Psi_l \mid \Psi_l\,\mathcal{U}\,\Psi_l \mid \Psi_l\,\mathcal{W}\,\Psi_l \mid \mathsf{X}_f\Psi_l \mid \Psi_l\,\mathcal{U}_f\,\Psi_l \mid \Psi_l\,\mathcal{W}_f\,\Psi_l
\end{aligned}$$

Las características principales de esta lógica son dos. Por un lado, las fórmulas temporales cuentan con el operador $\rightsquigarrow$. La fórmula de la izquierda Ψ_l está basada en LTL (linear-time temporal logic) [BK08] y nos permite caracterizar el conjunto de trazas sobre las que se evaluará la fórmula de la derecha Ψ_b,

esta última con una semántica basada en CTL. En segundo lugar, para cada operador temporal se incorporan operadores análogos, pero incorporando la presencia de fallas. Estos operadores nos permiten describir comportamiento con fallas. Si bien nos interesa mantener los operadores deónticos para la descripción de sistemas tolerantes a falla, éstos pueden ser definidos a partir de estos operadores.

Definamos formalmente la semántica de la lógica. Comenzamos definiendo la relación $\models$ de satisfacción de fórmulas de estado de fCTL sobre estructuras de Kripke con fallas.

Definición 3.10 *Dada una estrutura de Kripke con fallas $M = \langle S, S_0, R, F, I \rangle$ la semántica de los operadores de estado se define recursivamente con relaciones de satisfactibilidad sobre estados y sobre trazas (utilizaremos el mismo símbolo $\models$ para ambas) de la siguiente manera:*
Para $s \in S$

- $M, s \models \top$.
- $M, s \models p_i \Leftrightarrow p_i \in I(s)$, *donde* $p_i \in AP$.
- $M, s \models \neg\varphi \Leftrightarrow$ *no* $M, s \models \varphi$.
- $M, s \models \varphi \to \varphi' \Leftrightarrow (M, s \models \neg\varphi)$ *o* $(M, s \models \varphi')$.
- $M, s \models \mathsf{A}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \models \psi$ *implica* $M, \sigma \models \psi'$, *para toda traza* σ *tal que* $\sigma[0] = s$.
- $M, s \models \mathsf{E}(\psi \rightsquigarrow \psi') \Leftrightarrow M, \sigma \models \psi$ *y* $M, \sigma \models \psi'$, *para alguna traza* σ *tal que* $\sigma[0] = s$.

La relación de satisfacción de arriba hace uso de la satisfacción de fórmulas sobre trazas de fCTL que se amplía de la siguiente manera:

- $M, \sigma \models \mathsf{X}\varphi \Leftrightarrow M, \sigma[1] \models \varphi$.
- $M, \sigma \models \mathsf{X}_f\varphi \Leftrightarrow \sigma[0, 1] \in F$ *y* $M, \sigma[1] \models \varphi$
- $M, \sigma \models \varphi\, \mathcal{U}\, \varphi' \Leftrightarrow$ *existe* $j \geq 0$ *tal que* $M, \sigma[j] \models \varphi'$ *y para toda* $0 \leq k < j$, *se cumple* $M, \sigma[k] \models \varphi$.
- $M, \sigma \models \varphi\, \mathcal{U}_f\, \varphi' \Leftrightarrow M, \sigma[0] \models \varphi$ *y existe* $j \geq 1$ *tal que* $\sigma[j-1, j] \in F$ *y* $M, \sigma[j] \models \varphi'$ *y para toda* $1 \leq k < j$, *se cumple* $\sigma[k-1, k] \notin F$ *y* $M, \sigma[k] \models \varphi$.
- $M, \sigma \models \varphi\, \mathcal{W}\, \varphi' \Leftrightarrow$ *o existe* $j \geq 0$ *tal que* $M, \sigma[j] \models \varphi'$ *y para toda* $0 \leq k < j$ *se cumple* $M, \sigma[j] \models \varphi$, *o para toda* $j \geq 0$ *se tiene* $M, \sigma[j] \models \varphi$.
- $M, \sigma \models \varphi\, \mathcal{W}_f\, \varphi' \Leftrightarrow$ *o* $M, \sigma[0] \models \varphi$ *y existe* $j \geq 1$ *tal que* $\sigma[j-1, j] \in F$ *y* $M, \sigma[j] \models \varphi'$ *y para toda* $1 \leq k < j$, *se cumple* $\sigma[k-1, k] \notin F$ *y* $M, \sigma[k] \models \varphi$, *o para toda* $j \geq 0$ *se tiene* $\sigma[j, j+1] \notin F$ *y* $M, \sigma[j] \models \varphi$.

Denotamos con $M \vDash \varphi$ cuando $M, s \vDash \varphi$ se cumple para cualquier estado s de M, y con $\vDash \varphi$ cuando $M \vDash \varphi$ se cumple para cualquier estructura de Kripke con fallas M.

Definimos los operadores sobre trazas G y F que expresan que una propiedad se cumple siempre (globalmente) o eventualmente en el futuro como es usual:

$\mathsf{G}(\varphi) = \varphi\, \mathcal{W}\, \bot$

$\mathsf{F}(\varphi) = \top\, \mathcal{U}\, \varphi$

También podemos definir el operador F_f que permite expresar que en algún momento en el futuro se cumple una propiedad determinada inmediatamente después de la ocurrencia de una falla.

$\mathsf{F}_f(\varphi) = \top\, \mathcal{U}_f\, \varphi$

Denotamos simplemente con F_f o X_f cuando φ sea $true$, i.e $\mathsf{F}_f = \mathsf{F}_f(\top)$ y $\mathsf{X}_f = \mathsf{X}_f(\top)$, respectivamente.

También podemos definir el operador F_n que nos permite expresar que eventualmente en el futuro se cumple una determinada propiedad, sin la ocurrencia de fallas.

$\mathsf{F}_n(\varphi) = \neg\mathsf{X}_f\, \mathcal{U}\, \varphi$

Estos operadores permiten incorporar caracterizaciones a las trazas que distinguen ejecuciones normales (sin fallas) de ejecuciones con fallas. Por lo tanto, se pueden definir operadores deónticos tradicionales. Para ello caracterizamos en primer lugar las trazas normales $\mathcal{N}$ con una fórmula de traza

$$\mathcal{N} = \top\, \mathcal{W}_f\, \bot$$

Por lo tanto los operadores deónticos pueden ser definidos de la siguiente manera

- $\mathbf{O}(\psi) = \mathsf{A}(\mathcal{N} \rightsquigarrow \psi)$
- $\mathbf{P}(\psi) = \mathsf{E}(\mathcal{N} \rightsquigarrow \psi)$

donde ψ es una fórmula de traza.

Notar que, a diferencia de dCTL, los operadores de permiso y obligación no incluyen en su semántica una propiedad universal que establece que la propiedad se cumple a lo largo de la traza. Por lo tanto, para expresar el operador de obligación o de permiso correspondiente a dCTL es necesario anidarlo con el operador G.

Definimos esta notación para simplificar algunas expresiones más utilizadas.

- $\mathsf{A}(\psi) = \mathsf{A}(\top \rightsquigarrow \psi)$
- $\mathsf{E}(\psi) = \mathsf{E}(\top \rightsquigarrow \psi)$

- $\mathsf{AG}(\varphi) = \mathsf{A}(\mathsf{G}(\varphi))$
- $\mathsf{AF}(\varphi) = \mathsf{A}(\mathsf{F}(\varphi))$
- $\mathsf{EG}(\varphi) = \mathsf{E}(\mathsf{G}(\varphi))$
- $\mathsf{EF}(\varphi) = \mathsf{E}(\mathsf{F}(\varphi))$
- $\mathbf{O}\mathsf{G}(\varphi) = \mathbf{O}(\mathsf{G}(\varphi))$
- $\mathbf{O}\mathsf{F}(\varphi) = \mathbf{O}(\mathsf{F}(\varphi))$
- $\mathbf{P}\mathsf{G}(\varphi) = \mathbf{P}(\mathsf{G}(\varphi))$
- $\mathbf{P}\mathsf{F}(\varphi) = \mathbf{P}(\mathsf{F}(\varphi))$
- $\mathsf{AX}(\varphi) = \mathsf{A}(\mathsf{X}(\varphi))$
- $\mathsf{EX}(\varphi) = \mathsf{E}(\mathsf{X}(\varphi))$
- $\mathbf{O}\mathsf{X}(\varphi) = \mathbf{O}(\mathsf{X}(\varphi))$
- $\mathbf{P}\mathsf{X}(\varphi) = \mathbf{P}(\mathsf{X}(\varphi))$

donde φ es una fórmula de estado y ψ es una fórmula de traza. También aplicamos operadores de trazas sobre fórmulas de estado. Esto se puede hacer expresando cualquier fórmula de estado φ como la fórmula de trazas $\bot \, \mathcal{U} \, \varphi$.

Para definir obligación y permiso, utilizamos el predicado $\mathcal{N}$ para caracterizar las trazas sin fallas en el antecedente de los operadores A y E, respectivamente. Esta misma característica puede utilizarse para modelar o restringir el comportamiento de fallas. Por ejemplo, en el caso de que quisiéramos evaluar si se cumplen determinadas propiedades en el caso de que las fallas ocurran de tal o cual manera. Así como caracterizamos al conjunto de trazas que no tienen fallas (con $\mathcal{N}$), podemos expresar:

- $\mathcal{F}_1 = \top \, \mathcal{U}_f \, \mathcal{N}$ al conjunto de trazas que tienen sólo una falla.
- $\mathcal{F}_{\leq 1} = \top \, \mathcal{W}_f \, \mathcal{N}$ al conjunto de trazas que tienen una o menos fallas.
- $\mathcal{F}_{\leq 2} = \top \, \mathcal{W}_f \, \mathcal{F}_{\leq 1}$ al conjunto de trazas que tienen dos o menos fallas.

El mismo principio podría emplearse para describir otras restricciones en el comportamiento de las fallas en relación con el sistema. Por ejemplo, en el caso de contar con algún predicado, digamos $good$, que nos caracterice a los estados buenos; entonces podemos describir *"el conjunto de trazas que no tienen más de dos fallas entre estados que satisfacen* $good$*"* con la siguiente fórmula

$\mathsf{G}(\mathsf{F}_n(good) \vee \mathsf{F}_f(\mathsf{F}_n(good)) \vee \mathsf{F}_f(\mathsf{F}_f(\mathsf{F}_n(good))))$

Esta caracterización puede ser de utilidad cuando tenemos formas de garantizar que algo bueno pasará antes de la ocurrencia de una tercera falla consecutiva, y poder determinar si algunas propiedades se cumplen en esos casos.

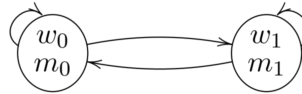


Figura 3.2: Modelo simple de una celda de memoria, sin fallas.

Podríamos expresar, por ejemplo, que el sistema es tolerante a dos fallas, pero no tres.

3.3. Casos de estudio

Veremos dos ejemplos que nos permiten ilustrar la presentación de los modelos y el uso de la lógica para expresar propiedades de tolerancia a fallas: el modelo de una celda de memoria con triple redundancia y un modelo de un protocolo token ring con tres nodos y pérdida de token.

3.3.1. Celda de memoria

Consideremos un sistema compuesto por una celda de memoria simple, que guarda un bit de información y soporta operaciones de lectura y escritura. Este sistema puede ser modelado con una estructura de Kripke como la mostrada en la Figura 3.2, donde cada estado mantiene el valor actual de la celda de memoria (m_i, para $i = 0, 1$) y la última operación de escritura realizada (w_i, para $i = 0, 1$). Evidentemente, en este sistema el resultado de la lectura depende del valor corriente de la celda. Una propiedad que podemos asociar con este modelo es que el valor de lectura coincide con el valor de la última escritura realizada. Esta propiedad puede ser expresada utilizando CTL como sigue

$$\mathsf{AG}((m_0 \rightarrow w_0) \wedge (m_1 \rightarrow w_1))$$

Esta propiedad puede ser considerada como requisito o *especificación* del sistema; y la implementación del sistema, descripto en la Figura 3.2, debería satisfacerla. Este modelo expresa un comportamiento ideal que no tiene en cuenta fallas de ningún tipo, por lo que no necesitamos operadores con fallas.

Consideremos ahora un escenario con fallas. Supongamos que cuando el valor de una celda es 1, ésta puede perder inesperadamente la carga e interpretarse como un 0. En este caso, una implementación sin redundancia no puede garantizar la especificación, dado que luego de haber escrito un 1 la lectura de la celda contiene un 0, dando lugar a una violación de la especificación. Es interesante mencionar que, si bien la especificación no se cumple, la propiedad $\mathsf{OG}((m_0 \rightarrow w_0) \wedge (m_1 \rightarrow w_1))$, que sólo cuantifica sobre trazas sin fallas, sí se cumple. Dicho de otra forma, si no ocurren fallas, el sistema funciona correctamente.

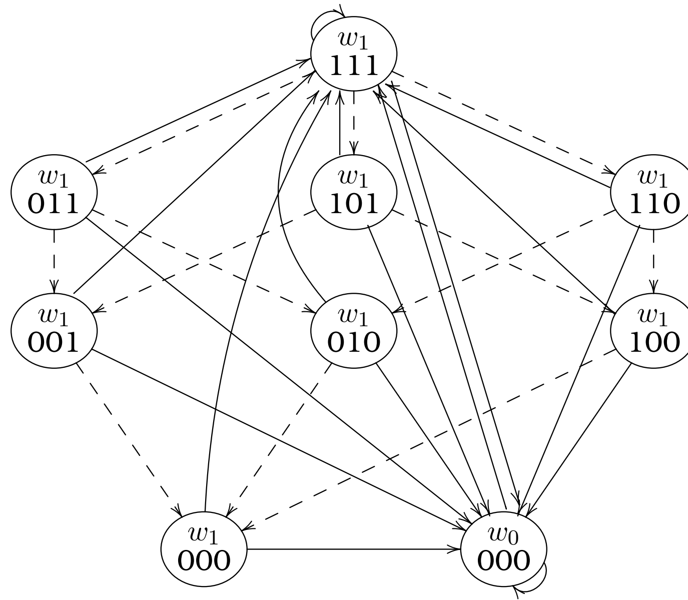


Figura 3.3: Implementación de una celda de memoria con triple redundancia y fallas.

Para contemplar la posibilidad de la ocurrencia de fallas, debemos modificar la implementación. Un mecanismo típico utilizado para tolerar este tipo de fallas en sistemas aeroespaciales es la triple redundancia. En particular, se puede implementar utilizando tres celdas de memoria en lugar de una, y decidiendo por mayoría el valor de lectura. En concreto, la operación de escritura se realiza escribiendo simultáneamente en las tres celdas, mientras que la operación de lectura devuelve el valor que se repite por lo menos en dos de las celdas. Este modelo se muestra en la Figura 3.3. Cada estado se describe con la variable w_i, que representa la última operación de escritura realizada, y tres bits descriptos por las variables booleanas c_0, c_1 y c_2, que representan el valor corriente de cada celda de memoria. La ocurrencia de una falla, que cambia una celda de memoria con valor 1 a un valor 0, se representa con una línea discontinua. Las líneas continuas indican operaciones normales, en este caso de escritura.

La operación de lectura se puede expresar como $r_1 = (c_0 \wedge c_1) \vee (c_0 \wedge c_2) \vee (c_1 \wedge c_2)$, de manera que se lea 1 en el caso de que por lo menos dos celdas de memoria sean 1. r_0 puede definirse simplemente como la negación de r_1.

Ahora podemos discutir sobre las propiedades esperadas en el nuevo modelo de memoria. Debemos actualizar los requisitos originales de la celda de memoria con respecto a la nueva implementación. En este caso, actualizamos el valor de lectura original m_i por la nueva representación definida r_i.

$$\mathsf{AG}((r_0 \rightarrow w_0) \wedge (r_1 \rightarrow w_1))$$

Lamentablemente, esta propiedad tampoco se cumple en esta nueva implementación si varias fallas ocurren. De todas formas, esta propiedad es útil, en tanto su verificación (e.g. via model checking) puede producir contraejemplos que nos ayuden a entender situaciones en las cuales la especificación se viola. Nuevamente, la misma propiedad sin la presencia de fallas sí es válida en este modelo.

$$\mathbf{O}\mathsf{G}((r_0 \to w_0) \wedge (r_1 \to w_1))$$

La motivación de introducir mecanismos de tolerancia a fallas es garantizar que el sistema siga funcionando correctamente, incluso bajo la presencia de fallas. En este caso, no cualquier escenario con fallas nos llevará a una violación de la especificación. Podemos debilitar la propiedad anterior limitando el comportamiento de las fallas, expresando por ejemplo que ocurra como máximo una falla, la cual sigue siendo válida.

$$\mathsf{A}(\mathcal{F}_{\leq 1} \rightsquigarrow \mathsf{G}(r_0 \to w_0) \wedge (r_1 \to w_1)))$$

Una propiedad más interesante aún, es no restringir la cantidad de fallas en toda la traza a 1, sino contemplar que pueda ocurrir cualquier cantidad de fallas siempre y cuando entre falla y falla ocurra por lo menos una escritura. Este supuesto se basa en una técnica utilizada para estos casos, en la cual el sistema recorre periódicamente las celdas de memoria reescribiendo sus propios datos. Para ello, podemos representar los estados buenos ($good$) como los dos estados posibles que se obtienen luego de una escritura, i.e., $good \doteq c1 = c2 = c3 = w$. La propiedad que expresa que la especificación vale si no ocurre más de una falla entre escritura y escritura ($good$) se puede expresar de la siguiente manera:

$$\mathsf{A}(\mathsf{G}(\mathsf{X}_f \to \mathsf{X}_f(\mathsf{F}_n(good))) \rightsquigarrow \mathsf{G}((r_0 \to w_0) \wedge (r_1 \to w_1)))$$

Otra propiedad interesante que nos puede resultar útil expresar es la posibilidad de recuperación. Es decir, independientemente de si estamos en un estado de falla, o de la cantidad de fallas previas, siempre es posible *recuperarse* a un estado bueno ($good$) en el que el sistema se reestablece. Esta propiedad la podemos expresar de la siguiente manera:

$$\mathsf{AG}(\neg good \to \mathbf{P}\mathsf{F}(good))$$

3.3.2. Protocolo Token Ring

Consideremos ahora otro caso de estudio. Supongamos que tenemos un sistema simple compuesto por tres nodos, cuyas actividades son reguladas

Figura 3.4: Modelo de un protocolo Token Ring con pérdida de token.

por el protocolo *token ring*. En el sistema original, los tres nodos están conectados mediante una topología de anillo, y un token es pasado a través de los nodos de manera que el nodo que lo tiene en un tiempo particular es aquel que cuenta con permiso de acceso a determinados recursos, por ejemplo, permiso para enviar información a través de la red. No es difícil pensar en algunas propiedades de ejemplo que expresen los requisitos del sistema, como que siempre hay exactamente un nodo que tiene el token, o cuando un nodo tiene el token, eventualmente lo pasa al siguiente en el anillo.

Una falla simple en este contexto podría ser que, debido a una falla en el canal de comunicación, el token se pierda al ser transmitido por alguno de los nodos. Si el período de uso del token para cada nodo tiene un límite máximo, se puede implementar un mecanismo de detección utilizando un timeout (si un nodo no vio el token por más de n veces el tiempo máximo, donde n es el número de nodos). Un modelo de la situación, incluyendo el mecanismo de detección y recuperación de fallas, se describe en la Figura 3.4. Cada estado n_i corresponde a que el token lo tiene el nodo i; cuando el token se pierde, ningún nodo lo tiene; y cuando la detección del token perdido se activa, un nuevo token se genera y es entregado al nodo 0.

La especificación del sistema puede ser descripta utilizando fCTL:

$$\mathsf{AG}((n_0 \wedge \neg n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge \neg n_1 \wedge n_2))$$
$$\mathsf{AG}(n_i \rightarrow \mathsf{AX}(n_{i \oplus 1}))$$

donde $\oplus$ es la suma módulo tres. Para mayor simplicidad, asumimos que cada nodo tiene el token exáctamente un instante de tiempo (en el próximo paso, el token pertenece al siguiente nodo). Para expresar que estas propiedades valen cuando no ocurren fallas en el sistema, podemos utilizar el operador de obligación de fCTL que evalúa la fórmula en trazas sin fallas.

$$\mathbf{O}\mathsf{G}((n_0 \wedge \neg n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge n_1 \wedge \neg n_2) \vee (\neg n_0 \wedge \neg n_1 \wedge n_2))$$
$$\mathbf{O}\mathsf{G}(n_i \;\rightarrow\; \mathsf{AX}(n_{i \oplus 1}))$$

Ambos requisitos, que valen cuando no ocurren fallas, no se cumplen en el caso de que ocurran fallas, es decir que se pierda el token. La segunda propiedad, por ejemplo, se puede *relajar* en el caso que la quisiéramos mantener

bajo la presencia de fallas. La necesidad de que el token deba ser pasado en el próximo instante al siguiente nodo, puede ser adaptado a que sea pasado en algún momento en el futuro al próximo nodo.

$$\mathsf{AG}(n_i \rightarrow \mathsf{AF}(n_{i\oplus 1}))$$

Esta es una propiedad de progreso que vale en el sistema, siempre que el sistema garantice *strong fairness* [1] [Tel00]. Notar que, a pesar que fairness no es expresable en CTL, esta restricción se suele incorporar en varios model checkers para la verificación de propiedades de *liveness*, como es este caso.

Otra propiedad interesante de tolerancia a fallas es la que expresa que las fallas son la única causa para la pérdida del token. En otras palabras, si al token lo tiene un nodo en particular n_i, el token es pasado al siguiente nodo o una falla ocurre. Esta idea la podemos expresar de la siguiente manera:

$$\mathsf{AG}(n_i \rightarrow \mathsf{X}_f \ \vee \ \mathsf{X}(n_{i\oplus 1}))$$

3.4. Tolerancia a fallas con fCTL

Los modelos de los sistemas vistos en este capítulo (estructuras de Kripke y variantes) están basados en propiedades sobre los estados, a diferencia del anterior que ponía foco en las acciones (LTSs y LTSs reactivos). En este marco, una manera de describir o caracterizar el buen funcionamiento del sistema es a través de predicados sobre el conjunto de estados que caractericen los estados buenos del sistema.

Para definir la tolerancia a fallas, Gärtner [Gär99] retoma el trabajo de Arora y Gouda [AG93] que la definen basados en la idea presentada por Dijkstra de *self-stabilization* [Dij74]. En estos trabajos podemos identificar algunos conceptos relacionados con la tolerancia a fallas que a continuación definimos en términos de fCTL:

Definición 3.11 *Una fórmula de estado es una fórmula proposicional de fCTL.*

Definición 3.12 *Si φ es una fórmula de estado, decimos que es cerrada para una Estructura de Kripke con fallas M si y sólo si*
$M \models \varphi \rightarrow \mathsf{AX}(\varphi)$

Dado un sistema representado por una estructura de Kripke con fallas, que una fórmula de estado sea cerrada establece que, si partimos de un estado que la satisface, todas las transiciones la preservan, y en consecuencia, representa un invariante.

[1] Entendemos por strong fairness cuando se garantiza que una transición que está habilitada infinitas veces eventualmente es ejecutada.

Definición 3.13 *Si φ y ψ son predicados de estados, decimos que ψ converge a φ para una Estructura de Kripke con fallas M si y sólo si*
$M \models \psi \rightarrow \mathsf{A}(\psi\, \mathcal{U}\, \varphi)$

La noción de convergencia de una fórmula de estado ψ a una fórmula de estado φ, en un sistema dado, establece que si partimos de cualquier estado que cumpla ψ eventualmente llegaremos a un estado que cumpla φ. Esta propiedad, junto con la anterior son la base para la noción de *self-stabilization* mencionada anteriormente.

Cuando hablamos de tolerancia a fallas, consideramos a las fallas como situaciones excepcionales. En este sentido, es útil distinguir el comportamiento normal, de aquel que tiene fallas. Por lo tanto podemos definir una variante de los conceptos anteriores, considerando sólo el comportamiento normal. Esto lo hacemos con el operador $\mathbf{O}()$.

Definición 3.14 *Si φ es una fórmula de estado, decimos que es normalmente cerrada para una Estructura de Kripke con fallas M si y sólo si*
$M \models \varphi \rightarrow \mathbf{O}\mathsf{X}(\varphi)$

Definición 3.15 *Si φ y ψ son predicados de estados, decimos que ψ converge normalmente a φ para una Estructura de Kripke con fallas M si y sólo si*
$M \models \psi \rightarrow \mathbf{O}(\psi\, \mathcal{U}\, \varphi)$

A partir de estas ideas, podemos definir la tolerancia a fallas, tanto enmascarada (masking) como no esmascarada (nonmasking), de acuerdo a la definición presentada por Gärtner en [Gär99, p. 7].

Un sistema con fallas se dice **tolerante a fallas** para un invariante P si y sólo sí existe un predicado T para el cual los siguientes requisitos se cumplen:

- Cualquier estado donde vale P también satisface T.

  $P \ \rightarrow \ T$

- Comenzando de un estado que satisface T cualquier transición (ya sea normal o de falla) nos lleva a un estado que satisface T. Es decir, T es cerrada sobre el conjunto de transiciones posibles.

  $T \ \rightarrow \ \mathsf{AX}(T)$

- Comenzando en un estado que satisface T, si sólo ejecutamos transiciones normales, eventualmente alcanzaremos un estado que satisfaga P.

  $T \ \rightarrow \ \mathbf{O}\mathsf{F}(P)$

Hablamos de *masking fault tolerance* si la fórmula de estado T es equivalente a P, en el sentido que se cumplen en el mismo conjunto de estados. La idea central, es que el sistema se comporta de la misma manera incluso con la presencia de fallas, siendo la ocurrencia de éstas completamente transparente al

sistema. El ejemplo más conocido son los códigos de corrección de errores, en los que la misma operación de lectura encubre la presencia de errores (siempre y cuando la cantidad de fallas que ocurrieron sea menor a la cantidad de errores que corrige el código).

Cuando T no es equivalente a P tenemos comportamientos que sólo pueden ocurrir bajo la ocurrencia de fallas. Si bien Gärtner [Gär99] se refiere a este caso como *nonmasking fault tolerance*, consideramos que es necesario para efectivamente hablar de tolerancia, incorporar una característica de *safety* P_{min} a la definición, que represente las condiciones de mínima que el sistema debe cumplir para ser tolerante a fallas. En este sentido, consideramos un sistema *nonmasking fault tolerance* cuando $T \rightarrow P_{min}$, completando nuestra definición de tolerancia a fallas.

Definición 3.16 *Un sistema representado con una Estructura de Kripke con fallas M se dice* ***tolerante a fallas*** *para un invariante P y un invariante mínimo P_{min} si y sólo sí existe un predicado T que cumple las siguientes condiciones:*

- $M \models P \rightarrow T$
- $M \models T \rightarrow \mathsf{AX}(T)$
- $M \models T \rightarrow \mathbf{O}\mathsf{F}(P)$
- $M \models T \rightarrow P_{min}$

Otra característica de la tolerancia a fallas son los estados de error, en los términos de la definición 1.2. Es decir, un estado del sistema provocado por una falla que eventualmente puede generar un comportamiento incorrecto del propio sistema.

Definición 3.17 *Dados un sistema representado por una estructura de Kripke con fallas M y $good$ una fórmula de estado que caracteriza el buen comportamiento de sistema. Decimos que w es un estado de error si y sólo si*

$M, w \models \mathbf{P}\mathsf{F}(\neg good)$

La idea de **recuperación** de fallas está incluida explícitamente en nuestra definición de tolerancia a fallas. La condición $T \rightarrow \mathbf{O}\mathsf{F}(P)$ expresa que si dejan de ocurrir fallas, en algún momento el sistema vuelve a cumplir el invariante P.

Tolerancia a n fallas Cuando trabajamos con dispositivos de hardware y otros recursos finitos, más allá del diseño del sistema, las fallas que pueden ser toleradas son finitas, en tanto esos recursos sean indispensables para el funcionamiento del mismo. Por lo tanto tiene sentido, como vimos en el ejemplo 3.3.1, debilitar la noción de tolerancia a fallas y expresar el grado de tolerancia a partir de la cantidad de fallas que el sistema puede tolerar.

Puede ocurrir entonces que, si ocurren más de n fallas, el sistema no pueda recuperarse o no se cumpla la condición mínima de safety P, dadas por las últimas dos condiciones de la definición general de tolerancia a fallas. La condición que queremos debilitar en nuestra definición es $T \rightarrow \mathsf{AX}(T)$, restringiendo el conjunto T a los estados que podemos llegar desde cualquier estado que cumple el invariante con n o menos fallas. La manera que podemos expresar esto es utilizando el componente LTL del operador A que permite restringir las trazas sobre las que se evalúa la fórmula.

$P \rightarrow \mathsf{A}(\mathcal{T}_n \rightsquigarrow \mathsf{G}(T))$

donde $\mathcal{T}_n$ se construye recursivamente de la siguiente manera

$\mathcal{T}_0 = \mathsf{F}_n(P)$

$\mathcal{T}_{n+1} = \mathcal{T}_n \vee \mathsf{F}_f(\mathcal{T}_n)$

Con $\mathcal{T}_n$ caracterizamos las trazas que tienen como máximo n fallas hasta su recuperación (cuando se satisface el invariante P).

Definición 3.18 *Un sistema representado con una Estructura de Kripke con fallas M se dice que M* ***tolera*** *n* ***fallas*** *para un invariante P y un invariante mínimo P_{min} si y sólo sí existe un predicado T que cumple las siguientes condiciones:*

- $M \models P \rightarrow T$
- $M \models P \rightarrow \mathsf{A}(\mathcal{T}_n \rightsquigarrow \mathsf{G}(T))$
- $M \models T \rightarrow \mathsf{OF}(P)$
- $M \models T \rightarrow P_{min}$

donde $\mathcal{T}_n$ se construye recursivamente de la siguiente manera

$\mathcal{T}_0 = \mathsf{F}_n(P)$

$\mathcal{T}_{n+1} = \mathcal{T}_n \vee \mathsf{F}_f(\mathcal{T}_n)$

3.5. Consideraciones finales

En este capítulo reflejamos el trabajo de desarrollo de dos lógicas nuevas para abordar la tolerancia a fallas que son dCTL y fCTL. En primer lugar, presentamos dCTL que fue el producto del trabajo conjunto con Pablo Castro y Cecilia Kilmurray en el que exploramos una combinación de operadores deónticos con CTL. Para esta lógica elaboramos algunos casos de estudio presentados en [CKAA11]. fCTL surge entonces a partir de una revisión de dCTL, redefiniendo la representación de fallas y resignificando el operador $\rightsquigarrow$ que permite establecer condiciones de trazas sobre las propiedades a evaluar.

Esta nueva lógica permitió redefinir los operadores deónticos y presentar definiciones más precisas y simples de la tolerancia a fallas y otros conceptos como los estados de error y la recuperación. En relación a la tolerancia a

fallas, presentamos las nociones de masking y nonmasking en este marco, y pudimos debilitar el concepto a la noción de tolerar hasta n fallas. Finalmente, presentamos casos de estudio para ilustrar los alcances de la lógica.

The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

EWD249 Notes on estructured programming, On The Reliability of Mechanisms

4 Verificación de una lógica deóntica temporal

En este capítulo, estamos interesados en explorar el enfoque tomado por Castro y Maibaum en [CM09a], donde se introduce una lógica proposicional deóntica (PDL) y luego se extiende con nuevos operadores para expresar comportamiento temporal (DTL). Esta lógica incorpora elementos deónticos que permiten distiguir entre comportamiento normativo (i.e. sin fallas) y no normativo (i.e. con fallas) con una directa aplicación en la tolerancia a fallas.

Dentro de un enfoque formal del desarrollo de software, se consideran esenciales las técnicas de análisis (semi-)automático para cualquier método que se pueda utilizar efectivamente en la práctica. En particular, la posibilidad de chequear algorítmicamente si una fórmula PDL, o una fórmula DTL que incluye operadores temporales, es válida en un determinado sistema es de gran relevancia para utilización de esta lógica como parte del uso de métodos formales para la tolerancia a fallas. Afortunadamente, tanto PDL y su extensión temporal DTL son decidibles [CM09a]: un procedimiento de decisión de la lógica DTL que se basa en un cálculo de tableaux, se propone en [CM08]. Sin embargo, el procedimiento de decisión se desarrolló con el objetivo de probar que la lógica era decidible; de hecho, este cálculo tableaux que ha demostrado ser útil para la investigación de decidibilidad y complejidad de la lógica, no fue concebido como parte de una herramienta para la verificación formal. Debido a este hecho, no se tomaron consideraciones prácticas en la definición de este procedimiento de decisión, y no se ha implementado en una herramienta para el análisis de las especificaciones de tolerancia a fallas.

En este capítulo, nos dedicamos a la definición de un procedimiento de decisión para PDL y DTL, con el propósito de utilizarlo para la verificación automá-

tica. Nuestro enfoque consiste en caracterizar PDL/DTL con μ-calculus y luego usar un model checker de μ-calculus para verificar si un determinado sistema satisface un propiedad de tolerancia a fallas expresada en PDL/DTL. Aquí presentamos nuestra caracterización de PDL/DTL a μ-calculus y mostramos como un sistema tolerante a fallas, capturado por una estructura deóntica, puede ser analizado a través del uso de fórmulas PDL/DTL para describir propiedades de tolerancia a fallas y su posterior verificación. Además, mostramos que nuestra traducción de PDL/DTL a μ-calculus es correcta en el sentido de que un problema de model checking en PDL/DTL se reduce correctamente a un problema de model checking en μ-calculus; también mostramos que se mantienen los contraejemplos, es decir, que cualquier contraejemplo de μ-calculus que surja de la verificación de una propiedad de un modelo traducido, puede ser recuperado como un contraejemplo de la especificación original.

En el plano metodológico, exploramos la especificación de sistemas con estructuras deónticas que permiten expresar tanto estados como acciones, e incluso, diferenciar comportamientos normales y de fallas. El desarrollo de algunos ejemplos nos ayudan a visualizar la expresividad y potencia de la herramienta. Por otro lado, mostramos cómo la combinación de operadores deónticos con los temporales nos permiten capturar propiedades y características de los sistemas tolerantes a fallas.

Finalmente, mostramos otra traducción a μ-calculus, que resulta más eficiente para la tarea de model checking y que utilizamos para verificar propiedades de nuestros ejemplos en el model checker Mucke ([AKCA12]). Esta herramienta nos permitió encontrar contraejemplos no triviales de alguna propiedad que se suponía válida en el modelo.

4.1. Antecedentes

4.1.1. μ-calculus

El μ-calculus es un lenguaje que se desarrolló a partir de los operadores de punto fijo, que permitieron describir propiedades más interesantes ligadas a la terminación. Esta lógica, si bien es muy expresiva no es muy cómoda a la hora de describir propiedades. Con este lenguaje, es posible codificar varias de las lógicas modales más utilizadas, como lo son las lógicas temporales.

Como otras lógicas con operadores de punto fijo, el μ-calculus es un formalismo útil para analizar la expresividad y la complejidad algorítmica de lógicas temporales y modales en general. Una introducción detallada de μ-calculus puede encontrarse en [Sch04]. En esta sección, expondremos las definiciones básicas de este formalismo, ya que la utilizaremos como framework para interpretar las lógicas PDL y DTL.

Definición 4.1 *Una signatura o vocabulario es un par* $\langle \Phi, \Delta \rangle$, *donde* Φ *es un*

conjunto finito de variables proposicionales y Δ es un conjunto finito de nombres de acciones.

Definición 4.2 *Dada una signatura $\langle \Phi_1, \Delta_1 \rangle$[1] y un conjunto V de variables, el conjunto Φ_μ de fórmulas $\mu-$calculus se define como:*

- $\Phi_1 \subseteq \Phi_\mu$
- $V \subseteq \Phi_\mu$
- *si $\varphi, \varphi_1, \varphi_2 \in \Phi_\mu$, entonces $\varphi_1 \wedge \varphi_2 \in \Phi_\mu$ y $\neg\varphi \in \Phi_\mu$*
- *si $\varphi \in \Phi_\mu$ y $\alpha \in \Delta_1$, entonces $\langle\alpha\rangle\varphi \in \Phi_\mu$ y $[\alpha]\varphi \in \Phi_\mu$*
- *si $\varphi \in \Phi_\mu$, entonces $\mu R.\varphi \in \Phi_\mu$ y $\nu R.\varphi \in \Phi_\mu$.*

Las variables ligadas deben aparecer bajo un número par de negaciones.

Definición 4.3 *Los modelos para una lógica μ-calculus son estructuras de Kripke etiquetadas. Más precisamente, dada una signatura $\langle \Phi_1, \Delta_1 \rangle$, un modelo es una tupla $M_\mu = \langle S, T, L \rangle$, donde:*

- *S es un conjunto de estados.*
- *L es una función $L : \Phi_1 \to \wp(S)$ que asigna a cada proposición el conjunto de estados donde es verdadera.*
- *T es una función $T : \Delta_1 \to \wp(S \times S)$ transición donde, dada una acción, devuelve una relación binaria entre estados. Denotaremos $s \xrightarrow{a} s'$ si $(s, s') \in T(a)$.*

Definición 4.4 *Dado un modelo M_μ, un estado $s \in S$ y una fórmula φ, que no tiene variables libres, decimos que φ se satisface en el estado s para el modelo M ($s, M \models_\mu \varphi$) si y sólo si $s \in [\![\varphi]\!]_{M_\mu}\rho$, donde ρ representa el estado de las variables (a cada variable le asigna un valor) y la interpretación $[\![\varphi]\!]_{M_\mu}\rho$ está definida recursivamente como sigue:*

- $[\![p]\!]_{M_\mu}\rho = L(p)$ *para* $p \in \Phi_1$,
- $[\![R]\!]_{M_\mu}\rho = \rho(R)$ *para* $R \in V$,
- $[\![\neg\varphi]\!]_{M_\mu}\rho = S - [\![\varphi]\!]_{M_\mu}\rho$,
- $[\![\varphi \wedge \psi]\!]_{M_\mu}\rho = [\![\psi]\!]_{M_\mu}\rho \cap [\![\varphi]\!]_{M_\mu}\rho$,
- $[\![\langle a\rangle\varphi]\!]_{M_\mu}\rho = \{s \in S \mid \exists t[s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_{M_\mu}\rho]\}$,

[1]Utilizaremos el subíndice 1 cuando nos refiramos a signaturas de μ-calculus para distinguirlos de aquellos para fórmulas deónticas.

- $[\![[a]\varphi]\!]_{M_\mu}\rho = \{s \in S \mid \forall t[s \xrightarrow{a} t \wedge t \in [\![\varphi]\!]_{M_\mu}\rho]\}$,
- $[\![\mu R.\varphi]\!]_{M_\mu}\rho$ *es el mínimo punto fijo de la función* $\tau : \wp(S) \to \wp(S)$*, definida como:*
$$\tau(T) = [\![\varphi]\!]_{M_\mu}\rho[R \mapsto T]^2,$$
- $[\![\nu R.\varphi]\!]_{M_\mu}\rho$ *definida igual al anterior, pero utilizando el máximo punto fijo.*

Utilizaremos $[\![\varphi]\!]_M$ en lugar de $[\![\varphi]\!]_{M_\mu}$ cuando no pueda haber confusión.

4.2. Presentación de la lógica

4.2.1. Lógica Deóntica Proposicional (PDL)

PDL [CM09a] es una lógica deóntica proposicional con acciones y con operadores booleanos sobre las acciones. Al igual que en μ-calculus la signatura consiste en *acciones* y *proposiciones*.

Definición 4.5 *Una signatura o vocabulario de PDL es un par* $\langle \Phi, \Delta \rangle$*, donde* Φ *es un conjunto finito de variables proposicionales y* Δ *es un conjunto finito de nombres de acciones.*

Los nombres de acciones son utilizados para describir los eventos que pueden ocurrir durante la ejecución de un sistema. Intuitivamente, los eventos son identificados con los cambios de estado. Las acciones se pueden componer utilizando operadores de acciones y algunas constantes: $\emptyset$ representa la acción de abort, $\mathbf{U}$ sirve para representar la ejecución de cualquier evento, $\sqcup$ representa la elección no determinista de dos acciones, $\sqcap$ representa la ejecución paralela de dos acciones y, dada una acción α, $\neg\alpha$ denota la ejecución de cualquier evento que no esté representado por α (complemento). Dado un conjunto de acciones Δ_0, el conjunto de términos de acciones Δ se define como la clausura transitiva de Δ_0 utilizando los operadores antes mencionados. Formalmente podemos definir el conjunto de acciones compuestas Δ de la siguiente manera:

Definición 4.6 *Dada una signatura* $\langle \Phi_0, \Delta_0 \rangle$*, el conjunto* Δ *se define como:*

- $\Delta_0 \subseteq \Delta$,
- *Si* $\alpha \in \Delta$ *y* $\beta \in \Delta$*, entonces* $\alpha \sqcup \beta$*,* $\alpha \sqcap \beta \in \Delta$*,* $\overline{\alpha} \in \Delta$*,*
- *ningún otro elemento pertenece a* Δ*.*

Definición 4.7 *Dada una signatura* $\langle \Phi_0, \Delta_0 \rangle^3$*, el conjunto* Φ *de* fórmulas *sobre esta signatura se define como el mínimo conjunto que satisfaga lo siguiente:*

[2] $(\rho[R \mapsto T])$ es la asignación ρ "actualizada" con la sustitución $R \mapsto T$, i.e., mapea todos los elementos establecidos en ρ, con excepción de R que es mapeado a T.

[3] Utilizaremos el subíndice 0 cuando nos refiramos a signaturas deónticas (PDL o DTL).

- $\Phi_0 \subseteq \Phi$,
- $\top, \bot \in \Phi$,
- *si* $\alpha, \beta \in \Delta$, *entonces* $\alpha =_{act} \beta \in \Phi$,
- *si* $\varphi, \psi \in \Phi$, *entonces* $\varphi \wedge \psi \in \Phi$ *y* $\neg\varphi \in \Phi$,
- *si* $\varphi \in \Phi$ *y* $\alpha \in \Delta$, *entonces* $\langle\alpha\rangle\varphi \in \Phi$,
- *si* $\alpha \in \Delta$, *entonces* $P(\alpha) \in \Phi$, $P_w(\alpha) \in \Phi$.

De ahora en adelante utilizaremos letras griegas para las variables de acciones y letras romanas minúsculas como variables proposicionales.

Los modelos de PDL se describen mediante *estructuras deónticas* que son esencialmente estructuras de Kripke etiquetadas, donde se identifica como *permitidas* a un conjunto de transiciones. Podríamos pensar, entonces, a cada transición con uno de dos colores: verde, representando transiciones permitidas, y rojas, representando transiciones prohibidas o de falla.

Un punto interesante de este formalismo es el uso de álgebras Booleanas para darle semántica a las acciones y sus operadores. En particular, los átomos de estas álgebras se pueden pensar como acciones que contienen toda la información sobre el comportamiento posible de las acciones. Para ello será de utilidad el siguiente conjunto:

Definición 4.8 *Dada una signatura* $\langle\Phi_0, \Delta_0\rangle$, *donde* $\Delta_0 = \{a_0, \ldots, a_n\}$. *Se define el siguiente conjunto de* ***términos atómicos****:*

$$\mathcal{A}(\Delta_0) = \{*a_0 \sqcap \cdots \sqcap *a_n \mid *a_i = a_i \text{ o } *a_i = \overline{a_i} \text{ para todo } i\}$$

Observar que cada término de acción en $\mathcal{A}(\Delta)$, que llamamos términos atómicos, expresa si cada primitiva de acción en Δ_0 es ejecutuda o no, cuando la misma se ejecuta.

Además de la interpretación de los términos de acciones, las lógicas deónticas requieren distinguir eventos normales de anormales para poder modelar situaciones de permiso, obligación y prohibición de acciones determinadas. Esto da lugar a una variante de las estructuras de Kripke, llamadas estructuras deónticas, que consisten básicamente en estructuras de Kripke donde se incorpora la noción de *eventos permitidos*. La otra característica, es que las etiquetas de cada transición no son acciones de Δ, si no un conjunto de eventos con los que se interpretan los términos de acciones.

Definición 4.9 *Dada una signatura* $\langle\Phi_0, \Delta_0\rangle$, *una* ***estructura deóntica*** M *es una tupla* $\langle\mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P}\rangle$, *donde:*

- $\mathcal{W}$ *es un conjunto de estados,*

- $\mathcal{E}$ *es un conjunto no vacío de eventos.*
- $\mathcal{R} : \mathcal{E} \to \mathcal{W} \to \mathcal{W}$ *es una función en la cual por cada evento* $e \in \mathcal{E}$ *devuelve una función* $\mathcal{R}(e) : \mathcal{W} \to \mathcal{W}$. *Denotaremos como* $w \stackrel{e}{\to} w'$ *cuando* $(\mathcal{R}(e))(w) = w'$.
- $\mathcal{I}$ *es una* función interpretación, *tal que:*
  - *para cada* $p \in \Phi_0$ *se cumple* $\mathcal{I}(p) \subseteq \mathcal{W}$,
  - *for each* $\alpha \in \Delta_0$ *se cumple* $\mathcal{I}(\alpha) \subseteq \mathcal{E}$;

  Además la función $\mathcal{I}$ *debe cumplir las siguientes propiedades:*

  I.1 $|\mathcal{I}(\delta)| \leq 1$, *para todo* $\delta \in \mathcal{A}(\Delta_0)$,
  I.2 $\mathcal{E} = \bigcup_{\alpha_i \in \Delta_0} \mathcal{I}(\alpha_i)$.
- $\mathcal{P} \subseteq \mathcal{W} \times \mathcal{E}$ *es una relación que indica, para cada estado, el conjunto de eventos que están permitidos.*

Donde la función interpretación $\mathcal{I}$ *es extendida a cualquier término de acciones como sigue:*

- $\mathcal{I}(\alpha \sqcup \beta) = \mathcal{I}(\alpha) \cup \mathcal{I}(\beta)$,
- $\mathcal{I}(\alpha \sqcap \beta) = \mathcal{I}(\alpha) \cap \mathcal{I}(\beta)$,
- $\mathcal{I}(\overline{\alpha}) = \mathcal{E} - \mathcal{I}(\alpha)$,
- $\mathcal{I}(\emptyset) = \emptyset$,
- $\mathcal{I}(\mathbf{U}) = \mathcal{E}$.

La función *interpretación* $\mathcal{I}$ está definida sobre dos dominios: el de proposiciones atómicas y el de fórmulas de acciones. Sobre el dominio de las proposiciones atómicas $\mathcal{I}(p)$ establece el conjunto de estados donde la proposición p es válida. La *interpretación* de cada acción α es un conjunto de eventos. Intuitivamente, una acción se ejecuta cuando se ejecuta alguno de los eventos asociados a ésta. Las restricciones que debe cumplir $\mathcal{I}$ establecen básicamente que debe haber una relación uno a uno entre eventos y subconjuntos de acciones, es decir, que a cada evento lo podemos asociar con un único conjunto de acciones que se corresponde con el término atómico $\delta \in \mathcal{A}(\Delta_0)$ que lo genera. Por otro lado, la relación $\mathcal{P}$ establece qué eventos están permitidos y cuáles no para cada estado del modelo; lo que incorpora la característica deóntica al modelo.

Definición 4.10 *Dada una estructura deóntica* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ *y un estado* $w \in \mathcal{W}$, *la satisfactibilidad de una fórmula se define de la siguiente manera:*

- $w, M \models_{PDL} p \iff w \in \mathcal{I}(p)$ *con* $p \in \Phi_0$,
- $w, M \models_{PDL} \alpha =_{act} \beta \iff \mathcal{I}(\alpha) = \mathcal{I}(\beta)$,
- $w, M \models_{PDL} \neg\varphi \iff$ *no ocurre que* $w, M \models_{PDL} \varphi$,
- $w, M \models_{PDL} \varphi_1 \wedge \varphi_2 \iff w, M \models_{PDL} \varphi_1$ *y* $w, M \models_{PDL} \varphi_2$,
- $w, M \models_{PDL} \langle\alpha\rangle\varphi \iff$ *existe algún* $w' \in \mathcal{W}$ *y* $e \in \mathcal{I}(\alpha)$ *tal que* $w \xrightarrow{e} w'$ *y* $w', M \models_{PDL} \varphi$,
- $w, M \models_{PDL} P(\alpha) \iff$ *para todo* $e \in \mathcal{I}(\alpha)$, *se tiene* $\mathcal{P}(w, e)$,
- $w, M \models_{PDL} P_w(\alpha) \iff$ *existe algún* $e \in \mathcal{I}(\alpha)$ *tal que* $\mathcal{P}(w, e)$.

En esta definición, se puede ver que el "color" de las transiciones dadas por una estructura deóntica es capturada por la relación $\mathcal{P}$. Contamos con dos operadores deónticos para expresar permiso: el standard P y el "débil" P_w (del inglés weak). La obligación $\mathbf{O}()$ se define en términos de las otras dos:

$$\mathbf{O}(\alpha) = \mathbf{P}(\alpha) \ \wedge \ \neg\mathbf{P}_w(\overline{\alpha}).$$

4.2.2. Lógica Deóntica Temporal DTL

La lógica proposicional deóntica, PDL, que se introdujo en la sección anterior, tiene operadores deónticos para expresar permisos y obligaciones. En la lógica deóntica temporal, DTL, se incorporan operadores temporales que, combinados con los otros, permiten expresar propiedades asociadas a la tolerancia a fallas y, concretamente, nos permiten razonar sobre diferentes ejecuciones del sistema. La semántica de los operadores incorporados a DTL se corresponde con la de los operadores tradicionales de una lógica CTL. Formalmente:

Definición 4.11 *Dada una signatura de PDL* (Φ_0, Δ_0), *un conjunto de fórmulas temporales sobre esta signatura se define como el mínimo conjunto* Φ_T *que satisface:*

- $\Phi \subseteq \Phi_T$,
- *si* $\varphi, \psi \in \Phi_T$, *entonces* $\varphi \wedge \psi \in \Phi_T$ *y* $\neg\varphi \in \Phi_T$,
- *si* $\varphi, \psi \in \Phi_T$, *entonces* $\mathsf{AG}(\varphi) \in \Phi_T, \mathsf{AN}(\varphi) \in \Phi_T, \mathsf{A}(\varphi \, \mathcal{U} \, \psi) \in \Phi_T, \mathsf{E}(\varphi \, \mathcal{U} \, \psi) \in \Phi_T$.

Donde Φ *representa el conjunto de las fórmulas PDL 4.7 y* Δ *el conjunto de los términos de acciones definidos en 4.6.*

Otros operadores de CTL se pueden definir a partir de los ya definidos como se hace usualmente. Estos operadores temporales nos habilitan a razonar sobre trazas de ejecución. A continuación definimos formalmente lo que esto significa.

Definición 4.12 *Dada una estructura $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ y un estado inicial w, una **traza** es una secuencia etiquetada $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots$ de estados y eventos tal que $s_0 = w$ y para cada i se cumple $s_i \xrightarrow{e_i} s_{i+1} \in \mathcal{R}$.*

El conjunto de todas las trazas de w se denota $\Sigma(w)$.

Dada una traza π, utilizaremos la siguiente notación para referirnos a los estados, eventos y subtrazas de una traza.

- $\pi.i = s_i$,
- $\pi^{\rightarrow}.i = e_i$,
- $\pi[i, j]$ (donde $i \leq j$) es una subtraza $s_i \xrightarrow{e_i} \cdots \xrightarrow{e_{j-1}} s_j$,
- decimos que $\pi' \preceq \pi$, si π' es una subtraza inicial o **prefijo** de π; i.e. $s_0 \xrightarrow{e'_0} s'_1 \xrightarrow{e'_1} s'_2 \xrightarrow{e'_2} \dots s'_k \preceq s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \dots$ sii $s'_i = s_i$ para todo $i \leq k$ y $e'_i = e_i$ para todo $i < k$.

A continuación definimos cuándo una fórmula DTL se satisface. Observar que esta definición extiende la definición de satisfactibilidad de PDL.

Definición 4.13 *Dada una estructura $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$, un estado inicial $w_0 \in \mathcal{W}$ y una traza $\pi \in \Sigma(w_0)$, la relación $\models_{DTL}$ se define como sigue:*

- $\pi, i, M \models_{DTL} \varphi \Longleftrightarrow \pi.i, M \models_{PDL}$, *si* $\varphi \in \Phi$,
- $\pi, i, M \models_{DTL} \neg\varphi \Longleftrightarrow$ *no* $\pi, i, M \models_{DTL} \varphi$,
- $\pi, i, M \models_{DTL} \varphi \wedge \psi \Longleftrightarrow \pi, i, M \models_{DTL} \varphi$ *y* $\pi, i, M \models_{DTL} \psi$,
- $\pi, i, M \models_{DTL} \mathsf{AN}\varphi \Longleftrightarrow$ *para todo* $\pi' \in \Sigma(\pi,0)$ *tal que* $\pi[0, i] \preceq \pi'$ *se tiene que* $\pi', i + 1, M \models_{DTL} \varphi$,
- $\pi, i, M \models_{DTL} \mathsf{AG}\varphi \Longleftrightarrow$ *para todo* $\pi' \in \Sigma(\pi,0)$ *tal que* $\pi[0, i] \preceq \pi'$ *se tiene que* $\forall j \geq i : \pi', j, M \models_{DTL} \varphi$,
- $\pi, i, M \models_{DTL} \mathsf{A}(\varphi\, \mathcal{U}\, \psi) \Longleftrightarrow$ *para todo* $\pi' \in \Sigma(\pi,0)$ *tal que* $\pi[0, i] \preceq \pi'$ *se tiene que* $\exists j \geq i : \pi', j, M \models_{DTL} \psi$ *y* $\forall k : i \leq k < j : \pi', k, M \models_{DTL} \varphi$,
- $\pi, i, M \models_{DTL} \mathsf{E}(\varphi\, \mathcal{U}\, \psi) \Longleftrightarrow$ *existe* $\pi' \in \Sigma(\pi,0)$ *tal que* $\pi[0, i] \preceq \pi'$ *se tiene que* $\exists j \geq i : \pi', j, M \models_{DTL} \psi$ *y* $\forall k : i \leq k < j : \pi', k, M \models_{DTL} \varphi$.

Dada una estructura $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$, un estado inicial $w_0 \in \mathcal{W}$ y una fórmula φ, decimos que $M, w_0 \models \varphi$ si y sólo si $\forall \pi \in \Sigma(w_0) : \pi, 0, M \models_{DTL} \varphi$.

4.2.3. Complejidad de PDL

Haremos un pequeño repaso de la complejidad de PDL. Primero definimos la función Gen que luego utilizamos para nuestra traducción y nos sirve para la prueba de complejidad.

Definición 4.14 *Sea $\langle \Phi_0, \Delta_0 \rangle$ una signatura, y $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ una estructura deóntica sobre la signatura. La función Gen : $\mathcal{E} \to \wp(\Delta_0)$ se define como:*

$$Gen(e) = \{a \mid a \in \Delta_0 \wedge e \in \mathcal{I}(a)\}$$

Lema 4.1 *Gen es inyectiva.*

Prueba: Sean $e, e' \in \mathcal{E}$ eventos tales que $Gen(e) = Gen(e') = \{a_1, a_2, \ldots, a_n\}$. Por I.2 sabemos que $n \neq 0$.

Como $Gen(e) = Gen(e')$ los términos atómicos δ_e y $\delta_{e'}$ que los representan son iguales. Veamos que $e' = e$.

Tomamos $*a_i = a_i$ si $a_i \in Gen(e)$ y $*a_i = \overline{a_i}$ caso contrario. Entonces:

$$\begin{aligned}
& \mathcal{I}(\delta_e) = \mathcal{I}(*a_0 \sqcap \cdots \sqcap *a_n) \\
\equiv & \{ \text{ definición de } \sqcap \} \\
& \mathcal{I}(\delta_e) = \mathcal{I}(*a_0) \cap \cdots \cap \mathcal{I}(*a_n) \\
\equiv & \{ \text{ definición de } *a_i \} \\
& \mathcal{I}(\delta_e) = \bigcap_{i \in Gen(e)} \mathcal{I}(a_i) \cap \bigcap_{j \notin Gen(e)} \mathcal{I}(\overline{a_j}) \\
\equiv & \{ \text{ definición de } \mathcal{I}(\overline{\alpha}) \} \\
& \mathcal{I}(\delta_e) = \bigcap_{i \in Gen(e)} \mathcal{I}(a_i) \cap \bigcap_{j \notin Gen(e)} \mathcal{E} - \mathcal{I}(a_j) \\
\equiv & \{ \text{ prop. conjuntos } \} \\
& \mathcal{I}(\delta_e) = \bigcap_{i \in Gen(e)} \mathcal{I}(a_i) \cap \left(\mathcal{E} - \bigcup_{j \notin Gen(e)} \mathcal{I}(a_j) \right)
\end{aligned}$$

Por la misma definición de $Gen(e)$ vale que $e \in \bigcap_{i \in Gen(e)} \mathcal{I}(a_i)$ y $e \notin \bigcup_{j \notin Gen(e)} \mathcal{I}(a_j)$. Por lo tanto, $e \in \mathcal{I}(\delta_e)$. De la misma manera se cumple $e' \in \mathcal{I}(\delta'_e)$.

Por I.1 tenemos que $|\mathcal{I}(\delta)| \leq 1$, para todo $\delta \in \mathcal{A}(\Delta_0)$. Como $\delta_e = \delta_{e'}$ tiene que valer $e = e'$

□

El siguiente lema establece que, dado un conjunto de primitivas de acciones de tamaño n, el número de eventos de cualquier modelos está acotado por 2^n.

Lema 4.2 *Dado $L = \langle \Phi_0, \Delta_0 \rangle$ y una estructura deóntica $M = \langle \mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P} \rangle$ sobre L, se cumple $|\mathcal{E}| \leq 2^{|\Delta_0|}$. Además, existe un modelo $M = \langle \mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P} \rangle$ tal que $|\mathcal{E}| = 2^{|\Delta_0|}$.*

Prueba:
Como la función $Gen : \mathcal{E} \to \wp(\Delta_0)$ es inyectiva (lema 4.1), tenemos que $|\mathcal{E}| \leq 2^{|\Delta_0|}$.

Ahora consideremos el conjunto $E' = \{e_s \mid s \in 2^{\Delta_0}\}$ y la interpretación $\mathcal{I}'(a) = \{e_s \in 2^{\Delta_0} \mid a \in s\}$. Esta interpretación satisface las condiciones $I.1$ e $I.2$ y, además, $|E'| = 2^{|\Delta_0|}$, con lo que probamos que existe una interpretación con $2^{|\Delta_0|}$ eventos.

□

El siguiente teorema utiliza una implementación de estructuras deónticas en términos de estructuras de datos computacionales para esbozar un procedimiento de model checking para PDL .

Teorema 4.3 *Dada una estructura deóntica M, un mundo w y una fórmula PDL φ, la relación $M, w \vDash \varphi$ puede ser decidida en tiempo $\mathcal{O}(|M| * |\varphi|)$, donde $|M|$ es el tamaño del modelo y M y $|\varphi|$ el tamaño de la fórmula φ.*

Prueba: Para la prueba presentamos una representación de M con estructuras de datos computacionales. Para cada proposición p_i consideremos un arreglo de bits de tamaño $|\mathcal{W}|$, que represente, para cada estado o mundo de $\mathcal{W}$, si la proposición p_i se cumple (es verdadero) o no en ese mundo. De manera similar, para cada acción primitiva a, consideramos un arreglo de bits de largo $|\mathcal{E}|$, que identifique los eventos que pertenecen a la interpretación de la acción. Para cada estado w también tenemos una matriz de tamaño $|\mathcal{E}|$ indicando los eventos habilitados en cada estado. Para la relación $\mathcal{R}$ utilizamos R, una matriz de adyancencia de tamaño $|\mathcal{W}|$: cada entrada de $R[w]$ tiene una lista de pares, digamos $(w, e) \in \mathcal{W} \times \mathcal{E}$, que captura el hecho de que $(w, w', e) \in \mathcal{R}$. Notar que para cada entrada $R[w]$, la lista tiene como máximo largo $|\mathcal{E}|$. La relación $\mathcal{P}$ se implementa de manera similar, con un arreglo de tamaño $|\mathcal{W}|$, donde cada entrada contiene un lista representando el conjunto de eventos permitidos en ese estado.

El algoritmo de model checking consiste en lo siguiente. Dada una fórmula φ, contruimos el árbol de sintaxis que la representa y luego calculamos el conjunto $SAT(\varphi)$ de estados que satisfacen φ. Este es un procedimiento de model checking *global*; para verificar $w, M \vDash \varphi$ sólo necesitamos verificar si $w \in SAT(\varphi)$. Una vez obtenido el árbol de sintaxis de la fórmula, calculamos los conjuntos SAT de cada subfórmula en orden bottom-up (del árbol de sintaxis). Para las hojas (variables proposicionales), los conjuntos se calculan recorriendo los arreglos de cada variable; este paso requiere tiempo $\mathcal{O}(|\mathcal{W}|)$ para cada proposición. Asumiendo que los estados que satisfacen la fórmula son almacenados en arreglos de bits de largo $|\mathcal{W}|$, cada operador booleano $\neg, \wedge, \vee$ se puede computar usando los operadores estándares de conjuntos en tiempo $\mathcal{O}(|\mathcal{W}|)$. Cuando se encuentra una acción α en el proceso, el conjunto de eventos que pertencen a la interpretación de la acción se calcula como detallamos

a continuación. Las acciones atómicas se calculan directamente utilizando los arreglos, los otros operadores se calculan utilizando operaciones de conjuntos sobre los arreglos de bits en tiempo $\mathcal{O}(|\mathcal{E}|)$. Este proceso lleva tiempo $\mathcal{O}(|\alpha|*|\mathcal{E}|)$, el cual está acotado por $\mathcal{O}(|\varphi|*|\mathcal{E}|)$.

Los casos interesantes son aquellos que involucran modalidades y fórmulas deónticas. Si tenemos $\langle\alpha\rangle\varphi$, entonces $SAT(\langle\alpha\rangle\varphi') = \{w \mid \exists(e,w') \in \mathcal{I}(\alpha) \times \mathcal{W} : w' \in SAT(\varphi')\}$.Este conjunto puede ser calculado en tiempo lineal utilizando las matrices correspondientes a α y $\mathcal{R}$ y las subfómulas ya calculadas, en un tiempo $\mathcal{O}(|\alpha|*|\mathcal{E}|+|\mathcal{R}|)$. De manera similar, el procesamiento de fórmulas deónticas del tipo $\mathsf{P}(\alpha)$ o $\mathsf{P}_w(\alpha)$ se puede calcular en tiempo $\mathcal{O}(|\alpha|*|\mathcal{E}|+|\mathcal{P}|)$. Por lo tanto, teniendo en cuenta que $|\mathcal{R}|$ y $|\mathcal{P}|$ están acotadas por el tamaño del modelo y $|\alpha|*|\mathcal{E}|$ está acotada por $|\varphi|*|\mathcal{E}|$, tenemos en el peor caso que $SAT(\varphi)$ puede ser calculada en tiempo $\mathcal{O}(|\varphi|*|M|)$.

□

Probamos que el model checking de una fórmula PDL puede ser decidida en tiempo lineal en relación al tamaño de la fórmula y el modelo. Es importante destacar que el tamaño del modelo puede ser exponencial con respecto al número de acciones primitivas (lema 4.2). Este hecho es bien conocido en model checking como el problema de la explosión de estados que afecta el model checking en general [CBJM96] y en particular nuestra aproximación.

4.3. Primera Traducción

4.3.1. Traducción de PDL a μ-calculus

Como explicamos en 4.1, el propósito de esta caracterización es utilizar herramientas de model checking de μ-calculus para verificar propiedades para sistemas tolerantes a fallas expresadas en PDL y DTL.

A través de la definición de la función Tr esperamos reducir el problema de model checking de PDL a model checking de μ-calculus. Formalmente, cada vez que

$$Tr^m(w,M) \not\models_\mu Tr(\varphi)$$

entonces se debe cumplir que

$$w,M \not\models_{DPL} \varphi$$

y vice versa. Por lo tanto la traducción Tr de PDL a μ-calculus debe satisfacer:

$$w,M \models_{PDL} \varphi \Longleftrightarrow Tr^m(w,M) \models_\mu Tr(\varphi).$$

En teoría, las traducciones entre lógicas que satisfacen esta propiedad se llaman *forward morphisms* [GR02]. Como mostraremos más adelante, esta propiedad permite garantizar que el problema de model checking se preserva con la traducción.

Comencemos formalmente con la traducción. En primer lugar tenemos que pensar cómo representar los modelos de DPL (estructuras deónticas) en términos de modelos en μ-calculus (estructuras de Kripke). Una de las diferencias consiste en el coloreo de los eventos $\mathcal{P}$ que puede ser representado fácilmente con nuevas proposiciones atómicas en la estructura de Kripke. La otra diferencia es que la semántica que se le da a las acciones en cada modelo es completamente diferente. En una estructura de Kripke etiquetada, las transiciones están etiquetadas con las acciones definidas en la signatura sobre la que está definida el modelo. En cambio, la transiciones entre estados en una estructura deóntica están etiquetadas con eventos que derivan de la interpretación de las acciones. La función Gen que definimos en 4.14 nos ayuda a relacionar las etiquetas representadas por un evento e en una estructura deóntica, con las etiquetas en una estructura de Kripke. Intuitivamente, *Gen*(e) corresponde al conjunto maximal de acciones cuya ejecución en paralelo genera el evento e. Observar que otra forma de representar el conjunto *Gen*(e) para un evento determinado e es con un término atómico $\delta_e \in \mathcal{A}(\Delta_0)$ definido de la siguiente manera: Si $\Delta_0 = \{a_0, a_1, ..., a_n\}$, tomamos $\delta_e = *a_0 \sqcap \cdots \sqcap *a_n$ con $*a_i = a_i$ si $a_i \in$ *Gen*(e) y $*a_i = \overline{a_i}$ caso contrario.

Definimos ahora la traducción Tr^m de los modelos de PDL a los modelos de μ-calculus. Notar que esta definición involucra también la definición de la signatura $\langle \Phi_1, \Delta_1 \rangle$ sobre la que se definirá la estructura de Kripke.

Definición 4.15 *Dada una signatura* $\langle \Phi_0, \Delta_0 \rangle$*, una estructura deóntica* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ *y un estado* $w \in \mathcal{W}$ *definimos la función* Tr^m *como:*

$$Tr^m(w, M) = w, M_\mu,$$

donde $M_\mu = \langle S, T, L \rangle$ *es un modelo de la signatura* $\langle \Phi_1, \Delta_1 \rangle$ *tal que:*

- $\Delta_1 = \wp(\Delta_0)$,
- $\Phi_1 = \Phi_0 \cup \{P_a \mid a \in \Delta_1\} \cup \{E_a \mid a \in \Delta_1\}$,
- $S = \mathcal{W}$,
- $T = \{w \stackrel{Gen(e)}{\longrightarrow} w' \mid w \stackrel{e}{\longrightarrow} w' \in \mathcal{R}\}$,
- $L(p) = I(p)$*, para todo* $p \in \Phi_0$,
- $L(P_a) = \{w \mid \exists e \in \mathcal{E} \ : \ (w, e) \in \mathcal{P} \wedge Gen(e) = a\}$*, para todo* $a \in \Delta_1$,
- $L(E_a) = \{s \mid \exists s' \ : \ s \stackrel{a}{\longrightarrow} s' \in T\}$.

En esta traducción el conjunto de estados y de transiciones se mantiene, aunque las etiquetas asociadas a cada transición se codifican de manera diferente. Específicamente, a cada evento se lo representa con el conjunto de acciones que ejecutadas paralelamente lo generan o, visto de otra fórma, cada

evento se representa con un término atómico (definición 4.8). Además, agregando proposiciones atómicas se pretende codificar, por un lado, el conjunto de eventos que se pueden ejecutar desde un determinado estado (E_a) y, por otro, cuáles de esas transiciones están permitidas (P_a).

Para la traducción de las fórmulas, teniendo en cuenta que las fórmulas PDL usan símbolos de Δ_0 para representar acciones mientras que en μ-calculus las fórmulas usan nombres de Δ_1 (i.e., subconjuntos de Δ_0), definimos la función *Set* para relacionar ambos conjuntos.

Definición 4.16 *La función Set* : $\Delta_0 \to \wp(\Delta_1)$ *se define como*

$$Set(\alpha) = \{a \mid a \in \Delta_1 \land \alpha \in a\}.$$

Podemos extender la función Set a todos los términos de acciones recursivamente.

- $Set(\emptyset) = \emptyset$,
- $Set(\boldsymbol{U}) = \Delta_1$,
- $Set(\neg\alpha) = \Delta_1 - Set(\alpha)$,
- $Set(\alpha \sqcup \beta) = Set(\alpha) \cup Set(\beta)$,
- $Set(\alpha \sqcap \beta) = Set(\alpha) \cap Set(\beta)$.

Finalmente estamos en condiciones de definir la función Tr, que traduce fórmulas PDL a fórmulas de μ-calculus.

Definición 4.17 *La función traducción Tr se define como sigue:*

- $Tr(p) = p$, *para cada* $p \in \Phi_0$,
- $Tr(\top) = \top$,
- $Tr(\bot) = \bot$,
- $Tr(\neg\varphi) = \neg Tr(\varphi)$,
- $Tr(\varphi_1 \land \varphi_2) = Tr(\varphi_1) \land Tr(\varphi_2)$,
- $Tr(\alpha =_{act} \beta) = \bigwedge_{a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta))} \neg E_a$,
- $Tr(\langle\alpha\rangle\varphi) = \bigvee_{a \in Set(\alpha)} \langle a \rangle Tr(\varphi)$,
- $Tr([\alpha]\varphi) = \bigwedge_{a \in Set(\alpha)} (E_a \to [a]Tr(\varphi))$,
- $Tr(P(\alpha)) = \bigwedge_{a \in Set(\alpha)} (E_a \to P_a)$,
- $Tr(P_w(\alpha)) = \bigvee_{a \in Set(\alpha)} P_a$.

4.3.2. Corrección de Tr

El siguiente teorema establece que la traducción que definimos anteriormente es correcta respecto del problema de model checking. La prueba se realiza por inducción estructural sobre las fórmuas PDL.

Teorema 4.4 *Sea* $\langle \Phi_0, \Delta_0 \rangle$ *una signatura,* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ *una estructura y* $w \in \mathcal{W}$ *un estado, entonces*

$$w, M \models \phi \Leftrightarrow Tr^m(w, M) \models_\mu Tr(\phi).$$

Prueba: Sea $(w, M_\mu) = Tr^m(w, M)$ con $M_\mu = (S, T, L)$ entonces

$$Tr^m(w, M) \models_\mu Tr(\phi) \Leftrightarrow w, M_\mu \models_\mu Tr(\phi) \Leftrightarrow w \in [\![Tr(\phi)]\!]_{M_\mu}$$

Por lo tanto tenemos que probar

$$w, M \models \phi \Leftrightarrow w \in [\![Tr(\phi)]\!]_{M_\mu}$$

La demostración la haremos por inducción estructural sobre la fórmula.

Caso $\phi = p$**:**

$$\begin{aligned}
& w, M \models p \\
\equiv & \{ \text{ def. de } \models \} \\
& w \in \mathcal{I}(p) \\
\equiv & \{ \text{ def. de } Tr^m(M) \} \\
& w \in L(p) \\
\equiv & \{ \text{ def. de } [\![]\!] \} \\
& w \in [\![p]\!] \\
\equiv & \{ \text{ def. de } Tr \} \\
& w \in [\![Tr(p)]\!]
\end{aligned}$$

Caso $\phi = \neg\varphi$**:**

$$\begin{aligned}
& w, M \models \neg\varphi \\
\equiv & \{ \text{ def. de } \models \} \\
& \text{no } w, M \models \varphi \\
\equiv & \{ \text{ Hipótesis Inductiva } \} \\
& \text{no } w \in [\![Tr(\varphi)]\!] \\
\equiv & \{ \text{ def. de } Tr^m(M) \text{ y propiedades de conjuntos } \} \\
& w \in [\![\neg Tr(\varphi)]\!] \\
\equiv & \{ \text{ def. de } Tr \} \\
& w \in [\![Tr(\neg\varphi)]\!]
\end{aligned}$$

Caso $\phi = \varphi \wedge \psi$**:**

$$
\begin{aligned}
& w, M \models \varphi \wedge \psi \\
\equiv & \{ \text{ def. de } \models \} \\
& w, M \models \varphi \wedge w, M \models \psi \\
\equiv & \{ \text{ Ind.Hyp. } \} \\
& w \in [\![Tr(\varphi)]\!] \wedge w \in [\![Tr(\psi)]\!] \\
\equiv & \{ \text{ def. de } [\![]\!] \text{ y prop. de conjuntos } \} \\
& w \in [\![Tr(\varphi) \wedge Tr(\psi)]\!] \\
\equiv & \{ \text{ def. de } Tr \} \\
& w \in [\![Tr(\varphi \wedge \psi)]\!]
\end{aligned}
$$

Caso $\phi = \alpha =_{act} \beta$**:**

$$
\begin{aligned}
& w, M \models \alpha =_{act} \beta \\
\equiv & \{ \text{ def. de } \models \} \\
& \mathcal{I}(\alpha) =_{Set} \mathcal{I}(\beta) \\
\equiv & \{ \text{ Gen. es inyectiva } \} \\
& \{Gen(e) \mid e \in \mathcal{I}(\alpha)\} = \{Gen(e) \mid e \in \mathcal{I}(\beta)\} \\
\equiv & \{ \text{ Lemma 4.8 } \} \\
& \{a \mid a \in Set(\alpha) \wedge E_a\} = \{a \mid a \in Set(\beta) \wedge E_a\} \\
\equiv & \{ \text{ conjuntos } \} \\
& Set(\alpha) \cap \{a \mid E_a\} = Set(\beta) \cap \{a \mid E_a\} \\
\equiv & \{ \text{ conjuntos } \} \\
& \forall a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta)) \rightarrow \ a \notin \{a \mid E_a\} \\
\equiv & \{ \text{ conjuntos } \} \\
& \forall a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta)) \rightarrow \ \neg E_a \\
\equiv & \{ \ L(E_a) \text{ no depende del estado } \} \\
& \bigwedge_{a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta))} w \notin L(E_a) \\
\equiv & \{ \text{ def. de } [\![]\!] \} \\
& \bigwedge_{a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta))} w \in [\![\neg E_a]\!] \\
\equiv & \{ \text{ def. de } [\![]\!] \} \\
& w \in [\![\bigwedge_{a \in (Set(\alpha) \cup Set(\beta)) - (Set(\alpha) \cap Set(\beta))} \neg E_a]\!]
\end{aligned}
$$

Caso $\phi = \langle \alpha \rangle \varphi$**:**

$$
w, M \models \langle \alpha \rangle \varphi
$$

$\equiv$ { def. de $\models$ }
existe $w' \in \mathcal{W}$ y $e \in \mathcal{I}(\alpha)$ tq $w \overset{e}{\to} w' \in \mathcal{R}$ y $w', M \models \varphi$
$\equiv$ { Ind. Hyp. }
existe $w' \in \mathcal{W}$ y $e \in \mathcal{I}(\alpha)$ tq $w \overset{e}{\to} w' \in \mathcal{R}$ y $w' \in [\![Tr(\varphi)]\!]$
$\equiv$ { Lema 4.10 }
existe $w' \in \mathcal{W}$ y $a \in Set(\alpha)$ tq $w \overset{a}{\to} w' \in \mathrm{T}$ y $w' \in [\![Tr(\varphi)]\!]$
$\equiv$ { propiedades de conjuntos }
$w \in \bigcup_{a \in Set(\alpha)} \{s \in S \mid \text{ existe } w' \text{ tq } s \overset{a}{\to} w' \in \mathrm{T} \text{ y } w' \in [\![Tr(\varphi)]\!]\}$
$\equiv$ { def. de $[\![]\!]$ }
$w \in \bigcup_{a \in Set(\alpha)} [\![\langle a \rangle Tr(\varphi)]\!]$
$\equiv$ { def. de $[\![]\!]$ }
$w \in [\![\bigvee_{a \in Set(\alpha)} \langle a \rangle Tr(\varphi)]\!]$
$\equiv$ { def. de Tr }
$w \in [\![Tr(\langle \alpha \rangle \varphi)]\!]$

Caso $\phi = P(\alpha)$:

$w, M \models P(\alpha)$
$\equiv$ { def. de $\models$ }
$\forall e \in \mathcal{I}(\alpha) \ : \ (w, e) \in \mathcal{P}$
$\equiv$ { Lema 4.8 }
$\forall e \ : \ a = Gen(e) \wedge a \in Set(\alpha) \wedge E_a \to (w, e) \in \mathcal{P}$
$\equiv$ { def. de P_a }
$\forall e \ : \ a = Gen(e) \wedge a \in Set(\alpha) \wedge E_a \to w \in L(P_a)$
$\equiv$ { Gen inyectiva, y biyectiva si restringimos su codominio (E_a) }
$\forall a \ : \ a \in Set(\alpha) \wedge E_a \to w \in L(P_a)$
$\equiv$ { $Set(\alpha)$ finito }
$\bigwedge_{a \in Set(\alpha)} E_a \to w \in L(P_a)$
$\equiv$ { def. de $L(E_a)$, no depende del estado w }
$\bigwedge_{a \in Set(\alpha)} w \in L(E_a) \to w \in L(P_a)$
$\equiv$ { def. de $[\![]\!]$ }
$\bigwedge_{a \in Set(\alpha)} w \in [\![E_a]\!] \to w \in [\![P_a]\!]$
$\equiv$ { def. de $[\![]\!]$ }

$$\bigwedge_{a \in Set(\alpha)} w \in [\![E_a \to P_a]\!]$$
$$\equiv \{ \text{ def. de } [\![]\!], \text{ conjuntos } \}$$
$$w \in [\![\bigwedge_{a \in Set(\alpha)} E_a \to P_a]\!]$$
$$\equiv \{ \text{ def. de } Tr \}$$
$$w \in [\![Tr(P(\alpha))]\!]$$

Caso $\phi = P_w(\alpha)$:

$$w, M \models P_w(\alpha)$$
$$\equiv \{ \text{ definición de } \models \}$$
$$\exists e \in \mathcal{I}(\alpha) \; : \; (w, e) \in \mathcal{P}$$
$$\equiv \{ \text{ Lema 4.11 } \}$$
$$\exists a \in Set(\alpha) \; : \; w \in L(P_a)$$
$$\equiv \{ \; Set(\alpha) \text{ finito } \}$$
$$\bigvee_{a \in Set(\alpha)} w \in L(P_a)$$
$$\equiv \{ \text{ definición de } L(P_a) \}$$
$$w \in \bigcup_{a \in Set(\alpha)} L(P_a)$$
$$\equiv \{ \text{ definición de } [\![]\!] \}$$
$$w \in \bigcup_{a \in Set(\alpha)} [\![P_a]\!]$$
$$\equiv \{ \text{ definición de } [\![]\!] \}$$
$$w \in [\![\bigvee_{a \in Set(\alpha)} P_a]\!]$$
$$\equiv \{ \text{ definición de } Tr \}$$
$$w \in [\![Tr(P_w(\alpha))]\!]$$

□

4.3.3. Traducción de DTL a μ-calculus

Para la traducción de DTL, que incorpora operadores temporales a partir de DTL, necesitamos extender la definición de Tr para que los incluya. El primer paso es considerar la traducción de los modelos; esto lo hacemos a través de Tr^{tm}. Para la semántica de lógicas temporales es necesario identificar los estados iniciales, por lo tanto la traducción también involucra otra función Tr^s.

Definición 4.18 *Sea* (Φ_0, Δ_0) *una signatura,* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$ *una estructura sobre esa signatura y* $w_0 \in \mathcal{W}$ *es estado inicial de* M*. Las funciones* Tr^{tm} *y* Tr^{s} *se definien como:*

$$\begin{aligned} Tr^{tm}(M) &= M_\mu \\ Tr^{s}(w) &= w \end{aligned}$$

donde $M_\mu = \langle S, T, L \rangle$ *es el modelo para* μ*-calculus definido sobre la signatura* $\langle \Phi_1, \Delta_1 \rangle$*, que se obtienen de la siguiente manera:*

- $\Delta_1 = \wp(\Delta_0)$,
- $\Phi_1 = \Phi_0 \cup \{P_a \mid a \in \Delta_1\} \cup \{E_a \mid a \in \Delta_1\}$,
- $S = \{s \mid s \in \mathcal{W}\}$,
- $T = \{s \stackrel{Gen(e)}{\longrightarrow} s' \mid s \stackrel{e}{\longrightarrow} s' \in \mathcal{R}\}$,
- $L(p) = \mathcal{I}(p)$ *para todo* $p \in \Phi_0$,
- $L(P_a) = \{w \mid \exists e \in \mathcal{E} \ : \ (w, e) \in \mathcal{P} \wedge Gen(e) = a\}$ *para todo* $a \in \Delta_1$,
- $L(E_a) = \{s \mid \exists s' \ : \ s \stackrel{a}{\longrightarrow} s' \in T\}$,

Avancemos ahora sobre la traducción de fórmulas DTL a μ-calculus.

Definición 4.19 *La traducción* Tr *de PDL a* μ*-calculus se extiende a DTL de la siguiente manera:*

- $\varphi \in \Phi$, $Tr(\varphi)$ *como la descripta en la definición 4.17,*
- $Tr(\neg\varphi) = \neg Tr(\varphi)$,
- $Tr(\varphi \wedge \psi) = Tr(\varphi) \wedge Tr(\psi)$,
- $Tr(\mathsf{AN}\varphi) = Tr([U]\varphi)$,
- $Tr(\mathsf{EN}\varphi) = Tr(\langle U \rangle\varphi)$,
- $Tr(\mathsf{AG}\varphi) = \nu R.(Tr(\varphi) \wedge \bigwedge_{a \in \Delta_1} [a]R)$,
- $\mathsf{A}(\varphi \,\mathcal{U}\, \psi) = \mu R.(Tr(\psi) \vee (Tr(\varphi) \wedge \bigwedge_{a \in \Delta_1} [a]R))$,
- $\mathsf{E}(\varphi \,\mathcal{U}\, \psi) = \mu R.(Tr(\psi) \vee (Tr(\varphi) \wedge \bigvee_{a \in \Delta_1} \langle a \rangle R))$.

4.3.4. Corrección de la extensión de Tr

La corrección de esta traducción extendida para DTL se prueba generalizando el teorema de corrección de la traducción de PDL. Para ello necesitamos una función para la traducción de trazas.

Definición 4.20 *La función Tr^p se define como sigue:*

$$Tr^p(w_0 \stackrel{e_0}{\to} w_1 \stackrel{e_1}{\to} w_2 \stackrel{e_2}{\to} \dots) = s_0 \stackrel{Gen(e_0)}{\to} s_1 \stackrel{Gen(e_1)}{\to} s_2 \stackrel{Gen(e_2)}{\to} \dots .$$

Observar que, como Gen es inyectiva, si tenemos una traza π obtenida a partir de la traducción, existe una única traza π' en el modelo original tal que $Tr^p(\pi') = \pi$. Además, por construcción Tr^p is suryectiva.

Lema 4.5 *Tr^p es biyectiva.*

El siguiente teorema nos permite relacionar la validez de una fórmula en μ-calculus con la validez en la lógica deóntica temporal DTL.

Teorema 4.6 *Dados un lenguaje (Φ_0, Δ_0), una estructura $M = (\mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P})$, un estado $w \in \mathcal{W}$ y una fórmula ϕ sobre ese lenguaje, se cumple*

$$w \in [\![Tr(\phi)]\!]_{Tr^{tm}(M)} \text{ si y sólo si } \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \phi \rangle$$

Prueba: Por inducción estructural sobre la fórmula ϕ.

Sea $M_\mu = Tr^{tm}(M)$.

Caso $\phi \in \Phi$

$$\begin{array}{l} \quad \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \phi \rangle \\ \equiv \{ \text{ Def. } \models_{DTL} \} \\ \quad \langle \forall \pi, i : w = Tr^p(\pi).i : \pi.i, M \models_{DPL} \phi \rangle \\ \equiv \{ \ w = Tr^p(\pi).i \ \to \ \pi.i = w \ \} \\ \quad \langle \forall \pi, i : w = Tr^p(\pi).i : w, M \models_{DPL} \phi \rangle \\ \equiv \{ \text{ Término constante } \} \\ \quad w, M \models_{DPL} \varphi \\ \equiv \{ \text{ Teorema 4.12 } \} \\ \quad w \in [\![Tr(\varphi)]\!]_{M_\mu} \end{array}$$

Caso $\phi = \mathsf{AG}\varphi$

Sea $Y = \{w \mid \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models \mathsf{AG}\varphi \rangle\}$. Vamos a demostrar que $Y = [\![Tr(\mathsf{AG}\varphi)]\!]$

$$\begin{aligned} Tr(\mathsf{AG}\varphi) &= \nu R.(Tr(\varphi) \wedge \bigwedge_{a \in \Delta_1} [a]R) \\ &= \text{mayor punto fijo de } \tau \end{aligned}$$

Donde

$$\begin{aligned} \tau(S) &= [\![Tr(\varphi) \wedge \bigwedge_{a\in\Delta_1} [a]R]\!]\ e[R \leftarrow S] \\ \tau(S) &= [\![Tr(\varphi)]\!]\ e[R \leftarrow S] \cap [\![\bigwedge_{a\in\Delta_1} [a]R]\!]\ e[R \leftarrow S] \\ \tau(S) &= [\![Tr(\varphi)]\!]\ e[R \leftarrow S] \cap \bigcap_{a\in\Delta_1} [\![[a]R]\!]\ e[R \leftarrow S] \end{aligned}$$

1. $Y \subseteq [\![Tr(\mathsf{AG}\varphi)]\!]$

   Veamos que Y es un punto fijo de τ.

$$\begin{aligned}
& w \in \tau(Y) \\
\Rightarrow\ & w \in ([\![Tr(\varphi)]\!] \cap \bigcap_{x\in\Delta_1} [\![[x]Y]\!]) \\
\Rightarrow\ & w \in [\![Tr(\varphi)]\!] \wedge \bigwedge_{x\in\Delta_1} w \in [\![[x]Y]\!]) \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \bigwedge_{x\in\Delta_1} \left\langle \forall w' : w \xrightarrow{x} w' : w' \in Y \right\rangle \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \left\langle \forall w' : w \xrightarrow{x} w' : w' \in Y \right\rangle \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \\
& \quad \left\langle \forall w' : w \xrightarrow{x} w' : \langle \forall \pi, i : w' = Tr^p(\pi).i : \pi, i, M \models_{DTL} AG\varphi \rangle \right\rangle \\
\Rightarrow\ & \{ \text{ distrib. , cambio de variable } (i \to i+1) \ \} \\
& \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \\
& \quad \langle \forall w', i, \pi : w = Tr^p(\pi).i \wedge w' = Tr^p(\pi).(i+1) : \pi, i+1, M \models_{DTL} AG\varphi \rangle \\
\Rightarrow\ & \{ \text{ Def. de } AG \ \} \\
& \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \langle \forall w', i, \pi \ : \\
& \quad w = Tr^p(\pi).i \wedge w' = Tr^p(\pi).(i+1) \ : \ \langle \forall \pi', j : \pi[0, i+1] \preceq \pi' \wedge j \geq i+1 : \pi', j, M \models_{DTL} \varphi \rangle \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \langle \forall i, \pi \ : \ w = Tr^p(\pi).i \ : \\
& \quad \langle \forall w', \pi', j : \pi[0, i+1] \preceq \pi' \wedge j \geq i+1 \wedge w' = Tr^p(\pi).(i+1) : \pi', j, M \models_{DTL} \varphi \rangle \rangle \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \langle \forall i, \pi \ : \ w = Tr^p(\pi).i \ : \\
& \quad \langle \forall \pi', j : \pi[0, i] \preceq \pi' \wedge j \geq i+1 : \pi', j, M \models_{DTL} \varphi \rangle \rangle \\
\Rightarrow\ & \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \wedge \langle \forall \pi', j : \pi[0, i] \preceq \pi' \wedge j \geq i+1 : \pi', j, M \models_{DTL} \varphi \rangle \rangle
\end{aligned}$$

$$\Rightarrow\ \langle \forall i, \pi : w = Tr^p(\pi).i : \langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge j \geq i : \pi', j, M \models_{DTL} \varphi \rangle \rangle$$
$$\Rightarrow \{ \text{ Definición de AG } \}$$
$$\langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} AG\varphi \rangle$$
$$\Rightarrow w \in Y$$

Como Y es punto fijo de τ es menor que el máximo punto fijo, por lo que 1. queda demostrado.

2. $Y \supseteq [\![Tr(AG\varphi)]\!]$

   Para esta demostración es necesario definir Y_n

$$Y_n = \{w \mid \langle \forall i, \pi : w = Tr^p(\pi).i : \langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+n : \pi', j, M \models_{DTL} \varphi \rangle \rangle \}$$

   Intuitivamente el conjunto Y_n es el conjunto de estados que garantizan que todo camino que comienza en alguno de esos estados satisface φ en su tramo inicial de logitud n.

   Como consecuencia de esta definición se tiene que $Y = \bigcap_{n:0..\infty} Y_n$.

   Veamos primero que $Y_n \supseteq \tau^{(n+1)}(\top)$. Probaremos la propiedad por inducción en i_0.

   - Caso $n = 0$

$$w \in \tau^{n+1}(\top)$$
$$\Rightarrow w \in \tau(\top)$$
$$\Rightarrow w \in [\![Tr(\varphi)]\!] \wedge \bigwedge_{x\in\Delta_1} w \in [\![[x]\top]\!])$$
$$\Rightarrow w \in [\![Tr(\varphi)]\!] \wedge \bigwedge_{x\in\Delta_1} w \in [\![[x]\top]\!])$$
$$\Rightarrow\ \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \bigwedge_{x\in\Delta_1} \left\langle \forall w' : w \xrightarrow{x} w' : w' \in [\![\top]\!] \right\rangle$$
$$\Rightarrow\ \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \left\langle \forall w' : w \xrightarrow{x} w' :\ True \right\rangle$$
$$\Rightarrow\ \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge\ True$$
$$\Rightarrow\ \left\langle \forall i, \pi : w = Tr^p(\pi).i : \left\langle \forall \pi' : \pi[0,i] \preceq \pi' : \pi', i, M \models_{DTL} \varphi \right\rangle \right\rangle$$
$$\Rightarrow\ \left\langle \forall i, \pi : w = Tr^p(\pi).i : \left\langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+0 : \pi', j, M \models_{DTL} \varphi \right\rangle \right\rangle$$
$$\Rightarrow w \in Y_n$$

   - Caso $n > 0$

$$w \in \tau^{n+1}(\top)$$
$$\Rightarrow w \in \tau(\tau^n(\top))$$
$$\Rightarrow w \in [\![Tr(\varphi)]\!] \wedge \bigwedge_{x\in\Delta_1} w \in [\![[x]\tau^n(\top)]\!])$$

$$\Rightarrow \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \bigwedge_{x \in \Delta_1} \left\langle \forall w' : w \xrightarrow{x} w' : w' \in [\![\tau^n(\top)]\!] \right\rangle$$
$$\Rightarrow \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \left\langle \forall w' : w \xrightarrow{x} w' : w' \in [\![\tau^n(\top)]\!] \right\rangle$$
$$\Rightarrow \{ \text{ HI } \}$$
$$\langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \left\langle \forall w' : w \xrightarrow{x} w' : w' \in Y_{n-1} \right\rangle$$
$$\Rightarrow \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \langle \forall w' : w \xrightarrow{x} w' :$$
$$\langle \forall i, \pi : w' = Tr^p(\pi).i : \langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+n-1 : \pi', j, M \models_{DTL} \varphi \rangle \rangle \rangle$$
$$\Rightarrow \{ \text{ cambio de var } i \to i+1 \}$$
$$\langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge \langle \forall w', i, \pi : w \xrightarrow{x} w' \wedge w' = Tr^p(\pi).(i+1) :$$
$$\langle \forall \pi', j : \pi[0,i+1] \preceq \pi' \wedge i+1 \leq j \leq i+1+n-1 : \pi', j, M \models_{DTL} \varphi \rangle \rangle$$
$$\Rightarrow \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi \rangle \wedge$$
$$\langle \forall i, \pi : w = Tr^p(\pi).i : \langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge i+1 \leq j \leq i+n : \pi', j, M \models_{DTL} \varphi \rangle \rangle$$
$$\Rightarrow \langle \forall i, \pi : w = Tr^p(\pi).i : \langle \forall \pi', j : \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+n : \pi', j, M \models_{DTL} \varphi \rangle \rangle$$
$$\Rightarrow w \in Y_n$$

Veamos lo que queríamos demostrar: $Y \supseteq [\![Tr(AG\varphi)]\!]$

$$w \in \bigcap_{i=0\ldots\infty} \tau^i(\top)$$
$$\Rightarrow w \in \bigcap_{i=1\ldots\infty} \tau^i(\top) \cap \tau^0(\top)$$
$$\Rightarrow w \in \bigcap_{i=0\ldots\infty} \tau^{(i+1)}(\top)$$
$$\Rightarrow \bigwedge_{i=0\ldots\infty} w \in \tau^{(i+1)}(\top)$$
$$\Rightarrow \{ \text{ por la propiedad demostrada anteriormente } \}$$
$$\bigwedge_{n:0..\infty} w \in Y_n$$
$$\Rightarrow w \in \bigcap_{n:0..\infty} Y_n$$
$$\Rightarrow w \in Y$$

Caso $\phi = \mathsf{EG}\varphi$:
La demostración de este caso es análoga al caso $\phi = \mathsf{AG}\varphi$.
Caso $\phi = \mathsf{A}(\varphi\, \mathcal{U}\, \psi)$:
Para este caso se resuelve de manera análoga al caso $\phi = \mathsf{E}(\varphi\, \mathcal{U}\, \psi)$.
Caso $\phi = \mathsf{E}(\psi\, \mathcal{U}\, \varphi)$:
Sea $Y = \{w \mid \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models \mathsf{E}(\psi\, \mathcal{U}\, \varphi) \rangle \}$. Vamos a demostrar que $Y = [\![Tr(\mathsf{E}(\psi\, \mathcal{U}\, \varphi))]\!]$

$$Tr(\mathsf{E}(\psi\, \mathcal{U}\, \varphi)) = \mu R.(Tr(\psi) \vee (Tr(\varphi) \wedge \bigvee_{x \in \Delta_1} \langle x \rangle R))$$

$= \text{menor punto fijo de } \tau$

Donde

$$\begin{aligned}
\tau(S) &= [\![Tr(\psi) \vee \left(Tr(\varphi) \wedge \bigvee_{x\in\Delta_1} \langle x\rangle R\right)]\!]\ e[R \leftarrow S] \\
&= [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap [\![\bigvee_{x\in\Delta_1} \langle x\rangle S]\!]\right) \\
&= [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \bigcup_{x\in\Delta_1} [\![\langle x\rangle S]\!]\right) \\
&= [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \bigcup_{x\in\Delta_1} \{w \mid \left\langle \exists w' : w \xrightarrow{x} w' : w' \in S\right\rangle\right) \\
&= [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \{w \mid \left\langle \exists w' : w \xrightarrow{a} w' : w' \in S\right\rangle\right)
\end{aligned}$$

1. $Y \supseteq [\![Tr(E(\psi U \varphi)]\!]$

   Veamos que Y es un punto fijo de τ.

$$\begin{aligned}
& w \in \tau(Y) \\
\Rightarrow\ & w \in [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \{w \mid \left\langle \exists w' : s_x^w \xrightarrow{x} s_x^w : w' \in Y\right\rangle\right) \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \psi\rangle \vee \\
& \quad \left(\langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi\rangle \wedge \left\langle \exists w' : w \xrightarrow{x} w' : w' \in Y\right\rangle\right) \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \psi\rangle \vee \Big(\langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models_{DTL} \varphi\rangle \wedge \\
& \quad \left\langle \exists w' : w \xrightarrow{x} w' : \langle \forall \pi, i : w' = Tr^p(\pi).i : \pi, i, M \models E(\psi U \varphi)\rangle\right\rangle \Big) \\
\Rightarrow\ & \langle \forall \pi, i : w = Tr^p(\pi).i : \pi, i, M \models E(\psi U \varphi)\rangle \\
\Rightarrow\ & w \in Y
\end{aligned}$$

   Como Y es punto fijo de τ es mayor que el mínimo punto fijo, por lo que $Y \supseteq [\![Tr(\mathsf{E}(\psi\ \mathcal{U}\ \varphi)]\!]$ queda demostrado.

2. $Y \subseteq [\![Tr(\mathsf{E}(\psi\ \mathcal{U}\ \varphi))]\!]$

   Para esta demostración es necesario definir Y_n

$$Y_n = \{w \mid \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models E_n(\varphi U \psi)\rangle\}$$

Donde

$$\pi, i, M, w \models E_n(\varphi U \psi) \equiv$$
$$\langle \exists \pi', j : \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+n : \pi', j, M \models_{DTL} \psi \wedge \langle \forall k : i \leq k < j : \ \pi, k, M \models \varphi \rangle \rangle$$

Intuitivamente el conjunto Y_n es el conjunto de estados que garantizan que existe un camino que comienza en ese estado y cuyo tramo inicial de longitud menor o igual a n termina en un estado que cumple ψ y cumple φ en los estados intermedios. Como consecuencia de esta definición se tiene que $Y = \bigcup_{n:0..\infty} Y_n$.

Veamos primero que $Y_n \subseteq \tau^{(n+1)}(\perp)$. Probaremos la propiedad por inducción en i_0.

• Caso $n = 0$

$$\begin{aligned}
& w \in Y_0 \\
\Rightarrow\ & \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models E_0(\varphi U \psi) \rangle \\
\Rightarrow\ & \Big\langle \forall i, \pi \ : \ w = Tr^p(\pi).i \ : \ \Big\langle \exists \pi', j \ : \ \pi[0,i] \preceq \pi' \wedge i \leq j \leq i+0 \ : \\
& \qquad\qquad \pi', j, M \models_{DTL} \psi \wedge \langle \forall k : i \leq k < j : \ \pi, k, M \models \varphi \rangle \Big\rangle \Big\rangle \\
\Rightarrow\ & \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models \psi \rangle \\
\Rightarrow\ & w \in [\![Tr(\psi)]\!] \\
\Rightarrow\ & w \in [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \{ w \mid \left\langle \exists w' : w \xrightarrow{a} w' : w' \in \perp \right\rangle \right) \\
\Rightarrow\ & w \in \tau(\perp)
\end{aligned}$$

• Caso $n > 0$

$$\begin{aligned}
& w \in Y_n \\
\Rightarrow\ & \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models E_n(\varphi U \psi) \rangle \\
\Rightarrow\ & \langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models \psi \rangle \vee \Big(\langle \forall i, \pi : w = Tr^p(\pi).i : \pi, i, M \models \varphi \rangle \wedge \\
& \quad \Big\langle \exists w' : w \xrightarrow{a} w' : \langle \forall i, \pi : w' = Tr^p(\pi).i : \pi, i, M \models E_{n-1}(\varphi U \psi) \rangle \Big\rangle \Big) \\
\Rightarrow\ & w \in [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \{ w \mid \left\langle \exists w' : w \xrightarrow{a} w' : w' \in Y_{n-1} \right\rangle \right) \\
\Rightarrow\ & w \in [\![Tr(\psi)]\!] \cup \left([\![Tr(\varphi)]\!] \cap \{ w \mid \left\langle \exists w' : w \xrightarrow{a} w' : w' \in \tau^n(\top) \right\rangle \right) \\
\Rightarrow\ & w \in \tau(\tau^n(\top)) \\
\Rightarrow\ & w \in \tau^{n+1}(\top)
\end{aligned}$$

Como consecuencia

$$
\begin{aligned}
& s \in Y \\
\Rightarrow\ & s \in \bigcup_n Y_n \\
\Rightarrow\ & \bigvee_n w \in Y_n \\
\Rightarrow\ & \{\ \ Y_n \subseteq \tau^{(n+1)}(\bot)\ \ \} \\
& \bigvee_n w \in \tau^{n+1}(\bot) \\
\Rightarrow\ & s \in \bigcup_n \tau^{n+1}(\bot) \\
\Rightarrow\ & s \in \bigcup_n \tau^{n+1}(\bot) \cup \tau^0(\bot) \\
\Rightarrow\ & s \in \bigcup_n \tau^{n}(\bot)
\end{aligned}
$$

□

Como consecuencia de este teorema, se demuestra la corrección de la traducción de DTL.

Corolario 4.7 *Dada una signatura* $\langle \Phi_0, \Delta_0 \rangle$*, una estructura* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$*, un estado inicial* $w_0 \in \mathcal{W}$ *y una fórmula* φ*, se cumple:*

$$w_0, M \models_{DTL} \varphi \leftrightarrow w_0, Tr^{tm}(M) \models_\mu Tr(\varphi)$$

Podemos remarcar que la biyección entre trazas (donde cada estado también mantiene su equivalencia) hacen posible obtener, a partir de los contraejemplos en mu-calculus de las fórmulas traducidas, un contraejemplo para el modelo original.

4.3.5. Lemas auxiliares

Aquí se detallan las pruebas de algunos lemas utilizados para las demostraciones previas.

Lema 4.8 $\{Gen(e) \mid e \in \mathcal{I}(\alpha)\} = \{a \mid a \in Set(\alpha) \wedge E_a\}$

Prueba:

$$\{a \mid a \in Set(\alpha) \wedge E_a\}$$

$\equiv$ { definición de $L(P_a)$ }
$\{a \mid a \in Set(\alpha) \wedge a \in Im(Gen)\}$
$\equiv$ { conjuntos }
$\{a \mid \exists e \ : \ a = Gen(e) \wedge a \in Set(\alpha)\}$
$\equiv$ { Gen inyectiva }
$\{Gen(e) \mid Gen(e) \in Set(\alpha)\}$
$\equiv$ { Lema 4.9 }
$\{Gen(e) \mid e \in \mathcal{I}(\alpha)\}$

□

Lema 4.9 $e \in \mathcal{I}(\alpha) \Longleftrightarrow Gen(e) \in Set(\alpha)$

Prueba:

$Gen(e) \in Set(\alpha)$
$\equiv$ { definición de Set }
$\alpha \in Gen(e)$
$\equiv$ { definición de Gen }
$e \in \mathcal{I}(\alpha)$

□

Lema 4.10 *Dados* $w, w' \in \mathcal{W}$ *y* $\alpha \in \Delta$

$$\exists e \in \mathcal{I}(\alpha) \ : \ w \stackrel{e}{\rightarrow} w' \in \mathcal{R} \Longleftrightarrow \exists a \in Set(\alpha) \ : \ w \stackrel{a}{\rightarrow} w' \in T$$

Prueba:

$\Rightarrow$) Sea $e \in \mathcal{I}(\alpha)$ tal que $w \stackrel{e}{\rightarrow} w' \in \mathcal{R}$. Se define $a = Gen(e)$. Por definición de T se tiene $w \stackrel{a}{\rightarrow} w' \in$ T vamos a probar que $a \in Set(\alpha)$. Si tenemos $\alpha \in \Delta_0$, $Set(\alpha)$ es el conjunto de subconjuntos de Δ_0 que contienen α, entonces a pertenece a $Set(\alpha)$ cuando α pertenece a a, pero esto es cierto dado que $e \in \mathcal{I}(\alpha)$ y por definición de $Gen(e) = \{\alpha \mid \alpha \in \Delta_0 \wedge e \in \mathcal{I}(\alpha)\}$.

Ahora, por inducción estructural en α

-$\alpha = \emptyset$**:** no es posible, dado que $e \in \mathcal{I}(\alpha)$

-$\alpha =$ **U:** $a \in \mathbf{U} = \mathcal{P}(\Delta_0)$.

-$\alpha = \beta \sqcup \gamma$**:** $a \in Set(\alpha)$ sii $a \in Set(\beta)$ o $a \in Set(\gamma)$; por otro lado, $e \in \mathcal{I}(\alpha)$ sii $e \in \mathcal{I}(\beta)$ o $e \in \mathcal{I}(\gamma)$.

- Los otros casos son similares.

$\Leftarrow$) Sea $a \in Set(\alpha)$ tal que $w \overset{a}{\to} w' \in$ T. Por definición de T, existe $e \in \mathcal{E}$ tal que $a = Gen(e)$ y $w \overset{e}{\to} w' \in \mathcal{R}$.

$\square$

Lema 4.11

$$\bigvee_{e \in \mathcal{I}(\alpha)} (w, e) \in \mathcal{P} \Longleftrightarrow \bigvee_{a \in Set(\alpha)} \exists e \in \mathcal{E} \ : \ (w, e) \in \mathcal{P} \wedge Gen(e) = a$$

Dado $w \in \mathcal{W}$ *y* $\alpha \in \Delta$

$$\exists e \in \mathcal{I}(\alpha) \ : \ (w, e) \in \mathcal{P} \Longleftrightarrow \exists a \in Set(\alpha) \ : \ w \in L(P_a)$$

Prueba: Esta prueba es similar a la del lema 4.10.

$\Rightarrow$) Sea $e \in \mathcal{I}(\alpha)$ tal que $\langle w, e \rangle \in \mathcal{P}$. Se define $a = Gen(e)$. Por definición de $L(P_a)$ se tiene $w \in L(P_a)$, probemos que $a \in Set(\alpha)$. Cuando $\alpha \in \Delta_0$, $Set(\alpha)$ es el conjunto de subconjuntos de Δ_0 que contienen α, por lo tanto a está en $Set(\alpha)$ si α está en a, pero esto es verdadero porque $e \in \mathcal{I}(\alpha)$ y por definición de $Gen(e) = \{\alpha \mid \alpha \in \Delta_0 \wedge e \in \mathcal{I}(\alpha)\}$. Por inducción estructural en α se prueba para todo $\alpha \in \Delta$, como en el lema anterior.

$\Leftarrow$) Sea $a \in Set(\alpha)$ una acción tal que $w \in L(P_a)$. Por definición de $L(P_a)$ existe $e \in \mathcal{E}$ tal que $a = Gen(e)$ y $(w, e) \in \mathcal{P}$.

$\square$

Lema 4.12 *Dada una signatura* $\langle \Phi_0, \Delta_0 \rangle$*, una estructura* $M = \langle \mathcal{W}, \mathcal{R}, \mathcal{E}, \mathcal{I}, \mathcal{P} \rangle$*, un estado inicial* $w_0 \in \mathcal{W}$ *y una fórmula no temporal* φ*, se tiene:*

$$s_a^w \in [\![Tr(\varphi)]\!]_{Tr^{tm}(M)} \equiv w, M \models_{DPL} \varphi \qquad \textit{para todo } a$$

Este lema se prueba de la misma manera que el teorema 4.4.

4.4. Segunda Traducción

La primera traducción permite traducir de manera independiente la signatura, el modelo y cada fórmula, para luego poder evaluar si una determinada fórmula se satisface o no.

$$w, M \models_{DTL} \varphi \Longleftrightarrow Tr^m(w, M) \models_\mu Tr(\varphi).$$

Si embargo, el modelo y la fórmula de μ-calculus obtenida con esta traducción son exponenciales en relación al modelo y la fórmula original. Esta explosión surge principalmente del conjunto de acciones $\Delta_1 = 2^{\Delta_0}$ [4].

[4] Recordar que denotamos con Δ_0 el conjunto de acciones de DTL y con Δ_1 el de μ-calculus.

En esta sección, proponemos una nueva traducción más eficiente, en la que tanto la signatura y, en consecuencia, el modelo dependen de la fórmula que se quiere verificar.

$$w, M \models_{DTL} \varphi \Longleftrightarrow Tr^{m}_{\langle \Phi_1, \Delta_1 \rangle}(w, M) \models_{\mu} Tr_{\langle \Phi_1, \Delta_1 \rangle}(\varphi)$$

Donde $\langle \Phi_1, \Delta_1 \rangle = Tr^{L}_{\varphi}(\langle \Phi_0, \Delta_0 \rangle)$.

4.4.1. Segunda traducción de PDL a μ-calculus

Comenzamos con la caracterización de la lógica proposicional deóntica PDL en términos de μ-calculus, para continuar luego con la traducción de la lógica temporal deóntica DTL a μ-calculus. Para ello, introducimos las siguientes funciones auxiliares.

Definición 4.21 *Sea φ una fórmula PDL sobre la signatura $\langle \Phi_0, \Delta_0 \rangle$. Definimos recursivamente la función $actionSet(\varphi)$ como sigue:*

- *actionSet*$(p) = \emptyset$, *donde* $p \in \Phi_0 \cup \{\top, \bot\}$,
- *actionSet*$(\alpha =_{act} \beta) = \emptyset$,
- *actionSet*$(\neg\varphi) =$ *actionSet*(φ) ,
- *actionSet*$(\varphi \wedge \psi) =$ *actionSet*$(\varphi) \cup$ *actionSet*(ψ),
- *actionSet*$(\langle\alpha\rangle\varphi) = \{\alpha\} \cup$ *actionSet*(ψ),
- *actionSet*$(\mathbf{P}(\alpha)) = \{\alpha\}$,
- *actionSet*$(\mathbf{P}_w(\alpha)) = \{\alpha\}$.

Notar que, dado una signatura $\langle \Phi_0, \Delta_0 \rangle$, un modelo M y una fórmula φ sobre esa signatura, se tiene $actionSet(\varphi) \subseteq \Delta$ [5]. Por lo tanto, se cuenta con una interpretación $\mathcal{I}$ de cada elemento del conjunto.

Definición 4.22 *Sea φ una fórmula PDL sobre la signatura $\langle \Phi_0, \Delta_0 \rangle$. Definimos recursivamente la función $equalSet(\varphi)$ como sigue:*

- *equalSet*$(p) = \emptyset$, *donde* $p \in \Phi_0 \cup \{\top, \bot\}$,
- *equalSet*$(\alpha =_{act} \beta) = \{\alpha =_{act} \beta\}$,
- *equalSet*$(\neg\varphi) =$ *equalSet*(φ) ,
- *equalSet*$(\varphi \wedge \psi) =$ *equalSet*$(\varphi) \cup$ *equalSet*(ψ),

[5] conjunto de términos de acciones a partir de Δ_0. Def. 4.6

- *equalSet*$(\langle\alpha\rangle\varphi) =$ *equalSet*(φ),
- *equalSet*$(\mathbf{P}(\alpha)) = \emptyset$,
- *equalSet*$(\mathbf{P}_w(\alpha)) = \emptyset$,

Para definir la traducción a μ-calculus necesitamos traducir la signatura, traducir el modelo y traducir la fórmula. En esta traducción, el modelo y la signatura dependen de la fórmula. De esta manera logramos reducir la complejidad del modelo y la fórmula para lograr un método de model checking más eficiente.

Definición 4.23 (Tr^L_φ) *Dada una signatura $\langle\Phi_0, \Delta_0\rangle$ y una fórmula PDL φ, se define $Tr^L_\varphi(\langle\Phi_0, \Delta_0\rangle) = \langle\Phi_1, \Delta_1\rangle$ donde*

- $\Delta_1 = actionSet(\varphi)$,
- $\Phi_1 = \Phi_0 \cup \{AP_a \mid a \in \Delta_1\} \cup \{EP_a \mid a \in \Delta_1\} \cup \{EQ_x \mid x \in equalSet(\varphi)\}$,

donde AP_a, EP_a y EQ_x son símbolos proposicionales nuevos definidos a partir de cada $a \in \Delta_1$ y $x \in equalSet(\varphi)$, respectivamente.

Definición 4.24 (Tr^M_φ) *Dada una signatura $\langle\Phi_0, \Delta_0\rangle$, una estructura deóntica $M = \langle\mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P}\rangle$ y una fórmula PDL φ, definimos $Tr^M_\varphi(M) = \langle S, T, L\rangle$ donde $\langle\Phi_1, \Delta_1\rangle = Tr^L_\varphi(\langle\Phi_0, \Delta_0\rangle)$ y*

- $S = \mathcal{W}$,
- $T = \{w \xrightarrow{a} w' \mid a \in \Delta_1 \wedge \exists e \in \mathcal{I}(a) :\ w \xrightarrow{e} w' \in \mathcal{R}\}$,
- $L(p) = I(p)$, *para cada* $p \in \Phi_0$,
- $L(AP_a) = \{w \mid \forall e \in \mathcal{I}(a)$ *se cumple* $(w, e) \in \mathcal{P}\}$, *para cada* $a \in \Delta_1$,
- $L(EP_a) = \{w \mid \exists e \in \mathcal{I}(a)$ *tal que* $(w, e) \in \mathcal{P}\}$, *para cada* $a \in \Delta_1$,
- $L(EQ_{\alpha =_{act} \beta}) = \begin{cases} \mathcal{I}(\alpha) = \mathcal{I}(\beta) \to S \\ \mathcal{I}(\alpha) \neq \mathcal{I}(\beta) \to \emptyset \end{cases}$,

Por razones de legibilidad, omitiremos el subíndice φ cuando sea posible.

Definición 4.25 *Dada uan signatura $\langle\Phi_0, \Delta_0\rangle$, una estructura deóntica $M = \langle\mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P}\rangle$ y una fórmula PDL φ, y tomando $\langle\Phi_1, \Delta_1\rangle = Tr^L_\varphi(\langle\Phi_0, \Delta_0\rangle)$, definimos recursivamente la función Tr de la siguiente manera:*

- $Tr(p) = p$, *para cada* $p \in \Phi_0$,
- $Tr(\top) = \top$,

- $Tr(\bot) = \bot$,
- $Tr(\neg\varphi) = \neg Tr(\varphi)$,
- $Tr(\alpha =_{act} \beta) = EQ_{\alpha =_{act} \beta}$
- $Tr(\langle\alpha\rangle\varphi) = \langle\alpha\rangle Tr(\varphi)$,
- $Tr(\mathbf{P}(\alpha)) = AP_{\alpha}$,
- $Tr(\mathbf{P}_w(\alpha)) = EP_{\alpha}$.
- $Tr(\varphi_1 \wedge \varphi_2) = Tr(\varphi_1) \wedge Tr(\varphi_2)$,

Las traducciones entre sistemas lógicos han sido estudiados a lo largo de los años por la comunidad [GB92, GR02]. En este contexto, los sistemas lógicos son capturados por términos abstractos. Las traducciones más usuales entre sistemas lógicos son los llamados *morfismos* y *comorfismos* (o representaciones). En ambos casos, las traducciones de modelos y fórmulas van en direcciones opuestas. Más precisamente, un morfismo entre sistemas lógicos L y L' traduce modelos de L en modelos de L' y fórmulas de L' en fórmulas de L, de una forma que preserve las propiedades; los comorfismos se comportan en el sentido inverso. En ambos casos, tienen las características de conexiones de Galois.

Nuestra traducción difiere de los morfismos y comorfismos en el sentido de que mapea modelos y fórmulas en "el mismo sentido". Este tipo de traducción se llama *forward morphism* [GR02]. Cabe destacar, que este tipo de morfismos es el que necesitamos, dado que son los que mejor sirven para el problema de model checking (nuestro objetivo principal). En el apartado 4.4.3, veremos que las trazas del modelo traducido, pueden ser reconstruidas como trazas del modelo original. Intuitivamente, esto significa que preserva contraejemplos, una propiedad fundamental para la tarea de model checking.

4.4.2. Corrección de la segunda traducción Tr

El siguiente teorema establece que nuestra traducción es correcta respecto al problema de model checking. Se prueba por inducción estructural sobre las fórmulas PDL.

Teorema 4.13 *Dada una signatura $\langle\Phi_0, \Delta_0\rangle$, una fórmula ϕ sobre esa signatura, una structure deóntica $M = \langle\mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P}\rangle$ y un estado $w \in \mathcal{W}$, la función Tr definida en 4.25 preserva satisfactibilidad. Es decir, si $M_\mu = Tr^M(M)$ y $\phi_\mu = Tr(\phi)$ entonces*

$$w, M \models \phi \Leftrightarrow w, M_\mu \models_\mu \phi_\mu.$$

Prueba: Haremos la prueba por inducción sobre la fórmula ϕ. Para mayor claridad utilizaremos denotamos $\phi_\mu = Tr(\phi)$, $\varphi_\mu = Tr(\varphi)$, $\psi_\mu = Tr(\psi)$. También utilizaremos los símbolos T y L para referirnos al conjunto de transiciones y a la función de etiquetado de M_μ, respectivamente.

Caso $\phi = p$**:**

$w, M \models p$
$\equiv$ { def. de $\models$ }
$w \in \mathcal{I}(p)$
$\equiv$ { def. de Tr }
$w \in L(p)$
$\equiv$ { def. de $[\![ ]\!]$ }
$w \in [\![p]\!]$
$\equiv$ { def. of Tr }
$w \in [\![\phi_\mu]\!]$

Caso $\phi = \neg\varphi$**:**

$w, M \models \neg\varphi$
$\equiv$ { def. de $\models$ }
no $w, M \models \varphi$
$\equiv$ { Hipótesis Inductiva }
no $w \in [\![Tr(\varphi)]\!]$
$\equiv$ { def. de $Tr^m(M)$ y propiedades de conjuntos }
$w \in [\![\neg Tr(\varphi)]\!]$
$\equiv$ { def. de Tr }
$w \in [\![Tr(\neg\varphi)]\!]$

Caso $\phi = \varphi \wedge \psi$**:**

$w, M \models \varphi \wedge \psi$
$\equiv$ { def. de $\models$ }
$w, M \models \varphi \wedge w, M \models \psi$
$\equiv$ { Hip. inductiva }
$w \in [\![\varphi_\mu]\!] \wedge w \in [\![\psi_\mu]\!]$
$\equiv$ { Def. de $[\![ ]\!]$ y prop. de conjuntos }
$w \in [\![\varphi_\mu \wedge \psi_\mu]\!]$
$\equiv$ { def. de Tr }
$w \in [\![\phi_\mu]\!]$

Caso $\phi = \alpha =_{act} \beta$**:** Por definición de Tr tenemos que $\phi_\mu = EQ_{\alpha =_{act} \beta}$. Tenemos que considerar dos casos: el caso en el que $\mathcal{I}(\alpha) = \mathcal{I}(\beta)$ y el caso

$\mathcal{I}(\alpha) \neq \mathcal{I}(\beta)$. En ambos casos la prueba es análoga, por lo que probamos el prumer caso.

Como $\mathcal{I}(\alpha) = \mathcal{I}(\beta)$, se cumple $w, M \models \alpha =_{act} \beta$ por definición de $\models$.Por otro lado, se tiene que $w \in [\![EQ_{\alpha=_{act}\beta}]\!]$, por definición de $[\![]\!]$, ya que $L(EQ_{\alpha=_{act}\beta}) = S$.

Caso $\phi = \langle\alpha\rangle\varphi$**:**

$$
\begin{aligned}
& w, M \models \langle\alpha\rangle\varphi \\
\equiv & \{ \text{ def. de } \models \} \\
& \text{existe } w' \in \mathcal{W} \text{ y } e \in \mathcal{I}(\alpha) \text{ tal que } w \xrightarrow{e} w' \in \mathcal{R} \text{ y } w', M \models \varphi \\
\equiv & \{ \text{ Hip. inductiva } \} \\
& \text{existe } w' \in \mathcal{W} \text{ y } e \in \mathcal{I}(\alpha) \text{ tal que } w \xrightarrow{e} w' \in \mathcal{R} \text{ y } w' \in [\![\varphi_\mu]\!] \\
\equiv & \{ \text{ Def. de } T \text{ y } S, \text{ prop. de conjuntos } \} \\
& \text{existe } w' \in S \text{ tal que } w \xrightarrow{\alpha} w' \text{ y } w' \in [\![\varphi_\mu]\!] \\
\equiv & \{ \text{ def. de } [\![]\!] \text{ y prop. de conjuntos } \} \\
& w \in [\![\langle\alpha\rangle\varphi_\mu]\!].
\end{aligned}
$$

Caso $\phi = \mathbf{P}(\alpha)$**:** We have, by definition of Tr, that $\phi_\mu = AP_\alpha$.

Since $L(AP_\alpha)$ is defined as $\{w \mid \forall e \in \mathcal{I}(a)$ such that exists some $w' \in \mathcal{W}$ and $w \xrightarrow{e} w'$, it holds $\mathcal{P}(w, e)\}$, $w, M \models \mathbf{P}(\alpha)$ holds iff $w \in [\![AP_\alpha]\!]$, by definition of $\models$ and $[\![\ ]\!]$.

$$
\begin{aligned}
& w, M \models \mathbf{P}(\alpha) \\
\equiv & \{ \text{ def. of } \models \} \\
& \forall e \in \mathcal{I}(\alpha)(w, e) \in \mathcal{P} \\
\equiv & \{ \text{ def de } AP_\alpha \} \\
& w \in L(AP_\alpha) \\
\equiv & \{ \text{ def. of } [\![]\!] \} \\
& w \in [\![AP_\alpha]\!] \\
\equiv & \{ \text{ def. of } Tr \} \\
& w \in [\![\phi_\mu]\!]
\end{aligned}
$$

Caso $\phi = \mathbf{P}_w(\alpha)$**:** Se demuestra de manera similar al caso anterior.

□

En relación a nuestro objetivo de definir una herramienta de model checking para esta lógica deóntica, un requisito fundamental es que la traducción definida se pueda realizar en un tiempo razonable. Esto se muestra en el siguiente teorema.

Teorema 4.14 *Dado un un modelo PDL M y una fórmula φ de PDL, la traducción de M en un modelo de μ-calculus $M_\mu = \langle S, T, L\rangle$ se puede obtener en tiempo $\mathcal{O}(|M| * |\varphi|)$.*

Prueba: S se obtiene en $\mathcal{O}(|\mathcal{W}|)$ y L en $\mathcal{O}(|\mathcal{I}|)$, ambos acotados por $\mathcal{O}(|M|)$. Para calcular T procesamos la interpretación de las acciones φ, que se puede calcular $\mathcal{O}(|M| * |\varphi|)$, como se muestra en el teorema 4.3. Como T es una relación de transición para cada acción en Δ (acotada por $|\varphi|$) y estas transiciones son las representadas en $\mathcal{R}$, el cálculo de T está acotado por $\mathcal{O}(|M| * |\varphi|)$.

□

Teorema 4.15 *La traducción de una fórmula PDL φ a una fórmula de μ-calculus φ_μ se puede obtener en tiempo $\mathcal{O}(|\varphi|)$.*

La prueba es inmediata teniendo en cuenta que la fórmula obtenida tiene el mismo o menor tamaño que la fórmula original.

Teniendo en cuenta que las fórmulas de PDL se traducen a un fragmento de $\mu-$calculo que cuenta con algoritmos de model checking de tiempo polinómico, se verifica el siguiente corolario:

Corolario 4.16 *Dado un modelo PDL M y una fórmula de PDL φ, la relación $M \vDash_{PDL} \varphi$ puede ser verificada en tiempo $\mathcal{O}(|M| * |\varphi|)$ utilizando Tr^M y Tr y los procedimientos de model checking de μ-calculus.*

Prueba: Siguiendo la definición 4.25, podemos ver las fórmulas son traducidas al fragmento libre alternado (alternation free fragment) de μ-calculus, para el cual el problema de model checking es lineal [Eme96] con respecto al tamaño del modelo y la fórmula. Por el teorema 4.15, el resultado es directo.

□

Estos resultados nos indican que podemos utilizar procedimientos de model cheking estandares de μ-caluclus para verificar si una determinada estructura deóntica satisface una fórmula PDL, en un tiempo razonable.

4.4.3. Segunda traducción de DTL a μ-calculus

Ahora extenderemos la definición de Tr para cubrir las fórmulas temporales de DTL. Para ello extendemos las definiciones de las funciones $actionSet$ y $equalSet$ para incluir operadores temporales de la siguiente manera:

Definición 4.26 *Sea φ una fórmula DTL sobre la signatura $\langle \Phi_0, \Delta_0\rangle$. Extendemos la función $actionSet(\varphi)$ definida recursiamente en 4.21 de la siguiente manera:*

- *actionSet*($\mathsf{EG}\varphi$) $= \{\mathbf{U}\} \cup$ *actionSet*(φ),

- *actionSet*$(\mathsf{E}(\varphi\,\mathcal{U}\,\psi) = \{\mathbf{U}\} \cup$ *actionSet*$(\varphi) \cup$ *actionSet*(ψ),

Definición 4.27 *Sea φ una fórmula DTL sobre la signatura $\langle\Phi_0, \Delta_0\rangle$. Extendemos la función $equalSet(\varphi)$ definida recursiamente en 4.22 de la siguiente manera:*

- *equalSet*$(\mathsf{EG}\varphi) = \emptyset$,
- *equalSet*$(\mathsf{E}(\varphi\,\mathcal{U}\,\psi)) = \emptyset$.

La traducción de la signatura y el modelo se define de la misma manera que en las definiciones 4.23 y 4.24 con los conjuntos $actionSet$ y $equalSet$ actualizados.

Definición 4.28 *Sea $\langle\Phi_0, \Delta_0\rangle$ una signatura, φ una fórmula sobre esa signatura, $M = \langle\mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P}\rangle$ una estructura deóntica, y $w_0 \in \mathcal{W}$ un estado inicial en M. Extendemos la definición de la función $Tr_{\langle\Phi_1,\Delta_1\rangle}$ definida en 4.25 a fórmulas DTL como sigue:*

- *Para $\varphi \in \Phi$, $Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi)$ se define como se describe en la definición 4.25,*
- $Tr_{\langle\Phi_1,\Delta_1\rangle}(\neg\varphi) = \neg Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi)$,
- $Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi \wedge \psi) = Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi) \wedge Tr_{\langle\Phi_1,\Delta_1\rangle}(\psi)$,
- $Tr_{\langle\Phi_1,\Delta_1\rangle}(\mathsf{EG}\varphi) = \nu R.(Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi) \wedge \langle\mathbf{U}\rangle R)$,
- $Tr_{\langle\Phi_1,\Delta_1\rangle}(\mathsf{E}(\varphi\,\mathcal{U}\,\psi)) = \mu R.(Tr_{\langle\Phi_1,\Delta_1\rangle}(\psi) \vee (Tr_{\langle\Phi_1,\Delta_1\rangle}(\varphi) \wedge \langle\mathbf{U}\rangle R))$.

$\langle\Phi_1, \Delta_1\rangle$ es definida como en 4.23, pero omitimos los subíndices cuando sea posible.

La traducción utiliza la semántica de punto fijo estándar para los operadores temporales [BCJM00].

4.4.4. Corrección de la segunda traducción extendida Tr

La corrección de la traducción extendida se prueba generalizando el teorema de corrección de la troducción para PDL. Veamos primero la traducción de las trazas que mapea el conjunto de trazas de M a M_μ.

Definición 4.29 *La función Tr^p se define como sigue:*

$$Tr^p(w_0 \xrightarrow{e_0} w_1 \xrightarrow{e_1} w_2 \xrightarrow{e_2} \dots) = \{w_0 \xrightarrow{a_0} w_1 \xrightarrow{a_1} w_2 \xrightarrow{a_2} \cdots \mid e_i \in \mathcal{I}(a_i) \wedge a_i \in \Delta_1\}.$$

El siguiente teorema muestra la corrección de la función de traducción de fórmulas DTL.

Teorema 4.17 *Dado una signatura $\langle \Phi_0, \Delta_0 \rangle$, una estructura deóntica $M = \langle \mathcal{W}, \mathcal{E}, \mathcal{R}, \mathcal{I}, \mathcal{P} \rangle$, un estado inicial $w_0 \in \mathcal{W}$ y una fórmula ϕ, si $M_\mu = Tr^M(M)$ y $\phi_\mu = Tr(\phi)$ entonces*

$$w_0 \in [\![\phi_\mu]\!]_{M_\mu} \Leftrightarrow w_0, M \models \phi.$$

Prueba: Para los operadores modales y deónticos se sigue la prueba en el teorema 4.13. Para los operadores temporales nuestra prueba se basa en la prueba de la traducción de CTL a μ-calculus en [CBJM96].

Caso $\phi = \mathsf{EG}\varphi$: Tenemos $\phi_\mu = \nu R.(\varphi_\mu \wedge \langle \mathbf{U} \rangle R)$.

Sea $Y = \{s \mid s, M \models \mathsf{EG}\varphi\}$. Tenemos que probar: $Y = [\![\phi_\mu]\!] = [\![\nu R.(\varphi_\mu \wedge \langle \mathbf{U} \rangle R)]\!]$

La expresión $\nu R.(\varphi_\mu \wedge \langle \mathbf{U} \rangle R)$ se define como el mayor punto fijo de τ, donde:

$$\tau(S) = [\![\varphi_\mu \wedge \langle \mathbf{U} \rangle R]\!] e[R \leftarrow S]$$
$$\tau(S) = [\![\varphi_\mu]\!] \cap [\![\langle \mathbf{U} \rangle S]\!]$$

Observar que por hipótesis inductiva y definición de T vale que

$$s \in \tau(S) \Longleftrightarrow s, M \models \varphi \wedge \exists s', e \text{ tal que } s \xrightarrow{e} s' \wedge s \in S.$$

Para probar que ambos conjuntos son iguales, probamos la doble inclusión. A continuación en (1) probamos que Y es un punto fijo de τ, por lo que está incluido en el máximo punto fijo; en (2) probamos que Y es el máximo punto fijo.

(1) $Y \subseteq [\![\phi_\mu]\!]$

Sea $s \in \tau(Y)$. Por definición de τ y Y tenemos que

$$s, M \models \varphi \wedge \exists s_1, e \text{ tal que } s \xrightarrow{e} s_1 \wedge s_1, M \models \mathsf{EG}\varphi.$$

Dado que $s_1, M \models \mathsf{EG}\varphi$, existe una traza $s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots$ tal que $s_i, M \models \varphi$ para $i > 0$. Entonces, podemos extender a $s_0 \xrightarrow{e_0} s_1 \xrightarrow{e_1} s_2 \xrightarrow{e_2} \ldots$, donde $s_i, M \models \varphi$ para $i \geq 0$, y $s = s_0$ y $e = e_0$. Por lo tanto, vale $s, M \models \mathsf{EG}\varphi$, es decir, $s \in Y$.

(2) $Y \supseteq [\![\phi_\mu]\!]$

Sea $s \in [\![\phi_\mu]\!]$. Como τ es continuo, el mayor punto fijo de τ es $\bigcap_i \tau^i(\top)$. Ahora consideremos i_0 un natural tal que $\tau^{i_0}(\top) = \tau(\tau^{i_0}(\top))$ [6]. Entonces, $[\![\phi_\mu]\!] = \tau^{i_0}(\top)$.

Si $s \in \tau^{i_0}(\top)$ entonces s también pertenece a $\tau(\tau^{i_0}(\top))$. Por lo tanto,

[6] i_0 existe dado que M es finito.

$$s, M \models \varphi \wedge \exists s_1, e_0 : s \stackrel{e_0}{\to} s_1 \wedge s_1 \in \tau^{i_0}(\top).$$

Pero en este caso se tiene que $s_1 \in \tau^{i_0}(\top)$, y nuevamente vale

$$s_1, M \models \varphi \wedge \exists s_2, e_1 : s_1 \stackrel{e_1}{\to} s_2 \wedge s_2 \in \tau^{i_0}(\top).$$

Continuando de la misma manera, se puede contruir una traza $\pi = s \stackrel{e_0}{\to} s_1 \stackrel{e_1}{\to} s_2 \stackrel{e_2}{\to} \ldots$ tal que $\forall i\ s_i, M \models \varphi$. Por lo tanto, $s \in Y$.

Caso $\phi = \mathsf{E}(\varphi\ \mathcal{U}\ \psi)$**:** Tenemos $\phi_\mu = \mu R.(\psi_\mu \vee (\varphi_\mu \wedge \langle \mathbf{U} \rangle R))$.

Sea $Y = \{s \mid s, M \models \mathsf{E}(\varphi\ \mathcal{U}\ \psi)\}$. Tenemos que probar: $Y = [\![\phi_\mu]\!] = [\![\mu R.(\psi_\mu \vee (\varphi_\mu \wedge \langle \mathbf{U} \rangle R))]\!]$.

La expresión $\mu R.(\psi_\mu \vee (\varphi_\mu \wedge \langle \mathbf{U} \rangle R))$ se define como el menor punto fijo de τ, donde:

$$\tau(S) = [\![\psi_\mu \vee (\varphi_\mu \wedge \langle \mathbf{U} \rangle R)]\!]e[R \leftarrow S]$$
$$\tau(S) = [\![\psi_\mu]\!] \cup ([\![\varphi_\mu]\!] \cap [\![\langle \mathbf{U} \rangle S]\!])$$

Observar que por hipótesis inductiva y definición de T vale

$$s \in \tau(S) \iff s, M \models \psi \vee (s, M \models \varphi \wedge \exists s', e \text{ tal que } s \stackrel{e}{\to} s' \wedge s \in S)$$

.

Para probar la igualdad proba mos la doble inclusión.

(1) $Y \supseteq [\![\phi_\mu]\!]$. Vamos a probar que Y es un punto fijo de τ y por lo tanto está incluido en el menor punto fijo.
Sea $s \in \tau(Y)$. Por definición de τ y Y, tenemos

$$s, M \models \psi \vee (s, M \models \varphi \wedge \exists s_1, e_0 \text{ tq } s \stackrel{e_0}{\to} s_1 \wedge s_1, M \models \mathsf{E}(\varphi\ \mathcal{U}\ \psi)).$$

En el caso $s, M \models \psi$, cualquier traza que comienza en s satisface $s, M \models \mathsf{E}(\varphi\ \mathcal{U}\ \psi))$, entonces $s \in Y$.
En el caso $s, M \models \varphi \wedge \exists s_1, e_0$ tq $s \stackrel{e_0}{\to} s_1 \wedge s_1, M \models \mathsf{E}(\varphi\ \mathcal{U}\ \psi)$, existe un natural k y una traza $s_1 \stackrel{e_1}{\to} s_2 \stackrel{e_2}{\to} \ldots$ tal que $s_i, M \models \varphi$ para $0 < i < k$ y $s_k, M \models \psi$. Entonces tenemos una traza $s \stackrel{e_0}{\to} s_1 \stackrel{e_1}{\to} s_2 \stackrel{e_2}{\to} \ldots$ tal que $s, M \models \mathsf{E}(\varphi\ \mathcal{U}\ \psi)$, es decir, $s \in Y$.

(2) $Y \subseteq [\![\phi_\mu]\!]$
Sea $Y_n = \{s | s, M \models \mathsf{E}_n(\varphi \mathcal{U}\ \psi)\}$, donde:

$$s, M \models \mathsf{E}_n(\varphi \mathcal{U}\ \psi) \iff \exists \pi \in \Sigma(s)\ \exists k \leq n : \pi, k, M \models \psi \wedge \forall j < k\ \pi, j, M \models \varphi.$$

Tenemos

$$s \in Y \Longrightarrow s \in \bigcup_n Y_n \stackrel{\star}{\Longrightarrow} s \in \bigcup_n \tau^{n+1}(\bot) \Longrightarrow s \in \bigcup_n \tau^n(\bot) \Longrightarrow s \in [\![\phi_\mu]\!].$$

Sólo resta probar la implicación $\star$, que se sigue de $Y_n \subseteq \tau^{n+1}(\bot)$. Esto se prueba por inducción en n.

Caso $n = 0$
Si $s \in Y_0$, por definición de Y_0 tenemos que $s, M \models \psi$. Por hipótesis inductiva se cumple $s \in [\![\psi_\mu]\!]$. Por lo tanto, $s \in \tau(\bot)$.

Caso $n + 1$
Si $s \in Y_{n+1}$ entonces existe $\pi \in \Sigma(s)$ y $k \leq n + 1$ tal que

$$\pi, k, M \models \psi \wedge \forall j < k \ \pi, j, M \models \varphi.$$

Si $k = 0$ entonces $s, M \models \psi$, y por hipótesis inductiva $s \in [\![\psi_\mu]\!]$. Por lo tanto, para cada S vale que $s \in \tau(S)$. Tomando $S = \tau^n(\bot)$, este caso queda demostrado.
Cuando $k > 0$ tenemos que

$$s, M \models \varphi \wedge s \stackrel{e}{\to} s' \wedge \pi', k', M \models \psi \wedge \forall j < k' \ \pi', j, M \models \varphi,$$

donde $k' = k - 1$, $e = \pi^{\to}$,0, $s' = \pi$,1 y $\pi' = \pi[1, \infty]$. Notar que $s = \pi$,0 y $\pi' \in \Sigma(s')$.
Dado que $k' \leq n$, tenemos que

$$s, M \models \varphi \wedge \exists s', e \text{ tal que } s \stackrel{e}{\to} s' \wedge s', M \models \mathsf{E}_n(\varphi \mathcal{U} \ \psi).$$

Por hipótesis inductiva y definición de τ concluimos que $s \in \tau^{n+1}(\bot)$.

□

Es interesante mencionar que, dado que existe una función uno a uno entre estados (y tanto los estados como sus traducciones mantienen propiedades equivalentes), para cada traza en el modelo traducido, digamos un contraejemplo de una fórmula traducida, se puede contruir una traza en el modelo original que nos sirve como contraejemplo de la fórmula original. En otras palabras, los contraejemplos obtenidos utilizando model checkers de μ-caluclus pueden ser traducidos sistemáticamente en contraejemplos de la especificación original en DTL.

4.5. Casos de estudio

En esta sección desarrollamos dos casos de estudio de sistemas tolerantes a fallas. El primero es un caso de productor-consumidor en el cual los componentes se comunican a través de un canal con fallas. Este modelo es bastante simple, pero nos sirve para ejemplificar la traducción y algunas características de la lógica como lenguaje de especificación de sistemas tolerantes a fallas. El segundo caso corresponde a un diseño de circuito tolerante a retrasos (delay), un ejemplo conocido de sistema tolerante a fallas. Este caso de estudio nos agrega un beneficio adicional: puede ser escalado de una manera controlada, aumentando la cantidad de entradas (inputs). Lo utilizamos para comparar con la traducción previa y con el objeto de confirmar la escalabilidad obtenida del nuevo mecanismo de traducción.

En ambos casos, dado que el objetivo es una herramienta de model checking, utilizaremos la segunda traducción definida, que es más eficiente en complejidad. Por otro lado, utilizamos la herramienta de model checking Mucke [Bie97] para verificar los resultados de las especificaciones en DTL. Esta implemetación en Mucke se realizó en colaboración con Pablo Castro y Cecilia Kilmurray. Una versión preliminar del trabajo se encuentra publicada en [AKCA12].

4.5.1. Productor-consumidor

El primer caso de estudio consiste en un escenario de comunicación simple, compuesto por un productor, un consumidor y un canal utilizado para comunicar los dos primeros. Los modelos en la Figura 4.1 describen gráficamente estos componentes. Con el objeto de incorporar fallas y mecanismos de tolerancia a fallas, consideramos un canal con "ruido", por lo cual algunos mensajes pueden perderse. Para afrontar la pérdida de mensajes, el productor y consumidor utilizan un simple protocolo de comunicación que fuerza reenvíos del útlimo mensaje hasta que llega el mensaje correspondiente de confirmación. La única acción prohibida es la correspondiente a la pérdida de mensaje en el canal de comunicación. Los operadores deónticos nos permiten, indirectamente, referirnos a las ejecuciones normales (permitidas) y a las ejecuciones de falla (prohibidas).

En este capítulo no profundizaremos en la forma en que los componentes son sincronizados, pero básicamente, si una acción normal o de fallas se sincroniza con una acción de falla, la "acción compuesta" también se considera de falla. [7] Dependiendo del comportamiento del canal, el envío de un mensaje (snd) puede ser sincronizado con una acción permitida ($pass$) o con una acción de falla, i.e. de pérdida de mensaje ($lose$). El sistema resultante de componer el

[7] Estas "acciones compuestas" están representadas en el modelo como eventos.

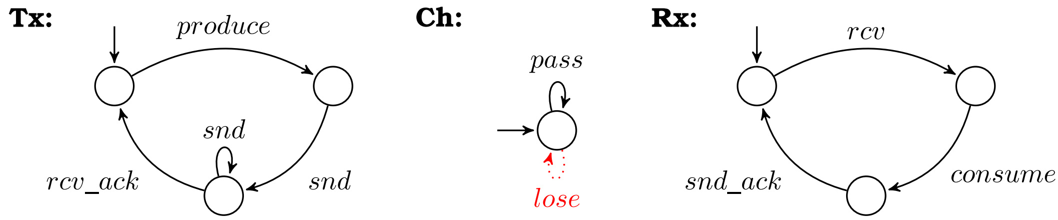


Figura 4.1: Un sistema productor-consumidor con un canal con fallas.

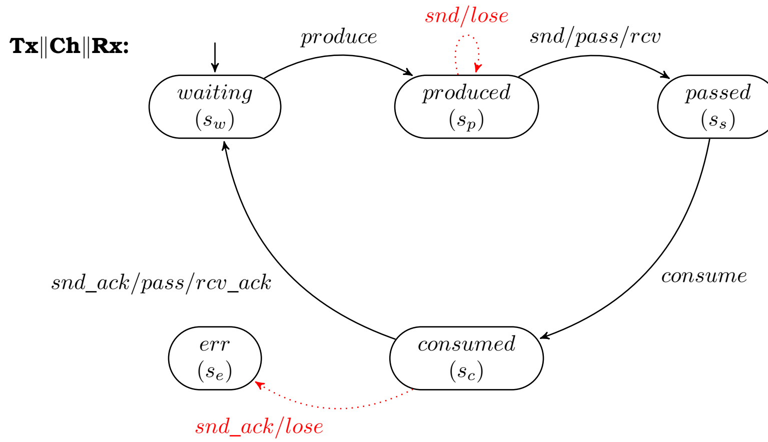


Figura 4.2: Composición del productor, consumidor y el canal con fallas.

productor, el consumidor y el canal se muestra en la Figura 4.2, para el cual los eventos de falla son $snd/lose$ y $snd_ack/lose$

En este ejemplo, la signatura de la estructura deóntica está dada por los siguientes conjuntos de proposiciones y acciones.

$$\Phi_0 = \{waiting, produced, passed, consumed\},$$
$$\Delta_0 = \{produce, consume, snd, rcv, snd_ack, rcv_ack, pass, lose\}.$$

Formalmente, la estructura deóntica para el modelo de productor-consumidor es la siguiente:

- El conjunto de estados $\mathcal{W}$ y de transiciones $\mathcal{R}$ se definen como en la Figura 4.2.
- El conjunto de eventos se define de la siguiente manera:

  $$\mathcal{E} = \{produce, consume, snd/lose, snd/pass/rcv, snd_ack/lose, snd_ack/pass/rcv_ack\},$$

- La interpretación de las variables proposicionales es la descripta en la Figura 4.2 y la interpretación de las acciones esta dada por

- $\mathcal{I}(produce) = \{produce\}$,
- $\mathcal{I}(consume) = \{consume\}$,
- $\mathcal{I}(snd) = \{snd/lose, snd/pass/rcv\}$,
- $\mathcal{I}(rcv) = \{snd/pass/rcv\}$,
- $\mathcal{I}(lose) = \{snd/lose, snd_ack/lose\}$,
- $\mathcal{I}(pass) = \{snd/pass/rcv, snd_ack/pass/rcv_ack\}$,
- $\mathcal{I}(rcv_ack) = \{snd_ack/pass/rcv_ack\}$,
- $\mathcal{I}(snd_ack) = \{snd_ack/lose, snd_ack/pass/rcv_ack\}$.

- Las transiciones permitidas representadas en $\mathcal{P}$ son las flechas continuas en la Figura 4.2. Las transiciones con las etiquetas (eventos) $snd/lose$ y $snd_ack/lose$ no están permitidas.

Observar que por cuestiones de una mejor comprensión del modelo, nombramos los eventos como la sincronización paralela de las acciones que lo componen.

Propiedades

Veamos ahora algunas propiedades de interés que podemos expresar en DTL, y que a raíz de la traducción pueden ser verificadas automáticamente.

P1 Cuando el sistema está en el estado *waiting* está obligado a producir un item:

$waiting \;\rightarrow\; \mathbf{O}(produce)$

P2 Luego de producir un item, si no ocurren fallas, el sistema está obligado a consumir:

$[\, produce \,]\; [\, \overline{lose} \,]\; \mathbf{O}(consume)$

P3 Si un ítem es producido entonces ningún otro ítem puede ser producido hasta que éste sea consumido.

$produced \;\rightarrow\; \mathsf{A}(\mathbf{O}(\overline{produce})\; \mathcal{U}\; consumed)$

P4 Cuando un ítem es consumido, no se conumen ítems adicionales hasta que un nuevo ítem es producido.

$consumed \rightarrow \mathsf{AN}(\mathsf{A}(\neg consumed\; \mathcal{W}\; produced))$

P5 Cuando un ítem es producido, todas las formas de ejecutar la acción *send* están permitidas.

$[\, produce \,]\; \mathbf{P}(snd)$

Traducción

Como nuestra traducción depende de la fórmula a verificar, en particular el conjunto de acciones que aparencen en las fórmulas, calculamos en primer lugar el conjunto $actionSet(\varphi)$, donde tomamos $\varphi = P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5$.

$$actionSet(P1 \wedge P2 \wedge P3 \wedge P4 \wedge P5) = \{produce, \overline{produce}, consume, \overline{consume}, \overline{lose}, \mathbf{U}, snd\}$$

Notar que $\overline{consume}$ aparece a partir de la definición de obligación.

La signatura de μ-calculus se obtiene traduciendo la signatura según la definición 4.23 :

- $\Delta_1 = \{produce, \overline{produce}, consume, \overline{consume}, \overline{lose}, \mathbf{U}, snd\}$,
- $\Phi_1 = \Phi_0 \cup \{AP_a \mid a \in \Delta_1\} \cup \{EP_a \mid a \in \Delta_1\}$,

y el modelo se obtiene traduciendo la estructura deóntica según la definición 4.24:

- $S = \mathcal{W} = \{s_w, s_p, s_s, s_c, s_e\}$,
- $T = \{s_w \xrightarrow{a} s_p | a \in \{produce, \overline{consume}, \overline{lose}, \mathbf{U}\}\} \cup \{s_p \xrightarrow{a} s_p | a \in \{\overline{produce}, \overline{consume}, snd, \mathbf{U}\}\} \cup \{s_p \xrightarrow{a} s_s | a \in \{\overline{produce}, \overline{consume}, \overline{lose}, snd, \mathbf{U}\}\} \cup \{s_s \xrightarrow{a} s_c | a \in \{\overline{produce}, consume, \overline{lose}, \mathbf{U}\}\} \cup \{s_c \xrightarrow{a} s_e | a \in \{\overline{produce}, \overline{consume}, \mathbf{U}\}\} \cup \{s_c \xrightarrow{a} s_w | a \in \{\overline{produce}, \overline{consume}, \overline{lose}, \mathbf{U}\}\}$,
- $L(p)$ como se describe en la Figura 4.2 para $p \in \Phi_0$,
- $L(AP_{produce}) = L(EP_{produce}) = \{s_w\}$
  $L(AP_{consume}) = L(EP_{consume}) = \{s_s\}$
  $L(AP_{\overline{produce}}) = \{s_w, s_s, s_e\}$
  $L(EP_{\overline{produce}}) = \{s_p, s_s, s_c\}$
  $L(EP_{\neg consume}) = \{s_w, s_p, s_c\}$
  $L(AP_{\neg consume}) = \{s_w\}$
  $L(AP_{\overline{lose}}) = \{s_w, s_p, s_s, s_c, s_e\}$
  $L(EP_{\overline{lose}}) = \{s_w, s_p, s_s, s_c\}$
  $L(EP_{snd}) = \{s_p\}$
  $L(AP_{snd}) = \emptyset$
  $L(EP_{\mathbf{U}}) = \{s_w, s_p, s_s, s_c\}$
  $L(AP_{\mathbf{U}}) = \{s_w, s_s, s_e\}$

Notar que si sólo quisiéramos verificar la fórmula **P1**, el conjunto de acciones Δ_1 se reduciría al conjunto $\{produce, \overline{produce}\}$, por lo que el modelo sería más pequeño. En consecuencia, cuando verificamos sistemas, las propiedades (fórmulas) a ser verificadas jugan un rol importante, ya que las acciones que aperencen afectan el tamaño del modelo resultante.

Las propiedades del modelo expresadas en DTL se traducen a μ-calculus como sigue:

TP1 $Tr(waiting \;\rightarrow\; \mathbf{O}(produce)) =$

$$waiting \;\rightarrow\; AP_{produce} \;\wedge\; \neg EP_{\overline{produce}}$$

TP2 $Tr([\,produce\,]\,[\,\overline{lose}\,]\,\mathbf{O}(consume)) =$

$$[\,produce\,]\,[\,\overline{lose}\,]\,AP_{consume} \wedge \neg EP_{\overline{consume}}$$

TP3 $Tr(produced \;\rightarrow\; \mathsf{A}(\mathbf{O}(\overline{produce})\;\mathcal{U}\;consumed)) =$

$$produced \;\rightarrow\; \mu R.\,(consumed \;\vee\; (AP_{\neg produce} \;\wedge\; \neg EP_{produce} \;\wedge\; [\,\mathbf{U}\,]\,R))$$

TP4 $Tr(consumed \rightarrow \mathsf{AN}(\mathsf{A}(\neg consumed\;\mathcal{W}\;produced))) =$

$$consumed \;\rightarrow\; [\,\mathbf{U}\,]\,((\mu R.\,(produced \;\vee\; (\neg consumed \;\wedge\; [\,\mathbf{U}\,]\,R)) \vee \nu R.\,(\neg consumed \;\wedge\; [\,\mathbf{U}\,]\,R)$$

TP5 $Tr([\,produce\,]\,P(snd)) =$

$$[\,produce\,]\,AP_{snd}$$

Recordar que el operador de obligación $\mathbf{O}(\alpha)$ se define como $\mathbf{P}(\alpha) \;\wedge\; \neg\mathbf{P}_w(\overline{\alpha})$

Empleamos el model checker Mucke para verificar estas fórmulas. Las propiedades **P1**, **P2**, **P4** se encontraron válidas en el modelo, mientras que **P3** y **P5** no lo son. Sorprendentemente, el problema en **P3** se pueden encontrar a simple vista; el contraejemplo encontrado por el model checker es el siguiente:

$$s_p \overset{\mathbf{U}}{\rightarrow} s_p \overset{\mathbf{U}}{\rightarrow} s_s \overset{\mathbf{U}}{\rightarrow} s_c$$

Notar que la transición $s_p \overset{\mathbf{U}}{\rightarrow} s_p$ no está permitida, y en consecuencia no vale la obligatoriedad, falsificando la fórmula $\mathbf{O}(\overline{produce})$.

4.5.2. Muller C-Element

El segundo caso de estudio es una formalización de un circuito tolerante a retrazos (*delay-insensitive circuit*) conocido como *Muller C-Element* [MC79]. El circuito tiene dos entradas booleanas (llamadas x, y) y una salida booleana (llamada z). Idealmente, la salida es $true$ cuando las dos entradas son $true$, y

$false$ cuando ambas entradas son $false$. El comportamiento de falla se introduce considerando retrasos en las entradas, de manera que la salida se mantiene en el mismo estado hasta que ambas entradas coinciden; es decir, memoriza el último estado en el que las dos entradas coinciden, tolerando de esa manera el comportamiento de falla del circuito. Este circuito es un ejemplo estándar de tolerancia a falla, como fue presentado en [AG93]. También es utilizado para presentar TLA en [Lam95].

Un C-element puede ser implementado con un circuito de mayoría con tres entradas. Se agrega una entrada u al circuito que realimenta la salida z. El modelo que captura esta implementación se muestra en la Figura 4.3. Cada estado, etiquetado con xyu/z, indica el valor de cada entrada x, y y u y la salida z, respectivamente. Se asume que el circuito de mayoría funciona correctamente, es decir, la salida del circuito siempre es el valor obtenido de la mayoría de sus entradas. También consideramos retrasos en la realimentación de z a u. Por ejemplo, aquella reflejada en la transición del estado $110/1$ al estado $010/0$, que significa que la entrada x cambió antes que u recibiera el valor de z. Veremos que nuestro modelo tolera retrasos en sus entradas, pero no tolera retrasos en la realimentación de la salida z a u. Por simplicidad, asumimos que el valor de u no cambia al mismo momento que cambian x o y.

Para este ejemplo consideramos la siguiente signatura:

$$\Phi_0 = \{x, y, z, u\},$$
$$\Delta_0 = \{cx, cy, cu\},$$

donde cx es la acción de cambiar el bit corresponidente a x, y de la misma manera para y y u. Recordemos que z cambia aplicando la mayoría a x, y, u, por lo tanto no consideramos una acción que cambie este bit. La estructura deóntica se define como sigue:

- el conjunto de estados $\mathcal{W}$ y la relación de transición $\mathcal{R}$ se describen en la Figura 4.3;
- el conjunto de eventos

$$\mathcal{E} = \{cx/cy, cx, cy, cu\};$$

- la interpretación de las variables proposicionales se muestran en la Figura 4.3, y la interpretación de las acciones está dada por
  - $\mathcal{I}(cx) = \{cx, cx/cy\}$,
  - $\mathcal{I}(cy) = \{cy, cx/cy\}$,
  - $\mathcal{I}(cu) = \{cu\}$;
- los eventos permitidos en $\mathcal{P}$ se representan con flechas continuas en la Figura 4.3.

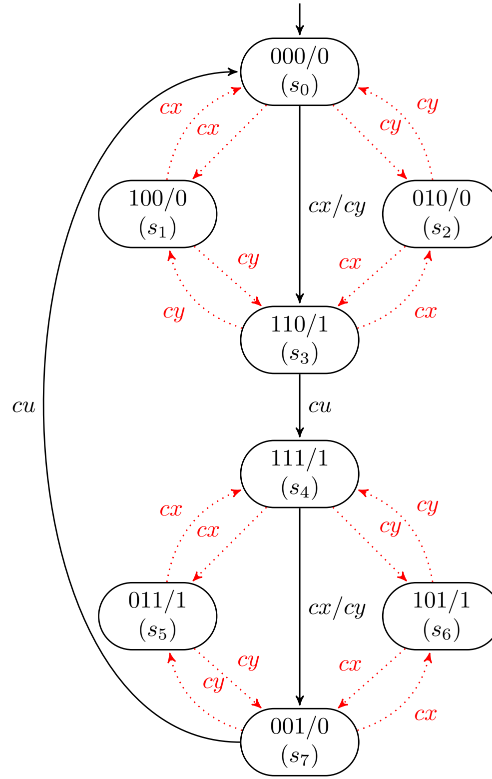


Figura 4.3: Un modelo de una celda Muller C-Element implementada por un circuito de mayoría.

Observar que para mayor claridad en la presentación, utilizamos el nombre de evento cx/cy para referirnos al evento generado por la ejecución paralela de las acciones cx y cy.

Propiedades

Las propiedades que queremos verificar son las siguientes:

C1 Es obligatorio cambiar x e y simultáneamente, o es obligatorio realimentar la salida a la entrada u.

$$\mathbf{O}(cx \sqcap cy) \vee \mathbf{O}(cu)$$

C2 Cuando no hay fallas, la salida coincide con el valor de las entradas.

$$\mathsf{AG}\neg v \to (x \equiv y) \wedge (y \equiv z)$$

C3 Cuando x, y y z coinciden, z maintiene su valor hasta que cambien x e y.

$$\neg x \wedge \neg y \wedge \neg z \to \mathsf{A}(\neg z \, \mathcal{U} \, x \wedge y)$$

Traducción

Como mostramos en el caso de estudio previo, el modelo traducido depende de la fórmula a ser verificada. En primer lugar calculamos el conjunto $actionSet$:

$$actionSet(\mathbf{C1} \wedge \mathbf{C2} \wedge \mathbf{C3}) = \{\mathbf{U}, cx \sqcap cy, cu, \overline{cx \sqcap cy}, \overline{cu}\}.$$

Recordar que la obligación está definida en términos de permisos de acciones negadas.

Para este caso de estudio se obtiene la siguiente signatura y modelo traducido con las funciones definidas en 4.23 y 4.24

- $\Delta_1 = \{\mathbf{U}, cx \sqcap cy, cu\}$;
- $\Phi_1 = \Phi_0 \cup \{AP_a \mid a \in \Delta_1\} \cup \{EP_a \mid a \in \Delta_1\}$;
- $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7\}$;
- $T = \{s_i \xrightarrow{\mathbf{U}} s_j \mid \exists e : s_i \xrightarrow{e} s_j \in R\} \cup \{s_3 \xrightarrow{cu} s_4\} \cup \{s_7 \xrightarrow{cu} s_0\} \cup \{s_0 \xrightarrow{cx \sqcap cy} s_3\} \cup \{s_4 \xrightarrow{cx \sqcap cy} s_7\} \cup \{s_i \xrightarrow{\neg cu} s_j \mid \exists e : s_i \xrightarrow{e} s_j \in R \wedge (s_i, s_j) \notin \{(s_3, s_4), (s_7, s_0)\}\} \cup \{s_i \xrightarrow{\neg cx \sqcap cy} s_j \mid \exists e : s_i \xrightarrow{e} s_j \in R \wedge (s_i, s_j) \notin \{(s_0, s_3), (s_4, s_7)\}\}$;
- $L(p)$ como se define en la Figura 4.3 para $p \in \Phi_0$;
- $L(AP_{cx \sqcap cy}) = L(EP_{cx \sqcap cy}) = \{s_0, s_4\}$;
- $L(AP_{cu}) = L(EP_{cu}) = \{s_7\}$;
- $L(AP_{\neg cu}) = L(EP_{\neg cu})\{s_0, s_4\}$;
- $L(AP_{\neg cx \sqcap cy}) = L(EP_{\neg cx \sqcap cy}) = \{s_3, s_7\}$.

Las fórmulas traducidas resultan de la siguiente manera:

TC1 $(AP_{cx \sqcap cy} \wedge \neg EP_{\neg cx \sqcap cy}) \vee (AP_{cu} \wedge \neg EP_{\neg cu})$

TC2 $\nu R.((\neg v \rightarrow (x \equiv y) \wedge (y \equiv z)) \wedge [\mathbf{U}]R)$

TC3 $\neg x \wedge \neg y \wedge \neg z \rightarrow \mu R.((x \wedge y) \vee (\neg z \wedge [\mathbf{U}]R))$

Las fórmulas se verificaron utilizando Mucke. Las fórmulas **TC1** y **TC2** se encontraron válidas, mientras que la fórmula **TC3** es falsa. El contraejemplo de la última propiedad está dado por la siguiente traza: $s_7 \rightarrow s_5$. En esta traza el valor de y cambia antes que el valor de u sea actualizado, por lo que el valor de z es incorrecto.

Como mencionamos anteriormente, este caso de estudio nos permite redimencionar el problema de una manera controlada. Utilizamos esta característica para evaluar si la traducción de DTL a μ-calculus, que preserva linealmente

N. Inputs	N.States	N.Transitions	Time	Memory	Time F.Translation	Mem. F.Translation
2	8	20	0.032s	19.7MB	0.040s	19.7MB
3	16	76	0.037s	19.7MB	0.046s	20MB
4	32	260	0.054s	19.7MB	0.083s	20.6MB
5	64	844	0.083s	20.5MB	0.143s	22.8MB
6	128	2660	0.146s	22MB	SO	-
7	256	8236	0.328s	26.8MB	SO	-

Tabla 4.1: Verificación de la propiedad **C2** con Mucke variando la cantidad de entradas al modelo C-Element. Performance de la traducción 4.25 y la versión previa de esta traducción mostrada en [AKCA12].

el tamaño de las fórmulas a lo largo de la traducción, efectivamente presenta una mejor escabilidad. Los experimentos se ejecutaron en una máquina vitual GNU/Linux de 32-bits con 4GB de memoria dedicada, sobre una MacBook Air i5 1.4GhZ de 8GB. Tomamos la propiedad **TC2**, aquella en la cual el tiempo de verificación tomó más tiempo en evaluarse para dos entradas, y evaluamos la performace del proceso de model checking variando la cantidad de entradas. En la Tabla 4.1 se muestran los resultados obtenidos de verificar la propiedad **C2** en un sistema C-element de hasta 7 entradas. La columna “N. Input” muestra el número de entradas usadas en el experimento, “N.States” (resp. “N.Transitions”) indica el número de estados (resp. transiciones) del modelo, las columnas “time” (resp. “Memory”) muestran el tiempo (resp. espacio) necesario para la verificación. Sólo a modo ilustativo se incluyen las columnas “Time F.Translation” and “Memory F.Translation” (utilizamos F para referirnos a la versión previa, “Former”) que indican el tiempo y el espacio, respectivamente, necesario para verificar la propiedad con versión previa de esta traducción descripta en [AKCA12] con características exponenciales. Vale la pena destacar que para un circuito con más de 4 entradas la propiedad obtenida con la versión previa de la traducción no se pudo resolver con Mucke. Dado su tamaño, la ejecución lleva a un *stack overflow* que se inidca con SO en la tabla.

4.6. Consideraciones finales

En este capítulo estudiamos una lógica deóntica temporal (DTL), que ya había sido propuesta para la especificación y el razonamiento de sistemas confiables [8], con el objetivo de asentar las bases para una herramienta que soporte esta lógica. Dasarrollamos dos traducciones de esta lógica a μ-calculus. La primera, de caracterísitcas exponenciales tanto en el modelo como en el tamaño de las fórmulas. Esta traducción tiene interés teórico en la medida en que muestra la factibilidad de realizar una traducción de la signatura y la estructura deóntica independientemente de la fórmula a verificar; es decir, que

[8] También se suele utilizar en castellano la palabra *dependibles* para referirnos al mismo concepto.

los modelos de μ-calculus (estructuras de Kripke) son suficientemente expresivos para capturar las estructuras deónticas. La segunda traducción, de menor complejidad, nos permitió demostrar que el problema de model checking puede ser resuelto en tiempo polinomial. Esto nos habilita una solución indirecta para resolver el problema de model checking de especificaciones en PDL con la utilización de model checkers de μ-calculus.

Desde un punto de vista técnico, las traducciones de DTL a μ-calculus involucran principalmente dos funciones, una que traduce la estructura deóntica en una estructura de Kripke y otra que convierte la fórmula DTL a μ-calculus. Se demuestra en ambos casos la corrección de la traducción, mostrando que el problema de model checking de fórmulas en PDL y DTL se reducen al problema de model checking de μ-calculus. En particular, mostramos que los contraejemplos se preservan, es decir, que cualquier contraejemplo que encontremos en el modelo traducido de μ-calculus, cuando verificamos una fórmula, puede ser reconstruido como contraejemplo de la fórmula original. También probamos la complejidad de la segunda traducción, más apta para el problema de model checking.

Por último, desarrollamos casos de estudio basados en la verificación de propiedades de sistemas tolerantes a fallas. Estos ejemplos nos permitieron mostrar usos concretos de la lógica y las estructuras deónticas para modelar sistemas tolerantes a fallas y algunas de sus propiedades más características. Además, mostramos la traducción a μ-calculus de estos sistemas y sus propiedades, y las verificamos con el model checker de μ-calculus Mucke.

La ingeniería de sofware en general, y sobre los sistemas confiables (o dependibles) en particular, están en una constante búsqueda de mecanismos más sistemáticos y estandarizados de desarrollo de sofware. Entendemos que metodologías de desarrollo de software sistemático requieren no sólo formalismos apropiados para capturar los aspectos relevantes del software a un nivel de abstracción apropiado, si no herramientas que permitan operar sobre estas verificaciones. Consideramos que el aporte hecho en este capítulo es importante, en la medida que brinda una herramienta de verificación automática para sistemas especificados en DTL. Esta lógica combina operdores temporales y deónticos en sus fórmulas, y permite expresar características sobre acciones y transiciones, y propiedades sobre los estados, lo que la hace muy potente para expresar características de los sistemas tolerantes a fallas.

We should never forget that programmers live in a world of artefacts, a fact that distinguishes them from most other scientists. The programmer should not ask how applicable the techniques of sound programming are, he should create a world in which they are applicable [...].

EWD 1305.

5 Conclusiones

El tema de esta tesis surgió de mi experiencia laboral en la Comisión Nacional de Actividades Espaciales (CONAE), en la que diseñábamos componentes para desarrollo de dispositivos aeroespaciales y de acceso al espacio. En ese marco conocí la conceptualización de la *tolerancia a fallas* planteada desde la ingeniería aeroespacial. Con un abordaje informal, se proponían recomendaciones en el diseño y recetas ad hoc para problemas conocidos. Pese a la informalidad, esta conceptualización permitía incorporar mayor rigurosidad al desarrollo de estos sistemas, y una nueva dimensión para analizarlos y evaluar su desempeño.

En este marco, consideramos que el uso de métodos formales para especificar, desarrollar y razonar sobre sistemas críticos, en particular sistemas aeroespaciales, es de suma relevancia. Los métodos formales nos proponen abordar los problemas desde una perspectiva diferente, no sólo incorporando herramientas lógico-matemáticas que contribuyen al diseño y el análisis de los mismos, sino también para el abordaje de los propios conceptos que surgen desde lo informal. Esto último, permite separar técnicas ad hoc y recomendaciones generales poco específicas, de aquellas que agregan un valor diferencial a la conceptualización general de la tolerancia a fallas y las características generales de los sistemas.

El recorrido del capítulo 1 en el que sistematizamos conceptos, clasificaciones y técnicas de la tolerancia a fallas del campo informal, permitió apropiarnos y revisar desde la perspectiva formal esos conceptos. Dado que los métodos formales permiten analizar la corrección del software, entendemos que las fallas provendrían del entorno o componentes externas no controladas por el diseñador, por lo que nuestro abordaje excluye los errores o *bugs* originados

en la implementación. Esto presenta una nueva perspectiva de análisis, sobre todo cuando analizamos las diferentes clasificaciones de fallas que existen en la bibliografía informal.

Como síntesis de esa etapa de recorrido bibliográfico, revisamos varios conceptos que pudimos formalizar posteriormente con lógicas específicas. Además, pudimos delinear consideraciones generales en relación al marco metodológico para abordar el diseño de sistemas tolerantes a fallas. En particular, aprendimos la importancia de diferenciar sistema de entorno, y la necesidad de contar con herramientas que permitan identificar diferentes componentes o subsistemas.

El capítulo 2 es de carácter exploratorio, mientras que el capítulo 3 presenta una lógica para la tolerancia a fallas, mejor elaborada y basada en la experiencia con lógicas deónticas previas. A través de ambos capítulos sintetizamos la búsqueda de lenguajes (lógicas) que permitan describir y razonar sobre los sistemas tolerantes a fallas. En el capítulo 2 destacamos la diferenciación de la especificación y el modelo de implementación que permitió identificar las fallas como eventos externos y ajenos a la especificación. A su vez, formalizamos conceptos de falla (fail), error, fallo (faliure) y algunas clasificaciones como la duración, el estado o la autonomía de las fallas. Otro aporte que nos interesa destacar es la idea de *modelo de fallas*, que propone describir el comportamiento de las fallas con un modelo específico que condiciona las posibles trazas sobre las que se evalúa el comportamiento general del sistema, entendiendo que las fallas pueden tener su propia lógica de ocurrencia y correlación que se diferencia de los eventos normales del sistema. Esa construcción permitió ampliar la noción de tolerancia a fallas.

Si bien realizamos nuestra primera formalización utilizando LTS reactivos, vemos que no aprovechamos suficientemente las particularidades de la *reactividad* para el trabajo que realizamos. A su vez, el tratar de describir características y propiedades con lenguajes operacionales, surgió una fuerte necesidad de explorar lenguajes más descriptivos que permitan simplificar estas definiciones. Por eso, en los capítulos 3 y 4 nos enfocamos en lógicas concretas.

A diferencia de los LTSs reactivos que se basan en transiciones y no cuentan con propiedades de estados, las lógicas abordadas en el capítulo 3, dCTL y fCTL, se basan en CTL. Los modelos de ambas lógicas se definen como variantes de estructuras de Kripke en el que distinguimos algunos estados o transiciones, respectivamente, como parte de un comportamiento *de falla*. Estos modelos, basados en estados, permitieron resignificar los conceptos de convergencia y recuperación, ya trabajados por Arora y Gouda en [AG93] y retomados posteriormente por varios autores dentro de la bibliografía de los métodos formales. Además, el evento (transición) de falla incorporado en el modelo, permitió incorporar operadores deónticos a las lógicas. Estos operadores distinguen en su semántica el comportamiento *normal*, sin la ocurrencia de fallas, del *anormal*, con ocurrencia de fallas, por lo que podemos verificar distintas propiedades

para cada caso. Con esto se logra recuperar resultados de lógicas modales y deónticas para ser aplicados a la tolerancia a fallas y permitiendo abordar los sistemas tolerantes a fallas de manera más conceptual.

La lógica fCTL es el resultado de la exploración y trabajo con dCTL, que evolucionó para dar lugar a una nueva lógica más acorde a las propiedades y conceptos que se quieren expresar. Esta lógica permitió expresar nuevas propiedades de la tolerancia a fallas, e incluso ampliar (debilitar en términos lógicos) el concepto mismo de tolerancia a fallas. Por ejemplo, se pudo expresar la idea de que un sistema tolera n fallas. En este sentido, retomamos la idea de *modelo de fallas* elaborada en el capítulo 2, incorporando elementos a las lógicas que permitieran describir condiciones al comportamiento sobre el cual evaluar propiedades (el antecedente del operador $\leadsto$). Esta idea, además de los operadores deónticos, es la que más nos interesó desde la perspectiva de la tolerancia a fallas al desarrollar tanto dCTL como fCTL.

La síntesis de esta tesis se desarrolló en el capítulo 4 en el que trabajamos con una lógica (DTL) que sintetiza varias de las ideas que surgieron en el recorrido previo y avanza sobre la elaboración de una herramienta para la verificación automática de propiedades sobre los modelos de la lógica (model checking). Los modelos de la lógica incorporan tanto eventos identificados como etiquetas en las transiciones como variables o propiedades de estado, lo cual da más expresividad a la hora de describir propiedades. Esta lógica también incorpora elementos deónticos como los trabajados en el capitulo 3. Como principal contribución de este capítulo, se realizaron dos traducciones de la lógica a μ-calculus con sus respectivas pruebas para poder contar con herramientas concretas de verificación automática de propiedades sobre la lógica.

Finalmente, consideramos que se puede profundizar más en los usos de la lógica y de estrategias metodológicas para describir y razonar sobre la tolerancia a fallas. Si bien en este capítulo se hizo poco énfasis en la metodología, consideramos que esta lógica, y principalmente la particularidad de darle semántica a las acciones con conjuntos de eventos, puede ser la puerta de entrada para poder describir componentes de manera más abstracta e independiente; e incorporar en los modelos, a partir del conjunto de eventos las posibles *interpretaciones* que resulten de la interrelación de las componentes. Al separar componentes de entorno, el cual no podemos controlar como el caso de las fallas, de las componentes a diseñar, podemos realizar modelos más acordes a la realidad. En este contexto, se presentaron estas estrategias a partir de casos de estudio que sería interesante seguir explorando para su generalización.

It is not the task of the University
to offer what society asks for, but
to give what society needs.

EWD 1305.

Código Mucke de casos de estudio

Mucke es un model checker simbólico para μ-calculus presentado en [Bie97]. Esta herramienta permite la verificación de propiedades de sistemas basados en estados, escritos en un código estilo *C*. A continuación se muestra el código fuente de los casos de estudio presentados en 4.5.1 y 4.5.2. Notar que la función T (que devuelve un booleano), además de modelar las transiciones de la estructura de Kripke, codifica algunas funciones auxiliares para referirnos a conjuntos de estados particulares, como por ejemplo la función `Produce`. Finalmente, formalizamos cada propiedad descripta en los casos de estudio 4.5.1 y 4.5.2 como funciones. Por ejemplo, la propiedad **P1** es capturada por la función `P1` que toma como entrada un estado. Mucke puede ser utilizado para verificar si los estados del modelo que satisfacen estas propiedades.

Para una introducción detallada de Mucke consultar [Bie97].

Código Mucke para caso de estudio productor-consumidor

A continuación se detalla el código Mucke para el caso de estudio detallado en 4.5.1.

```
enum S { waiting, produced, passed, consumed };
enum Delta1 {produce, consume, snd_lose, snd_pass_rcv, snd_ack_lose,
    snd_ack_pass_rcv_ack,snd};


bool T(S s, S t)
  case
    s = waiting : t = produced;
```

```
   s = produced : t = produced | t = passed;
   s = passed : t = consumed;
   s = consumed : t = produced | t = waiting;
  esac
;

/* ********************** Actions **************************** */
/*
* produce(s) states that s was reached after performing a produce
*/
bool Produce(S s)
s = produced
;

/*
* consume(s) states that s was reached after performing a consume
*/
bool Consume(S s)
s = consumed
;

/*
* snd_lose(s) states that s was reached after performing a send/lose
*/
bool Snd_lose(S s)
s = produced
;

/*
* snd_pass_rcv(s) states that s was reached after performing a
   snd_pass_rcv
*/
bool Snd_pass_rcv(S s)
s = passed
;

/*
* snd_ack_lose(s) states that s was reached after performing a
   snd_ack_lose
*/
bool Snd_ack_lose(S s)
s = produced
;

/*
* snd_ack_pass_rcv_ack(s) states that s was reached after performing
   a snd_ack_pass_rcv_ack
*/
bool Snd_ack_pass_rcv_ack(S s)
```

```
s = waiting
;

/*
* Lose(s) states that s was reached after performing a lose
*/
bool Lose(S s)
s = produced
;

/*
* U(s) states that s was reached after performing any action.
*/
bool U(S s)
( s = produced | s = waiting | s = passed | s = consumed)
;

/* ********* APa ********** */
//APa is true in all states where "a" is always allowed.
bool APa(Delta1 a, S s)
  case
    a = produce: s = waiting;
    a = consume : s = passed;
    a = snd_pass_rcv : false;
    a = snd_ack_pass_rcv_ack : false;
    a = snd_lose : false;
    a = snd_ack_lose : false;
    a = snd : false;
  esac
;

/* ********* EPa ********** */
//EPa is true in all states where "a" is allowed in some scenarios.
bool EPa(Delta1 a, S s)
  case
    a = produce: s = waiting;
    a = consume : s = passed;
    a = snd_pass_rcv : s = produced;
    a = snd_ack_pass_rcv_ack : false;
    a = snd_lose : s = produced;
    a = snd_ack_lose : false;
  esac
;

/* ********* APna ********** */
//APna is true in all states where "!a" is always allowed.
bool APna(Delta1 a, S s)
  case
    a = produce: s = passed;
```

```
    a = consume : s = waiting;
    a = snd_pass_rcv : false;
    a = snd_ack_pass_rcv_ack : false;
    a = snd_lose : ( (s=waiting) | (s=produced) | (s=passed)|
        (s=consumed) );
    a = snd_ack_lose : ( (s=waiting) | (s=produced) | (s=passed)|
        (s=consumed) );
  esac
;

/* ********* EPna ********** */
//EPna is true in all states where "!a" is allowed in some scenarios.
bool EPna(Delta1 a, S s)
  case
    a = produce: ((s=produced) | (s=passed)| (s=consumed));
    a = consume : ((s=produced) | (s=waiting)| (s=consumed));
    a = snd_pass_rcv : false;
    a = snd_ack_pass_rcv_ack : false;
    a = snd_lose : ( (s=waiting) | (s=produced) | (s=passed)|
        (s=consumed) );
    a = snd_ack_lose : ( (s=waiting) | (s=produced) | (s=passed)|
        (s=consumed) );
  esac
;

// O(produce) == APa(produce) & !EPna(produce)
bool Oproduce(S s)
( APa(produce,s) & !EPna(produce,s) )
;

// O(consume) == APa(consume) & ! EPna(consume)
bool Oconsume(S s)
 ( APa(consume,s) & !EPna(consume,s) )
;

// OnotProduce(s)
bool OnotProduce(S s)
( APna(produce,s) & !EPa(produce,s) )
;

// is true when an action different from Lose was executed.
bool NotLose(S s)
(Produce(s) | Consume(s) | Snd_pass_rcv(s) | Snd_ack_pass_rcv_ack(s)
    )
;

// notConsumed
bool notConsumed(S s)
(s=passed | s=waiting | s=produced )
```

```
;

// A ( O (!produce) U consumed )
mu bool A_OnotProd_U_consume(S s)
( s=consumed | ( OnotProduce(s) & (forall S t. !T(s,t) |
    A_OnotProd_U_consume(t) ) ) )
;

// A ( !consumed U produced) = u.R ( produced | ( !consumed & [U]R )
    )
mu bool A_notConsumed_U_produced(S s)
(s=produced | ( notConsumed(s) & (forall S t. !T(s,t) |
    A_notConsumed_U_produced(t) ) ) )
;

//neverConsume = G (!consumed) = ( v.R(!consumed & [U]R )
nu bool neverConsume(S s)
( s!=consumed & (forall S t. !T(s,t) | neverConsume(t) ) )
;

// AN_A_nConsumed_W_Produced == [U] { u.R ( produced | (! consumed &
    [U]R ) ) or ( v.R(!consumed & [U]R ) ) }
bool AN_A_nConsumed_W_Produced(S s)
( forall S t. !T(s,t) | ( A_notConsumed_U_produced(t) |
    neverConsume(t) ) );


/* *****************************Properties
    ************************** */

/* P1 = Whenever the system is in the waiting state, it is obliged
    to produce an item.
*/
bool P1(S s)
( (s != waiting) | Oproduce(s) )
;

/* P2 = After the production of an item, if no fault occurs, then
    the system is obliged to consume.
*/
bool P2(S s)
   (exists S t1. T(s,t1) & Produce(t1) & (exists S t2. T(t1,t2) &
      !Lose(t2) & Oconsume(t2) ) )
;

/* P3 = If an item has been produced, then no new item ought to be
    produced until the current item has been consumed.
*/
bool P3(S s)
```

```
  (s!=produced | A_OnotProd_U_consume(s) )
;

/* P4 = When a item has been consumed, no additional items are
    consumed until some new item is produced.
*/
bool P4(S s)
  ( s!=consumed | AN_A_nConsumed_W_Produced(s) )
;

/* P5 = When an item is produced, all possible ways of performing
    the send action are allowed.
*/
bool P5(S s)
  ( Produce(s) & APa(snd,s) )
;
```

Código Mucke para caso de estudio Muller C-Element

A continuación se detalla el código Mucke para el caso de estudio detallado en 4.5.2.

```
enum S { s0, s1, s2, s3, s4, s5, s6, s7 };
enum Delta1 {cx, cy, cu, cx_cy };

bool T(S s, S t)
  case
    s = s0 : t = s1 | t = s2 | t = s3 ;
    s = s1 : t = s0 | t = s3;
    s = s2 : t = s0 | t = s3;
    s = s3 : t = s1 | t = s2 | t = s4;
    s = s4 : t = s5 | t = s6 | t = s7 ;
    s = s5 : t = s4 | t = s7;
    s = s6 : t = s4 | t = s7;
    s = s7 : t = s5 | t = s6 | t = s0 ;
  esac
;

/* ********************** Actions **************************** */
/*
* Cx(s) states that s was reached after performing a change in "x"
*/
bool Cx(S s)
(s=s0 | s=s1 | s=s2 | s=s3 | s=s4 | s=s5 | s=s6 | s=s7 )
;

/*
```

```
* Cy(s) states that s was reached after performing a change in "y"
*/
bool Cy(S s)
(s=s0 | s=s1 | s=s2 | s=s3 | s=s4 | s=s5 | s=s6 | s=s7 )
;

/*
* Cy(s) states that s was reached after performing a change in "y"
*/
bool Cu(S s)
(s=s0 | s=s4 )
;

/*
* Cx_Cy(s) states that s was reached after performing a change in
    "x" and "y" simultaneously
*/
bool Cx_Cy(S s)
(s=s3 | s=s7 )
;


/* ********* APa ********** */
//APa is true in all states where "a" is always allowed.
bool APa(Delta1 a, S s)
  case
    a = cu: ( (s=s3) | (s=s7) );
    a = cx_cy : ( (s=s0) | (s=s4) );
    a = cx : false;
    a = cy : false;
  esac
;

/* ********* EPa ********** */
//EPa is true in all states where "a" is allowed in some scenarios.
bool EPa(Delta1 a, S s)
  case
    a = cu: ( (s=s3) | (s=s7) );
    a = cx_cy : ( (s=s0) | (s=s4) );
    a = cx : false;
    a = cy : false;
  esac
;

/* ********* APna ********** */
//APna is true in all states where "!a" is always allowed.
bool APna(Delta1 a, S s)
  case
    a = cu: ( (s=s0) | (s=s4) );
```

```
    a = cx_cy : ( (s=s3) | (s=s7) ) ;
  esac
;

/* ********* EPna ********** */
//EPna is true in all states where "!a" is allowed in some scenarios.
bool EPna(Delta1 a, S s)
  case
    a = cu: ( (s=s0) | (s=s4) );
    a = cx_cy : ( (s=s3) | (s=s7) );
  esac
;

// O(cx_cy)
bool Ocx_cy(S s)
( APa(cx_cy,s) & !EPna(cx_cy,s) )
;

//O(cu)
bool Ocu(S s)
( APa(cu,s) & !EPna(cu,s) )
;

//CoincideXY: is true in all states where the values of "x" and "y"
    are the same.
bool CoincideXY(S s)
( s=s0 | s=s3 | s=s4 | s=s7 )
;

//CoincideYZ : is true in all states where the values of "y" and "z"
    are the same.
bool CoincideYZ(S s)
( s=s0 | s=s3 | s=s4 | s=s7 )
;

//CoincideUZ: is true in all states where the values of "u" and "z"
    are the same.
bool CoincideUZ(S s)
( s=s0 | s=s1 | s=s2 | s=s4 | s=s5 | s=s6 )
;

//Violation: is true in all states where the values of "x" and "y",
    or "u" and "z" are different.
bool Violations(S s)
 !CoincideXY(s) | !CoincideUZ(s)
;

//x: is true in all states where the value of "x" is 1.
bool x(S s)
```

```
( s=s1 | s=s3 | s=s4 | s=s6)
;

//y: is true in all states where the value of "y" is 1.
bool y(S s)
( s=s2 | s=s3 | s=s4 | s=s5)
;

//z: is true in all states where the value of "z" is 1.
bool z(S s)
( s=s3 | s=s4 | s=s5 | s=s6)
;

// AG(!Violations(s) -> coincideXY(s) & coincideYZ(s) )
nu bool AG_notV_coincideIO(S s)
( ( Violations(s) | ( CoincideXY(s) & CoincideYZ(s) ) ) & (forall S
    t. !T(s,t) | AG_notV_coincideIO(t) ) )
;

// A (!z U (x & y)
mu bool A_nz_U_xy(S s)
( ( x(s) & y(s) ) | ( !z(s) & (forall S t. !T(s,t) | A_nz_U_xy(t) )
    ) )
;

/* *****************************Properties
    ************************** */

/* C1 = It is obligatory to change "x" and change "y"
    simultaneously, or it is obligatory to feed back the output to
    input "u".
*/
bool C1(S s)
( Ocx_cy(s) | Ocu(s) )
;

/* C2 = When no violations are present, the output contains the same
    value as the inputs.
 */
bool C2(S s)
  AG_notV_coincideIO(s)
;

/* C3 = when "x", "y" and "z" coincides, "z" maintains its value
    until "x" and "y" change.
*/
bool C3(S s)
  ( ( x(s) | y(s) | z(s) ) | A_nz_U_xy(s) )
;
```

Bibliografía

[ACD$^+$06] ARLAT, J., CROUZET, Y., DESWARTE, Y., FABRE, J.-C., LAPRIE, J.-C., AND POWELL, D. Tolérance aux fautes. *Les Editions Vuibert, J. Akoka, I. Comyn-Wattiau (Eds)* (2006), 241–270.

[AG93] ARORA, A., AND GOUDA, M. G. Closure and convergence: A foundation of fault-tolerant computing. *Software Engineering 19*, 11 (1993), 1015–1027.

[AK98] ARORA, A., AND KULKARNI, S. S. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the The 18th International Conference on Distributed Computing Systems* (Washington, DC, USA, 1998), ICDCS '98, IEEE Computer Society, p. 436.

[AKCA12] ACOSTA, A., KILMURRAY, C., CASTRO, P. F., AND AGUIRRE, N. Model checking propositional deontic temporal logic via a u-calculus characterization. In *SBMF* (2012), pp. 3–18.

[Avi85] AVIZIENIS, A. The n-version approach to fault-tolerant software. *IEEE Trans. Software Eng. 11*, 12 (1985), 1491–1501.

[BCJM00] BEREZIN, S., CLARKE, E., JHA, S., AND MARRERO, W. *Model Checking Algorithms for the μ-Calculus.* MIT Press, Cambridge, MA, USA, 2000, p. 309–337.

[BF15] BURLYAEV, D., AND FRADET, P. Formal verification of automatic circuit transformations for fault-tolerance. In *Formal Methods in Computer-Aided Design, FMCAD 2015, Austin, Texas, USA, September 27-30, 2015* (2015), R. Kaivola and T. Wahl, Eds., IEEE, pp. 41–48.

[Bie97] BIERE, A. μcke - efficient μ-calculus model checking. In *CAV* (1997), O. Grumberg, Ed., vol. 1254 of *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, Springer, pp. 468–471.

[BJRT09] BUTLER, M., JONES, C. B., ROMANOVSKY, A., AND TROUBITSYNA, E., Eds. *Methods, Models and Tools for Fault Tolerance*, vol. 5454 of *Lecture Notes in Computer Science.* Springer, 2009.

[BK08] BAIER, C., AND KATOEN, J.-P. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[CBJM96] CLARKE, E., BEREZIN, S., JHA, S., AND MARRERO, W. Model checking algorithms for the μ-calculus, 1996.

[CES86] CLARKE, E. M., EMERSON, E. A., AND SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. 8*, 2 (April 1986), 244–263.

[CGP99] CLARKE, J. E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. The MIT Press, 1999.

[CKAA11] CASTRO, P. F., KILMURRAY, C., ACOSTA, A., AND AGUIRRE, N. dctl: A branching time temporal logic for fault-tolerant system verification. In *Software Engineering and Formal Methods* (Berlin, Heidelberg, 2011), G. Barthe, A. Pardo, and G. Schneider, Eds., Springer Berlin Heidelberg, pp. 106–121.

[CM08] CASTRO, P. F., AND MAIBAUM, T. S. E. A tableaux system for deontic action logic. In *Proceedings of 9th International Conference on Deontic Logic in Computer Science , Luxembourg.* (2008), pp. 34–48.

[CM09a] CASTRO, P. F., AND MAIBAUM, T. S. E. Deontic action logic, atomic boolean algebra and fault-tolerance. *Journal of Applied Logic 7*, 4 (2009), 441–466.

[CM09b] CASTRO, P. F., AND MAIBAUM, T. S. Reasoning about system-degradation and fault-recovery with deontic logic. *Methods, Models and Tools for Fault Tolerance* (2009), 25–43.

[Dij74] DIJKSTRA, E. W. Self-stabilization in spite of distributed control. *Commun. ACM 17*, 11 (1974), 643–644.

[dLF02] DE LEMOS, R., AND FIADEIRO, J. L. An architectural support for self-adaptive software for treating faults. In *WOSS '02: Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), ACM, pp. 39–42.

[Dub13] DUBROVA, E. *Fault-Tolerant Design*. Springer Publishing Company, Incorporated, 2013.

[DvdHT02] DASHOFY, E. M., VAN DER HOEK, A., AND TAYLOR, R. N. Towards architecture-based self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), ACM, pp. 21–26.

[Eme96] EMERSON, E. A. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models* (1996), pp. 185–214.

[Gab08] GABBAY, D. M. Reactive kripke models and contrary to duty obligations. In *DEON* (2008), R. van der Meyden and L. van der Torre, Eds., vol. 5076 of *Lecture Notes in Computer Science*, Springer, pp. 155–173.

[Gab12] GABBAY, D. M. Introducing reactive kripke semantics and arc accessibility. *Ann. Math. Artif. Intell. 66*, 1-4 (2012), 7–53.

[Gab13] GABBAY, D. M. *Reactive Kripke Semantics*. Springer Publishing Company, Incorporated, 2013.

[Gär99] GÄRTNER, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys 31*, 1 (1999), 1–26.

[GB92] GOGUEN, J. A., AND BURSTALL, R. M. Institutions: Abstract model theory for specification and programming. *J. ACM 39*, 1 (jan 1992), 95–146.

[GJM03] GHEZZI, C., JAZAYERI, M., AND MANDRIOLI, D. *Fundamentals of software engineering (2. ed.)*. Prentice Hall, 2003.

[GMK02] GEORGIADIS, I., MAGEE, J., AND KRAMER, J. Self-organising software architectures for distributed systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), ACM, pp. 33–38.

[GR02] GOGUEN, J. A., AND ROSU, G. Institution morphisms. *Formal Asp. Comput. 13*, 3-5 (2002), 274–307.

[GS02] GARLAN, D., AND SCHMERL, B. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems* (New York, NY, USA, 2002), ACM, pp. 27–32.

[GSRRU07] GHOSH, D., SHARMAN, R., RAGHAV RAO, H., AND UPADHYAYA, S. Self-healing systems - survey and synthesis. *Decis. Support Syst. 42*, 4 (2007), 2164–2185.

[Jal94] JALOTE, P. *Fault Tolerance in Distributed Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[JHAL09] JEFFORDS, R. D., HEITMEYER, C. L., ARCHER, M., AND LEONARD, E. I. A formal method for developing provably correct fault-tolerant systems using partial refinement and composition. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The*

Netherlands, November 2-6, 2009. Proceedings (2009), A. Cavalcanti and D. Dams, Eds., vol. 5850 of *Lecture Notes in Computer Science*, Springer, pp. 173–189.

[KK07] KOREN, I., AND KRISHNA, C. M. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[Lam95] LAMPORT, L. Tla in pictures. *IEEE Trans. Software Eng. 21*, 9 (1995).

[MC79] MEAD, C., AND CONWAY, L. *Introduction to VLSI systems*. Addison-Wesley, 1979.

[MH01] MONIN, J. F., AND HINCHEY, M. G. *Understanding Formal Methods*. Springer-Verlag, Berlin, Heidelberg, 2001.

[MM06] MAGEE, J., AND MAIBAUM, T. Towards specification, modelling and analysis of fault tolerance in self managed systems. In *Proceedings of the 2006 International Workshop on Self-Adaptation and Self-Managing Systems* (New York, NY, USA, 2006), SEAMS '06, Association for Computing Machinery, p. 30–36.

[Nel90] NELSON, V. P. Fault-tolerant computing: Fundamental concepts. *IEEE Computer 23*, 7 (1990), 19–25.

[O'R17] O'REGAN, G. *Concise Guide to Formal Methods - Theory, Fundamentals and Industry Applications*. Undergraduate Topics in Computer Science. Springer, 2017.

[Par06] PARNAS, D. *Mathematical Approaches to Software Quality*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006, ch. 8, pp. 143–175.

[PST13] POTIRON, K., SEGHROUCHNI, A. E. F., AND TAILLIBERT, P. *From Fault Classification to Fault Tolerance for Multi-Agent Systems*. Springer Publishing Company, Incorporated, 2013.

[Pul01] PULLUM, L. L. *Software Fault Tolerance Techniques and Implementation*. Artech House, Inc., Norwood, MA, USA, 2001.

[Rav11] RAVINDRAN, K. Probabilistic fault-tolerance of distributed services: A paradigm for dependable applications. In *Sixth International Conference on Availability, Reliability and Security, ARES 2011, Vienna, Austria, August 22-26, 2011* (2011), IEEE Computer Society, pp. 75–82.

[Sah06] SAHA, G. K. Software based fault tolerance: a survey. *Ubiquity 7*, 25 (2006), 1–1.

[Sch04] SCHNEIDER, K. *Verification of Reactive Systems, Formal Methods and Algorithms*. Springer, 2004.

[SWH94] SURI, N., WALTER, C. J., AND HUGUE, M. M. *Advances in ULTRA-Dependable Distributed Systems*. IEEE Computer Society Press, Washington, DC, USA, 1994.

[Tel00] TEL, G. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[Wil00] WILFREDO, T. Software fault tolerance: A tutorial. Tech. rep., NASA Langley Technical Report Server, 2000.